



Curs 10

Thread-uri și sincronizare

Overview

- Crearea aplicațiilor multithread
- Elemente de sincronizare
- Apeluri asincrone
- Contexte de sincronizare



În cadrul unui proces codul este executat de către un thread. Puteți folosi clasele **Microsoft .NET Framework** pentru a crea obiecte ce reprezintă fire de execuție, pentru a executa cod și pentru a sincroniza firele de execuție.


Thread-urile rulează în cadrul unui domeniu de aplicație, care este o unitate de izolare în cadrul unui proces .NET Framework.

În acest modul veți învăța cum să folosiți clasele .NET Framework pentru a construi aplicații multithread, cum să realizați servicii Windows și clase pentru instalarea serviciilor și cum să accesați și să configurați domenii de aplicație.

Obiective

După completarea acestui modul veți fi capabili să:

- ☐ Descrieți și să implementați aplicații multithread prin folosirea namespace-ului **System.Threading**.
- ☐ Creați, instalați și controlați un serviciu Windows prin folosirea namespace-ului **System.ServiceProcess**.
- ☐ Descrieți și creați domenii de aplicație.



Crearea aplicațiilor multithread

- ▶ Introducere în thread-uri
- ▶ Crearea thread-urilor
- ▶ Folosirea clasei **Thread**
- ▶ Obiecte de sincronizare a thread-urilor
- ▶ Folosirea thread-urilor **Thread Pool**
- ▶ Folosirea cronometrelor
- ▶ Realizarea de apeluri asincrone
- ▶ Folosirea thread-urilor în codul de interfață cu utilizatorul





11/20/2014 3

Fiecare proces are cel puțin un fir de execuție. Puteți folosi aceste thread-uri pentru a scrie cod bazat pe task-uri, executat concurent. Totuși concurența prezintă probleme, în special în ceea ce privește accesul la date. Acest capitol prezintă clasele furnizate de **.NET Framework** pentru scrierea codului multithread.

Obiective:

După completarea acestui capitol veți fi capabili să:

- ☐ Descrieți scopul firelor de execuție.
- ☐ Scrieți cod pentru a crea thread-uri.
- ☐ Setați proprietățile unui thread pentru a controla execuția acestuia.
- ☐ Scrieți cod pentru a sincroniza accesul thread-urilor la resurse.
- ☐ Folosiți fire de execuție thread pool.
- ☐ Folosiți obiecte Timer pentru a apela cod la anumite intervale de timp.
- ☐ Folosiți thread-uri pentru a realiza apeluri asincrone de metode.



Introducere **Threading**

- ▶ Thread-uri Windows
 - ▶ O unitate de execuție
 - ▶ Permite dezvoltatorilor să se orienteze pe implementări bazate pe task-uri
 - ▶ Îmbunătățește performanța aplicațiilor dependente I/O
 - ▶ Poate reduce performanța aplicațiilor dependente CPU
- ▶ Thread-uri .NET Framework
 - ▶ Thread-uri logice bazate de obicei pe thread-uri Windows
 - ▶ .NET Framework este proiectat a fi thread-aware

11/20/2014 4

.NET Framework a fost realizat astfel încât să fie țină cont de firele de execuție. Dacă o clasă este sensibilă la thread-uri se poate realiza o corupere a datelor, dacă mai mult de un thread încearcă să acceseze aceeași instanță a unei clase, în același moment.

Biblioteca **.NET Framework** cuprinde astfel de clase, de aceea trebuie să vă asigurați că accesați obiectele create folosind aceste clase într-un mod sigur.

I/O Bound Code

Un computer prezintă un număr fix de procesoare, și fiecare procesor poate rula un singur fir de execuție la un moment dat. Fiecare fir de execuție dintr-o aplicație este planificat să ruleze pe procesor pentru o scurtă durată de timp. Această planificare poate adăuga un overhead în lucrul cu thread-uri.

Dacă este codul dependent **I/O**, codul apelează un dispozitiv de intrare/ieșire, iar când un thread realizează un astfel de apel, thread-ul este suspendat până când dispozitivul și-a terminat task-ul. Dacă acest fir de execuție folosește apelul **I/O** pentru a realiza alt

task, performanța task-ului este afectată.

Folosirea thread-urilor pentru apeluri **I/O** îmbunătățește scalabilitatea unei aplicații și poate mări performanța codului dumneavoastră.

Dacă este codul dependent de **CPU**, deoarece codul nu realizează apeluri **I/O**, un fir de execuție pierde mai mult timp executând codul pe **CPU**.

Cod bazat pe task-uri

Atunci când folosiți thread-uri, codul devine bazat pe task-uri. Dacă aveți un task de lungă durată, puteți porni acest task pe un thread separat și preluați rezultatul când task-ul s-a terminat. În timp ce acest task este executat aplicația dumneavoastră poate realiza alte operații ce nu au legătură cu task-ul inițial de lungă durată.

Windows Threads

Windows furnizează obiecte fire de execuție și manipulează planificarea acestor fire de execuție. Ținând cont de prioritatea thread-urilor, **Windows** decide care thread va rula și pentru cât timp.

Deoarece întotdeauna sunt mai multe thread-uri decât numărul de procesoare din computer, **Windows** permite fiecărui fir de execuție să ruleze pentru o scurtă perioadă de timp, înainte să fie suspendat și alt thread va urma să ruleze. Acest lucru se numește planificare.


Componente ale interfeței cu utilizatorul, precum ferestre și controale, prezintă

afinitate pentru firele de execuție și pot fi accesate doar de thread-ul care le-a creat. În plus, obiectele **COM** sunt scrise astfel încât să țină cont de thread-uri, astfel acestea sunt marcate fie sigure pentru acces multithread, fie ca având afinitate pentru thread-uri.

Majoritatea metodelor sunt apelate sincron, însă unele metode sunt apelate asincron, ceea ce înseamnă că thread-ul curent începe execuția metodei pe un alt thread, iar apelul se întoarce imediat. Metodele sincrone se mai numesc blocante.


Thread-uri .NET Framework

Un obiect fir de execuție **.NET Framework** este un fir de execuție logic. În majoritatea cazurilor, este folosit un thread **Windows**. **.NET Framework** ține cont de obiecte **COM**, dar uneori poate fi necesar să specificați că o metodă va fi folosită pentru a executa obiecte **COM** pe un singur fir de execuție.



