



Politechnika Łódzka

Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki

Programowanie sieciowe

Wykład 5 Programowanie wywołań asynchronicznych



Opracował: dr hab. inż. Radosław Wajman

Delegacje

- ▶ Delegacja pozwala utworzyć nowy typ danych, który definiuje sygnaturę metody.
- ▶ Obiekty delegacji mogą wywoływać metody zgodne z ich sygnaturą.
Ogólna deklaracja delegacji:

```
[modyfikatory] delegate typ nazwaDelegacji ( ... ) ;
```

- ▶ np.

```
delegate int Policz (string s); // <- Deklaracja (typu) delegacji mogącej wywoływać metody  
                                // pobierające parametr typu string i zwracające int.  
  
Policz p = Policz( nazwaMetody ); // <- Utworzenie obiektu delegacji  
p( );                             // <- Wywołanie metody wskazywanej przez delegację
```

Delegacje

- ▶ Jeśli utworzymy delegację typu **void**, to będzie ona mogła wskazywać i wywoływać wiele metod jednocześnie (delegacje inne niż **void** nie mogą rejestrować wielu metod, gdyż nie miało by sensu zwracanie wyniku wywołania wielu metod na raz).
- ▶ W takim wypadku rejestrujemy kolejne metody, które ma wywołać delegacja dzięki operatorowi **+=**, natomiast wyrejestrowujemy je za pomocą **-=**.
- ▶ Przykładowo:

```
class A {  
    delegate void ListyFunkcji ( );           // Deklaracja delegacji typu void.  
  
    public static void Main( ) {  
  
        ListyFunkcji listaInicjalizacyjna = null; // <- Utworzenie obiektu delegacji  
        listaInicjalizacyjna += przywitaj;       // <- Zarejestrowanie dodatkowej funkcji  
  
        ...                                     // Następnie obiekt  
        listaInicjalizacyjna();                 // moglibyśmy przekazać do funkcji lub wywołać.  
    }  
  
    void przywitaj() {...}                     // <- Funkcja zarejestrowana w delegacji.  
}
```

Delegacje - Callback

- ▶ Callback to kod wykonywalny przekazywany jako parametr do innego kodu,
- ▶ Implementowany jest m.in. za pomocą delegacji,
- ▶ Sposób na wywołanie czasochłonnych zadań i na informowanie w miejscu delegacji, gdy zadanie się zakończy.

```
public delegate void WorkCompletedCallback(string result);
```

```
1 reference  
public void DoWork(WorkCompletedCallback callback)  
{  
    callback("Hello world");  
}
```

```
1 reference  
public void Test()  
{  
    WorkCompletedCallback callback = TestCallback;  
    DoWork(callback);  
}
```

```
1 reference  
public void TestCallback(string result)  
{  
    Console.WriteLine(result);  
}
```

Operator lambda - anonimowość

- Operator lambda jest funkcją anonimową służącą do tworzenia delegatów,
- Za jego pomocą można pisać funkcje lokalne, które mogą być przekazywane jako argumenty lub zwracane jako wartość wywołania funkcji,
- Aby utworzyć wyrażenie lambda, należy po lewej stronie operatora lambda => określić parametry wejściowe (jeśli istnieją) i umieścić blok wyrażenia lub instrukcji po prawej.
- Wyrażenie lambda $x \Rightarrow x * x$ określa parametr o nazwie x i zwraca wartość x kwadratów, a następnie całość można przypisać do typu delegata:

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

Operator lambda - anonimowość

- Operator lambda (`=>`) ma takie samo pierwszeństwo jak operator przypisania (`=`) i następuje łączność do prawej strony,
- Ogólna forma to:
`(input parameters) => expression`
- Nawiasy są opcjonalne tylko wtedy, gdy lambda ma jeden parametr wejściowy; w przeciwnym razie są wymagane.
Co najmniej dwa parametry wejściowe są rozdzielane przecinkami w nawiasach:
`(x, y) => x == y`
- Czasami jest trudne lub wręcz niemożliwe dla kompilatora wywnioskowanie typów wejściowych dlatego można określić typy jawnie:
`(int x, string s) => s.Length > x`
- Określanie braku parametrów wejściowych za pomocą pustych nawiasów:
`() => SomeMethod()`

Instrukcja lambda

- Instrukcja lambda jest podobna do wyrażenia lambda, z tym że instrukcje są ujęte w nawiasy klamrowe:

```
(input parameters) => {statement;}
```

- Treść instrukcji lambda może składać się z dowolnej liczby instrukcji

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

Zdarzenia

- Zdarzenia używane są do oprogramowania czynności zachodzących w trakcie działania aplikacji,
- Takim zdarzeniem może być ruch myszą, kliknięcie w obrębie komponentu, przesunięcie kontrolki i wiele innych,
- Po umieszczeniu przykładowej kontrolki na formie i kliknięciu w niego, generowany jest automatycznie kod:

```
private void button1_Click(object sender, EventArgs e)
{
}
}
```


Zdarzenia

- Jak to się dzieje, że po naciśnięciu przycisku wykonywany jest kod znajdujący się w tej metodzie?
- Zdarzenia w dużej mierze opierają się właśnie na delegatach,
- W pliku **.designer.cs* można odnaleźć instrukcje, które odpowiadają za utworzenie przycisku Button na formularzu oraz za przypisanie określonej metody do danego zdarzenia:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

- Zdarzenie Click zadeklarowane jest w klasie Control

```
public event EventHandler Click;
```

- Delegaty deklarujemy z użyciem słowa kluczowego **delegate**, a zdarzenia — z użyciem słowa **event**.

Fraza EventHandler nie należy do słów kluczowych języka C#. Jest to nazwa delegatu określonego w przestrzeni System:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Zdarzenia

- Nic nie stoi zatem na przeszkodzie, aby przypisać do danego zdarzenia więcej niż jedną metodę:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Procedura zdarzeniowa #1");
}

private void myButton_Click(object sender, EventArgs e)
{
    MessageBox.Show("Procedura zdarzeniowa #2");
}

private void Form1_Load(object sender, EventArgs e)
{
    button1.Click += button1_Click;
    button1.Click += myButton_Click;
}
```

JAVA: od klasy anonimowej do wyrażenie lambda

- Klasa anonimowa to stworzenie jedynej instancji klasy w miejscu jej użycia.
- Definicja klasy anonimowej połączona jest z tworzeniem jej jedynej instancji.
- Klasy anonimowe zawsze są klasami wewnętrznymi.

```
// using an anonymous class
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello");
    }
};
```

JAVA: od klasy anonimowej do wyrażenie lambda

- W Javie zapis wyrażenia lambda jest identyczny, jak w C#,
- Runnable jest interfejsem funkcyjnym tzn. ma tylko jedną metodę,
- Wyrażenie lambda domyślnie ją nadpisuje.

```
// using an anonymous class
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello");
    }
};

// using a lambda expression
Runnable r2 = () -> System.out.println("Hello");
```

JAVA: wyrażenie lambda

```
int zmienna = 1;
Runnable r = new Runnable() {
    @Override
    public void run() {
        // tu jest ok!
        int zmienna = 2;
    }
};

Runnable r2 = () -> {
    // compile error: nie można zadeklarować lokalnej zmiennej w wyrażeniu Lambda
    // jeśli zadeklarowana jest taka sama zmienna w strefie zawierającej wyrażenie Lambda
    int zmienna = 2;
};
```

JAVA: wyrażenie lambda

```
// Jakiś interface funkcyjny
interface Task {
    public void execute();
}

// Metody przeciążone
public static void go(final Runnable r) {
    r.run();
}
public static void go(final Task t) {
    t.execute();
}

// Ich wywołanie

// Gdy klasa anonimowa nie ma problemu ze sprecyzowaniem typu
go(new Task() {
    @Override
    public void execute() {
        System.out.println("Hello");
    }
});

// Przy użyciu wyrażenia lambda expression mamy błąd kompilacji!!
// Wywołanie metody nie jest jednoznaczne
go(() -> {
    System.out.println("Hello");
});

// Aby jednoznacznie wskazać metodę
go((Task) () -> {
    System.out.println("Hello");
});
```

Lambda asynchronicznie: C#

- Można łatwo tworzyć wyrażenia lambda i instrukcje, które obejmują wywołania asynchroniczne za pomocą słów kluczowych **async** i **await**,
- Przykład zawiera program obsługi zdarzeń, który wywołuje i czeka na metodę asynchroniczną

```
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);
    }
}
```

Lambda asynchronicznie

- Sam program obsługi zdarzeń można dodać, używając lambdy asynchronicznej,
- Aby dodać taką obsługę, należy użyć modyfikator **async** przed listą parametrów lambda:

