Programowanie Współbieżne

C#

Materiały

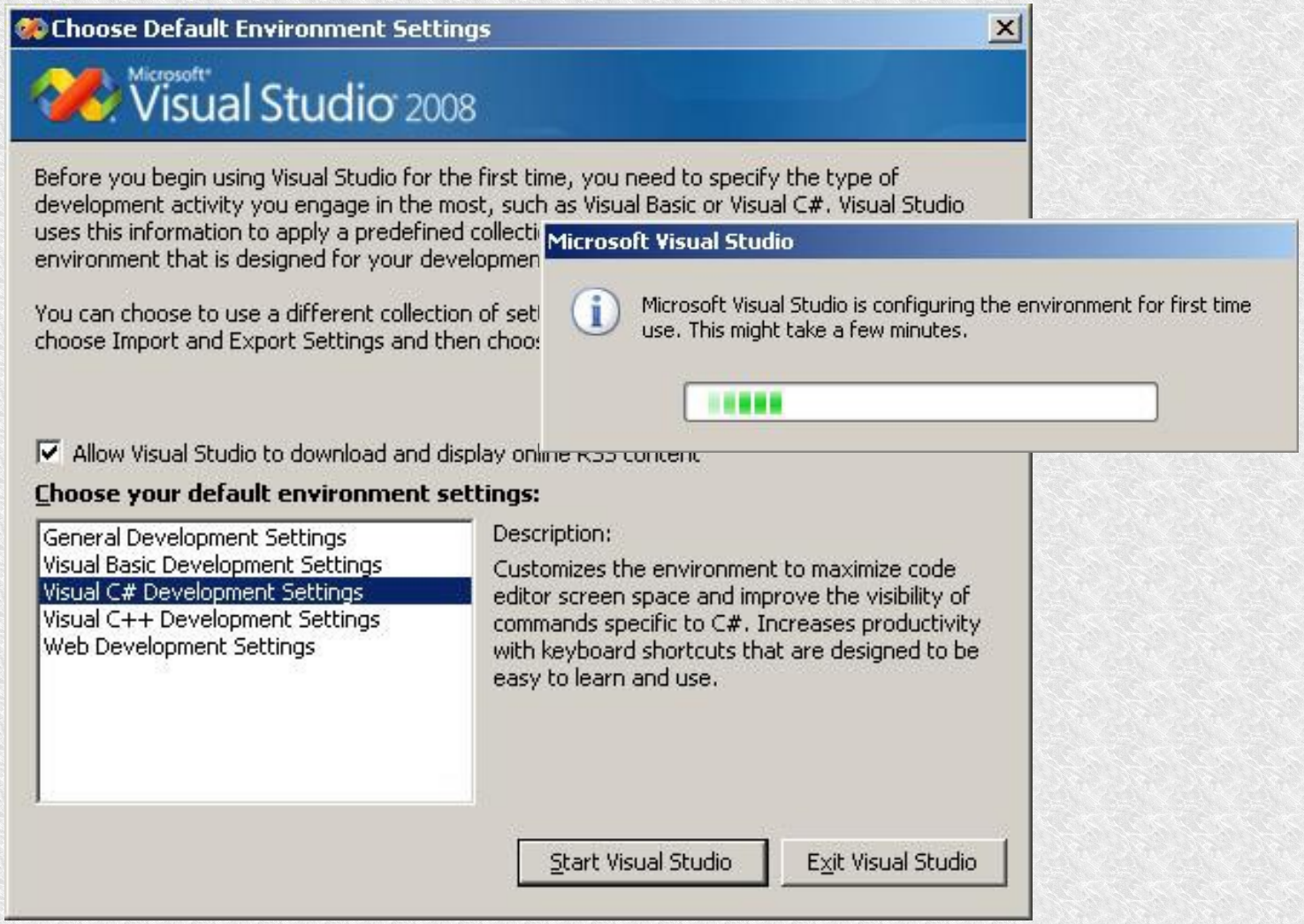
http://www.microsoft.com/express/download/

