УДК 004.432.2, 004.422.83

ПОРІВНЯННЯ МОЖЛИВОСТЕЙ Task.Run ТА BackgroundWorker ДЛЯ ПОБУДОВИ БАГАТОПОТОКОВИХ ПРОГРАМ МОВОЮ C#

Світлана Ярошко, Сергій Ярошко

Львівський національний університет імені Івана Франка,
вул. Університетська, 1, м. Львів, 79000, e-mail: serhiy.yaroshko@lnu.edu.ua

На прикладі побудови застосунку мовою C# виконано порівняння можливостей екземплярів класів *BackgroundWorker* та *Task*. Застосунок графічними засобами демонструє процес впорядкування масиву цілих чисел різними методами сортування. Завдяки використанню засобів рефлексії у програмі користувач може сам обрати методи сортування, випробувати їх на різних наборах вхідних даних. Показано, що метод *Task.Run()* зручніший у використанні. Запуск паралельного потоку відбувається у зрозуміліший спосіб, для отримання інформації про хід сортування використовують строго типізований об'єкт, а для дострокової зупинки потоку – загальний механізм переривання виконання платформи .Net.

Ключові слова: потік виконання, багатопотоковий застосунок, клас Task, зупинка потоку виконання, відображення ходу обчислень, алгоритми сортування, рефлексія.

1. Вступ

У програмі мовою C# окремий потік виконання можна запустити багатьма способами: за допомогою асинхронного виклику делегата, через екземпляр класу *Thread*, через асинхронний шаблон на основі подій (компонент *BackgroundWorker*), через екземпляр класу *Task* [1]. Саме останній радять використовувати розробники платформи .Net. У одній з попередніх публікацій [2] ми детально описали використання *BackgroundWorker* у застосунку *Windows Forms*, що візуалізує процес виконання різних алгоритмів упорядкування послідовності значень, що працюють у окремих потоках. Компонент *BackgroundWorker* був дуже прогресивним на момент створення, проте сьогодні з'явилися досконаліші технології. Ми порівняємо його можливості та способи використання з можливостями екземпляра *Task*, зокрема, отриманого методом *Task.Run()* на прикладі такого ж застосунку. Подібне порівняння на умовному прикладі можна знайти в [3]. Особливості використання *Task* для розпаралелення розв'язування задачі комівояжера генетичним алгоритмом ми описали в [4].

Нагадаємо, що застосунок візуалізації алгоритмів упорядкування запускає на виконання у паралельних потоках декілька методів сортування масиву цілих чисел, надає користувачеві можливість зупинити кожен з методів, якщо виконання триває занадто довго. Кожен з масивів зображається на окремій графічній панелі набором відрізків відповідної довжини. Потоки сортування повідомляють застосунок про обміни елементів сортованого масиву, що спричиняє перемальовування відповідної пари відрізків. Кожен потік повідомляє про своє завершення. Отож, до інструменту створення потоку висувають такі вимоги:

- зручний запуск методу впорядкування;
- отримання інформації про хід виконання та можливість відобразити її у вікні застосунку;

- можливість зреагувати в потоці застосунку на завершення потоку сортування;
- можливість зупинити потік достроково.

Порівняємо засоби для задоволення цих вимог, які надають *BackgroundWorker* та *Task.Run()*.

2. БАЗОВА ВЗАЄМОДІЯ

Базова взаємодія передбачає створення потоку, запуск його на виконання і отримання результату. Для використання *BackgroundWorker* потрібно виконати кілька кроків: створити екземпляр класу, налаштувати опрацювання події *DoWork* та викликати метод *RunWorkerAsync*. Схематично це можна зобразити так.

```
static void MainThread()
{
   int[] arrayToSort = GetArray();
   BackgroundWorker bgw = new BackgroundWorker();
   bgw.DoWork += SortingWork; // метод для виконання в потоці
   bgw.RunWorkerAsync(arrayToSort);  // запуск на виконання
}

// метод повинен мати спеціальний інтерфейс
static void SortingWork(object sender, DoWorkEventArgs e)
{
   // метод сортування отримує масив з аргументу події
   OppositeSort((int[])e.Argument);
}

// метод сортування масиву зустрічним обміном
static void OppositeSort(int[] array) { ... }
```

Бачимо, що виклик методу сортування розташовано досить далеко від місця створення екземпляра *BackgroundWorker* та запуску потоку на виконання, а передавання параметрів – неочевидне, що, безумовно, утруднює розуміння тексту програми. Запис можна дещо скоротити за допомогою використання лямбда-виразів, які ми використовуватимемо надалі.