```
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);
    }
}
```


Asynchroniczność

- ▶ Asynchroniczność jest istotna dla działań, które mogą powodować blokowanie, np. kiedy aplikacja uzyskuje dostęp do sieci web.
- ▶ Dostęp do zasobów sieci web bywa powolny lub opóźniony i jeśli takie działanie jest zablokowane w procesie synchronicznym, cała aplikacja musi czekać.
- ▶ W procesie asynchronicznym aplikacja może kontynuować wykonywanie innych zadań, które nie są zależne od zasobów sieci web przed zakończeniem zadania mogącego powodować blokowanie.
- ▶ Poniżej wymienione zostały interfejsy API środowiska .NET Framework 4.5 oraz WinRT, które zawierają metody z obsługą wywołań asynchronicznych:

Obszar aplikacji	Obsługa interfejsów API, która obejmuje metody asynchroniczne
Dostęp do sieci Web	HttpClient , SyndicationClient
Praca z plikami	StorageFile , StreamWriter , StreamReader , XmlReader
Praca z obrazami	MediaCapture , BitmapEncoder , BitmapDecoder
Programowanie WCF	Operacje synchroniczne i asynchroniczne

Asynchroniczność

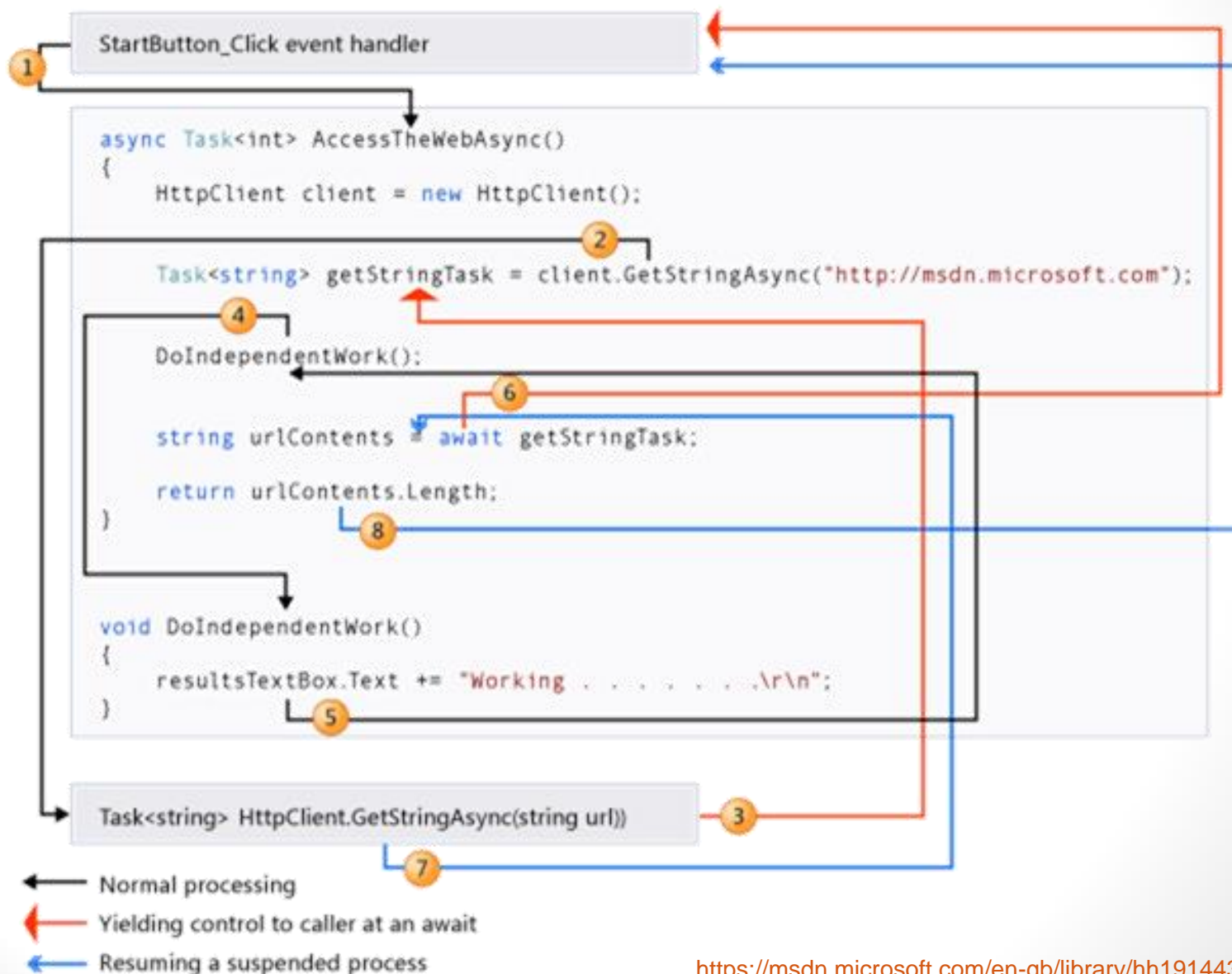
- Asynchroniczność okazuje się szczególnie cenna w przypadku aplikacji, które uzyskują dostęp do wątku interfejsu użytkownika, ponieważ wszystkie działania związane z interfejsem użytkownika korzystają zazwyczaj z jednego wątku.
- Jeśli w aplikacji synchronicznej jest zablokowany jakikolwiek proces, to wszystko jest zablokowane.
Aplikacja przestanie odpowiadać, i może uznać, że uległa awarii zamiast po prostu czekać.
- Przy użyciu metod asynchronicznych, aplikacja dalej odpowiada interfejsowi.

Łatwość implementacji