http://msdn.microsoft.com/en-us/library/aa645740(VS.71).aspx

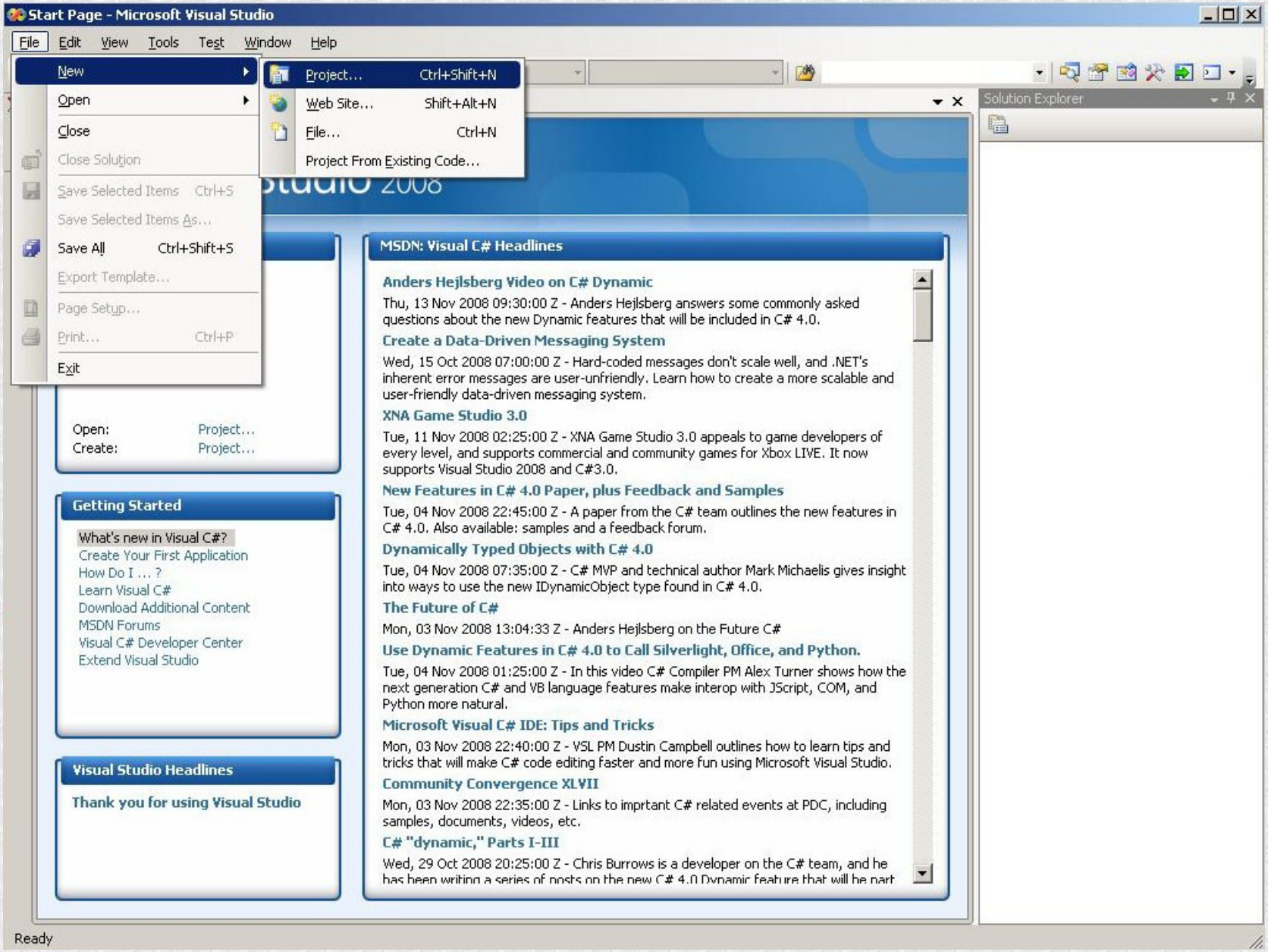
http://www.albahari.com/threading/

http://www.centrumxp.pl/

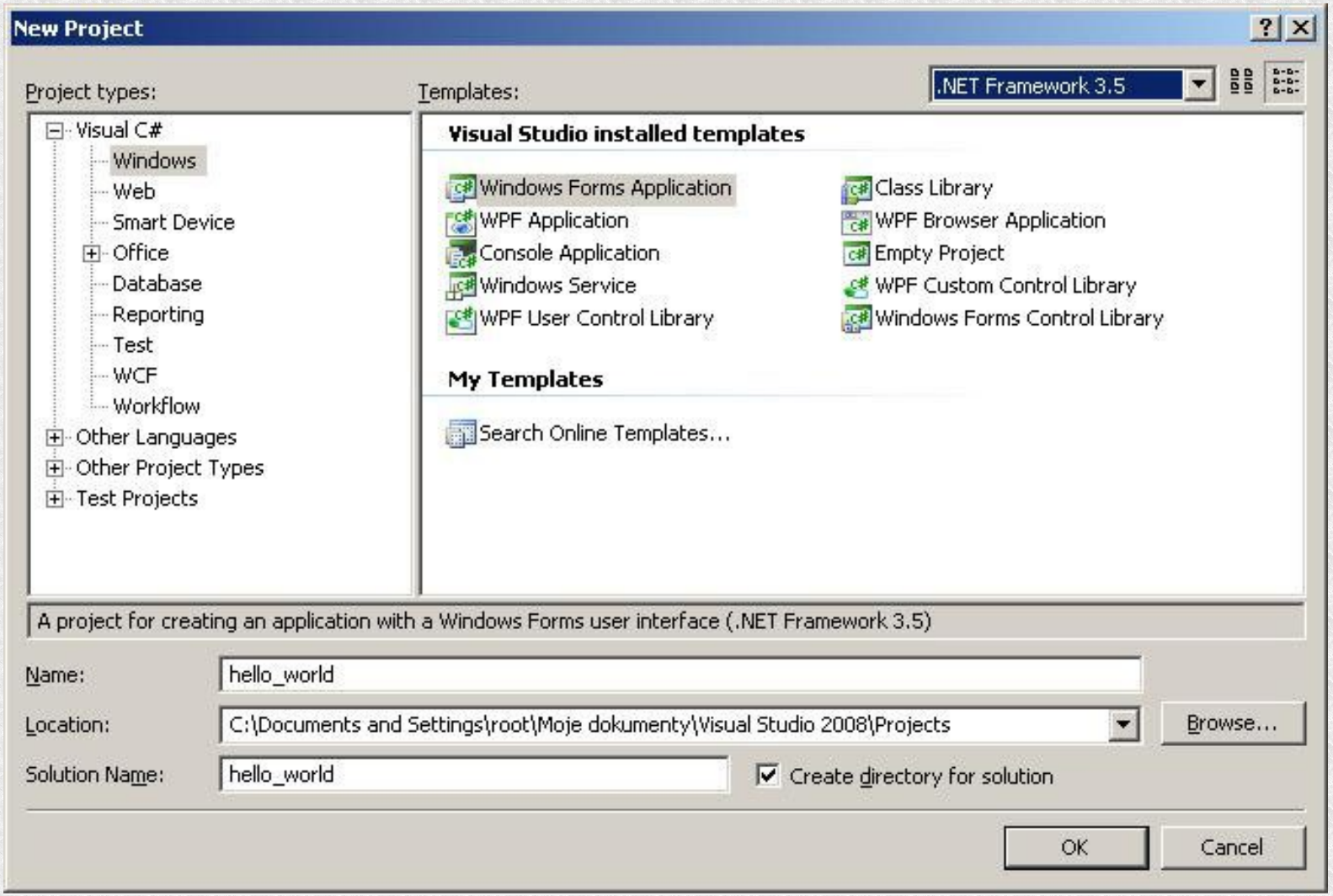
Pierwsze kroki



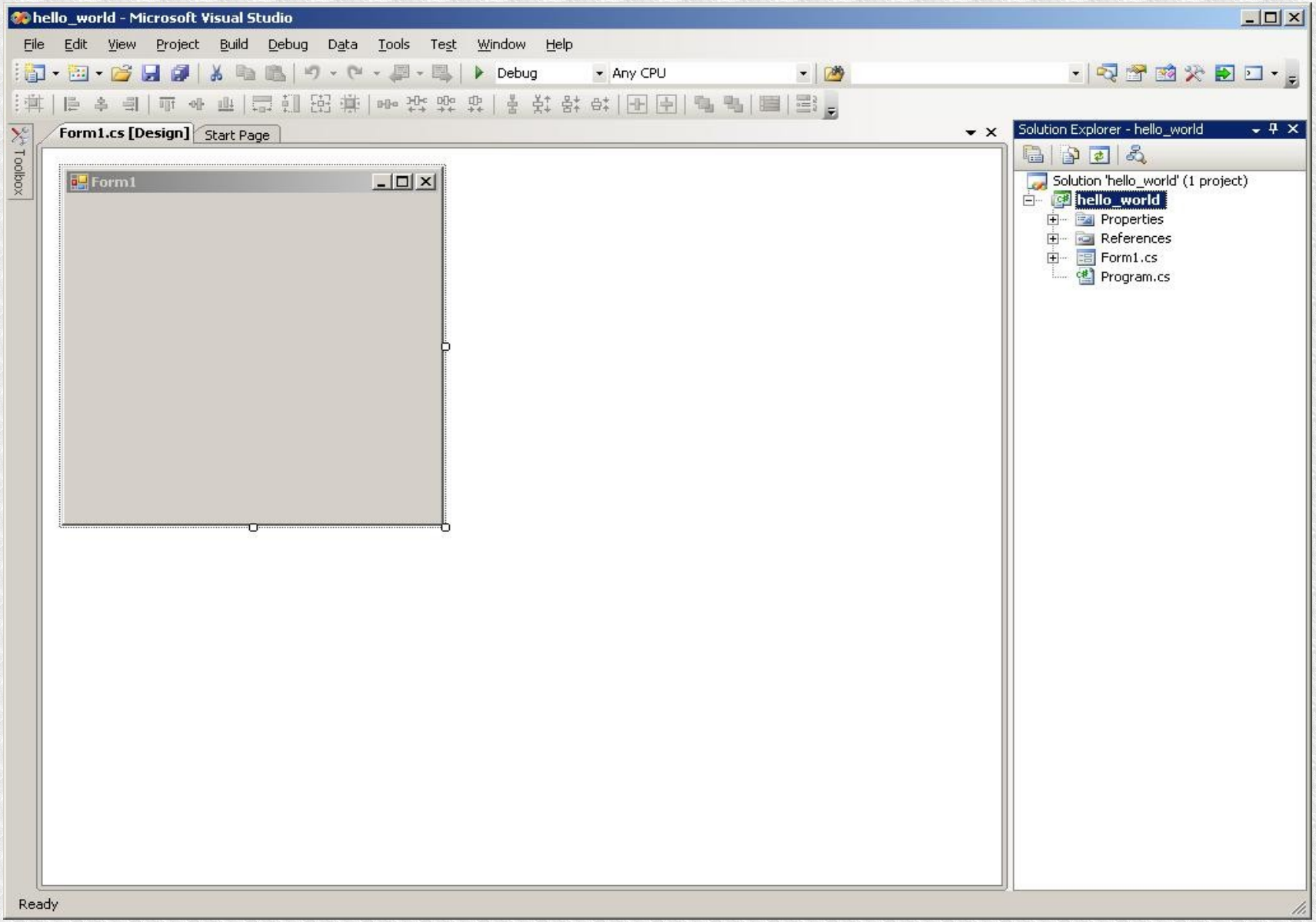
Pierwsze kroki



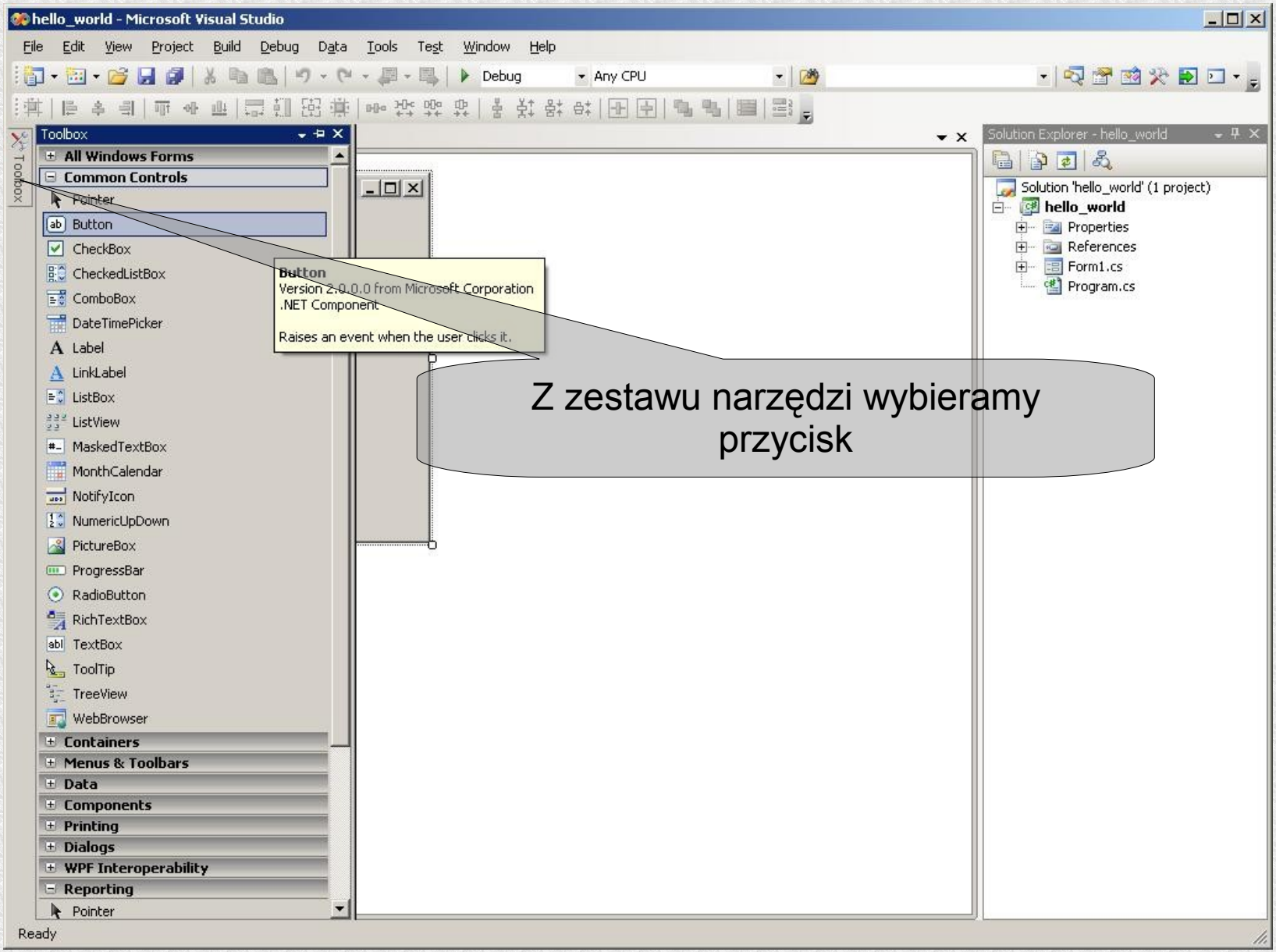
Pierwsze kroki



Pierwsze kroki

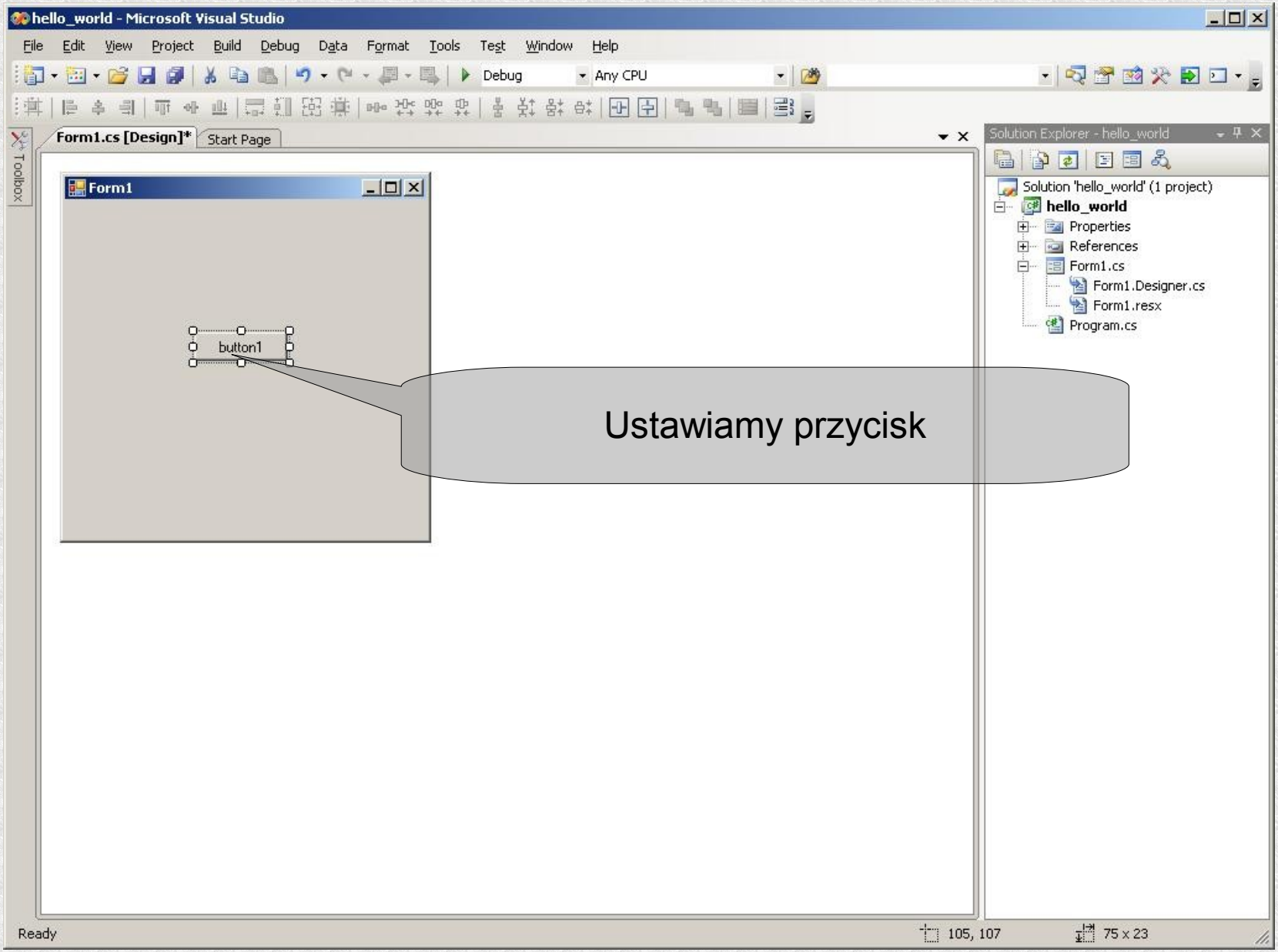


Pierwsze kroki

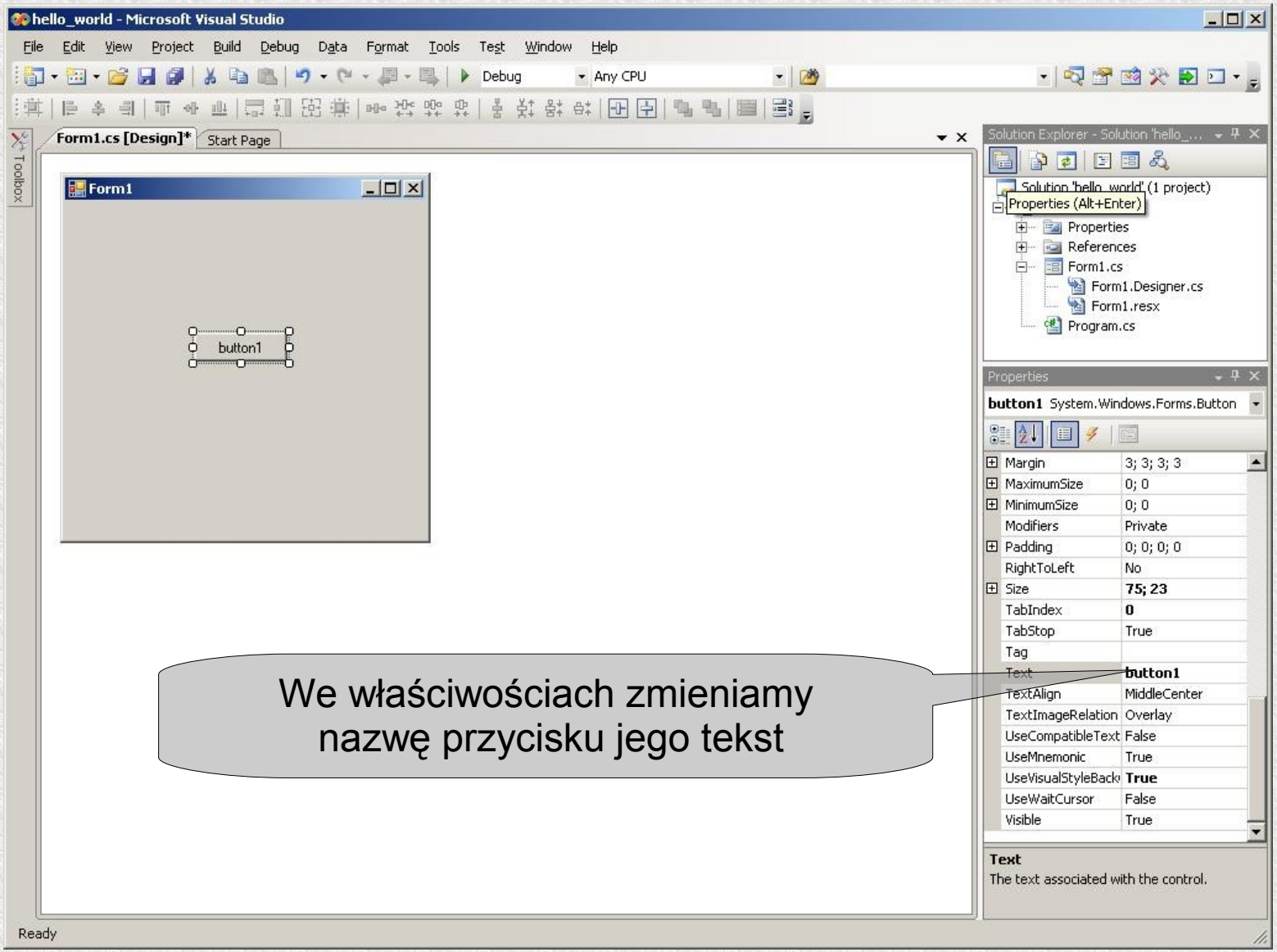


Z zestawu narzędzi wybieramy przycisk

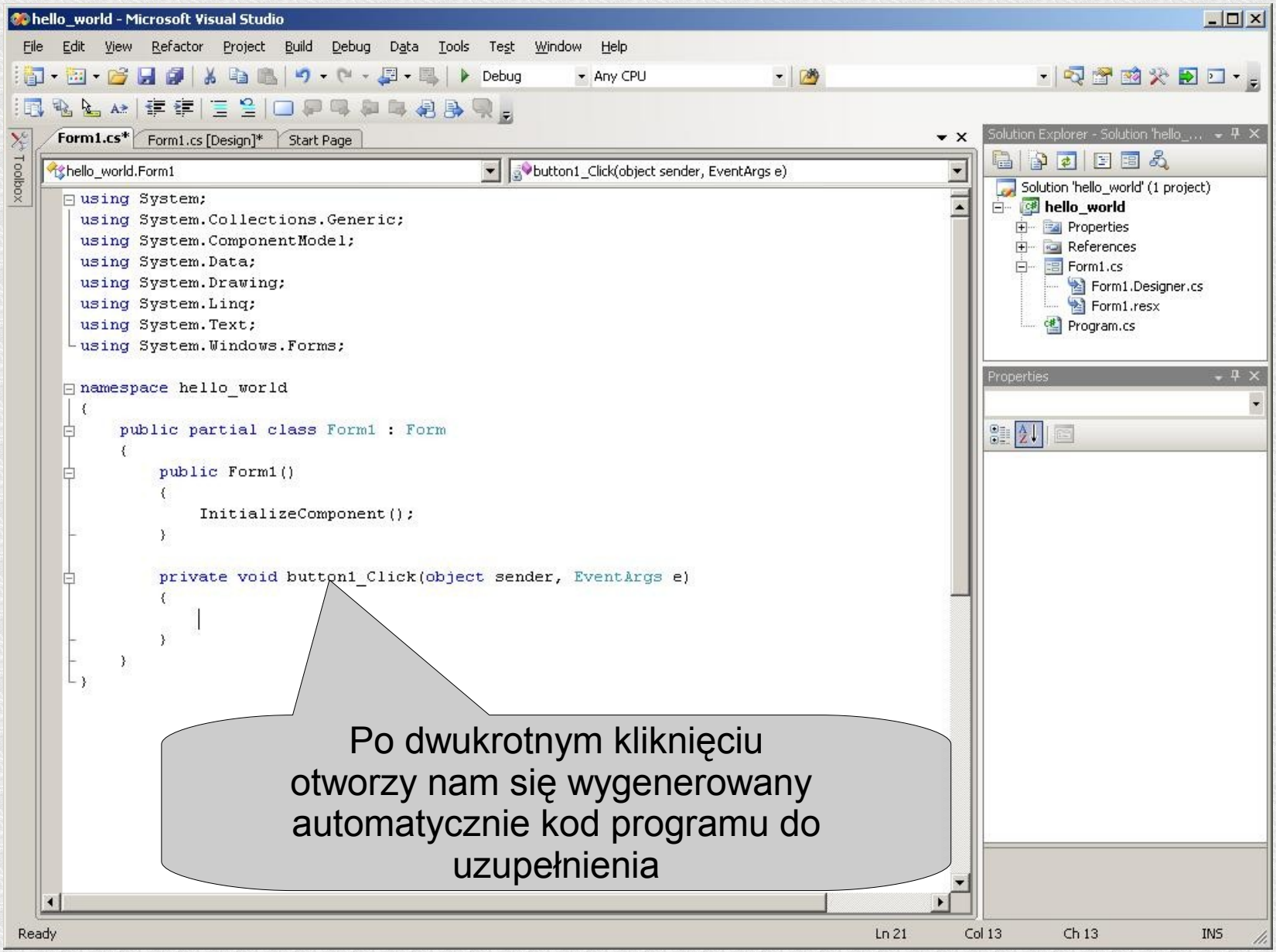
Pierwsze kroki



Pierwsze kroki



Pierwsze kroki



Pierwsze kroki

Następnie uzupełniamy nasz kod jedną linią:

```
MessageBox.Show("Hello world");
```

Kompilacja i uruchomienie F5 lub ctrl+F5

Wątki

Kiedy przydają nam się wątki?

Gdy chcemy by nasz program reagował gdy wykonuje w tle jakieś „ciężkie” zadanie

Różnego rodzaju procesy – serwery. Podczas oczekiwania na dane na jednym wątku, program może coś wykonywać na innym.

Gdy mamy program, który wykonuje sporo obliczeń (np. kompresja plików multimedialnych) i chcemy je w jakiś sposób zrównoleglić. Efekt będzie odczuwalny gdy fizycznie będziemy dysponować wieloma rdzeniami. Ilość tą można sprawdzić za pomocą **Environment.ProcessorCount**

Wątki

Kiedy wątki mogą nam szkodzić?

•Gdy będzie ich za dużo. Czas przełączania i alokacji zbyt kosztowny,

•Gdy zadanie wykonywane przez wątek będzie krócej trwało niż powołanie danego wątku.

•Gdy w pełni nie przewidzimy interakcji pomiędzy wątkami, debugowanie jest bardzo kłopotliwe.

•Gdy używamy dużo dysku, nie powinniśmy powoływać wiele wątków, a raczej jeden, dwa i szeregować zadania odczytu i zapisu. (ktoś próbował skopiować z płyty CD/DVD kilka plików naraz?)

Wątki

- Gdy operujemy na wątkach musimy dodać do programu

```
using System.Threading;
```

- Każdy program ma przynajmniej jeden wątek, zwany wątkiem głównym