Те саме завдання за допомогою *Task* можна виконати простіше

```
static void MainThread()
{
   int[] arrayToSort = GetArray();
   // створити і запустити потік одним рядком
   Task.Run(() => { OppositeSort(arrayToSort); });
}
// метод сортування масиву зустрічним обміном
static void OppositeSort(int[] array) { ... }
```

Зазвичай метод сортування не повертає ніяких значень, проте, він міг би це робити. Наприклад, під час сортування можна лічити виконані обміни елементів масиву і повертати обчислену кількість як результат. У цьому випадку інтерфейс методу сортування зазнає незначних змін: його тип стане *int* замість *void*. Складніше буде налаштувати отримання результату за допомогою *BackgroundWorker*. Метод, що

працюватиме в потоці, повинен помістити результат у властивість *Result* аргументу події *DoWork*. Доведеться також визначити опрацювання події завершення виконання і прочитати властивість *Result* її аргументу. Додаткова складність полягає в тому, що обидві властивості *Result* мають тип *object*, і користувачеві потрібно точно знати тип результату та застосувати операцію приведення типу.

```
static int MainThread()
{
   int[] arrayToSort = GetArray();
   BackgroundWorker bgw = new BackgroundWorker();

   // вміст потоку можна задати лямбда-виразом
   // метод сортування отримує масив з аргументу події,
   // а результат поміщає в спеціальну властивість цього аргументу
   bgw.DoWork += (sender, doWorkEventArg) =>
                 { OppositeSort((int[])doWorkEventArg.Argument); };

   // окремий лямбда-вираз для отримання результату
   bgw.RunWorkerComleted += (sender, completedEventArg) =>
   {
      int result = (int)completedEventArg.Result;
      Console.WriteLine("Count of exchanges is " + result);
   };

   bgw.RunWorkerAsync(arrayToSort);  // запуск на виконання
}

// метод впорядкування масиву зустрічним обміном
static int OppositeSort(int[] array) { ... }
```

Щоб передати знайдену кількість обмінів у головну програму, доведеться докласти додаткових зусиль.

У випадку використання *Task* доповнення до наведеного вище коду будуть мінімальними. *Task.Run* викликають у операторі *await*. Функція, передана в нього, повертає результат звичайним способом. Тип результату зберігається. До того ж, строго типізований результат повернеться одразу в головну програму.

```
async static void MainThread()
{
   int[] arrayToSort = GetArray();
   // створити, запустити потік і отримати результат одним рядком
   int result = await Task.Run(() =>
                  { return OppositeSort(arrayToSort); });
}
// метод впорядкування масиву зустрічним обміном
static int OppositeSort(int[] array) { ... }
```

3. ЗАВЕРШЕННЯ ПОТОКУ ВИКОНАННЯ

Повернімося до того сценарію, коли метод сортування має тип *void* і нічого не повертає. Зазвичай після завершення потоку виконання потрібно виконати насту-

пні кроки. Наприклад, у нашому застосунку після завершення потоку сортування потрібно зменшити лічильник активних потоків і приховати кнопку примусового завершення потоку.