- Za pomocą dwóch słów kluczowych (**async** i **await**), można użyć zasobów w programie .NET Framework lub Windows Runtime (WinRT) do utworzenia metody asynchronicznej niemalże podobnie łatwo jak metody synchronicznej.
- Metody asynchroniczne definiowane za pomocą **async** i **await** są określane jako metody **async**.
- Zainteresowanych odsyłam do strony:
<https://msdn.microsoft.com/en-us/library/mt674882.aspx>

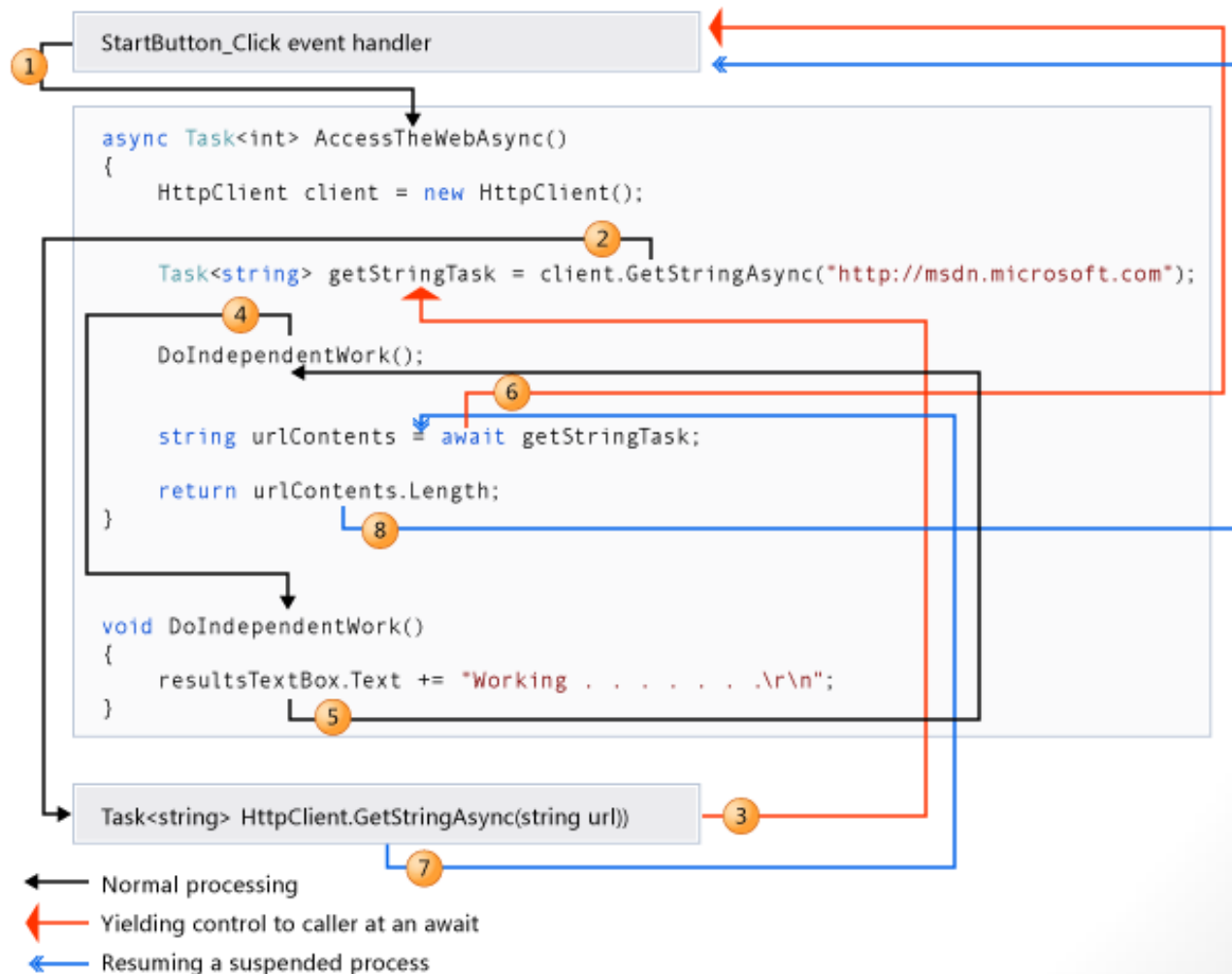
Jak to działa?

- Poniżej przykład wywołania metody pobierającej dane z sieci w trybie asynchronicznym



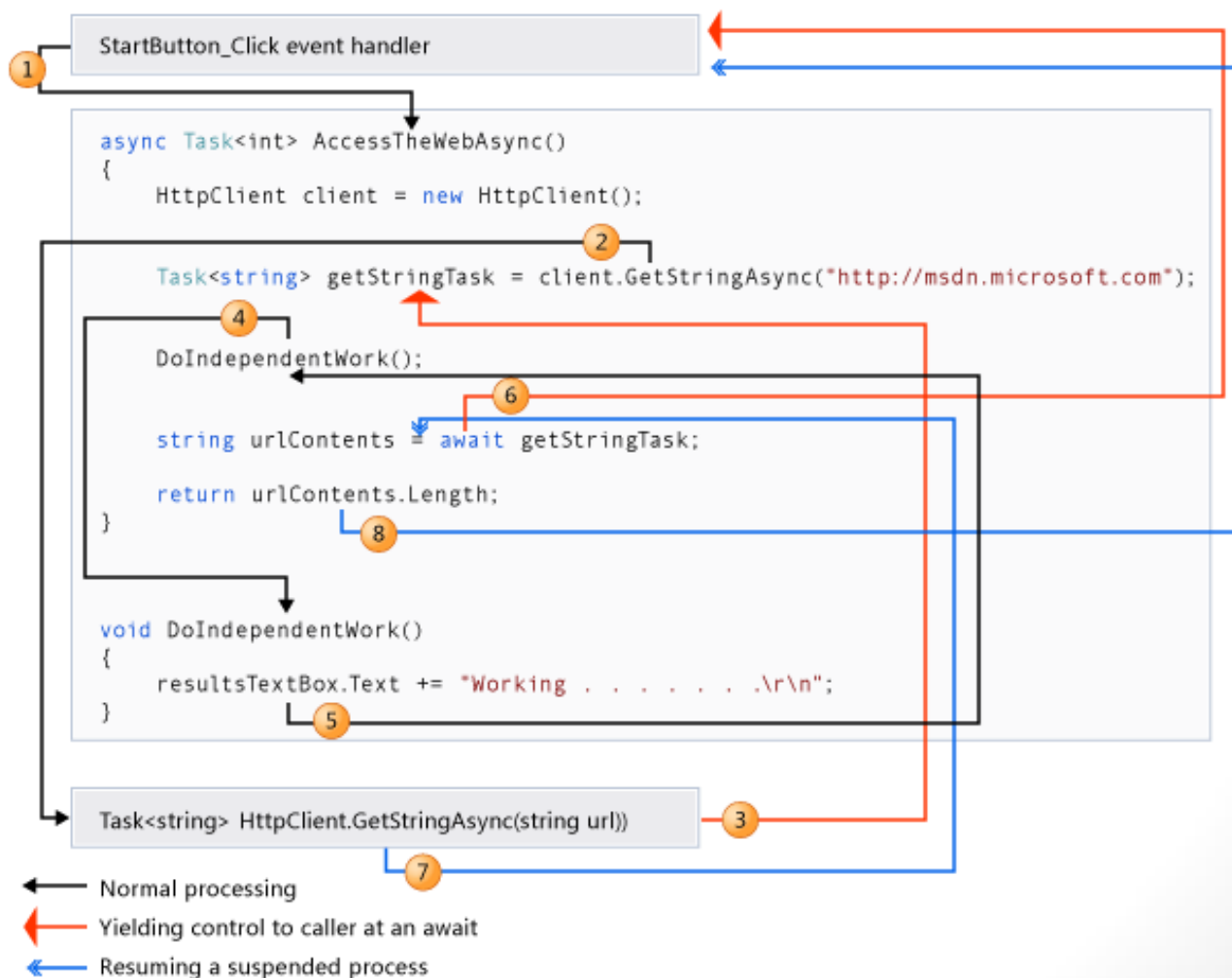
Jak to działa?

1. Procedura obsługi zdarzeń związana z kliknięciem przycisku wywołuje asynchronicznie i czeka na **AccessTheWebAsync**.



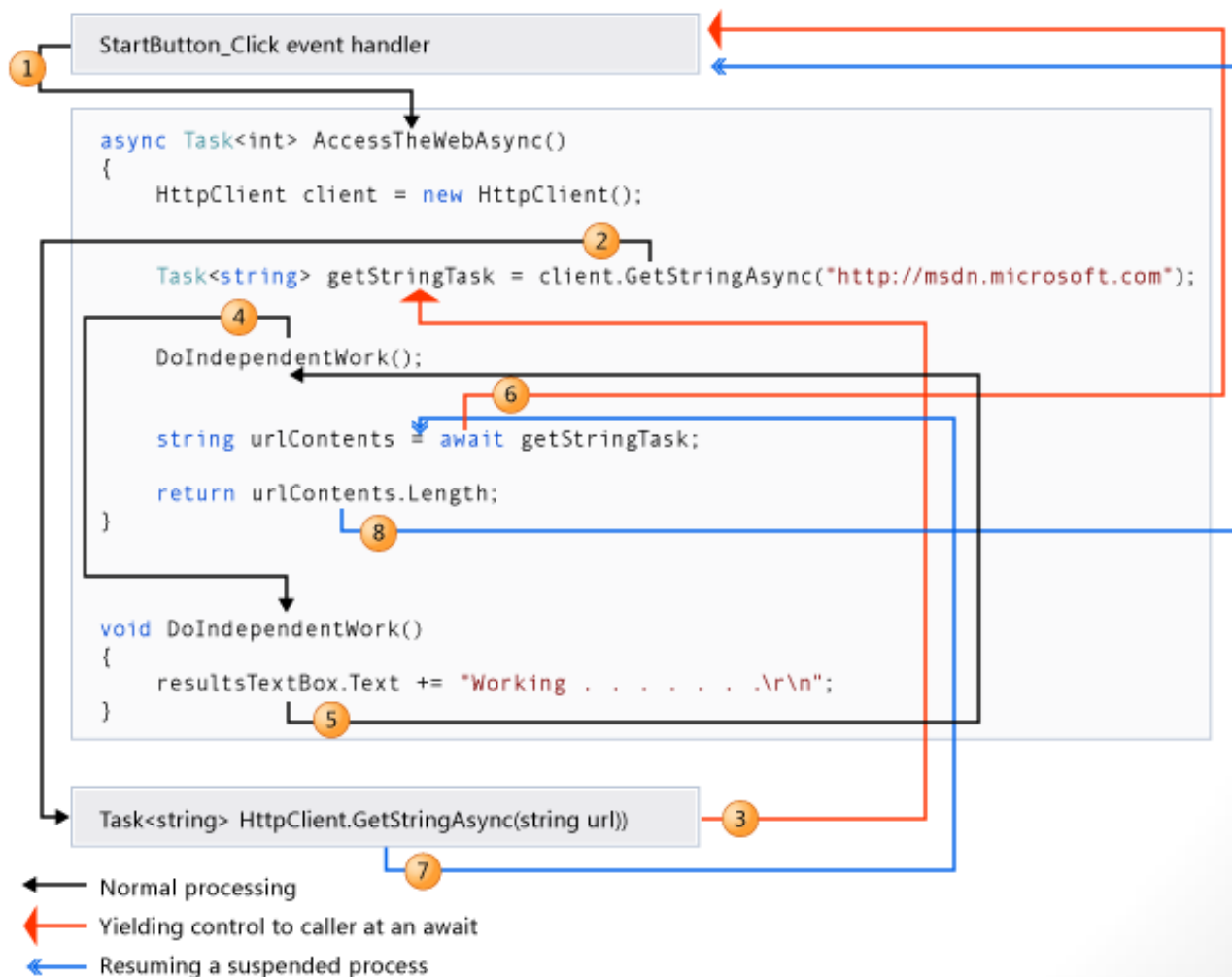
Jak to działa?

2. **AccessTheWebAsync** tworzy obiekt [HttpClient](#) i wywołuje asynchronicznie jego metodę [GetStringAsync](#) w celu pobrania zawartości witryny sieci Web jako zmienną **String**.



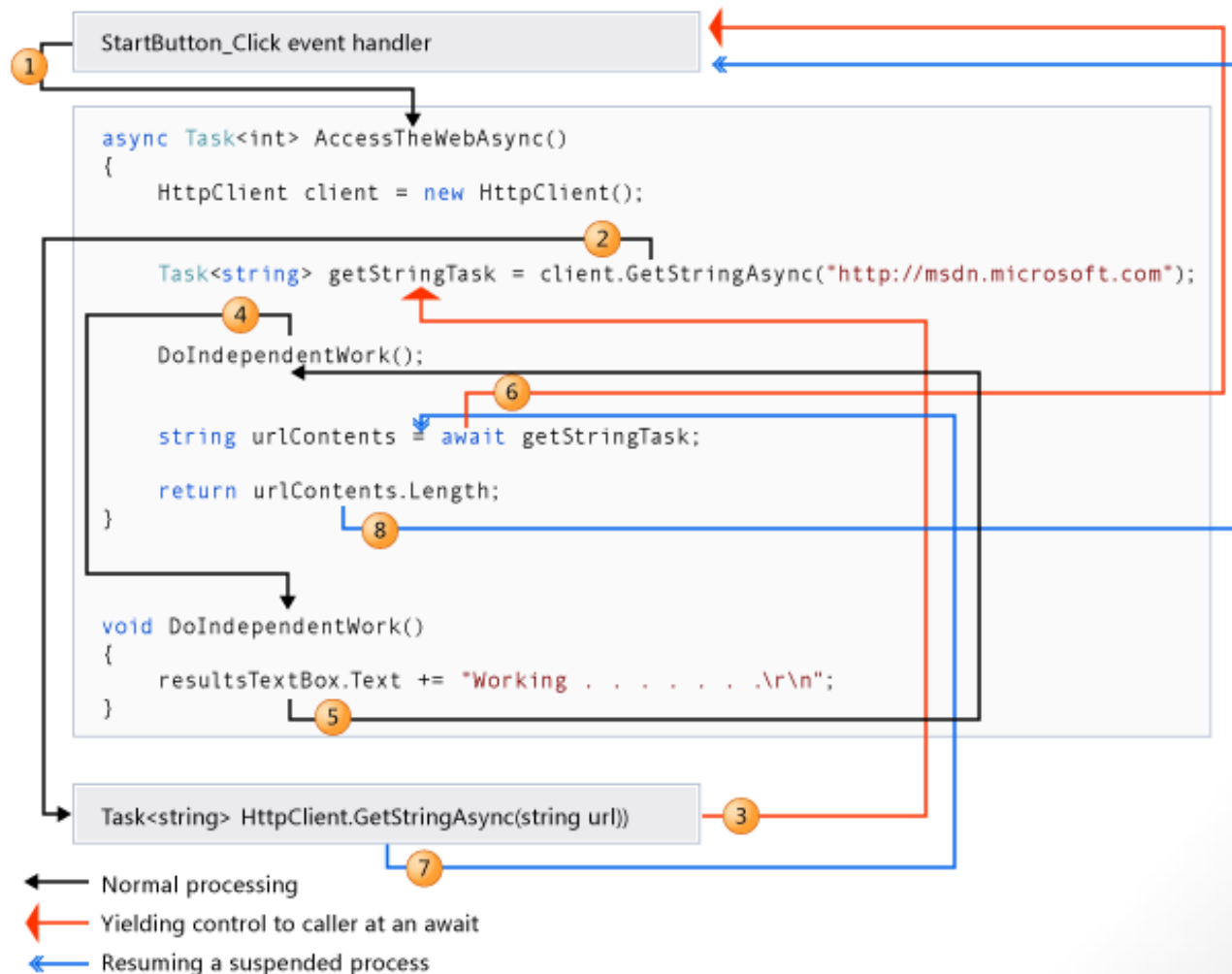
Jak to działa?

3. Załóżmy, że coś się dzieje w metodzie **GetStringAsync**, coś co wstrzymuje jej postęp. Być może musi czekać na pobranie lub jest to jakieś inne działanie blokujące witryny sieci Web. Aby uniknąć blokowania aplikacji, **GetStringAsync** przekazuje z powrotem sterowanie do obiektu wywołującego **AccessTheWebAsync**.



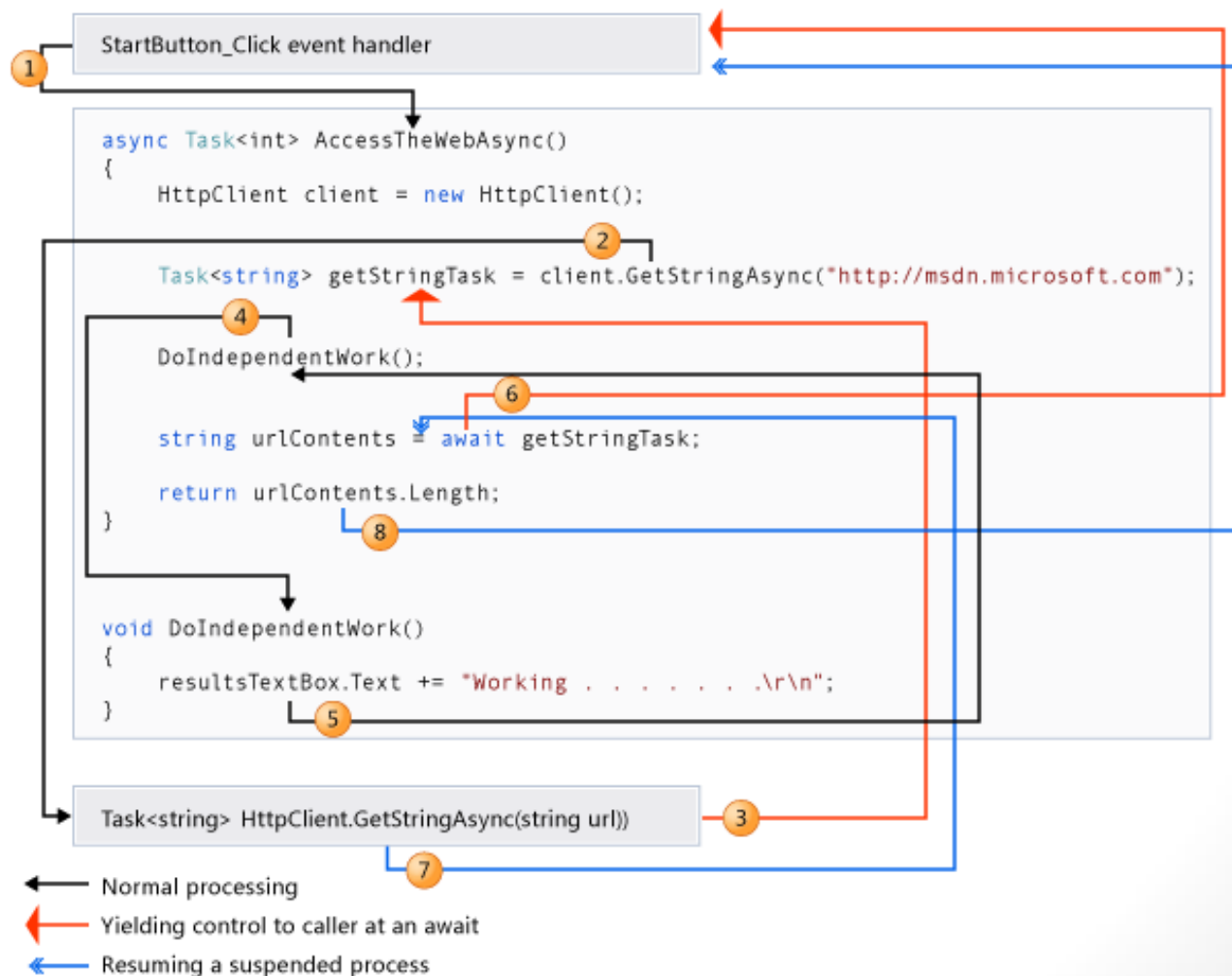
Jak to działa?

3. **GetStringAsync** zwraca obiekt **Task<TResult>**, gdzie **TResult** jest typu **String**, a z kolei **AccessTheWebAsync** przypisuje zadanie do zmiennej **getStringTask**. Obiekt **Task** przedstawia ciągły proces wywołania **GetStringAsync**, ze zobowiązaniem utworzenia faktycznej wartości ciągu po ukończeniu pracy.