Crearea thread-urilor

- ▶ Crearea în două etape
 - ▶ Creați un nou obiect thread, pasați procedura thread-ului la constructor
 - ▶ Apelați metoda **Start** pentru a rula procedura thread-ului
- ▶ Delegatul **ThreadStart**
 - ▶ Pentru o procedură de thread fără niciun parametru
- ▶ Delegatul **ParameterizedThreadStart**
 - ▶ Pentru o procedură de thread cu un obiect parametru
 - ▶ Parametrul este dat prin intermediul metodei **Start**



11/20/2014 5

Un obiect fir de execuție **.NET Framework** este un fir de execuție logic. Atunci când creați un thread, creați infrastructura, iar thread-ul este creat la apelul metodei **Thread.Start**.

Pornirea unui thread

Trebuie să folosiți un proces de construcție în două etape: creați obiectul fir de execuție, furnizați procedura acestuia și apoi apelați metoda **Start** pentru a porni procedura thread-ului. Thread-ul își continuă execuția până când este aruncată o excepție, este apelată metoda **Abort** de către alt thread sau procedura thread-ului își completează task-urile.

Cele două metode prin care puteți crea un obiect fir de execuție determină modul în care veți apela metoda **Start**:

Dați ca parametru constructorului un delegat **ThreadStart**. Acest delegat nu are niciun parametru; astfel nu dați niciun parametru metodei **Start**.

Dați ca parametru constructorului un delegat **ParameterizedThreadStart**. Acest delegat este pentru o metodă cu un singur parametru, pe care îl dați prin intermediul metodei **Start**.

Crearea thread-urilor

► Exemplu

```
void CreateThread()
{
    Thread t = new Thread(new ParameterizedThreadStart(ThreadProc));

    //obiectul a fost creat dar metoda nu a fost apelată
    t.Start(1000); // metoda este pornită pe noul thread
}

void ThreadProc(object param)
{
    int count = (int) param;
    for (int x = 0; x < count; x++)
    {
        ...
    }
}
```

itacad.ro
you@itacad.ro

Folosirea clasei **Thread**

- ▶ Fiecare thread poate avea una din mai multe stări date de **ThreadState**
- ▶ **Unstarted, Running, WaitSleepJoin, Stopped și Aborted**

Categorie	Metode și proprietăți
COM	SetApartmentState
Securitate	CurrentPrincipal
Localizare	CurrentCulture CurrentUICulture
Starea thread-ului	Priority ThreadState
Regiuni critice	BeginCriticalRegion EndCriticalRegion

11/20/2014
7

Proprietățile unui obiect **Thread** determină modul în care codul este executat. Clasa **Thread** conține metode și proprietăți care afectează modul de execuție al thread-ului.

COM -Fiecare thread poate executa obiecte COM. Un obiect COM rulează în cadrul unui apartament, iar .NET Framework introduce thread-ul automat într-un apartament atunci când este făcut primul apel la un obiect COM. Implicit firul de execuție se alătură la Multi Threaded Apartment(MTA), însă puteți schimba tipul apartamentului apelând metoda SetApartmentState.

Securitate - Fiecare fir de execuție rulează sub un pricipiu de securitate, care poate fi accesat prin intermediul proprietății CurrentPrincipal

Localizare - Fiecare fir de execuție, fir de execuție lucrător sau fir de execuție de interfață cu utilizatorul este localizat. Puteți folosi proprietățile CurrentCulture sau CurrentUICulture pentru a schimba cultura.

Starea thread-ului - Există multe proprietăți prin care putem să obținem informații despre starea firelor de execuție. Spre exemplu, proprietatea Priority indică prioritatea firului de execuție, iar ThreadState indică dacă thread-ul rulează, este suspendat sau s-a terminat.

Regiuni critice - Puteți folosi BeginCriticalRegion și EndCriticalRegion pentru a marca o regiune de cod critică.

Folosirea clasei **Thread**

- ▶ Fiecare thread poate avea una din mai multe stări date de **ThreadState**
- ▶ **Unstarted, Running, WaitSleepJoin, Stopped și Aborted**

Categorie	Metode și proprietăți
Spațiul local de stocare	Spațiul local de stocare este implementat folosind slot-uri de date.
Contexte și domenii	Metode și proprietăți ale clasei Thread
Controlul thread-ului	Metode ale clasei Thread
Afinitatea thread-urilor	BeginThreadAffinity EndThreadAffinity

11/20/2014



8

Spațiul local de stocare - Fiecare fir de execuție poate avea date asociate, numite spațiu local de stocare al thread-ului. În .NET Framework spațiul local de stocare este implementat folosind slot-uri de date. Fiecare fir de execuție are un slot de date, care are același nume, dar conține date diferite pentru fiecare thread.

Contexte și domenii - Fiecare thread se execută într-un domeniu de aplicație în cadrul unui context. Clasa Thread furnizează metode și proprietăți pentru accesul domeniului și contextului curent.

Controlul thread-ului - Fiecare thread poate fi controlat prin intermediul metodelor clasei Thread.

Afinitatea thread-urilor - Puteți folosi BeginThreadAffinity și EndThreadAffinity pentru a vă asigura că execuția codului este realizată doar de un anumit thread.

 	Obiecte de sincronizare a thread-urilor	
	Obiect	Descriere
	EventWaitHandle	Un fir de execuție poate semnala unul sau mai multe fire de execuție care așteaptă.
	Mutex	Un singur fir de execuție deține controlul asupra obiectului protejat de mutex.
	Semaphore	Mai multe fire de execuție dețin controlul asupra obiectului protejat de semafor.
	Monitor	Furnizează un lock pentru protejarea codului.
	ReaderWriterLock	Un fir de execuție poate scrie date, mai multe fire de execuție pot citi datele.
11/20/2014		9

Obiectele de sincronizare sunt folosite pentru a îndeplini două scopuri:

Comunicarea între thread-uri. Obiectul de sincronizare este folosit de unul dintre firele de execuție pentru a indica unui alt thread că poate realiza un anumit task.

Accesul sigur la date. Sincronizarea obiectelor este folosită astfel încât un singur thread să acceseze cod și date la un moment dat.