- Każdy wątek ma swój oddzielny stos więc zmienne lokalne są modyfikowane niezależnie.
- Zmienne globalne są współdzielone przez wątki (często wymagana synchronizacja)

Wątki

Przykład uruchomienia metod działających jako wątki.

```
private void naszWatekBezParametrow()
        {
        MessageBox.Show("wątek ZUPELNIE bez parametrów");
        }

private void naszWatek(object o)
        {
        MessageBox.Show("jestem sobie " +
   Thread.CurrentThread.Name + "\nWiadomość to: " +
   (string)o);
        }

private void naszWatek()
        {
        MessageBox.Show("wątek bez parametrów, przeciążony");
        }
```

Wątki

Przykład uruchomienia bez parametrów

```
Thread watek = new Thread(new
                  ThreadStart(naszWatekBezParametrow));

//Thread watek = new Thread(naszWatekBezParametrow);
  //kompilator sam sobie resztę doda

//Thread watek = new Thread(new ThreadStart(naszWatek));

//Thread watek = new Thread(naszWatek); //Gdy mamy
  przeciążone metody (bez i z parametrami kompilator nie wie
  czy zastosować ThreadStart czy ParameterizedThreadStart

watek.Start();

//jako wątek bez parametrów przy próbie Start(wiadomosc)
  zakończy się błędem podczas uruchomienia

//string wiadomosc = "jakaś wiadomość";

//watek.Start(wiadomosc);
```

Wątki

Przykład uruchomienia z parametrami

```
private void naszWatekZParametrem(object wiadomosc)
          {
              MessageBox.Show((string)wiadomosc);
          }


Thread watek = new Thread(new
  ParameterizedThreadStart(naszWatekZParametrem));

//Thread watek = new Thread(naszWatekZParametrem);

//Thread watek = new Thread(new
  //ParameterizedThreadStart(naszWatek));

//watek.Start(); //Możliwy taki start ale nie będzie
  parametrów

watek.Start("jakaś wiadomość będąca parametrem");
```

Wątki

Przykład uruchomienia anonimowego

```
private void naszWatekZKonkretnymParametrem(string
   wiadomosc)
   {
      MessageBox.Show("Nasz wątek z konkretnym parametrem