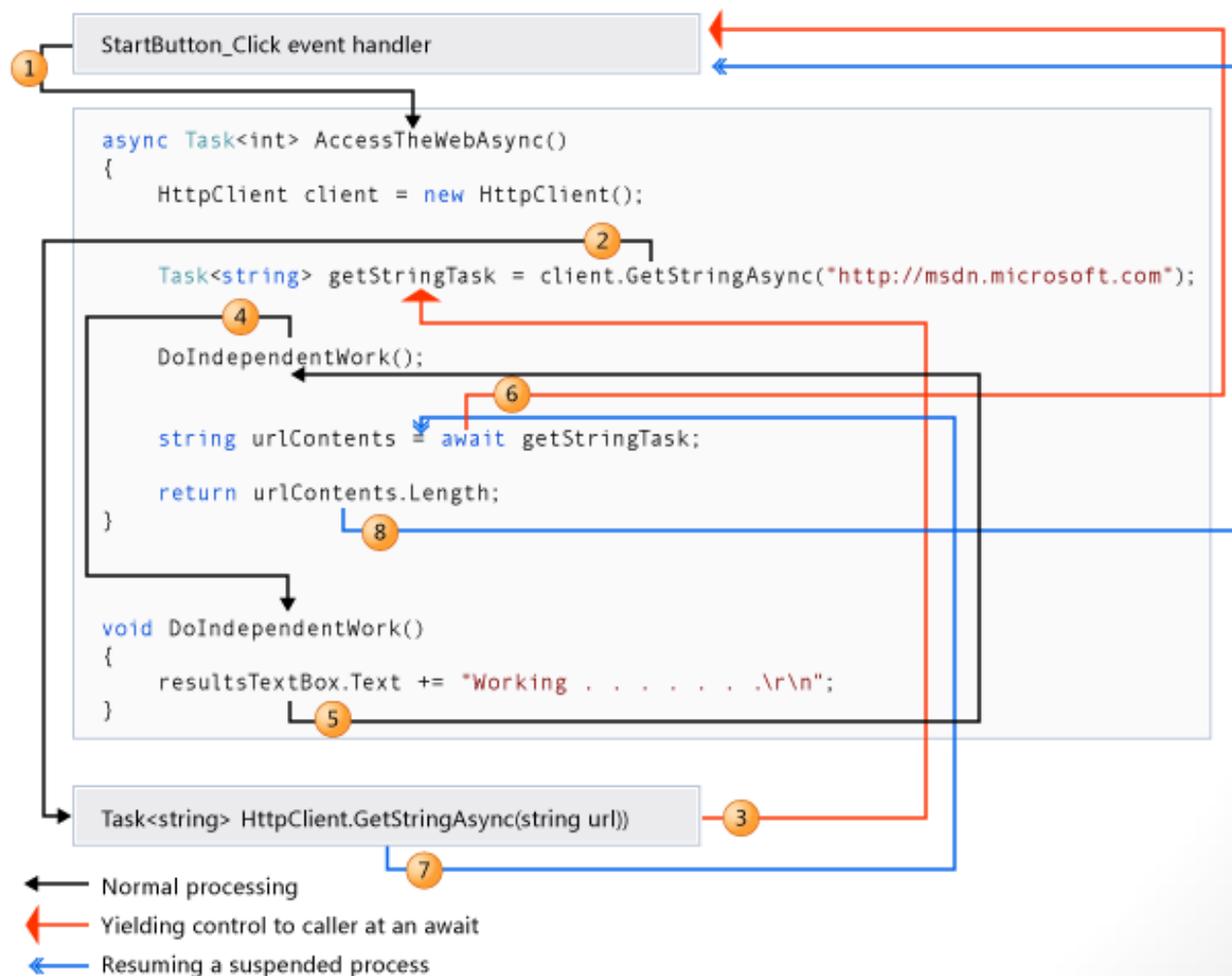
Jak to działa?

4. Ponieważ **getStringTask** nie zakończyło się jeszcze, **AccessTheWebAsync** może kontynuować wykonywanie innych zadań, które są niezależne od wyniku końcowego **GetStringAsync**. Ta praca jest reprezentowana przez wywołanie metody synchronicznej **DoIndependentWork**.



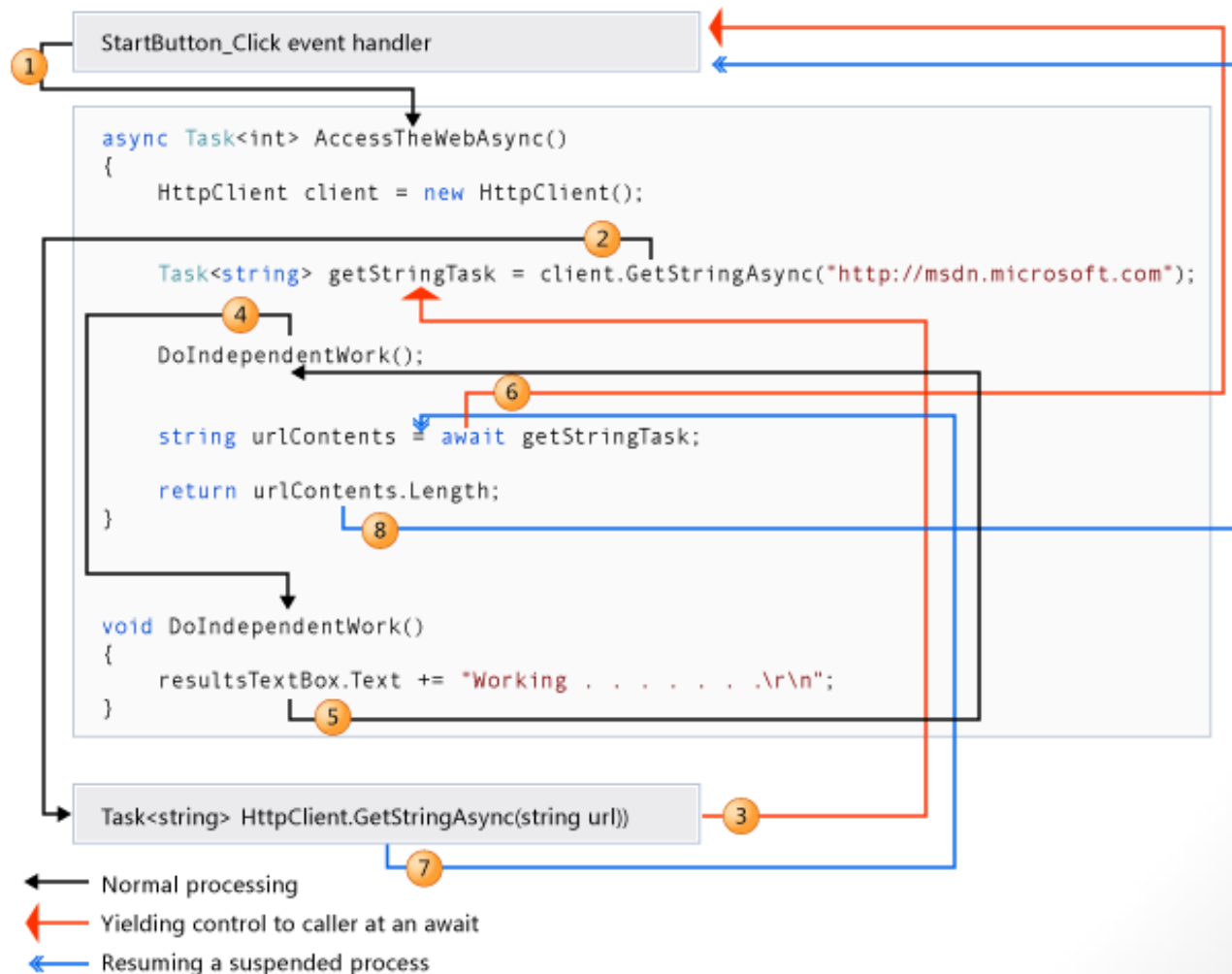
Jak to działa?

5. **DoIndependentWork** jest metodą synchroniczną, która działa i powraca.



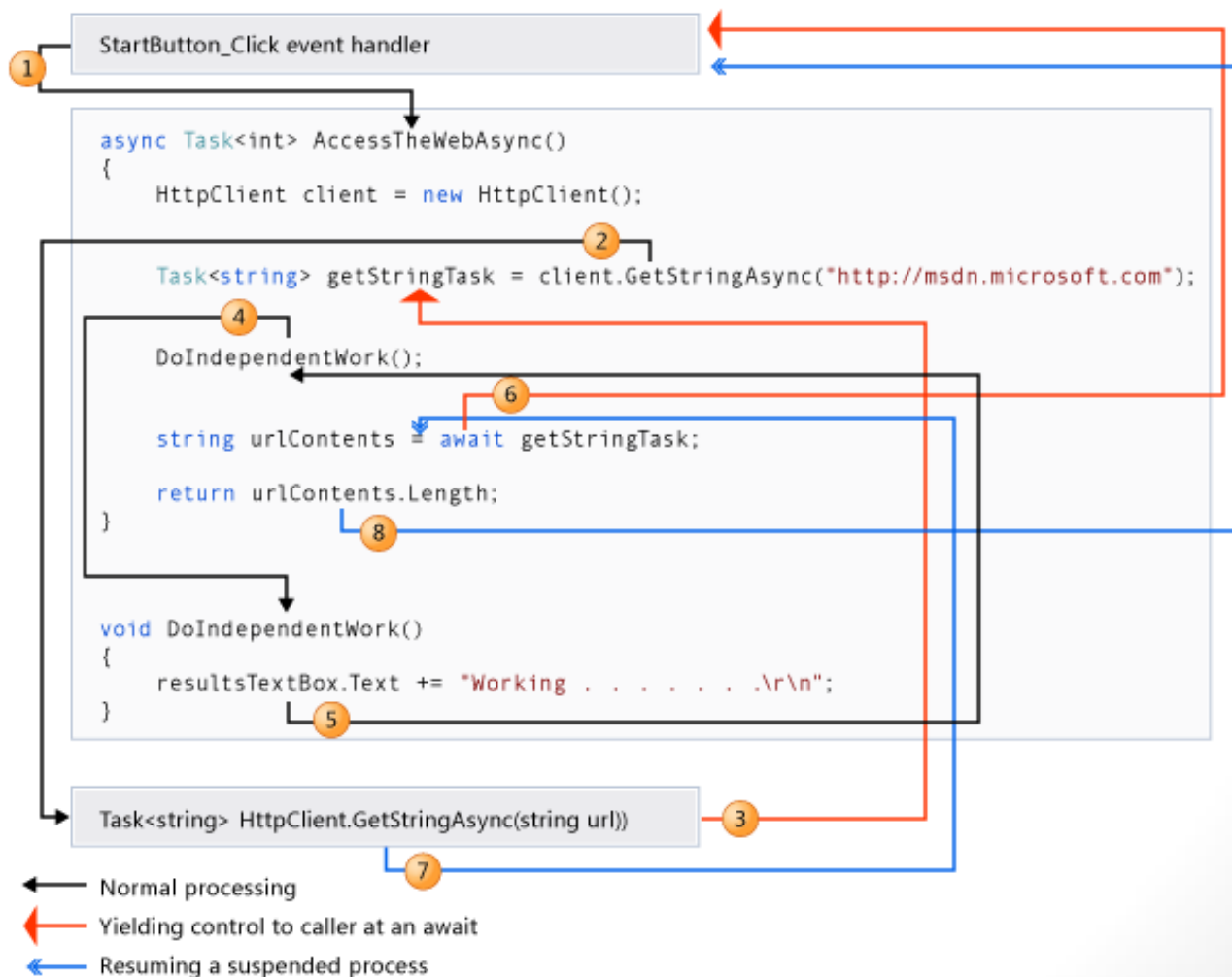
Jak to działa?

6. Załóżmy, że metoda **AccessTheWebAsync** nie ma nic więcej do roboty i wciąż musi czekać na wynik **getStringTask**, który jest jej potrzebny, aby obliczyć i zwrócić długość pobranej strony.



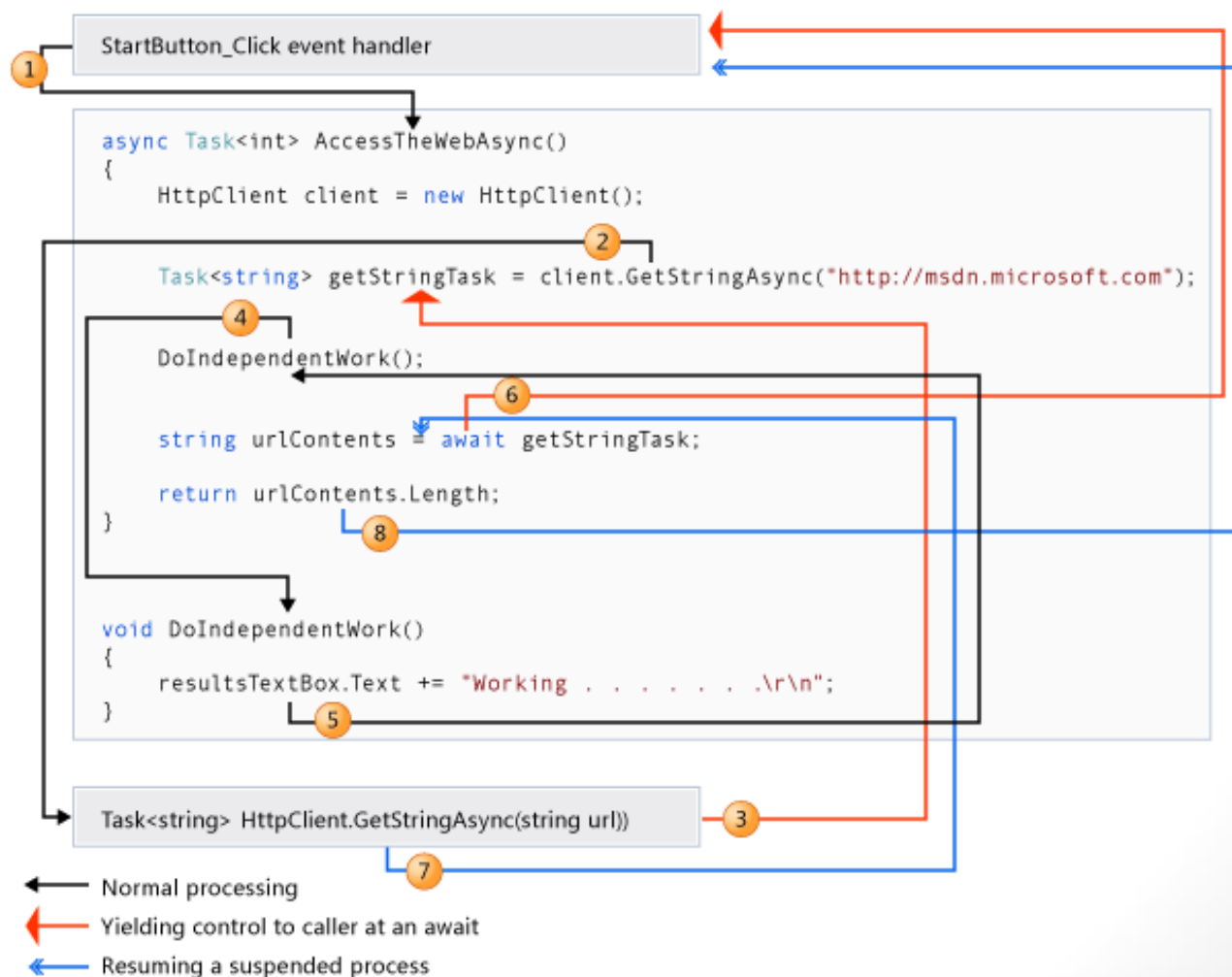
Jak to działa?

6. W związku z tym **AccessTheWebAsync** używając operatora **await** zawiesza swoje działanie i przekazuje sterowanie do metody, która ją wywołała. Zwraca **Task<int>** do obiektu wywołującego. Obiekt jest obietnicą utworzenia w wyniku liczby całkowitej.



Jak to działa?

6. W metodzie obsługującej zdarzenie wciśnięcia przycisku działanie jest kontynuowane. Obiekt wywołujący może wykonywać inne czynności, które nie są zależne od wyników **AccessTheWebAsync**.

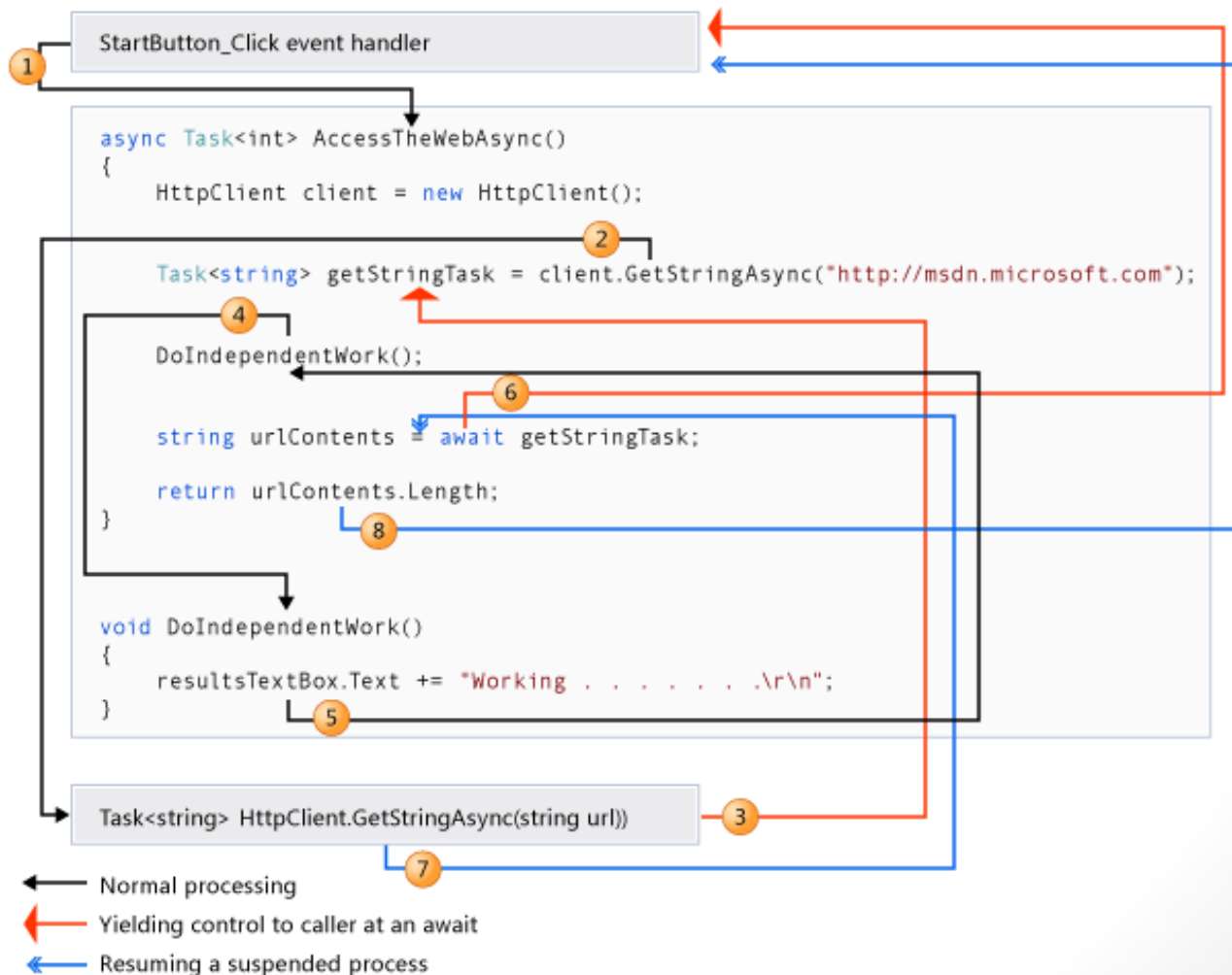


Jak to działa?

7. **GetStringAsync** kończy i zwraca wynik w postaci obiektu **Task**.

Zwrócony obiekt typu **String** nie jest zwracany przez wywołanie **GetStringAsync** w taki sposób, który można by oczekiwać.

(Należy pamiętać, że metoda zwróciła już obiekt **Task** w kroku 3).

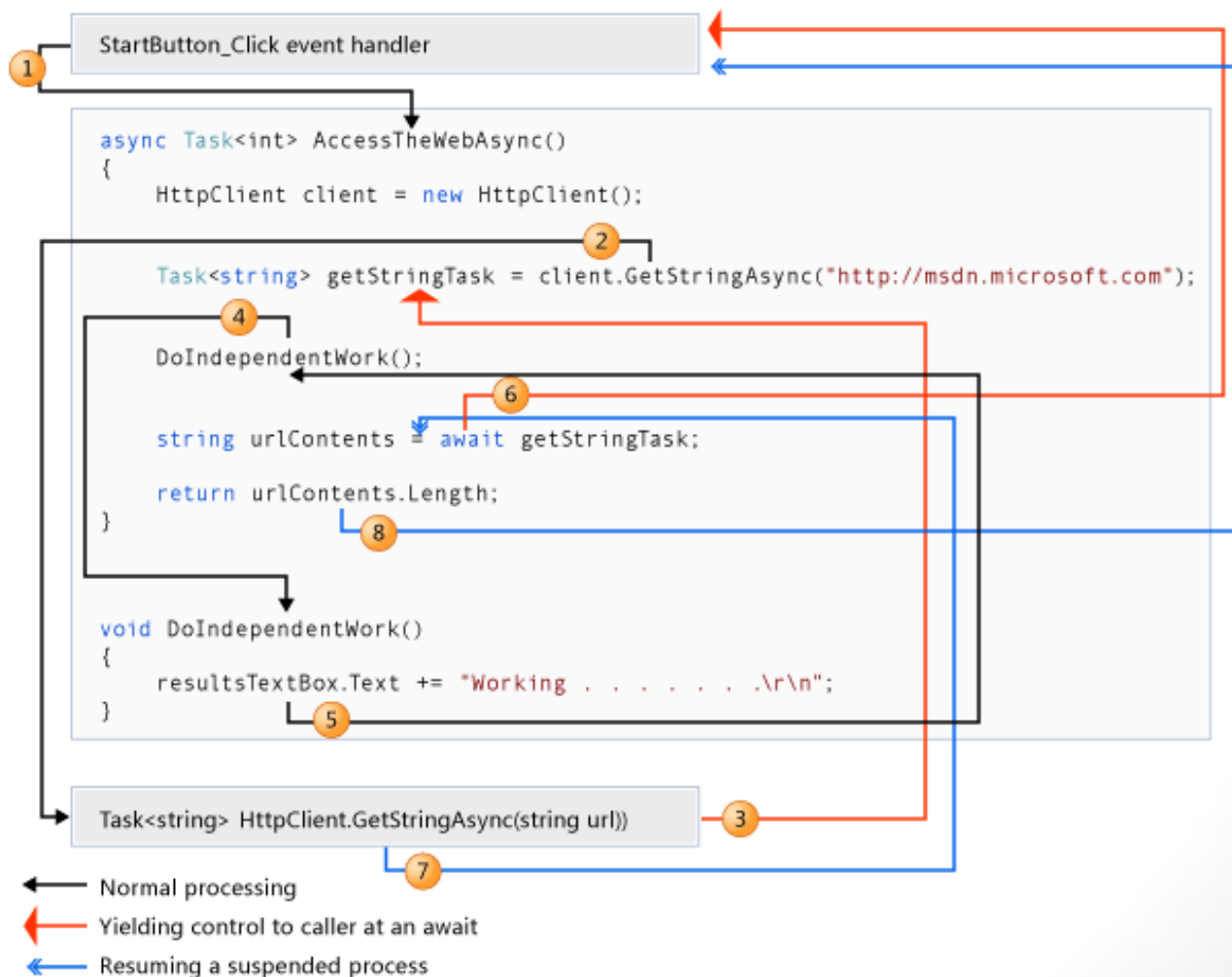


Jak to działa?