.NET Framework furnizează clasele **EventWaitHandle**, **Mutex** și **Semaphore**, care înglobează obiectele de sincronizare Windows event, mutex și semaphore. În plus, .NET Framework furnizează obiecte mai complexe de sincronizare precum **Monitor** și **ReaderWriterLock** pentru a furniza acces la cod sensibil la thread-uri.

Cod si date sensibile firelor de execuție

Trebuie acordată multă atenție datelor din cod, care sunt accesate de mai multe thread-uri. Dacă datele pot fi scrise, înseamnă că în timp ce un thread scrie datele, un alt thread poate accesa această valoare, deci thread-ul care citește datele, poate citi valori parțial actualizate. De asemenea, orice cod care accesează date partajate, sunt în mod similar sensibile la thread-uri(ex.: clase colecție). Un obiect colecție poate returna un

enumerator pentru a da acces la obiectele din colecție; deci, nu ar trebui eliminate elemente din colecție atunci când un enumerator este folosit.

Puteți folosi sincronizarea thread-urilor pentru a proteja datele și codul sensibil de accesul multithread.

Obiecte de sincronizare a thread-urilor

WaitHandle



Clasele EventWaitHandle, Mutex și Semaphore sunt derivate din clasa WaitHandle.

Toate obiectele WaitHandle au o caracteristică importantă: sunt fie setate(semnalate), fie resetate.

Metode:

WaitOne - blocheaza thread-ul apelant până când obiectul devine semnalat

WaitAny și **WaitAll** – metode statice, primesc ca parametru un vector de obiecte WaitHandle; blochează thread-ul apelant până când unul sau toate obiectele sunt semnalate



Obiecte de sincronizare a thread-urilor

- **ManualResetEvent**
- **AutoResetEvent**

11/20/2014 11

Spre exemplu, dacă doriți să creați mai multe thread-uri pentru a executa un task, și doriți ca toate thread-urile să pornească în același moment, puteți să sincronizați task-urile folosind un obiect **ManualResetEvent**.

```
void MultipleThreads()  
{  
    ManualResetEvent mre = new  
    ManualResetEvent(false);  
    // Neamnat  
    for(int i = 0; i < 10; ++i)  
    {  
        new Thread  
        (new ParameterizedThreadStart(ThreadProc)).Start(mre);  
    }  
}
```



```
    }  
    // Toate thread-urile sunt blocate  
    mre.Set();    // Le pornim pe toate  
}  
  
void ThreadProc(object obj)  
{  
    waitHandle wait = obj as waitHandle;  
    wait.WaitOne();  
    // Blocare până când obiectul de  
    // sincronizare este semnalat  
    ...  
}
```

Obiectul este partajat între thread-uri și este semnalat faptul că toate thread-urile pot să pornească execuția. Când acest eveniment este semnalat prin apelarea metodei **Set**, toate thread-urile sunt eliberate. Dacă folosiți **AutoResetEvent** în loc de **ManualResetEvent**, atunci un singur thread este eliberat pentru fiecare apel al metodei **Set**.

Dacă doriți să protejați codul și datele de acces de către thread-uri multiple, ar trebui să folosiți un mutex, precum în exemplele următoare:


```
Mutex mutex;
Object protectedObject;
void dowork()
{
    mutex = new Mutex(false); //Acest thread nu
    detine mutex-ul
    for(int i = 0; i < 10; ++i)
    {
        new Thread(new
ThreadStart(ThreadProc)).Start();
    }
}
void ThreadProc()
{
    ...
    mutex.WaitOne(); // Obtine mutexul
    DoSomethingWithObject(protectedObject);
    mutex.Release(); //Elibereaza mutexul
    ...
}
```

Mutexul este folosit pentru a proteja accesul la câmpul **protectedObject**. Metoda **DoSomethingWithObject** modifică valoarea obiectului, deci un singur thread poate apela această metodă la un anumit moment dat. La creare acesta este nesemnalat, iar primul thread care apelează metoda **WaitOne** va deține mutexul. Dacă un alt thread va apela metoda **WaitOne** acesta se va bloca. După ce thread-ul care deține mutexul și-a terminat task-urile, eliberează mutexul și un alt thread poate apela metoda **WaitOne** pentru a prelua controlul asupra mutexului.



Obiecte de sincronizare a thread-urilor

➤ Clasa Monitor

- obiecte corupte

Exemplu:

```
void One()
{
    lock(this);
    {
        // Doar One sau Two pot fi apelate, dar nu amândouă simultan
    }
}

void Two()
{
    lock(this);
    {
        // Doar One sau Two pot fi apelate, dar nu amândouă simultan
    }
}
```

11/20/2014 12

Locking Access

În exemplul anterior s-a prezentat o modalitate de a proteja codul astfel încât un singur thread să îl acceseze la un anumit moment. .NET Framework furnizează clase precum **Monitor** cu o funcționalitate similară și cu flexibilitate mai mare. Clasa **Monitor** poate sincroniza pe baza oricărui obiect, dacă apeleți metoda **Enter** înainte de codul critic și **Exit** imediat după acesta. Trebuie să dați ca parametru la ambele metode un obiect care va fi folosit pentru sincronizare.

Puteți obține obiectul Type pentru o clasă prin apelarea metodei GetType. Există un singur obiect Type pentru o clasă într-un domeniu de aplicație; deci, dacă folosiți acest obiect pentru clasa curentă, ca obiect de sincronizare, atunci codul protejat de clasa Monitor se poate executa pe o singură instanță a clasei la un moment dat. Altfel dacă nu apeleți metoda **Exit** puteți bloca toate thread-urile.



```
void one()
```

```
{
    lock(this);
    {
        // Doar One sau Two pot fi apelate, dar nu amândouă simultan
    }
}

void Two()
{
    lock(this);
    {
        // Doar One sau Two pot fi apelate, dar nu amândouă simultan
    }
}
```

Clasa **Monitor** (folosită de instrucțiunea lock) este bună pentru protejarea codului însă pot apărea probleme. Este importantă protejarea datelor deoarece dacă două thread-uri încearcă să actualizeze același obiect, în același moment, obiectul respectiv poate deveni corupt. O operație de citire este sigură deoarece nu se încearcă modificarea obiectului. Puteți folosi instrucțiunea lock pentru a restricționa scrierea, însă o abordare mai flexibilă este clasa **ReaderWriterLock**.