Ми вже знайомі з опрацюванням події *BackgroundWorker.RunWorkerComleted*. Саме тут, окрім отримання результату, описують наступні кроки.

```
static int MainThread()
{
   int[] arrayToSort = GetArray();
   BackgroundWorker bgw = new BackgroundWorker();
   bgw.DoWork += (sender, doWorkEventArg) =>
                 { OppositeSort((int[])doWorkEventArg.Argument); };
   bgw.RunWorkerComleted += (sender, completedEventArg) =>
                 { ExecuteNextSteps(); };
   bgw.RunWorkerAsync(arrayToSort);  // запуск на виконання
}
static void OppositeSort(int[] array) { ... }
```

Продовження виконання *task.Run()* можна задати двома різними способами. Якщо виклик виконано в операторі *await*, то основний потік дочекається завершення потоку сортування, а тоді виконає наступні в інструкції.

```
async static void MainThread()
{
   int[] arrayToSort = GetArray();
   await Task.Run(() => { OppositeSort(arrayToSort); });
   ExecuteNextSteps();
}
static void OppositeSort(int[] array) { ... }
```

Метод *ExecuteNextSteps* буде виконано в потоці *MainThread*. Інший спосіб використовує додаткові можливості класу *Task*: завдання можна об'єднувати в ланцюжки довільної довжини методом *ContinueWith*.

```
async static void MainThread()
{
   int[] arrayToSort = GetArray();
   await Task.Run(() => { OppositeSort(arrayToSort); }).
              ContinueWith(task => { ExecuteNextSteps(); });
}
static void OppositeSort(int[] array) { ... }
```

Тепер для виконання *ExecuteNextSteps* буде створено новий екземпляр *Task* і новий потік.

Якщо після завершення потоку обчислень потрібен доступ до елементів графічного інтерфейсу користувача, як у нашому застосунку, то перевагу треба віддавати першому способу, оскільки візуальні компоненти можна змінювати тільки в тому потоці, в якому вони були створені.

4. ПОВІДОМЛЕННЯ ПРО ХІД ОБЧИСЛЕНЬ

Головним завданням застосунку візуалізації алгоритмів упорядкування є відображення графічними засобами процесу сортування масиву. Для цього потрібно отримувати інформацію про обміни елементів масиву, виконані методом сортування. І *BackgroundWorker*, і *Task* надають такі можливості.

Для налаштування екземпляру *BackgroundWorkwer* потрібно виконати декілька кроків: задати його властивість *WorkerSupportsProgress* = *true*, задати опрацювання події *ProgressChanged* і помістити в метод сортування виклики *BackgroundWorker.ReportProgress(anInt)* або *BackgroundWorker.ReportProgress(anInt, anObject)*. Першим параметром має бути ціле число, другим – довільний об'єкт. Саме цими значеннями потік інформує про хід виконання. Метод опрацювання події *ProgressChanged* може отримати доступ до них відповідно через властивості *ProgressPercentage* та *UserState* аргументу події. Якщо одного цілого числа не достатньо для інформування, використовують другий варіант і властивість *UserState*. Вона має тип *object*, тому потребує приведення типу в головній програмі.

Зрозуміло, що метод сортування потребує доступу до екземпляра *BackgroundWorker*, що керує його потоком виконання. Для цього доведеться змінити інтерфейс методу і передавати в нього окрім масиву ще й екземпляр *BackgroundWorker*.

```
static int MainThread()
{
   int[] arrayToSort = GetArray();
   BackgroundWorker bgw = new BackgroundWorker();
   bgw.DoWork += (sender, doWorkEventArg) => { OppositeSort(
     (int[])doWorkEventArg.Argument, sender as BackgroundWorker); };

   // налаштування підтримки інформування про хід обчислень
   bgw.WorkerSupportsProgress = true;
   bgw.ProgressChanged += (sender, progressChangedEventArg) =>
             { VisualiseProgress(
              (Tuple<int,int>)progressChangedEventArg.UserState); };

   bgw.RunWorkerAsync(arrayToSort);  // запуск на виконання
}
static void OppositeSort(int[] array, BackgroundWorker worker)
{ ... Swap(array[i], array[j]);
      // перший параметр - зайвий, але мусимо його задати
      worker.ReportProgress(0, Tuple.Create(i, j)); ... }
```

Тепер екземпляр *BackgroundWorker* виконує подвійне завдання: підтримує виконання потоку та є безпосереднім учасником методу сортування, який сам і підтримує.