7. Pobrana strona jest zapisana w obiekcie **Task**, który reprezentuje ukończenie metody **GetStringTask**.

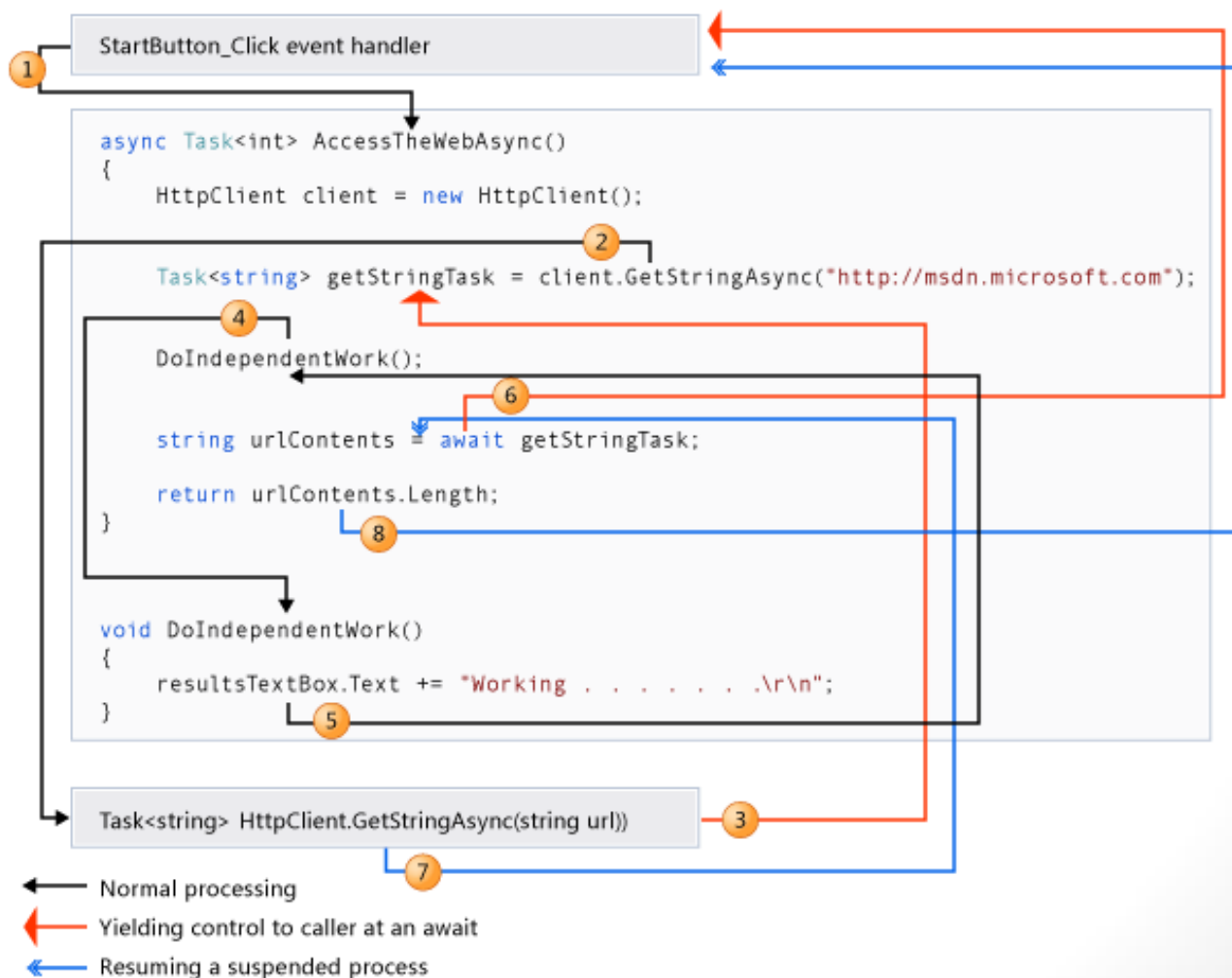
Operator **await** pobiera wyniki z obiektu **getStringTask**.

Instrukcja przypisania przypisuje pobrany wynik w **urlContents**.



Jak to działa?

8. Gdy metoda **AccessTheWebAsync** ma pobraną stronę, może obliczyć długość ciągu. Następnie jej praca jest również zakończona, a czekający program obsługi zdarzenia może wznowić działanie.



async i await - podsumownie

Metoda oznaczona jako **async** może:

- użyć **await** do wskazania punktów zawieszenia;

operator **await** poinformuje kompilator, że metoda **async** nie może kontynuować działań do czasu ukończenia procesu asynchronicznego;

w międzyczasie sterowanie powraca do obiektu wywołującego metodę **async**;

zawieszenie metody **async** na wyrażeniu **await** nie stanowi wyjścia z metody, dlatego też nie zostaną uruchomione bloki **finally**;

- być oczekiwana przez metody, które ją wywołują.

- Metoda **async** zazwyczaj zawiera jeden lub więcej wystąpień operatora **await**.
- Brak **await** nie powoduje błędu kompilatora.
- Jeśli metoda **async** nie używa **await**, aby oznaczyć punkt zawieszenia, pomimo **async** metoda jest wykonywana jak metoda synchroniczna.
- Kompilator generuje wówczas tylko ostrzeżenia dla takich metod.

Typy zwracane

Zwykle są to **Task** lub **Task<TResult>**.

Wewnątrz metody **async** operator **await** jest użyty do obiektu **Task**, który jest zwracany w wyniku wywołania innej metody **async**.

Jeżeli jest potrzeba zwrócenia wyniku wyrażeniem **return**, wówczas należy zastosować typ **Task<TResult>** i zastąpić **TResult** odpowiednim typem zwracanym.