Clasa **ReaderWriterLock** fie permite unui thread să modifice un obiect, fie permite mai multor thread-uri să citească obiectul respectiv. Metoda **AcquireWriteLock** blochează thread-ul atât timp cât există lock-uri de citire sau scriere.



Obiecte de sincronizare a thread-urilor

➤ **Metode:**

- **ReleaseWriterLock**
- **AcquireWriterLock**
- **ReleaseReaderLock**
- **AcquireReaderLock**

Exemplu - interschimbare referințe:

```
static ReaderWriterLock rwl = new ReaderWriterLock();
void Swap(ref object ref1, ref object ref2)
{
    rwl.AcquireReaderLock(1000);
    object temp = ref1;
    object temp2 = ref2;
    rwl.ReleaseReaderLock();
    rwl.AcquireWriterLock(1000);
    ref1 = temp2;
    ref2 = temp;
    rwl.ReleaseWriterLock();
}
```

11/20/2014 13

Când toate lock-urile sunt eliberate, thread-ul curent este eliberat și pune un lock de scriere asupra obiectului până când este apelată metoda **ReleaseWriterLock**.

Un thread apelează metoda **AcquireReaderLock** pentru a obține un lock de citire.


Această metodă blochează thread-ul doar dacă există un lock de scriere. Dacă nu există lock-uri de scriere, thread-ul pune un lock de citire până când este apelată metoda **ReleaseReaderLock**.

În următorul exemplu, metoda interschimbă referințele celor două obiecte:

```
static ReaderWriterLock rwl = new ReaderWriterLock();
```

```
void Swap(ref object ref1, ref object ref2)
{
    rwl.AcquireReaderLock(1000);
    object temp  = ref1;
    object temp2 = ref2;
    rwl.ReleaseReaderLock();
    rwl.AcquireWriterLock(1000);
    ref1 = temp2;
    ref2 = temp;
    rwl.ReleaseWriterLock();
}
```

Sunt obținute lock-uri de citire înainte de citirea obiectelor și stocarea valorilor în variabile temporare. În acest timp, **Swap** poate fi apelată de alte thread-uri, care pot citi valorile din aceste obiecte. La scrierea valorilor este obținut un lock de scriere , în acest moment un singur thread are acces la date.



Clasa **ThreadPool**

- ▶ Este o clasă statică
- ▶ Menține o mulțime de thread-uri generice lucrătoare
- ▶ Apelați metoda **QueueUserWorkItem** și trimiteți un delegat **WaitCallback** procedurii thread-ului
 - ▶ Thread disponibil → procedura este executată
 - ▶ Nu sunt thread-uri disponibile → cererea este pusă într-o coadă de așteptare


•SetMaxThreads

•SetMinThreads

•QueueUserWorkItem

•GetMaxThreads

•GetMinThreads



11/20/2014 14

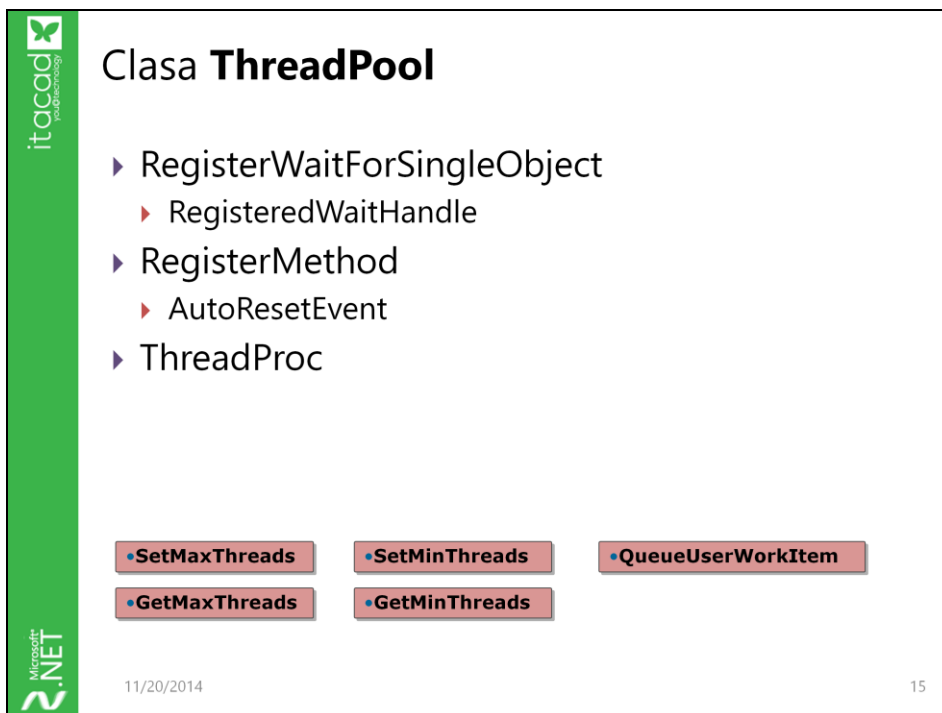
Fiecare proces generează o mulțime de fire de execuție numită thread pool. Această mulțime este folosită de către apelurile asincrone și de către alte clase din .NET Framework. Puteți folosi thread-uri din această mulțime prin intermediul clasei **ThreadPool**.

Folosirea clasei **ThreadPool**

Clasa **ThreadPool** este o clasă statică în care fiecare membru din clasă este static. Puteți folosi metodele **GetMinThreads** și **GetMaxThreads** pentru a obține numărul minim și maxim de thread-uri care se pot afla la un moment dat în thread pool. Pentru a porni execuția pe un fir de execuție thread pool, creați un delegat **WaitCallback** către metoda pe care doriți să o apelați, delegat pe care îl dați ca parametru metodei **QueueUserWorkItem**. Așa precum numele sugerează, delegatul este pus într-o coadă de așteptare, de unde va fi preluat de un fir de execuție din thread pool. Firele de execuție thread pool sunt thread-uri background; astfel dacă mai sunt fire de execuție thread pool care lucrează, acest lucru nu împiedică procesul să se închidă. Următorul cod ilustrează modul de utilizare a clasei **ThreadPool**.

```
void CallMethod()
```

```
{  
    ThreadPool.QueueUserWorkItem(new  
waitCallback(ThreadProc), 1000);  
}  
void ThreadProc(object obj)  
{  
    int i = (int)obj;  
    ...  
}
```

The slide is titled "Clasa **ThreadPool**". It features a green vertical bar on the left with the "itacad" logo and "you@academy" text. The main content lists several methods and events:

- ▶ RegisterWaitForSingleObject
 - ▶ RegisteredWaitHandle
- ▶ RegisterMethod
 - ▶ AutoResetEvent
- ▶ ThreadProc

Below the list, there are five buttons arranged in two rows:

- SetMaxThreads
- SetMinThreads
- QueueUserWorkItem
- GetMaxThreads
- GetMinThreads

At the bottom left, there is a "Microsoft .NET" logo. The date "11/20/2014" is at the bottom center, and the number "15" is at the bottom right.