   dostał   wiadomość: \n" + wiadomosc);
   }

string zmiennaWiadomosc;
   zmiennaWiadomosc = "Wiadomość przed utworzeniem wątku";

Thread watek = new Thread(delegate()
   { naszWatekZKonkretnymParametrem(zmiennaWiadomosc); }); //
   zastosowanie anonimowej metody, nie musimy podawać
   parametru object tylko możemy konkretnego typu np. string

zmiennaWiadomosc = "Wiadomość po utworzeniu wątka";

// Thread watek = new Thread(delegate()
      {  MessageBox.Show("wiadomość 1");
         MessageBox.Show("Wiadomość 2"); });
   watek.Start();
```

Wątki

Przykład uruchomienia z obiektu

```
public class RozneWatki
        {
            public void watek1()
             {
             MessageBox.Show("jestem sobie wątek1");
             }
            public void watek2()
             {
             MessageBox.Show("jestem sobie wątek2");
             }
        }

RozneWatki rozneWatki = new RozneWatki();
   Thread watek1 = new Thread(rozneWatki.watek1);
   Thread watek2 = new Thread(rozneWatki.watek2);
   watek1.Start();
   watek2.Start();
```

Wątki

Nazywanie wątków – pomoc w debugowaniu

```
Thread watek = new Thread(naszWatek);
watek.Name = "Ot taka nazwa";

private void naszWatek()
    {
    MessageBox.Show(Thread.CurrentThread.Name);
    }
```

Wątki

Wątki pierwszoplanowe i w tle

```
Thread watek = new Thread(naszWatek);
watek.IsBackground = true; //gdy true zamknięcie
                           głównego zamyka też potomny
watek.Start();

 private void naszWatek()
      {
      MessageBox.Show(Thread.CurrentThread.Name);
      }
```

Domyślnie **`IsBackground = false`** dlatego po zamknięciu głównej aplikacji nadal widzimy wątki z niej powstałe. Gdy ustawimy na **`true`** wyjście z wątka głównego powoduje natychmiastowe zakończenie wątków potomnych. Blok **`finaly`** jest pomijany. Jest to sytuacja nie pożądana dlatego powinniśmy poczekać na koniec wątków potomnych.