При використанні *Task* ролі розділено: для інформування про хід обчислень використовують окрему сутність, екземпляр класу *Progress*$<T>$, де T – довільний тип, а значення цього типу є змістом повідомлення. Як і в попередньому випадку, метод сортування повинен отримати додатковий параметр, екземпляр *Progress*$<T>$, і у належному місці викликати його метод *Report(a*T*)*.

```
async static void MainThread()
```

```
{
   int[] arrayToSort = GetArray();
   var progress = new Progress<Tuple<int,int>>(value =>
                                   { VisualiseProgress(value); });
   await Task.Run(() => { OppositeSort(arrayToSort, progress); });
}
static void OppositeSort(int[] array, Progress<Tuple<int,int>> p)
{ ... Swap(array[i], array[j]);
      p.Report(Tuple.Create(i, j)); ... }
```

Перевагою цього способу є розділення ролей об'єктів і збереження типу значення, надісланого у повідомленні.

5. ПЕРЕРИВАННЯ ОБЧИСЛЕНЬ

Застосунку, що виконує довготривалі обчислення, потрібно мати змогу перервати їх: або через вичерпання відведеного часу, або за бажанням користувача. Як і у випадку інформування про хід обчислень, переривання виконання *BackgroundWorker* потребує декількох додаткових налаштувань. Перш за все, потрібно задати значення *true* його властивості *WorkerSupportCancellation*. Щоб зупинити обчислення, головна програма повинна викликати метод *worker.CancelAsync()*. Цей метод встановить властивість *worker.CancellationPending* у *true*. Метод сортування повинен мати доступ до *worker*, щоб періодично перевіряти її, та на вимогу завершити виконання, надавши при цьому властивості *DoWorkEventArgs.Cancel* значення *true*. Щоб з'ясувати, чи були обчислення перервані достроково, потрібно перевірити властивість *RunWorkerCompletedEventArgs.Cancel* в методі опрацювання завершення обчислень.

```
static int MainThread()
{
   int[] arrayToSort = GetArray();
   BackgroundWorker bgw = new BackgroundWorker();
   bgw.DoWork += (sender, doWorkEventArg) => { OppositeSort(
     (int[])doWorkEventArg.Argument, sender as BackgroundWorker,
      doWorkEventArg); };

   // налаштування підтримки переривання обчислень
   bgw.WorkerSupportCancellation = true;
   bgw.RunWorkerComleted += (sender, completedEventArg) =>
       { if (completedEventArg.Cancel) InformAboutCancellation();
         ExecuteNextSteps(); };

   bgw.RunWorkerAsync(arrayToSort);  // запуск на виконання
   if (TimeOut()) bgw.CancelAsync(); // переривання виконання
}
static void OppositeSort(int[] array,
                    BackgroundWorker worker, DoWorkEventArgs arg)
{ ... if (worker.CancellationPending)
      { arg.Cancel = true; return; } ... }
```

Бачимо, що до багатьох обов'язків екземпляра *BackgroundWorker* та методу опра-

цювання події *RunWorkerComleted* додалися нові: *worker* повідомляє про вимогу зупинити виконання потоку, а метод – перевіряє, чи потік завершився на вимогу, чи звичайним чином. Окрім перевантаженості обов'язків є ще одна небезпека: можна просто забути перевірити властивість *completedEventArg.Cancel*, і виконання програми піде далі, ніби нічого не трапилося. Так можна пропустити нештатну ситуацію.

Для переривання обчислень у *Task.Run()* використовують загальний механізм переривання виконання, впроваджений у .Net 4.0: створюють екземпляр *CancellationTokenSource* і передають у асинхронний виклик маркер завершення *token*, який отримують з властивості *CancellationTokenSource.Token*. Як і в попередньому випадку, запущений на виконання код повинен періодично перевіряти стан маркера і переривати виконання на вимогу. Для цього є дві можливості. Перша – перевірити властивість *token.IsCancellationRequested* і виконати вихід з методу. Таке завершення, як і у випадку *BackgroundWorkwer*, для головної програми схоже на звичайне завершення обчислень. Про переривання свідчить значення *true* властивості *IsCancellationRequested* маркера переривання – вона зберігається після завершення асинхронного виклику. Друга можливість – викликати метод *token.ThrowIfCancellationRequested()*, який генерує відповідний виняток і таким чином перериває обчислення. Головна програма не має шансів не помітити таке завершення. Адже у цьому випадку асинхронний виклик потрібно виконати в інструкції *try/catch*. Щоб перервати обчислення, головна програма викликає *CancellationTokenSource.Cancel()*.