În plus, clasa **ThreadPool** furnizează o metodă numită **RegisterWaitForSingleObject** pe care o apelezi folosind un delegat, un timeout și un obiect **WaitHandle**. Metoda apelează delegatul atunci când obiectul de sincronizare este semnalat sau dacă s-a terminat perioada timeout.

Metoda **RegisterWaitForSingleObject** returnează un obiect **RegisteredWaitHandle**, care are o metodă numită **Unregistered** pe care o poți apela pentru a anula operația de așteptare a înregistrării. Următoarele exemple ilustrează modul de folosire a metodei **RegisterWaitForSingleObject**:

```
AutoResetEvent are = new AutoResetEvent(false);
void RegisterMethod()
{
    ThreadPool.RegisterWaitForSingleObject(
        are, new WaitOrTimerCallback(ThreadProc), null, -1, true);
}
void CallMethod()
{

```

```
are.Set();  
}  
void ThreadProc(object obj, bool timeout)  
{  
    ...  
}
```

Metoda **RegisterMethod** apelează metoda **RegisterWaitForSingleObject** care folosește obiectul de sincronizare **AutoResetEvent** și un delegat către metoda apelată când obiectul de sincronizare este semnalat. Al treilea parametru este o referință către un obiect care va fi dat mai departe ca parametru metodei apelate. Al patrulea parametru este o valoare de timeout.

Prin specificarea -1, este indicat faptul că timeout nu se va întâmpla niciodată. Ultimul parametru are valoarea true ceea ce indică faptul că metoda este apelată o singură dată.

Metoda **ThreadProc** are doi parametri. Primul sugerează starea obiectului, dat prin intermediul metodei **RegisterWaitForSingleObject**. Al doilea parametru indică dacă metoda a fost apelată fie deoarece obiectul de sincronizare a fost semnalat, fie deoarece a expirat perioada de timeout. Metoda **RegisterWaitForSingleObject** reprezintă o modalitate de a apela o metodă în mod repetat.

Folosirea **Timer**-elor

- ▶ Folosește un fir de execuție **ThreadPool** pentru a realiza apeluri repetate asupra unei funcții de utilizator
- ▶ Folosește un constructor pentru a porni timer-ul
 - ▶ Delegate **TimerCallback**
 - ▶ Valoare de delay înainte de pornirea timer-ului
 - ▶ Valoare pentru intervalul dintre apelurile consecutive
 - ▶ Obiect de stare

11/20/2014

16

Folosiți clasa **Timer** dacă doriți ca o metodă să fie apelată în mod repetat. Clasa **Timer** apelează metoda respectivă pe un fir de execuție thread pool.

Folosirea clasei **Timer**

Următoarele exemple de cod prezintă modul de folosire a clasei **Timer**.

```
void callTimer()
{
    Timer timer = new Timer(new
TimerCallback(ThreadProc), null, 0, 1000);
    //Metoda ThreadProc este apelată o dată la fiecare secundă
    Console.WriteLine("Press ENTER to end timer");
    Console.ReadLine();
    timer.Change(Timeout.Infinite, Timeout.Infinite);
    Console.WriteLine("Press ENTER to exit");
    Console.ReadLine();
}
```

```
{  
void ThreadProc(object obj)  
{  
    ...  
}
```

Constructorul are patru parametrii. Primul este un delegat către metoda apelată. Al doilea parametru este un obiect de stare, ce este preluat de către procedura thread-ului atunci când aceasta este apelată. Ultimii doi parametrii sunt o întârziere inițială și intervalul de timp între apelurile procedurii thread-urilor.

Realizarea apelurilor asincrone

► Delegat

- Compilatorul creează metode care vor fi apelate asincron pe un fir de execuție thread pool
- Nu puteți opri o metodă după ce a fost apelată
- Metodele delegat returnează o referință **IAsyncResult**

► IAsyncResult

- IsCompleted
- AsyncWaitHandle
- AsyncState


Atunci când declarați un delegat în codul dumneavoastră, compilatorul generează metode pe baza clasei delegat pentru a vă permite să apălați în mod asincron metoda delegat. Compilatorul creează un thread prin care porniți apelul metodei, iar această metodă apelează delegatul prin intermediul unui fir de execuție thread pool. De asemenea o metodă care face curățenie după ce apelul s-a terminat și returnează parametrii de output, dacă aceștia există.

Apelarea metodelor în mod asincron

Apelarea asincronă a unei metode este simplă. Creați delegatul pentru metodă și compilatorul face restul. Compilatorul creează două metode. Prima este numită **BeginInvoke**; aceasta prezintă parametrii în și in/out ai metodei originale și returnează o referință către o interfață **IAsyncResult**. Această metodă este folosită pentru a porni metoda inițială pe un fir de execuție thread pool, iar referința **IAsyncResult** vă permite să gestionați obiectul cu apel asincron. După oprirea metodei aceasta nu poate fi anulată. Obiectul cu apel asincron indică momentul când apelul s-a terminat. Interfața are două proprietăți **IsCompleted** și **AsyncWaitHandle**.

Proprietatea **IsCompleted** este o valoare booleană care este setată la valoarea true în

momentul în care apelul s-a terminat. Proprietatea **AsyncWaitHandle** este semnalată în momentul în care metoda s-a terminat. A doua metodă generată de compilator este numită **EndInvoke** și prezintă parametrii out și in/out ai metodei originale; ar trebui să apelați această metodă pentru a obține rezultatele și pentru a elibera resursele folosite de apelul asincron. Dacă sunt ridicate excepții în timpul execuției metodei, obiectul excepție este ridicat la apelarea metodei **EndInvoke**.