Zmiana pracy z tła do pierwszoplanowej i odwrotnie nie wpływa na priorytet.

Wątki

Priorytetowość

```
enum ThreadPriority { Lowest, BelowNormal, Normal,
  AboveNormal, Highest }
```

- Oznacza jak dużo czasu procesora przyznane jest dla danego wątka w grupie wątków jednego procesu.
- Ustawienie na Highest wcale nie oznacza że będzie to wątek czasu rzeczywistego. Trzeba by było również ustawić priorytet dla procesu.

```
Process.GetCurrentProcess().PriorityClass =
    ProcessPriorityClass.High;
```

- Jest jeszcze wyższy priorytet **Realtime**, wtedy nasz proces będzie działał nieprzerwanie, jednak gdy wejdzie w pętle nieskonczoną nie odzyskamy kontroli nad systemem.
- W przypadku gdy nasza aplikacja posiada (zwłaszcza skomplikowany) interfejs graficzny, też nie powinniśmy podnosić priorytetu, gdyż odświerzanie spowoduje zwolnienie całego systemu.

Wątki

Wyjątki

```
Try
     {
      watek.Start();
     }
   catch
      {
      MessageBox.Show("błąd przy uruchamianiu");
      }
```

- Taki uruchomienie zwróci nam jedynie wyjątek przy uruchamianiu, nie przechwycimy wyjątku rzuconego w wątku.
- Wyjątki z wątków mogą zakończyć aplikację, trzeba je przechwytywać na poziomie wątków.

Synchronizacja

Blokowanie

- Procesy zablokowane z powodu oczekiwania na jakieś zdarzenie, np. `Sleep, Join, lock, Semaphore` itp.
  Natychmiastowo zrzekają się czasu procesora, dodają `WaitSleepJoin` do właściwości `ThreadState` i nie kolejkują się do czasu odblokowania.
- Odblokowanie może nastąpić z 4 przyczyn:
  - Warunek odblokowania został spełniony
  - Minął timeout
  - Został przerwany przez `Thread.Interrupt`
  - Zosatł przerwany przez `Thread.Abort`

Synchronizacja

Oczekiwanie `Sleep i SpinWait`

```
static void Main()
   {
   Thread.Sleep (0); // zrzeknięcie się przydzielonego kwantu
   czasowego
   Thread.Sleep (1000); // uśpij na 1000 ms
   Thread.Sleep (TimeSpan.FromHours (1)); // uśpij na 1
   godzinę
   Thread.Sleep (Timeout.Infinite); // śpij wiecznie :) czyli
   do czasu przerwania.
   }
```

- Ogólnie Sleep powoduje rezygnację wątku z czasu procesora. Wątek taki nie jest kolejkowany przez podany czas.

```
Thread.SpinWait (100); // nic nie rób przez 100 cykli
```

- Wątek nie rezygnuje z procesora, jednak wykonuje na nim puste operacje. Nie jest w stanie `WaitSleepJoin` i nie może być przerwany przez `Interrupt`. Można zastosować gdy chcemy czekać bardzo krótko.

- Podobnie zachowuje się wątek podczas aktywnego czekania.

Synchronizacja

Oczekiwanie Join

```
Thread watek1 = new Thread(new
   ParameterizedThreadStart(naszWatekZParametrem));
   watek1.Start("Wątek z Join.");
   watek1.Join();
```

- Czekamy na zakończenie watku. Mechanizm, zbierania komunikatów nie jest zatrzymany, więc jak klikniemy jakiś guzik w wątku głównym to ostatecznie doczekamy się reakcji.

Sekcja krytyczna

Kilka wątków (n_wątków) robi to samo zadanie:

```
for (int ii = 0; ii < 1000000; ii++)
   {
        {
            licznik++;
        }
   }
```

Bez zabezpieczeń wynik będzie wynosił <= n_wątków*1000000

Sekcja krytyczna

lock

```
private object blokowacz = new object();

...

for (int ii = 0; ii < 1000000; ii++)
      {
      lock (blokowacz)
         {
         licznik++;
         }
      }
```

- W danym momencie tylko jeden wątek, może przebywać w chronionym obszarze inne będą czekały w kolejce FIFO
- Wątki czekające są w stanie **`WaitSleepJoin`**.
- Wątki takie można też zakończyć przez przerwanie lub abort

Sekcja krytyczna

Wybór obiektu który będzie blokował

- Musi to być typ referencyjny
- Zwykle jest związany z obiektami na których działamy np.:

```
class Bezpieczna {
    List <string> list = new List <string>();
    void Test() {
       lock (list) {
          list.Add ("Item 1");
  ...
```

- Powinniśmy stosować obiekty które są `private` by uniknąć niezamierzonej interakcji z zewnątrz
- Z tego samego powodu nie powinniśmy stosować np. `lock (this){}` lub `lock (typeof (Widget)) { ... }`
- Użycie obiektu do zablokowania fragmentu kodu nie powoduje automatycznie blokowania danego obiektu.

```
lock(list1) {
          list2.Add ("Item 1"); ...
```

Sekcja krytyczna

Monitor – rozwinięcie lock

- Lock faktycznie jest skrótem składniowym czegoś takiego:

```
Monitor.Enter(blokowacz);
   try
         {
         licznik++;
         }

finally
         {
         Monitor.Exit(blokowacz);
         }
```

- Wywołanie Monitor.Exit bez uprzedniego Monitor.Enter spowoduje rzucenie wyjątku

- Monitor posiada też metodę TryEnter gdzie możemy podać timeout jeżeli wejdziemy przed końcem czasu to zwróci true lub false jeżeli wystąpi timout.

Sekcja krytyczna

Interlocked – operacje atomowe

```
for (int ii = 0; ii < 1000000; ii++)
    {
    Interlocked.Increment(ref licznik);
    }
```

- Dodatkowo mamy do dyspozycji
  - `Add` Dodadawanie do dwóch liczb
  - `CompareExchange` porównanie i ewentualna podmiana
  - `Decrement` zmniejszenie
  - `Equals` czy równe
  - `Exchange` Zamiana
  - `Read` odczyt liczby 64b
  - `ReferenceEquals` porównanie dwóch referencji

Sekcja krytyczna

Blokowanie zagnieżdżone

```
static object x = new object();
  static void Main() {
  lock (x) {
     Console.WriteLine ("Zablokowałem");
     Nest();
     Console.WriteLine ("Odblokowałem");
     }
  //Tutaj odblokowane zupełnie
  }

static void Nest() {
  lock (x) {
  …//Tu podwójny lock
  }
  //Tu odblokowane tylko ostatnie zagnieżdżenie
  }
```

Wątek blokujący może blokować ile chce, jednak blokowany i tak będzie czekał na tym najbardziej zewnętrznym.

Sekcja krytyczna

- Kiedy blokować
  - Wszędzie tam gdzie wiele wątków może mieć dostęp do wspólnych zmiennych
  - Wszędzie tam gdzie chcemy mieć niepodzielność operacji, np. sprawdzenie warunku i wykonanie czegoś
- Na co uważać
  - Nie powinniśmy zbyt dużo blokować bo ciężko analizować taki kod a łatwo spowodować *DeadLock*
  - Zbyt duże fragmenty kodu wykonywane przez pojedynczy proces powodują zubożenie współbieżności.

Przerwanie wątku

- `Thread.Interrupt` – przerywa bieżące czekanie i powoduje rzucenie wyjątku `ThreadInterruptedException`

```
private void watekNieskonczony()
    {
    try {
        Thread.Sleep(Timeout.Infinite);
        }
    catch (ThreadInterruptedException ex)
        {
        MessageBox.Show("przechwycony wyjątek:"+ex.Message);
        }
    MessageBox.Show("A tu koniec");
    }
```

- Należy pamiętać, że przerywanie w ten sposób może być niebezpieczne, chyba, że wiemy dokładnie w którym miejscu jesteśmy i posprzątamy.

Przerwanie wątku

- `Thread.Abort` – Działa podobnie jak Interrupt z tą różnicą, że rzuca wyjątkiem `ThreadAbortException` oraz wyjątek jest ponownie rzucany pod koniec bloku catch, chyba że w bloku catch zastosujemy `Thread.ResetAbort();`
- Działanie jest podobne, jednak w przypadku `Interrupt` wątek przerywany jest tylko w momencie czekania, `Abort` może tego dokonać w dowolnym miejscu wykonywania, nawet w nienaszym kodzie.