```
// Signature specifies Task
async Task Task_MethodAsync()
{
    // . . .
    // The method has no return statement.
}

// Calls to Task_MethodAsync
Task returnedTask = Task_MethodAsync();
await returnedTask;
// or, in a single statement
await Task_MethodAsync();
```

```
// Signature specifies Task<TResult>
async Task<int> TaskOfTResult_MethodAsync()
{
    int hours;
    // . . .
    // Return statement specifies an integer result.
    return hours;
}

// Calls to TaskOfTResult_MethodAsync
Task<int> returnedTaskTResult = TaskOfTResult_MethodAsync();
int intResult = await returnedTaskTResult;
// or, in a single statement
int intResult = await TaskOfTResult_MethodAsync();
```

Możliwe jest również swobodne użycie właściwości Status, aby określić, czy **Task** wystartował, czy zakończył się, został anulowany lub też wyrzucił wyjątek.

Status jest reprezentowany przez typ wyliczeniowy TaskStatus.

Typy zwracane

Zwykle są to **Task** lub **Task<TResult>**.

Wewnątrz metody **async** operator **await** jest użyty do obiektu **Task**, który jest zwracany w wyniku wywołania innej metody **async**.

Jeżeli jest potrzeba zwrócenia wyniku wyrażeniem **return**, wówczas należy zastosować typ **Task<TResult>** i zastąpić **TResult** odpowiednim typem zwracanym.

- Każdy zwracany **Task** reprezentuje pracę w toku.
Task zawiera informacje o stanie procesu asynchronicznego i/lub ostatecznego wyniku procesu lub wyjątku, który zgłasza proces, jeśli się nie powiedzie.
- Metoda **async** może także zawierać **void** jako typ zwracany.
Jest to zabieg używany głównie do definiowania programów obsługi zdarzeń (**void** jest wymagany).
- Metoda **async** zwracająca **void** nie może być oczekiwana, a obiekt wywołujący taką metodę nie może przechwycić wyjątków przez nią generowanych.
- Metoda **async** nie można zadeklarować parametrów **ref** i **out**, ale może wywoływać metody mające takie parametry.

Metody asynchroniczne w Windows Runtime mają inny zestaw typów zwracanych:

- **IAsyncOperation<TResult>**, które odpowiada **Task<TResult>**
- **IAsyncAction**, które odpowiada **Task**
- **IAsyncActionWithProgress<TProgress>**
- **IAsyncOperationWithProgress<TResult, TProgress>**

Tworzenie i uruchamianie zadań

- Jednym ze sposobów niejawnego tworzenia i uruchamiania nowych zadań jest wywołanie metody ***Parallel.Invoke*** np.

```
Parallel.Invoke(() => DoSomeWork(), () => DoSomeOtherWork());
```

- W przykładzie wystartują dwa zadania równolegle,
- Innym ciekawym przykładem pokazującym rozszerzoną liczbę równoległych zadań jest:

[https://msdn.microsoft.com/en-us/library/dd460705\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd460705(v=vs.110).aspx)

Tworzenie i uruchamianie zadań

- Innym sposobem jest z kolei jawne podanie delegacji, która zahermetyzuje kod wykonywany w ramach zadania **Task** np.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        Thread.CurrentThread.Name = "Main";

        // Create a task and supply a user delegate by using a lambda expression.
        Task taskA = new Task( () => Console.WriteLine("Hello from taskA."));
        // Start the task.
        taskA.Start();

        // Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name);

        taskA.Wait();
    }
}

// The example displays output like the following:
//      Hello from thread 'Main'.
//      Hello from taskA.
```

Tworzenie i uruchamianie zadań

- Lub też z natychmiastowym uruchomieniem poprzez wywołanie metody **Task.Run**

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        Thread.CurrentThread.Name = "Main";

        // Define and run the task.
        Task taskA = Task.Run( () => Console.WriteLine("Hello from taskA."));

        // Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name);

        taskA.Wait();
    }
}

// The example displays output like the following:
//      Hello from thread 'Main'.
//      Hello from taskA.
```

Tworzenie i uruchamianie zadań

- Obiekty Task i Task<TResult> posiadają statyczną właściwość Factory, która zwraca instancję do obiektu TaskFactory. Możliwe jest zatem użycie **Task.Factory.StartNew()**.

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        Thread.CurrentThread.Name = "Main";

        // Better: Create and start the task in one operation.
        Task taskA = Task.Factory.StartNew(() => Console.WriteLine("Hello from taskA.));

        // Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name);

        taskA.Wait();
    }
}

// The example displays output like the following:
//      Hello from thread 'Main'.
//      Hello from taskA.
```

Tworzenie i uruchamianie zadań

- Aby poczekać na kompletne wykonanie wszystkich **Task**ów, można też wywołać metodę **Task.WaitAll** podając jako parametr posiadaną tablicę.
- Zgodnie z poprzednim przykładem będzie to następujące wywołanie

```
Task.WaitAll(taskArray);
```

- Więcej o obiekcie Task, o zarządzaniu ID oraz o dostępnych opcjach można znaleźć w:

[https://msdn.microsoft.com/en-us/library/dd537609\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd537609(v=vs.110).aspx)

Kolejkowanie zadań

- Często praktyką programowania asynchronicznego jest kolejkowanie kilku zadań tak, aby zakończenie jednego wykonywało zadanie następne,
- Proces ten nazywamy kontynuowaniem zadań (ang. *continuation tasks*),
- W ramach tego możliwe jest:
 - Przekazywanie danych z zadania poprzedniego do następnego,
 - Precyzyjne określenie warunków, względem których kontynuowanie nastąpi bądź nie,
 - Anulowanie kontynuacji w dowolnym momencie,
 - Wywołanie wielu kontynuacji równoległe z jednego zadania,
 - Wywołanie jednej kontynuacji, gdy zakończone zostaną wszystkie lub dowolne z wielu zadań poprzedzających,
 - Użycie kontynuacji do przechwycenia wyjątku wyrzuconego przez poprzednika.