Realizarea apelurilor asincrone

Exemplu:

```

delegate int AddDelegate(int l, int r);
class Data
{
    public int data;
    public ManualResetEvent mre = new ManualResetEvent(false);
}
int CallAsync(int l, int r)
{
    AddDelegate add = new AddDelegate(Add);
    Data d = new Data();
    IAsyncResult ar = add.BeginInvoke(l, r, new AsyncCallback(Callback), d);
    d.mre.WaitOne();
    return d.data;
}

```

11/20/2014 18

Umătorul exemplu prezintă realizarea unui apel asincron:

```

delegate int AddDelegate(int l, int r);
// Compilatorul genereaza:
// IAsyncResult BeginInvoke(int l, int r, AsyncCallback cb, object
obj);
// int EndInvoke(IAsyncResult ar);
int CallAsync(int l, int r)
{
    AddDelegate add = new AddDelegate(Add);
    IAsyncResult ar = add.BeginInvoke(l, r, null, null);
    //daca metoda nu s-a terminat asteapta 10 ms si reincearca
    while(!ar.IsCompleted) Thread.Sleep(10);
    return add.EndInvoke(ar);
}
int Add(int l, int r)

```

```
{  
  
    return l+r;  
  
}
```


Mecanismul de verificare dacă metoda s-a terminat este prin verificarea proprietății **IsCompleted**. Dacă aceasta are valoarea false se așteaptă 10 ms. A doua modalitate de a determina dacă metoda s-a terminat implică doi parametri adiționali generați de compilator pentru metoda **BeginInvoke**. Primul parametru este un delegat de tipul **AsyncCallback** și al doilea este un obiect de stare. Când metoda se termină delegatul **AsyncCallback** este apelat pe firul de execuție thread pool și dat referinței **IAsyncResult** și apoi obiectului cu apel asincron. Obiectul de stare este pasat prin intermediul proprietății **IAsyncResult.AsyncState**.

Obiectul cu apel asincron este o instanță a clasei **AsyncResult**. Această clasă are o proprietate numită **AsyncDelegate**, care este o referință către delegatul apelat. Puteți trimite mai departe referința **IAsyncResult** la o referință **AsyncResult** și apoi puteți trimite proprietatea **AsyncDelegate** delegatului apelat de dumneavoastră. Prin intermediul acestui delegat puteți apela metoda **EndInvoke** pentru a termina metoda inițială. Următorul exemplu prezintă modul de apelare asincronă a unei metode.

```
delegate int AddDelegate(int l, int r);  
  
class Data  
{  
    public int data;  
    public ManualResetEvent mre = new ManualResetEvent(false);  
}  
  
int CallAsync(int l, int r)  
{  
    AddDelegate add = new AddDelegate(Add);  
    Data d = new Data();  
    IAsyncResult ar = add.BeginInvoke(l, r, new  
    AsyncCallback(Callback), d);
```



```
        d.mre.waitOne();  
        return d.data;  
    }
```




Realizarea apelurilor asincrone

Exemplu - continuare:

```
void Callback(IAsyncResult iar)
{
    Data d = iar.AsyncState as Data;
    // Obține starea obiectului
    AsyncResult ar = iar as AsyncResult;
    AddDelegate add = ar.AsyncDelegate as AddDelegate;
    d.data = add.EndInvoke(iar);
    d.mre.Set(); // Datele pot fi citite
}

int Add(int l, int r)
{
    return l+r;
}
```



11/20/2014 19



```
void Callback(IAsyncResult iar)
{
    Data d = iar.AsyncState as Data;
    // Obține starea obiectului
    AsyncResult ar = iar as AsyncResult;
    AddDelegate add = ar.AsyncDelegate as
AddDelegate;
    d.data = add.EndInvoke(iar);
    d.mre.Set(); // Datele pot fi citite
}

int Add(int l, int r)
{
```

```
    return 1+r;  
}
```

Rezultatul metodei este un tip valoare. Dacă un tip valoare este trimis mai departe ca un obiect, are loc boxing, astfel încât obiectul este modificat nu valoarea originală.

Codul folosește o clasă separată astfel încât să poată fi trimisă o referință pe post de obiect de stare. La sfârșitul metodei, la runtime este apelată metoda **Callback** pe firul de execuție thread pool. Această metodă obține obiectul cu apel asincron, pentru a obține delegatul și apoi este apelată metoda **EndInvoke**. Metoda **Callback** folosește apoi obiectul de stare pentru a returna rezultatul funcției.



Codul pentru interfața cu utilizatorul

- ▶ **SynchronizationContext**
 - ▶ Clasele derivate apelează codul pe thread-ul adecvat
 - ▶ O metodă este apelată prin intermediul unui obiect **SendOrPostDelegate**
 - ▶ Post apelează metoda în mod asincron
 - ▶ Send apelează metoda în mod sincron
- ▶ **BackgroundWorker**
 - ▶ Furnizează trei evenimente pe care le folosiți pentru a apela un task care rulează în background

11/20/2014 20

Ferestrele și controalele de interfață cu utilizatorul prezintă afinitate pentru thread-uri, astfel orice cod care modifică interfața cu utilizatorul trebuie să fie apelat pe thread-ul care a creat fereastra respectivă.

Este important ca thread-ul de interfață cu utilizatorul să nu fie blocat de execuții lungi de operații, deoarece acest lucru va împiedica actualizarea interfeței cu utilizatorul. .NET Framework furnizează clase care vă permit să realizați apeluri asincrone pe thread-ul corect.



Lucrul cu thread-uri și codul de interfață cu utilizatorul

.NET Framework furnizează o interfață numită **ISynchronizeInvoke**, care vă permite să apelați cod pe thread-ul de interfață cu utilizatorul. Clasa **Control** (clasa de bază pentru toate controalele, inclusiv **Form**) implementează interfața **ISynchronizeInvoke**.