Stany wątku

- **`ThreadState`** – kombinacja bitowa trzech warstw.
- Uruchomienie, blokada, przerwanie wątku (`Unstarted, Running, WaitSleepJoin, Stopped, AbortRequested`)
- Pierwszoplanowość i drugoplanowość wątku (`Background, Foreground`)
- Postęp w zawieszeniu wątku (`SuspendRequested, Suspended`) używane przez przestarzałe metody
- Ostateczny stan wątku określa się przez sumę bitową tych trzech „Warstw”. I tak, może być np wątek

`Background, Unstarted`

lub

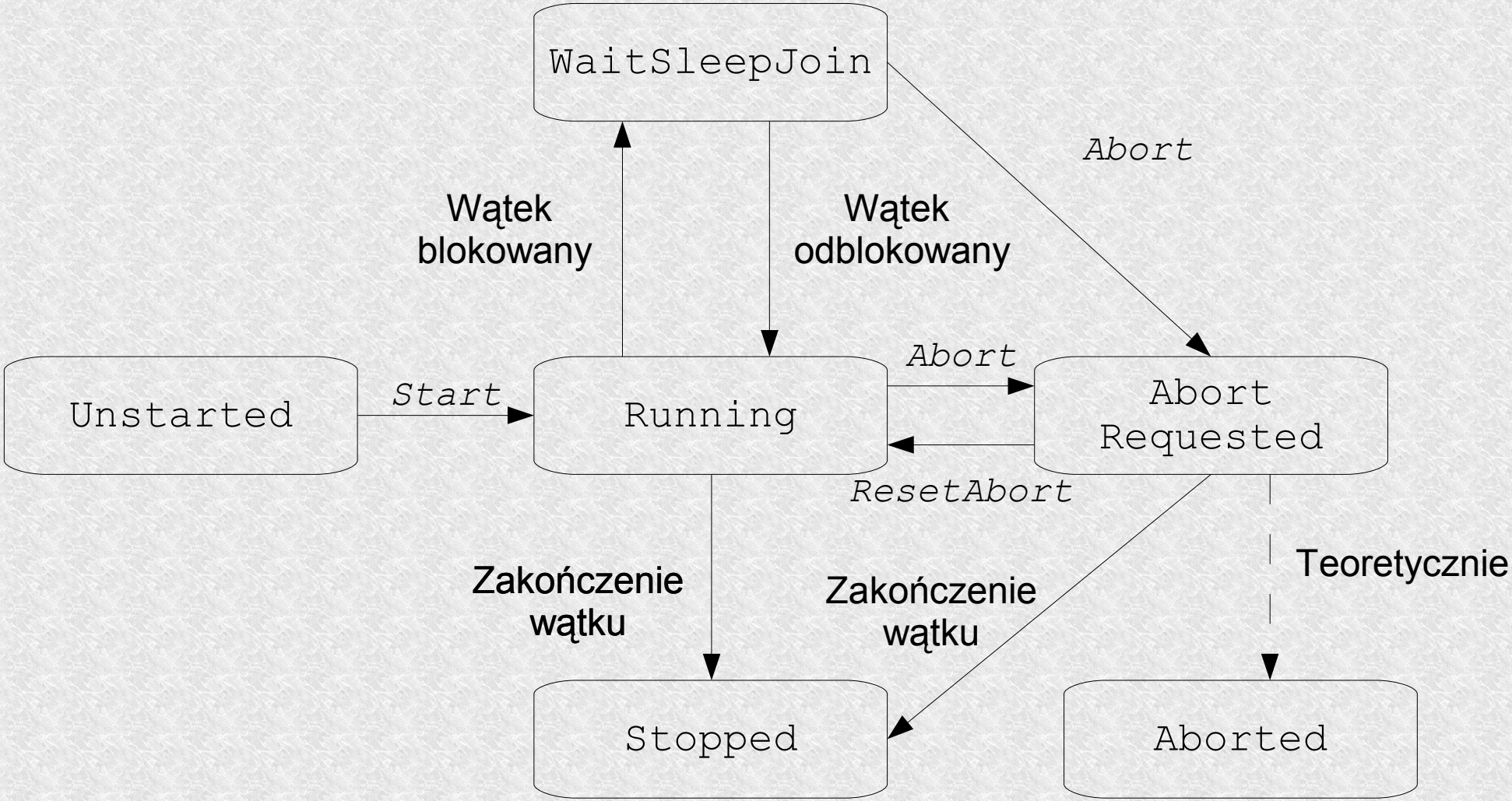
`SuspendRequested, Background, WaitSleepJoin`

Stany wątku

- W enumeracji ThreadState są też nigdy nie używane dwa stany: **`StopRequested`** i **`Aborted`**
- By jeszcze bardziej skomplikować, Running ma wartość 0
  więc porównanie
  `if ((t.ThreadState & ThreadState.Running) > 0)...`
  nic nam nie da
- Można się wspomóc IsAlive jednak zwraca false tylko przed startem i gdy się zakończy. Gdy jest zablokowany też jest true.
- Najlepiej napisać sobie swoją metodę:

```
public static ThreadState SimpleThreadState (ThreadState ts)
    {
    return ts & (ThreadState.Aborted |
                 ThreadState.AbortRequested |
                 ThreadState.Stopped |
                 ThreadState.Unstarted |
                 ThreadState.WaitSleepJoin);
    }
```

Stany wątku


WaitSleepJoin
Abort
Wątek blokowany
Wątek odblokowany
Unstarted
Start
Running
Abort
Abort Requested
ResetAbort
Zakończenie wątku
Zakończenie wątku
Teoretycznie
Stopped
Aborted

Wait Handles

- Win32 Api dostarcza trzech klas
  - `EventWaitHandle`
  - `Mutex`
  - `Semaphore`
- Wszystkie 3 bazują na abstrakcyjnej klasie `WaitHandle`
- `EventWaitHandle` ma dwie podklasy
  - `AutoResetEvent`
  - `ManualResetEvent`
- Różnią się one tylko sposobem wywołania konstruktora.
- `WaitHandles` pozwalają na nazwanie klas i używanie pomiędzy odrębnymi procesami

Wait Handles

`AutoResetEvent`

- Można porównać do bramki która przepuszcza tylko jeden proces za naciśnięciem jednego guzika.
- Gdy bramka jest otwarta proces lub wątek który wywoła metodę `WaitOne()` przechodzi przez bramkę jednocześnie ją zamykając
- Gdy bramka jest zamknięta proces ustawia się w kolejce.
- Każdy inny nie zablokowany proces może odblokować bramkę za pomocą wywołania metody `Set()`
- Jedno wywołanie `Set()` wpuści tylko jeden proces.
- Gdy nie będzie procesów w kolejce, `Set()` otworzy bramkę
- Gdy bramka jest już otwarta, następne `Set()` są ignorowane.

Wait Handles

AutoResetEvent

```
EventWaitHandle czekaczka = new EventWaitHandle (false,
                                    EventResetMode.Auto);
EventWaitHandle czekaczka = new AutoResetEvent (false);
```

Powyższe dwa wywołania są równoważne.

Pierwszy parametr określa czy bramka ma być podczas utworzenia otwarta.

Przykład (WaitHandles)

Wait Handles

`EventWaitHandle` - międzyprocesowe

```
EventWaitHandle czekaczka = new EventWaitHandle (false,
            EventResetMode.Auto,"Nasza nazwa czekaczki");
```

- Trzecim parametrem może być nazwa widziana przez wszystkie inne procesy w systemie.
- Gdy podczas tworzenia okaże sie że obiekt o podanej nazwie istnieje dostaniemy tylko referencję a czwarty parametr będzie false;

```
EventWaitHandle (false, EventResetMode.Auto,"Nasza nazwa
czekaczki", out czyNowy);
```

Przykład (WaitHandles)

Ready Go

Załóżmy, że mamy taki scenariusz

- Główny proces ma co chwilę nowe zadania do wykonania
- Zadania te mają być wykonane przez wątek
- Za każdym razem uruchamiany jest nowy wątek
- Przekazywane jest zadanie
- Po wykonaniu pracy wątek jest kończony

By zmniejszyć obciążenie wynikające z tworzenia wątków (czy nawet innych procesów) możemy postępować według poniższego algorytmu:

- Główny proces tworzy wątek
- Wątek czeka na zadanie
- Wykonuje zadanie
- Przechodzi w stan oczekiwania na kolejne zadanie

Ready Go

Najprostsza wersja producenta i konsumenta

```
static EventWaitHandle ready = new AutoResetEvent(false);
static EventWaitHandle go = new AutoResetEvent(false);
static volatile string zadanie;
static void Main(string[] args)
    {
        new Thread(Konsument).Start();
        for (int i = 1; i <= 5; i++) //przekaż 5 razy zadanie
        {
            ready.WaitOne(); // Czekamy na gotowość konsumenta
            zadanie = "a".PadRight(i, 'a'); // przygotowujemy zadanie
            go.Set(); // mówimy że dane gotowe do odbioru
        }
        ready.WaitOne(); zadanie = null; go.Set(); // każemy skończyć
        Console.ReadKey();
    }
static void Konsument()
    {
        while (true)
        {
            ready.Set(); // Informujemy producenta że jesteśmy gotowi
            go.WaitOne(); // i czekamy na dane
            if (zadanie == null) return; // gdy dostaniemy null kończymy
            Console.WriteLine(zadanie);
        }
    }
```