Застосунок візуалізації алгоритмів сортування надає користувачеві можливість зупинити будь-який алгоритм на власний розсуд. Тому переривання виконання не вважається помилкою, і ми використаємо перший зі згаданих способів.

```
  private CancellationTokenSource tokenSource;
  async static void MainThread()
  {
     int[] arrayToSort = GetArray();
     tokenSource = new CancellationTokenSource();
     await Task.Run(() =>
           { OppositeSort(arrayToSort, tokenSource.Token); });
  }
private void DoCancellation()
  {
     if (tokenSource != null) tokenSource.Cancel();
  }
  static void OppositeSort(int[] array, CancellationToken token)
  { ...
        if (token.IsCancellationRequested) return; ... }
```

Варто зауважити, що екземпляр *CancellationTokenSource* є "одноразовим": нема змоги використати його повторно після переривання, оскільки маркер переривання зберігає вимогу зупинки обчислень. Для повторного запуску того ж методу у *Task.Run()* потрібно створити новий екземпляр *CancellationTokenSource*.

6. АДАПТАЦІЯ МЕТОДУ СОРТУВАННЯ

Для того, щоб метод сортування масиву взаємодіяв із застосунком візуалізації, у нього мають бути додаткові параметри та виклики методів, про які ми вже згаду-

вали. Покажемо на конкретному прикладі, що саме потрібно змінити.

Традиційний метод сортування обмінами можна записати так.

```
public static void OppositeSort(int[] arrayToSort)
{
   for (int i = 0; i < arrayToSort.Length - 1; ++i)
      for (int j = arrayToSort.Length - 1; j > i; --j)
      {
         if (arrayToSort[i] > arrayToSort[j])
         {
            int t = arrayToSort[i];
            arrayToSort[i] = arrayToSort[j];
            arrayToSort[j] = t;
         }
      }
}
```

Адаптований метод залежить від двох додаткових параметрів: екземпляра $Progress<T>$ і маркера зупинки. Стан маркера доцільно перевіряти на кожному кроці вкладеного циклу, оскільки виконання цілого циклу може тривати досить довго. Про кожен виконаний обмін значень елементів масиву метод повідомлятиме через об'єкт *progress*.

Застосунок спроектовано максимально незалежним від бібліотеки методів сортування. Він завантажує її на етапі виконання та знаходить відповідні методи засобами рефлексії. Така архітектура дає змогу доповнювати бібліотеку методів без перебудови застосунку. Для полегшення пошуку методів їх позначено атрибутом *TaskNameAttribute*, спеціально створеним для цієї мети [5].

```
[TaskNameAttribute("Opposite exchange", LocalName = "Зустрічний обмін")]
  public static void OppositeSort(int[] arrayToSort,
            CancellationToken token, IProgress<Tuple<int, int>> progress)
{
   for (int i = 0; i < arrayToSort.Length - 1; ++i)
      for (int j = arrayToSort.Length - 1; j > i; --j)
      {
         // перевірка вимоги зупинити потік
         if (token.IsCancellationRequested)
               return;
         if (arrayToSort[i] > arrayToSort[j])
         {
            int t = arrayToSort[i];
            arrayToSort[i] = arrayToSort[j];
            arrayToSort[j] = t;
            // повідомляємо про обмін
            progress.Report(Tuple.Create(i, j));
         }
      }
}
```

7. АРХІТЕКТУРА ЗАСТОСУНКУ

Застосунок візуалізації збудовано з дотриманням моделі MVC [2, 4], у якій за відображення відповідають візуальні компоненти, в тому числі спеціально сконструйовані, модель постачає масиви цілих чисел, а контролер *SortController* пов'язує компоненти та масиви з *BackgroundWorker* і методами сортування. Як уже зрозуміло, повне налаштування *BackgroundWorker* вимагає чималих зусиль. У згаданому застосунку заради зручності його інкапсульовано в класі *BackgroundSorter*.