Dacă în codul dumneavoastră aveți un thread lucrător și doriți să actualizați interfața cu utilizatorul folosind acest thread, o modalitate de a face acest lucru este prin apelarea

metodelor interfeței **ISynchronizeInvoke** pe obiectul de interfață cu utilizatorul.

```
Label dateTime;  
delegate void UpdateDelegate(string s);  
void ThreadProc()  
{  
    // Actualizarea etichetei  
    UpdateDelegate del = new UpdateDelegate(UpdateLabel);  
    dateTime.Invoke(del, new  
object[] { DateTime.Now.ToString() });  
}  
void UpdateLabel(string s)  
{  
    dateTime.Text = s;  
}
```



Codul pentru interfața cu utilizatorul

- ▶ **WindowsFormsSynchronizationContext**
- ▶ *Exemplu:*

```
SynchronizationContext ctx;  
  
//Este apelata pe thread-ul de interfata cu utilizatorul  
void StartOperation(object sender, EventArgs e)  
{  
    ctx = SynchronizationContext.Current;  
    // Salvati contextul curent  
    Thread t = new Thread(new ThreadStart(ThreadProc));  
    t.Start();  
}  
  
void UpdateLabel(object state)  
{  
    result.Text = state as string;  
}
```

11/20/2014 21

Problema cu această abordare este dacă doriți să scrieți cod de bibliotecă, nu se cunoaște dacă este apelat codul pentru actualizarea unui obiect de interfață cu utilizatorul.

Clasa SynchronizationContext

Dacă aveți un thread lucrător și doriți ca acest thread să schimbe interfața cu utilizatorul pentru a reflecta progresul acesteia, trebuie să executați codul pe thread-ul de interfață cu utilizatorul. Clasa **SynchronizationContext** furnizează o modalitate de a face acest lucru.

Clasa **SynchronizationContext** este o clasă de bază, iar pentru aplicații Windows Forms .NET Framework furnizează clasa **WindowsFormsSynchronizationContext**. Clasa de bază oferă o proprietate statică numită **Current** care returnează un context potrivit thread-ului curent. Astfel dacă apelați proprietatea **SynchronizationContext.Current** pe un thread de interfață cu utilizatorul veți primi un obiect

WindowsFormsSynchronizationContext. Puteți folosi acest obiect de context pe un alt thread și toate apelurile făcute prin intermediul contextului vor fi executate pe thread-ul inițial. Următorul cod arată modul de utilizare al acestei clase:



```
Label result;
SynchronizationContext ctx;
//Este apelata pe thread-ul de interfata cu utilizatorul

void StartOperation(object sender, EventArgs e)
{
    ctx = SynchronizationContext.Current;
    // Salvati contextul curent
    Thread t = new Thread(new ThreadStart(ThreadProc));
    t.Start();
}

void ThreadProc()
{
    SendOrPostCallback callback = new
    SendOrPostCallback(UpdateLabel);
    for(int i = 0; i < 100; ++i)
    {
        Thread.Sleep(10);
        //Lucram cu interfata cu utilizatorul in
        contextul de interfata cu utilizatorul
        ctx.Post(callback, string.Format("{0}%
        completed", i));
    }
}
```

```
        ctx.Send(callback, "Completed"); // 0 alta  
modalitate  
}
```

```
void UpdateLabel(object state)  
{  
    result.Text = state as string;  
}
```

Codul pentru interfața cu utilizatorul

- ▶ **WindowsFormsSynchronizationContext**
- ▶ *Exemplu - continuare:*

```
void ThreadProc()
{
    SynchronizationContext callback = new SynchronizationContext(UpdateLabel);
    for(int i = 0; i < 100; ++i)
    {
        Thread.Sleep(10);
        //Lucram cu interfața cu utilizatorul în contextul de interfața cu utilizatorul
        ctx.Post(callback, string.Format("{0}% completed", i));
    }
    ctx.Send(callback, "completed"); // o alta modalitate
}
```

11/20/2014 22

Problema cu această abordare este dacă doriți să scrieți cod de bibliotecă, nu se cunoaște dacă este apelat codul pentru actualizarea unui obiect de interfață cu utilizatorul.

Clasa SynchronizationContext

Dacă aveți un thread lucrător și doriți ca acest thread să schimbe interfața cu utilizatorul pentru a reflecta progresul acesteia, trebuie să executați codul pe thread-ul de interfață cu utilizatorul. Clasa **SynchronizationContext** furnizează o modalitate de a face acest lucru.

Clasa **SynchronizationContext** este o clasă de bază, iar pentru aplicații Windows Forms .NET Framework furnizează clasa **WindowsFormsSynchronizationContext**. Clasa de bază oferă o proprietate statică numită **Current** care returnează un context potrivit thread-ului curent. Astfel dacă apeleți proprietatea **SynchronizationContext.Current** pe un thread de interfață cu utilizatorul veți primi un obiect

WindowsFormsSynchronizationContext. Puteți folosi acest obiect de context pe un alt thread și toate apelurile făcute prin intermediul contextului vor fi executate pe thread-ul inițial. Următorul cod arată modul de utilizare al acestei clase:



```
Label result;
SynchronizationContext ctx;
//Este apelata pe thread-ul de interfata cu utilizatorul

void StartOperation(object sender, EventArgs e)
{
    ctx = SynchronizationContext.Current;
    // Salvati contextul curent
    Thread t = new Thread(new ThreadStart(ThreadProc));
    t.Start();
}

void ThreadProc()
{
    SendOrPostCallback callback = new
    SendOrPostCallback(UpdateLabel);
    for(int i = 0; i < 100; ++i)
    {
        Thread.Sleep(10);
        //Lucram cu interfata cu utilizatorul in
        contextul de interfata cu utilizatorul
        ctx.Post(callback, string.Format("{0}%
        completed", i));
    }
}
```

```
        ctx.Send(callback, "Completed"); // 0 alta  
modalitate  
}
```

```
void UpdateLabel(object state)  
{  
    result.Text = state as string;  
}
```



Codul pentru interfața cu utilizatorul

- ▶ **BackgroundWorker**
 - ▶ **DoWork**
 - ▶ DoWorkEventArgs.Result
 - ▶ **ProgressChanged**
 - ▶ ProgressChangedEventArgs
 - ▶ WorkerReportsProgress
 - ▶ **RunWorkerCompleted**
 - ▶ RunWorkerCompletedEventArgs

11/20/2014 23

Metoda **StartOperation** este apelată pe thread-ul de interfață cu utilizatorul. Această metodă obține contextul curent și apoi pornește firul de execuție lucrător. În timp ce thread-ul lucrător realizează o operație de lungă durată, actualizează în mod periodic interfața cu utilizatorul prin apelul metodei **UpdateLabel**. Pentru a ne asigura că metoda **UpdateLabel** este apelată în contextul curent, este necesară crearea delegatului **SendOrPostCallback** către metodă și apelarea acestuia prin intermediul obiectului context.