Kolejka producent - konsument

- Wykorzystanie procesu drugoplanowego
- Producent kolejkuje elementy
- Konsument dekolejkuje elementy
- Rozwiązanie podobne do poprzedniego tylko nie blokujące

```
class ProducentKonsument : Idisposable
      {
      EventWaitHandle czekaczka = new AutoResetEvent(false);
      Thread konsumentWatek;
      Queue<string> kolejka = new Queue<string>();
      public ProducentKonsument()
            {
            konsumentWatek = new Thread(konsument);
            konsumentWatek.Name = "konsumentWatek";
            konsumentWatek.Start();
            }
```

Kolejka producent - konsument

```
void konsument()
     {
     while (true)
           {
           string mesg = null;
           lock (kolejka)
                if (kolejka.Count > 0)
                   {
                   mesg = kolejka.Dequeue();
                   if (mesg == null) return;
                   }
                if (mesg != null)
                   {
                   Console.WriteLine("odebralem: " + mesg);
                   Thread.Sleep(1000);
                   }
                else
                   {
                   Console.WriteLine("no to czekam...");
                   // Jeżeli nie ma więcej zadań to czekaj
                   czekaczka.WaitOne();
                   }
           }
     }
```

Kolejka producent - konsument

```
public void zakolejkuj(string mesg)
        {
        lock (kolejka)
             {
             kolejka.Enqueue(mesg);
             }
        czekaczka.Set();
        }

public void Dispose()
        {
        zakolejkuj(null);
        konsumentWatek.Join();
        czekaczka.Close();
        }
   }
```

Wait Handles

`ManualResetEvent`

```
EventWaitHandle czekaczka = new EventWaitHandle (false,
  EventResetMode.Manual);

EventWaitHandle czekaczka = new ManualResetEvent (false);
```

- Powyższe dwa wywołania są równoważne.
- Pierwszy parametr określa czy bramka ma być podczas utworzenia otwarta.
- Metoda `Set` wpuszcza wszystkich czekających lub wołających `WaitOne` dopóki nie zamknie się przez `Reset`.

Wait Handles

`Mutex`

- Działa tak samo jak `lock` z tym że może być używany pomiędzy procesami i jest około 100 razy wolniejszy (przy założeniu że nie blokujemy)
- Tak samo jak lock zapewnia wyłączny dostęp do bloku programu pomiędzy wywołaniem WaitOne a ReleaseMutex i musi być wywołany z tego samego wątka.
- Zaletą jest automatyczne zwolnienie mutexa nawet gdy aplikacja się zakończy nie wywołując `ReleaseMutex`

Wait Handles

Mutex

```
static Mutex mutex = new Mutex(false, "tu.kielce.pl mutex");
private void naszWatekZMutex(object o)
        {
            for (int ii = 0; ii < 1000000; ii++)
            {
                mutex.WaitOne();
                licznik++;
                mutex.ReleaseMutex();
            }
        }
```

Przykład Sekajca_krytyczna1

Wait Handles

`Semaphore`

Semafor jest jak licznik który nigdy nie może być mniejszy od 0. Operacja **`WaitOne`** zmniejsza ten licznik o 1 jeżeli jest 0 to dany wątek czeka aż inny zwiększy za pomocą **`Release`**.

W przypadku semafora, podnieść go może każdy inny wątek, nie tylko ten który go opuścił, tak jak to jest w przypadku lock czy Mutex.

Semafor jest nieco szybszy od Mutexa.

Wait Handles

Semaphore

```
static Semaphore semafor = new Semaphore(1, 1);

private void naszWatekZSemaphore(object o)
          {
              for (int ii = 0; ii < 1000000; ii++)
              {
                  semafor.WaitOne();
                  licznik++;
                  semafor.Release();
              }
          }
```

Przykład Sekajca_krytyczna1

Wait Handles

Wait, wait, wait...

WaitHandle.SignalAndWait – Jednoczesne wysłanie sygnału i czekanie. Można w ten sposób zrealizowanie np. spotkań.

```
private static EventWaitHandle wh1 =
    new
  EventWaitHandle(false,EventResetMode.AutoReset);
  private static EventWaitHandle wh2 =
    new
  EventWaitHandle(false,EventResetMode.AutoReset);
```

Jeden z wątków wywołuje:

```
WaitHandle.SignalAndWait(wh1, wh2);
```

Drugi z wątków wywołuje:

```
WaitHandle.SignalAndWait(wh2, wh1);
```

Wait Handles

Wait, wait, wait...

`WaitHandle.WaitAll(WaitHandle[] waitHandles)` –
Czekaj na pozwolenie od wszystkich z waitHandles

`WaitHandle.WaitAny(WaitHandle[] waitHandles)`
– Czekaj na pozwolenie od którego kolwiek z waitHandles

Wait Handles

`ContextBoundObject`

Automatyczne blokowanie wywołań metod z jednej instancji klasy.

```
using System.Runtime.Remoting.Contexts;

[Synchronization]
  public class JakasKlasa : ContextBoundObject
  {
  ...}
```

CLR (Common Language Runtime) zapewnia, że tylko jeden wątek może wywołać kod tej samej instancji obiektu w tym samym czasie. Sztuczka polega na tym, że podczas tworzenia obiektu klasy `JakasKlasa` tworzony jest obiekt proxy przez którego przechodzą wywołania metod klasy `JakasKlasa`.

Wait Handles

`ContextBoundObject`

Automatyczna synchronizacja nie może być stosowana do pól *protect static* ani klas wywodzących się od **`ContextBoundObject`** np Windows Form

Trzeba też pamiętać, że nadal nie rozwiązuje nam to problemu gdy wywołamy dla kolekcji coś takiego:

```
 BezpiecznaKlasa bezpieka = new
  BezpiecznaKlasa();

...

if (bezpieka.Count > 0) bezpieka.RemoveAt (0);
```

Wait Handles

ContextBoundObject

Jeżeli z bezpiecznego obiektu tworzony jest kolejny obiekt to automatycznie jest on też bezpieczny w tym samym kontekscie, chyba, że postanowimy inaczej za pomocą atrybutów.

```
[Synchronization (SynchronizationAttribute.REQUIRES_NEW)]

public class JakasKlasaB : ContextBoundObject { ...
```

NOT_SUPPORTED - równoważne z nieużywaniem Synchronized

SUPPORTED - dołącz do istniejącego kontekstu synchronizacji jeżeli jest stworzony z innego obiektu, w innym przypadku będzie niesynchronizowany

REQUIRED - (domyślny) dołącz do istniejącego kontekstu synchronizacji jeżeli jest stworzony z innego obiektu, w innym przypadku stwórz swój nowy kontekst synchronizacji

REQUIRES_NEW - Zawsze twórz nowy kontekst synchronizacji

Delegaty i zdarzenia

Obiektowy odpowiednik wskaźnika do funkcji z c/c++

klasa pochodną z klasy System.Delegate

Deklaracja wygląda jak deklaracja funkcji:

```
delegate void PrzykladowyDelegat(); //deklaracja delegatu
  która jest równoważna z deklaracją klasy
```

By wykorzystać delegat musimy stworzyć nowy obiekt tej klasy.

```
delegate ddd = new PrzykladowyDelegat(jakasFunkcja);
```

jakasFunkcja musi być tego samego typu co delegat.

Funkcjonalnie zachowuje się jak klasy wewnętrzne w javie z tym że w javie trzeba było tworzyć całą klasę tu tylko metodę.

Delegaty i zdarzenia

```
//PrzykladDelegate
  class Program
      {
      delegate void JakisDelegat();
      // metoda zgodna z deklaracją delegatu
      static void jakasFunkcja()
          {
          System.Console.WriteLine("Jakas Funkcja!?
  Przecieże to był delegat!");
          }
      static void Main(string[] args)
          {
           JakisDelegat ddd = new
  JakisDelegat(jakasFunkcja);
         // w tym miejscu delegujemy wywołanie metody
  jakasFunkcja()
           ddd();
           Console.ReadLine();
          }
      }
```

Delegaty i zdarzenia

Zdarzenia (events) są formą komunikacji informowania innych, że wystąpiła jakaś sytuacja.

Realizowane są przy pomocy delegatów. Przykład eventa wraz argumentami:

```
public delegate void TikTakEventHandler(object sender,
                                        TikTakEventArgs e);

public event TikTakEventHandler cykEvent;

public class TikTakEventArgs : EventArgs
          {
              public int ktoreCykniecie;
              public bool czyWiecznie;
          }
```

Delegaty i zdarzenia

Zasygnalizowanie zdarzenia:

```
TikTakEventArgs tta = new TikTakEventArgs();
  tta.czyWiecznie = true;
  for (int ii = 0;ii<ileRazy;ii++)
             {
             tta.ktoreCykniecie = ii;
             if (cykEvent!=null)
                cykEvent(this,tta);
             Thread.Sleep(1000);
             }
```

Przykład: PrzykladEventa

Delegaty i zdarzenia

Cross-Thread

W wielowątkowej aplikacji okienkowej nie można używać metod ani pól kontrolek, które nie zostały stworzone w danym wątku.