Kontynuacja po jednym poprzedniku

```
using System;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        // Execute the antecedent.
        Task<DayOfWeek> taskA = Task.Run( () => DateTime.Today.DayOfWeek );

        // Execute the continuation when the antecedent finishes.
        Task continuation = taskA.ContinueWith( antecedent => Console.WriteLine("Today is {0}.", antecedent.Result) )
    }
}

// The example displays output like the following output:
//      Today is Monday.
```

Kontynuacja po wielu poprzednikach

- Metoda `Task.WhenAll(IEnumerable<Task>)` tworzy zadanie „kontynuację”, które odzwierciedla wynik zakończenia swoich dziesięciu poprzedników.

```
using System.Collections.Generic;
using System;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        List<Task<int>> tasks = new List<Task<int>>();
        for (int ctr = 1; ctr <= 10; ctr++) {
            int baseValue = ctr;
            tasks.Add(Task.Factory.StartNew( (b) => { int i = (int) b;
                                                    return i * i; }, baseValue));
        }
        var continuation = Task.WhenAll(tasks);

        long sum = 0;
        for (int ctr = 0; ctr <= continuation.Result.Length - 1; ctr++) {
            Console.Write("{0} {1} ", continuation.Result[ctr],
                          ctr == continuation.Result.Length - 1 ? "=" : "+");
            sum += continuation.Result[ctr];
        }
        Console.WriteLine(sum);
    }
}

// The example displays the following output:
//      1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100 = 385
```

Opcje kontynuacji

- Metoda **Task.ContinueWith** posiada przeciążenie, dzięki któremu dodatkowo można podać jako parametr obiekt wyliczeniowy typu System.Threading.Tasks.TaskContinuationOptions, co pozwala określić dodatkowe warunki wystartowania kontynuacji np.:
 - Kontynuacja wystartuje tylko wtedy, gdy poprzednik zakończył się sukcesem,
 - Lub też tylko wtedy, gdy poprzednik zakończy niepowodzeniem,
- Jeśli warunek nie jest spełniony, gdy poprzednik jest gotowy do wezwania kontynuacji, wówczas natychmiast zadanie kontynuacji ustawiane jest w stan TaskStatus.Canceled i nie może zostać dalej wystartowane.
- Podobne przeciążenia ma metoda TaskFactory.ContinueWhenAll używana w przypadku kontynuowania po wielu poprzednikach.

Przekazywanie danych do kontynuacji

- Metoda [Task.ContinueWith](#) przekazuje jako argument do delegacji kontynuacji referencję **antecedent** do zadania poprzednika,
- Jeśli poprzednik zakończy się błędem lub zostanie przerwany, próba dostępu do właściwości [Result](#) spowoduje wyrzucenie [AggregateException](#).
- Można temu zapobiec używając opcji [OnlyOnRanToCompletion](#)

```
using System;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        var t = Task.Run( () => { DateTime dat = DateTime.Now;
                                if (dat == DateTime.MinValue)
                                    throw new ArgumentException("The clock is not working.");

                                if (dat.Hour > 17)
                                    return "evening";
                                else if (dat.Hour > 12)
                                    return "afternoon";
                                else
                                    return "morning"; });
        var c = t.ContinueWith((antecedent) => { Console.WriteLine("Good {0}!",
                                                                    antecedent.Result);
                                                Console.WriteLine("And how are you this fine {0}?",
                                                                    antecedent.Result); },
                             TaskContinuationOptions.OnlyOnRanToCompletion);
    }
}

// The example displays output like the following:
//      Good afternoon!
//      And how are you this fine afternoon?
```

Przekazywanie danych do kontynuacji

- Można też sprawdzić **Task.Status** i w zależności od jego wartości zdecydować o dalszym działaniu w kontynuacji.

```
var c = t.ContinueWith( (antecedent) => { if (t.Status == TaskStatus.RanToCompletion) {  
    Console.WriteLine("Good {0}!",  
        antecedent.Result);  
    Console.WriteLine("And how are you this fine {0}?",  
        antecedent.Result);  
}  
else if (t.Status == TaskStatus.Faulted) {  
    Console.WriteLine(t.Exception.GetBaseException().Message);  
}} );
```

Anulowanie kontynuacji

```
Random rnd = new Random();
var cts = new CancellationTokenSource();
CancellationToken token = cts.Token;
Timer timer = new Timer(Elapsed, cts, 5000, Timeout.Infinite);

var t = Task.Run( () => { List<int> product33 = new List<int>();
    for (int ctr = 1; ctr < Int16.MaxValue; ctr++) {
        if (token.IsCancellationRequested) {
            Console.WriteLine("\nCancellation requested in antecedent...\n");
            token.ThrowIfCancellationRequested();
        }
        if (ctr % 2000 == 0) {
            int delay = rnd.Next(16,501);
            Thread.Sleep(delay);
        }

        if (ctr % 33 == 0)
            product33.Add(ctr);
    }
    return product33.ToArray();
}, token);
```

```
Task continuation = t.ContinueWith(antecedent => { Console.WriteLine("Multiples of 33:\n");
```

```
try {
    continuation.Wait();
}
catch (AggregateException e) {
    foreach (Exception ie in e.InnerExceptions)
        Console.WriteLine("{0}: {1}", ie.GetType().Name,
            ie.Message);
}
```

Wyłapywanie wyjątków z kontynuacji

- Należy użyć metod [Wait](#), [WaitAll](#), lub [WaitAny](#), aby czekać na poprzednika i sterować pracą kontynuacji jednocześnie wyłapując wyjątki w sekcji **try**

```
using System;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        var task1 = Task<int>.Run( () => { Console.WriteLine("Executing task {0}",
                                                                Task.CurrentId);

                                                                return 54; });

        var continuation = task1.ContinueWith( (antecedent) =>
        { Console.WriteLine("Executing continuation task {0}",
                            Task.CurrentId);
          Console.WriteLine("Value from antecedent: {0}",
                            antecedent.Result);
          throw new InvalidOperationException();
        } );

        try {
            task1.Wait();
            continuation.Wait();
        }
        catch (AggregateException ae) {
            foreach (var ex in ae.InnerExceptions)
                Console.WriteLine(ex.Message);
        }
    }
}

// The example displays the following output:
//      Executing task 1
//      Executing continuation task 2
//      Value from antecedent: 54
//      Operation is not valid due to the current state of the object.
```


Wyjątki w wywołaniach asynchronicznych

- Taka implementacja nic nie daje

```
private async void ThrowExceptionAsync()
{
    throw new InvalidOperationException();
}
public void AsyncVoidExceptions_CannotBeCaughtByCatch()
{
    try
    {
        ThrowExceptionAsync();
    }
    catch (Exception)
    {
        // The exception is never caught here!
        throw;
    }
}
```

Implementacja serwera TCP w async (C#)

- Główna metoda programu
- i obsługa konsoli do wykrycia żądania zatrzymania serwera

```
static CancellationTokenSource cancellationToken = new CancellationTokenSource();
0 references
static void Main(string[] args)
{
    Parallel.Invoke(
        () => RunConsoleListener(),
        async () => await StartServer()
    );
}

1 reference
public static void RunConsoleListener()
{
    Console.ReadKey(true);
    Console.WriteLine("Sending cancellation token....");
    cancellationToken.Cancel();
    Console.ReadKey(true);
}
```

Implementacja serwera TCP w async (C#)

```
public static async Task StartServer()
{
    TcpListener listener = new TcpListener(new IPEndPoint(IPAddress.Any, 7));
    listener.Start();

    cancellationToken.Token.Register(listener.Stop);
    while (!cancellationToken.IsCancellationRequested)
    {
        try
        {
            TcpClient client = await listener.AcceptTcpClientAsync();
            var clientTask = HandleClient(client, cancellationToken)
                .ContinueWith((antecedent) => client.Dispose())
                .ContinueWith((antecedent) => Console.WriteLine("Client disposed.));
        }
        catch (ObjectDisposedException) when (cancellationToken.IsCancellationRequested)
        {
            Console.WriteLine("TcpListener stopped listening because cancellation was requested.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error handling client: {ex.Message}");
        }
    }
}
```

Implementacja serwera TCP w async (C#)

```
public static async Task HandleClient(TcpClient tcpClient, CancellationTokenSource cancellationToken)
{
    string clientEndPoint = tcpClient.Client.RemoteEndPoint.ToString();
    Console.WriteLine("Received connection request from " + clientEndPoint);
    try
    {
        cancellationToken.Token.Register(tcpClient.Close);
        //the above, if close client connection together with stopping listener request
        NetworkStream networkStream = tcpClient.GetStream();
        StreamReader reader = new StreamReader(networkStream);
        StreamWriter writer = new StreamWriter(networkStream);
        writer.AutoFlush = true;
        while (!cancellationToken.IsCancellationRequested)
        {
            char[] read = new char[1024];
            Console.WriteLine("Waiting for user request.");
            int bytes = await reader.ReadAsync(read, 0, read.Length);
            string request = new string(read, 0, bytes);
            if (bytes != 0)
            {
                Console.WriteLine("Received service request: " + request);
                Console.WriteLine("Response is: " + request + "\n");
                await writer.WriteAsync(read, 0, bytes);
            }
            else break; // Client closed connection
        }
        tcpClient.Close();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        if (tcpClient.Connected) tcpClient.Close();
    }
}
```

JAVA asynchronicznie (Future i Executor)

- Obiekt Executor wykonuje powierzone mu zadanie.
- Za pomocą wyrażenia lambda tworzy się instancje interfejsu Callable i implementuje jego metodę call(),
- Instancje przekazujemy do metody submit Executor'a,
- Egzekutor uruchamia zadanie w oddzielnym wątku zwracając referencje do obiektu Future reprezentującego proces i oczekiwanie na wynik w nim zaszyty,

```
public class Kalkulator {  
    public Future<Integer> policz(Integer liczba) {  
        return Executors.newSingleThreadExecutor().submit(() -> {  
            Thread.sleep(1000);  
            return liczba + liczba;  
        });  
    }  
}
```

JAVA: Future isDone() i get()

- Aby sprawdzić kompletność zadania, można wywołać na jego obiekcie metodę isDone()

```
Future<Integer> future = new Kalkulator().policz(10);  
  
while(!future.isDone()) {  
    System.out.println("Obliczam...");  
    Thread.sleep(300);  
}  
  
Integer wynik = future.get();
```

- Z kolei metoda get() pozwoli pobrać parametr przekazany przez zadanie,
- Istnieje przeciążenie get() pozwalające określić maksymalny czas oczekiwania na zakończenie zadania

```
Integer wynik = future.get(500, TimeUnit.MILLISECONDS);
```

po upływie którego wyrzucany jest wyjątek typu TimeoutException

JAVA: Anulowanie zadania

```
Future<Integer> future = new Kalkulator().policz(10);
```

```
boolean canceled = future.cancel(true);
```

- Możliwe jest, że anulowanie zadania zwróci **false**.
- Dzieje się tak głównie wtedy, kiedy zadanie zdążyło się samo zakończyć.
- Można też sprawdzić stan anulowania metodą:

```
future.isCancelled();
```

JAVA: mankamenty Future

- Objekt Future reprezentuje wynik, który będzie zwrócony przez trwającą operację,
- Oferuje metodę — `get` —, która zwróci wynik przetwarzania, gdy będzie ono zakończone,
- Problem w tym, że jej wywołanie **jest operacją blokującą** do momentu zakończenia przetwarzania,
- Takie rozwiązanie jest dość restrykcyjne i podważa sens mechanizmu asynchroniczności.