Clasa BackgroundWorker

Clasa are trei evenimente pe care le puteți folosi pentru a apela task-urile background și raporta progresul. Acestea sunt descrise în tabelul următor:

În toate cazurile, parametrul sender al metodei handler este o instanță a clasei **BackgroundWorker** care a ridicat evenimentul; puteți folosi acest parametru pentru a obține suport adițional pentru operație.



În exemplele următoare, sunt implementate metode handler pentru un obiect

BackgroundWorker:

DoWork - Metoda handler pentru acest eveniment execută task-ul background. Ar trebui să verificați în mod periodic dacă utilizatorul a cerut operația de anulare, dacă da ieșiți din metoda handler. Dacă operația generează un rezultat ar trebui să îl returnați prin intermediul proprietății `DoWorkEventArgs.Result`.

ProgressChanged - Metoda handler este informată de progresul curent al operației prin intermediul parametrului `ProgressChangedEventArgs`. Pentru a răspund eacestui eveniment trebuie setată la valoarea `true` proprietatea `WorkerReportsProgress`

RunWorkerCompleted - Metoda handler este informată dacă operația s-a terminat, a fost anulată sau a fost ridicată o excepție, prin intermediul parametrului `RunWorkerCompletedEventArgs`.



Codul pentru interfața cu utilizatorul

- ▶ **BackgroundWorker**
 - ▶ Implementare metode handler:

```
void StartOperation()
{
    worker = new BackgroundWorker();
    worker.WorkerSupportsCancellation = true;
    worker.DoWork += new DoWorkEventHandler(LongOperation);
    worker.WorkerReportsProgress = true;
    worker.ProgressChanged += new ProgressChangedEventHandler(OnProgress);
    worker.RunWorkerCompleted += new RunWorkerCompletedEventHandler(OnCompleted);
    worker.RunWorkerAsync();
}

void StopOperation()
{
    if(worker != null)
    {
        //Anuleaza operatia
        worker.CancelAsync();
    }
}
```

11/20/2014 24

```
BackgroundWorker worker = null;
```

```
Label progress;
```

```
void StartOperation()
```

```
{
```

```
    worker = new BackgroundWorker();
```

```
    worker.WorkerSupportsCancellation = true;
```

```
    worker.DoWork += new  
DoWorkEventHandler(LongOperation);
```

```
    worker.WorkerReportsProgress = true;
```

```
    worker.ProgressChanged +=
```

```
new ProgressChangedEventHandler(OnProgress);
```

```
    worker.RunWorkerCompleted +=
```

```
new RunWorkerCompletedEventHandler(OnCompleted);
```

```
    worker.RunWorkerAsync();
```

```
}  
void StopOperation()  
{  
    if(worker != null)  
    {  
        //Anuleaza operatia  
        worker.CancelAsync();  
    }  
}  
void LongOperation(object sender, DoWorkEventHandler  
e)  
{  
    BackgroundWorker thisworker = sender as  
BackgroundWorker;  
    for(int i = 0; i < 100; ++i)  
    {  
        //Verificam daca operatia a fost  
anulata  
        if(thisworker.CancellationPending)  
        {  
            e.Cancel = true;  
            return;  
        }  
        Thread.Sleep(100);  
    }  
}
```

```
        thisWorker.ReportProgress(x);
        //Indica nivelul de lucru realizat
    }
    e.Result = 100;
}

void OnProgress(object sender,
ProgressChangedEventArgs e)
{
    progress.Text = String.Format("{0}%
completed", e.ProgressPercentage);
}

void OnCompleted(object sender,
RunWorkerCompletedEventArgs e)
{
    if(!e.Cancelled && e.Error == null)
    {
        progress.Text =
String.Format("Result {0}", e.Result);
    }
    worker.Dispose();
    worker = null;
}
```


itacad
you@academy

Codul pentru interfața cu utilizatorul

- ▶ **BackgroundWorker**
 - ▶ Implementare metode handler - continuare:

```

void OnProgress(object sender, ProgressChangedEventArgs e)
{
    progress.Text = String.Format("{0}% completed", e.ProgressPercentage);
}
void OnCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    if(!e.Cancelled && e.Error == null)
    {
        progress.Text = String.Format("Result {0}", e.Result);
    }
    worker.Dispose();
    worker = null;
}
void LongOperation(object sender, DoWorkEventHandler e)
{
    ...
}

```

11/20/2014
25

```

BackgroundWorker worker = null;
Label progress;
void StartOperation()
{
    worker = new BackgroundWorker();
    worker.WorkerSupportsCancellation = true;
    worker.DoWork += new
DoWorkEventHandler(LongOperation);
    worker.WorkerReportsProgress = true;
    worker.ProgressChanged +=
new ProgressChangedEventArgs(OnProgress);
    worker.RunWorkerCompleted +=
new RunWorkerCompletedEventHandler(OnCompleted);
    worker.RunWorkerAsync();
}

```

```
}  
void StopOperation()  
{  
    if(worker != null)  
    {  
        //Anuleaza operatia  
        worker.CancelAsync();  
    }  
}  
void LongOperation(object sender, DoWorkEventHandler  
e)  
{  
    BackgroundWorker thisworker = sender as  
BackgroundWorker;  
    for(int i = 0; i < 100; ++i)  
    {  
        //Verificam daca operatia a fost  
anulata  
        if(thisworker.CancellationPending)  
        {  
            e.Cancel = true;  
            return;  
        }  
        Thread.Sleep(100);  
    }  
}
```

```
        thisWorker.ReportProgress(x);  
        //Indica nivelul de lucru realizat  
    }  
    e.Result = 100;  
}  
  
void OnProgress(object sender,  
ProgressChangedEventArgs e)  
{  
    progress.Text = String.Format("{0}%  
completed", e.ProgressPercentage);  
}  
  
void OnCompleted(object sender,  
RunWorkerCompletedEventArgs e)  
{  
    if(!e.Cancelled && e.Error == null)  
    {  
        progress.Text =  
String.Format("Result {0}", e.Result);  
    }  
    worker.Dispose();  
    worker = null;  
}
```

Sumar

- ▶ Crearea aplicațiilor multithread
- ▶ Elemente de sincronizare
- ▶ Apeluri asincrone
- ▶ Contexte de sincronizare