Gdy chcemy zapisać coś na formatce z innego wątku

```
private void niebezpieczneLiczydlo()
      {
      for (int liczba = 0; liczba < 20; liczba++)
            labelLicznik.Text = liczba.ToString();
      }
   ...
   Thread watekLicz = new Thread(niebezpieczneLiczydlo);
   watekLicz.Start();
```

Przykład: przykladCrossThread

dostaniemy:

Cross-thread operation not valid: Control 'labelLicznik' accessed from a thread other than the thread it was created on.

Trzeba posłużyć się pewną sztuczką, którą podpowiada MS.

Delegaty i zdarzenia

```
private void liczydloBezpieczne()
   {
   for (int liczba = 0; liczba < 20; liczba++)
        {
        if (this.labelLicznik.InvokeRequired)
           {
           // jest w innym wątku więc wymaga
           ZapiszTekst zapDel = new ZapiszTekst(zapiszTekst);
           //wywołuje delegata zapDel przekazując mu tablicę
           //parametrów w tym przypadku jeden parametr
           this.Invoke(zapDel, new object[]
 {liczba.ToString()});
           }
        else
           {
           // Gdy jest w tym samym wątku
           this.labelLicznik.Text = liczba.ToString();
           }
        Thread.Sleep(1000);
        }
   }
```

Apartment Threading

Jest logicznym kontenerem zawierającym:

Jeden wątek (Single-Threded Apartment)

Wiele wątków (Multi-Threded Apartment)

- Apartment może zawierać zarówno wątki jak i obiekty
- Kontekst Synchronizacji mógł tylko obiekty
- Obiekty takie są przypisane do Apartment'u przez cały okres trwania.
- Obiekty w kontekście synchronizacji mogą być wołane przez dowolne wątki (ale w trybie exclusive)
- Obiekty w Apartment mogą być wywołane tylko przez wątki w tym samym Apartmencie.

Apartment Threading

Jeżeli można by było przedstawić obrazowo kontener jako bibliotekę, obiekty jako książki a wątki jako osoby to:

- W przypadku kontekstu synchronizacji do biblioteki wchodził by sobie ktokolwiek ale zawsze pojedynczo (nie może być dwóch osób w bibliotece)
- W przypadku apartmentów mamy pracowników przypisanych do biblioteki.
  - Single. Jest jeden bibliotekarz i do niego odwołujemy się by coś przeczytał i nam powiedział co w danej książce jest
  - Multi. Jest wiele bibliotekarzy.

Apartment Threading

- Sygnalizowanie bibliotekarzowi , że chcemy skorzystać z jakiejś książki (obiektu) nosi nazwę „marshalling” (przekazywanie?).
- Metoda jest przekazywana (marshal) poprzez pracownika biblioteki i wykonywana na obiektach w bibliotece
- Marshalling jest zaimplementowany w „bibliotekarzu” przez system komunikatów w Windows Forms i jest przekazywane automatycznie
- Jest to mechanizm który cały czas sprawdza zdażenia związane z myszką i klawiaturą. Gdy nie zdążą być obsłużone są kolejkowane (FIFO).

Apartment Threading

- Wątki .Net są automatycznie przypisane do multi-thread-appartment chyba, że chcemy by było inaczej wtedy:

```
Thread t = new Thread (...);
t.SetApartmentState (ApartmentState.STA);
```

Lub

```
class Program {
[STAThread]
static void Main() {
...
```

Apartment Threading

- Gdy nasza aplikacja to czysty kod .Net Apartment nie ma znaczenia.
- Typy z System.Windows.Forms korzystają z kodu Win32 przeznaczonego do pracy w Single Thread Apartment. Z tego powodu w programach tego typu wymagane jest [STAThread].

Background Worker

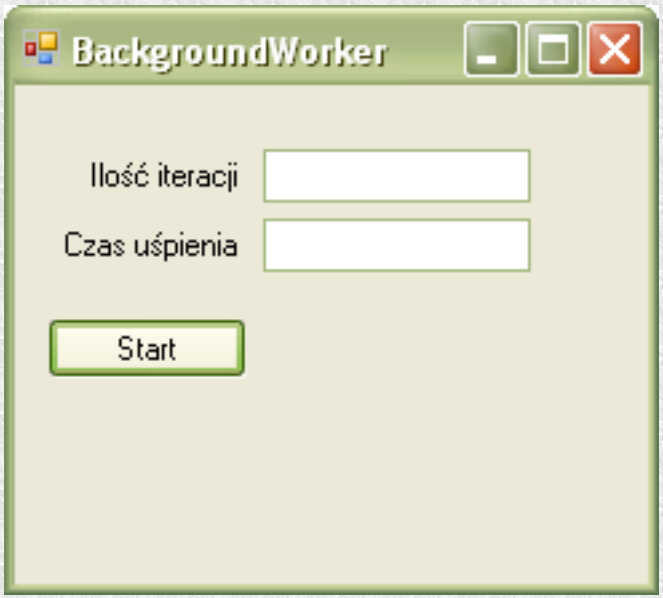
- Jest pomocną klasą z *System.ComponentModel* która dostarcza nam następującej funkcjonalności
  - Flaga *cancel* do zasygnalizowania końca, zamiast *Abort*
  - Standardowy protokół do do raportowania postępu, zakończenia i przerwania pracy
  - Implementacja *IComponent* pozwalająca na umieszczenie w VS Designerze.
  - Łapanie wyjątków w wątku *workera*
  - Możliwość zapisywania postępu bezpośrednio na formatkę w innym wątku (nie ma problemu z wywołaniem Cross-thread), nie musimy używać *Control.Invoke*

Background Worker

- Model tego typu wykorzystuje identyczną składnię, jak asynchroniczne delegaty
- Aby użyć BacgroundWorkera wystarczy poinformować go jaka metoda ma być wykonana w tle i wywołać RunWorkerAsync()
- Wątek główny kontynuuje działanie a w tle wykonywana jest funkcja zgłoszona do BackgroundWorkera.
- Gdy BW skończy sygnalizuje to zdarzeniem RunWorkerCompleted

Background Worker

- Zróbmy sobie formatkę jak niżej:



- Pola nazwane będą textBoxCzas oraz textBoxIlosc
- Guzik nazwany po prostu buttonStart
- Dodajemy backgroundWorkera przeciągając go z toolsów i zmieniamy jego nazwę na backgroundWorkerLiczydlo

Background Worker

- Dwukrotnie klikając w BW otworzy nam się kod wykonywany podczas zdarzenia DoWork
- Uzupełniamy go

```
private void backgroundWorkerLiczydlo_DoWork(object sender, DoWorkEventArgs e)
        {
            //uzyskaj obiekt wejsciowy
            ParametryDwa parametry = (ParametryDwa)e.Argument;
            for (int i = 0; i < parametry.iloscIteracji; i++)
            {
                Console.WriteLine("właśnie wykonuję iterację " + i);
                System.Threading.Thread.Sleep(parametry.czasUspienia);
            }
        }
```

- Klasa parametry:

```
class ParametryDwa
        {
            public int czasUspienia,iloscIteracji;
            public ParametryDwa(int _czasUspienia, int _iloscIteracji)
            {
                czasUspienia = _czasUspienia;
                iloscIteracji = _iloscIteracji;
            }
        }
```

Background Worker

- Dwukrotnie klikając w buttonStart otwiera nam się kod wykonany po kliknięciu w guzik (do uzupełnienia):

```
private void buttonStart_Click(object sender, EventArgs e)
        {
            try
            {
                //uzyskaj dane z formatki
                int czas = int.Parse(textBoxCzas.Text);
                int ilosc = int.Parse(textBoxIlosc.Text);
                //umieść to w obiekcie do wysyłki
                ParametryDwa param = new ParametryDwa(czas, ilosc);
                backgroundWorkerLiczydlo.RunWorkerAsync(param);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
```

- Uruchamiamy (F5), wypełniamy, klikamy start
- Backgroundworker wykonuje swoją pracę a my możemy dalej obsługiwać aplikację

Background Worker

- Gdy chcemy się dowiedzieć o końcu zadania i przekazać jakieś parametry wystarczy obsłużyć zdarzenie: RunWorkerCompleted .

```
private void backgroundWorkerLiczydlo_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
        {
            MessageBox.Show("Praca została wykonana\n" + e.Result);
        }
```

- oraz w DoWork przygotować wynik

```
private void backgroundWorkerLiczydlo_DoWork(object sender, DoWorkEventArgs e)
      {
          //uzyskaj obiekt wejsciowy
          ParametryDwa parametry = (ParametryDwa)e.Argument;
          for (int i = 0; i < parametry.iloscIteracji; i++)
          {
              Console.WriteLine("właśnie wykonuję iterację " + i);
              System.Threading.Thread.Sleep(parametry.czasUspienia);
          }
          string komunikat = "W sumie " + (parametry.iloscIteracji *
parametry.czasUspienia) + " milisekund";
          e.Result = komunikat;
      }
```


Przykład: BackgroundWorker