JAVA: CompletableFuture

- Objekt `CompletableFuture` implementuje `Future`, ale tak, że jest możliwe dołożenie pewnej dodatkowej logiki,

```
CompletableFuture<String> completableFuture  
    = CompletableFuture.supplyAsync(() -> "Hello");  
    .thenApply(s -> s + " World");  
  
assertEquals("Hello World", future.get());
```

- Najczęściej stosowane jest budowanie łańcucha, gdzie wynik poprzedniego ogniwa dostarczany jest do następnego, co wykonuje właśnie metoda `thenApply()`,
- Akceptuje ona, jako argument, instancję funkcji, która jako argument otrzymuje to, co zwróciła poprzednia.

JAVA: CompletableFuture

- Jeśli nie ma potrzeby późniejszego zwracania obiektu w dół łańcucha, można zastosować metodę, która pobiera argument ale zwraca `void`,
- Metoda *thenAccept()* obiektu *CompletableFuture* działa właśnie w ten sposób.
- Ostateczne wywołanie *future.get()* zwróci instancje do typu *void*.

```
CompletableFuture<String> completableFuture  
    = CompletableFuture.supplyAsync(() -> "Hello");  
    .thenAccept(s -> System.out.println("Computation returned: " + s));  
  
future.get();
```

JAVA: CompletableFuture

- Podobnie, jeśli zarówno nie ma potrzeby zwracania wartości oraz dodatkowo przyjmowania argumentów można wywołać metodę *thenRun*,
- Wywołanie w tym przypadku *future.get()* wypisze tylko stosowny komunikat w konsoli

```
CompletableFuture<String> completableFuture  
    = CompletableFuture.supplyAsync(() -> "Hello");  
    .thenRun(() -> System.out.println("Computation finished."));  
  
future.get();
```

JAVA: CompletableFuture

- Aby połączyć dwa obiekty *Future* w sekwencyjny łańcuch wywołań można też użyć metody *thenCompose*,
- Ta metoda, jako argument otrzymuje funkcję, która zwraca instancję *CompletableFuture*,
- Argumentem funkcji, jest wynik poprzedniej operacji, który może zostać wykorzystany w następnym wyrażeniu lambda

```
CompletableFuture<String> completableFuture  
    = CompletableFuture.supplyAsync(() -> "Hello")  
        .thenCompose(s -> CompletableFuture.supplyAsync(() -> s + " World"));  
  
assertEquals("Hello World", completableFuture.get());
```

- Metoda *thenCompose* otrzymuje funkcję, która zwraca inny obiekt tego samego typu.
- W ten sposób tworzony jest blok instancji *CompletableFuture*.

JAVA: CompletableFuture

- Z kolei, jeśli istnieje potrzeba uruchomienia dwóch niezależnych obiektów *Future* oraz wykonania pewnych operacji na ich wyniku, to należy użyć metody *thenCombine*,
- Metoda akceptuje *Future* i funkcje z dwoma argumentami i przetwarza je

```
CompletableFuture<String> completableFuture
    = CompletableFuture.supplyAsync(() -> "Hello")
        .thenCombine(CompletableFuture.supplyAsync(
            () -> " World"),
            (s1, s2) -> s1 + s2));

assertEquals("Hello World", completableFuture.get());
```

JAVA: CompletableFuture

- Prostszy przykład na przetwarzanie dwóch wyników obiektów Future,
- Który nie wymaga przerzucania żadnego wyniku w dół łańcucha,
- To metoda *thenAcceptBoth*:

```
CompletableFuture future = CompletableFuture.supplyAsync(() -> "Hello")
    .thenAcceptBoth(CompletableFuture.supplyAsync(
        () -> " World"),
        (s1, s2) -> System.out.println(s1 + s2));
```

JAVA: CompletableFuture

- W przypadku, gdy istnieje potrzeba uruchamiania wielu *Future* równoległe (nie sekwencyjnie), zazwyczaj oczekiwane jest zakończenie ich wszystkich i dopiero potem przetwarzanie spójnego wyniku,
- Statyczna metoda *CompletableFuture.allOf* pozwala oczekiwać na wiele obiektów *Future* jednocześnie, które podane jej są jako argument:

```
CompletableFuture<String> future1
    = CompletableFuture.supplyAsync(() -> "Hello");
CompletableFuture<String> future2
    = CompletableFuture.supplyAsync(() -> "Beautiful");
CompletableFuture<String> future3
    = CompletableFuture.supplyAsync(() -> "World");

CompletableFuture<Void> combinedFuture
    = CompletableFuture.allOf(future1, future2, future3);

// ...

combinedFuture.get();
```

JAVA: CompletableFuture -> wyjątki

- Obiekt *CompletableFuture* za pomocą specjalnej metody *handle* umożliwia przechwytywanie wyjątków,
- Do działania metoda otrzymuje dwa parametry:
 - Wynik przetwarzania (jeśli ukończony, gdy sukces),
 - Wyrzucony wyjątek (jeśli coś poszło nie tak),

```
String name = null;
```

```
// ...
```

```
CompletableFuture<String> completableFuture  
= CompletableFuture.supplyAsync(() -> {  
    if (name == null) {  
        throw new RuntimeException("Computation error!");  
    }  
    return "Hello, " + name;  
})).handle((s, t) -> s != null ? s : "Hello, Stranger!" + t.getMessage());
```


JavaScript: Obietnice

- Obietnice (ang. *Promises*) powstały z myślą o usprawnieniu pracy z kodem asynchronicznym.
- Ich podstawowymi zaletami są: lepsza kontrola w zakresie synchronizacji wywołań, obsługa błędów oraz lepsza czytelność.
- Obiekt **Promise** reprezentuje ewentualne zakończenie (lub porażkę) asynchronicznej operacji i jej wartości

JavaScript: Executor

```
new Promise( /* wykonawca */ function(resolve, reject) { ... } );
```

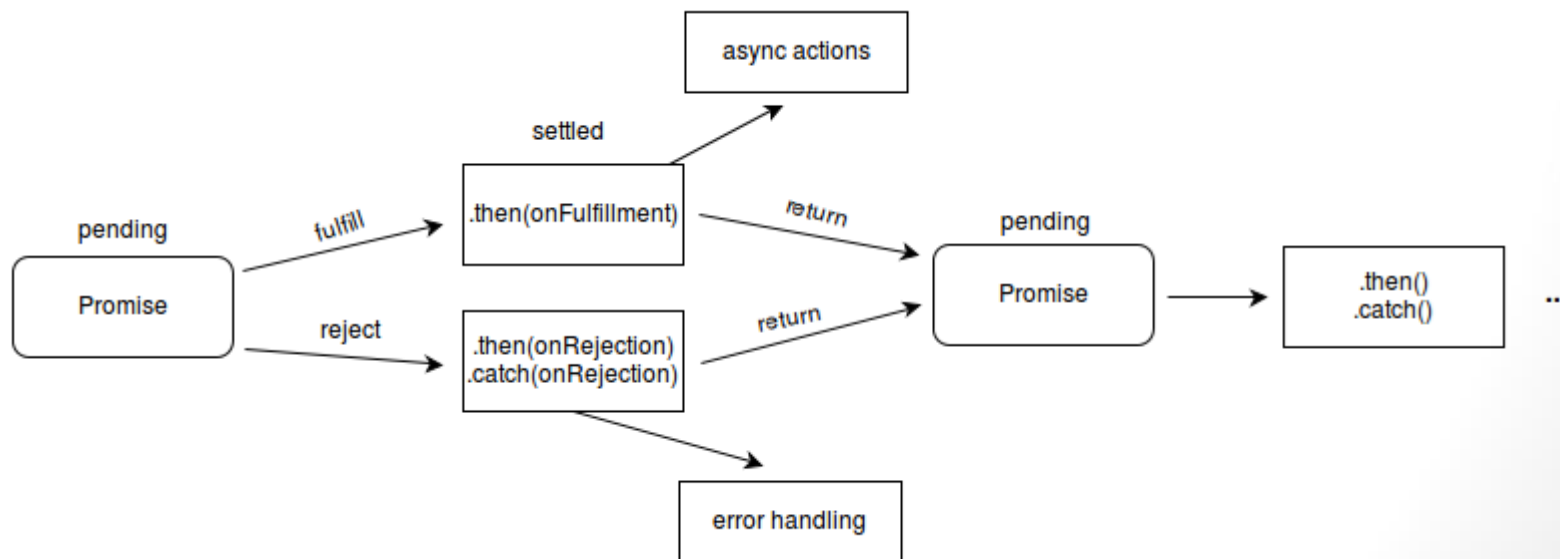
- Funkcja przekazywana z argumentami **resolve** i **reject**.
- Funkcja ta wykonywana jest natychmiast przez implementację obietnicy, przekazując funkcje **resolve** i **reject**
(ta funkcja jest wywoływana, zanim konstruktor Promise zwróci utworzony obiekt).
- Funkcje **resolve** i **reject** , gdy są wywoływane, odpowiednio rozwiązują lub odrzucają obietnicę.
- Ta funkcja zwykle inicjuje pracę asynchroniczną, a następnie, po jej zakończeniu, wywołuje funkcję **resolve** w celu rozwiązania obietnicy lub odrzuca ją, jeśli wystąpił błąd.
- Jeśli błąd zostanie zgłoszony w tej funkcji, obietnica zostaje odrzucona. Wartość zwracana tej funkcji jest ignorowana.

JavaScript: Obietnice

- Obietnica może znajdować się w jednym z trzech stanów:
 - spełniona/rozwiązana (fulfilled/resolved) –
gdy powiązane z nią zadanie zwraca żądaną wartość,
 - odrzucona (rejected) –
gdy zadanie nie zwraca wartości, np. z powodu wyjątku lub zwrócona wartość jest nieprawidłowa,
 - oczekująca (pending) –
to stan od rozpoczęcia żądania do otrzymania wyniku.

JavaScript: Obietnice

- Obowiązująca obietnica może być zakończona z wartością lub odrzucona z powodem (tj. błędem).
- Kiedy dzieje się jedna z tych opcji, wywoływane są zakolejkowane, powiązane za pomocą metody **then** procedury.
(Jeśli obietnica została już spełniona lub odrzucona, gdy dołączony jest odpowiedni moduł obsługi, zostanie on wywołany, więc nie ma potrzeby wyścigu między zakończeniem operacji asynchronicznej a jej obsługą).



JavaScript: Obietnice

```
var myPromise = new Promise((resolve, reject) => {  
  // resolve(...), gdy operacja asynchroniczna zakończyła się sukcesem  
  // reject(...), gdy z błędem.  
  
  // Użycie setTimeout(...) w celu symulacji działania zadania asynchronicznego.  
  setTimeout(function() {  
    resolve("Success!");  
  }, 250);  
  
  // lub  
  reject(new Error("Uh no!"));  
  
  // lub wyjątek  
  throw new Error("Uh no!"));  
});  
  
myPromise.then((successMessage) => {  
  console.log("Huraa! " + successMessage);  
}, (error) => {  
  console.log(error);  
});  
  
//lub z przechwyceniem wyjątku  
myPromise.catch(error => console.log(error));
```

```
p.then(onFulfilled[, onRejected]);  
  
p.then((value) => {  
  // fulfillment  
}, (reason) => {  
  // rejection  
});
```