Застосуємо такий підхід і зараз: покладемо обов'язок зі створення, налаштування, запуску та зупинки *Task* на екземпляр нового класу *TaskSorter*. Він пов'язуватиме, метод сортування, масив, маркер зупинки та екземпляр *Progress*$<T>$ і запускатиме новий потік виконання. Його обов'язком буде також дострокова зупинка потоку, за потреби, оновлення маркера зупинки, виконання необхідних дій після завершення потоку сортування.

```
public class InTaskSorter : IDisposable
{
   private CancellationTokenSource tokenSource; // джерело маркерів
   private Progress<Tuple<int,int>> progress = null;
   private int[] array  = null;           // і масив, і метод можна
   private MethodInfo sortMethod = null;  //         змінити
   private Action continuation = null;    // завершальні дії

   // більшість полів налаштовують-змінюють після створення
   public InTaskSorter()
   {
      tokenSource = new CancellationTokenSource();
   }
   public void Dispose() { tokenSource.Dispose(); }
   public void SetArray(int[] a) { array = a; }
   public void SetMethod(MethodInfo s) { sortMethod = s; }
   public void SetProgress(Action<Tuple<int,int>> p)
   {
      progress = new Progress<Tuple<int, int>>(p);
   }
   public void SetContinuation(Action c) { continuation = c; }

   public void Stop() { tokenSource.Cancel(); }
   public async void Start()
   {
      if (tokenSource.Token.IsCancellationRequested)
      {  // використаний маркер треба замінити
         tokenSource.Dispose();
         tokenSource = new CancellationTokenSource();
      }
      // створити і запустити нове завдання,
      await Task.Run(() =>
      {
```

```
            sortMethod?.Invoke(null,
               new object[] { array, tokenSource.Token, progress });
         }, tokenSource.Token);
         // дочекатися і виконати завершальні дії
         continuation();
      }
   }
```

Надалі в застосунку *SortController* працюватиме з колекцією *InTaskSorter*'ів замість *BackgroundSorter*. Продумана архітектура дозволила легко змінити клас контролера і використати в застосунку нову технологію запуску потоків.

Повний текст програми застосунку можна завантажити з [6].

8. ВИСНОВКИ

Ми порівняли можливості використання на практиці асинхронного шаблону на базі подій *BackgroundWorker* та асинхронних операцій *Task*, запущених оператором *await* у *async*-методі. Використання асинхронних операцій має низку переваг. Код запуску *Task.Run()* набагато простіший та зрозуміліший, схожий зовні на звичайний виклик методу. Для повернення результату *Task.Run()* використовує інструкцію *return*, як звичайний метод. Результат є строго типізований. Екземпляр *Task* ніяк не залучено до отримання інформації про хід виконання асинхронного потоку чи до його зупинки – для цього використовують окремі сутності: *Progress*<*T*> і *CancellationTokenSource* відповідно. Інформація про виконання є строго типізованою. Водночас *BackgroundWorker* оперує з даними невідомого типу *object*, змішує відповідальність одного методу за різні завдання та декількох налаштувань для однієї мети. Наприклад, метод опрацювання події *BackgroundWorker.RunWorkerCompleted* розпізнає помилки виконання, отримує результат та задає наступні після завершення дії.

Ми не висвітлили ще деякі переваги *Task*. Зокрема, він дає змогу безпомилково взаємодіяти з вкладеними асинхронними викликами, правильно розпізнавати та опрацьовувати винятки, що виникають в асинхронних потоках. Клас *Task* надає готові інструменти для синхронізації декількох асинхронних операцій. З цими та іншими перевагами *Task* читач зможе ознайомитися у [3].

1. Task Class [Електронний ресурс] / Microsoft Corporation // Microsoft Docs, 2020. – Режим доступу: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-6.0 – Назва з екрану.
2. Ярошко С. Використання шаблону асинхронної взаємодії потоків для візуалізації вибраних алгоритмів сортування в середовищі .Net / С. Ярошко, О. Ярошко // Вісник Львівського університету. Серія прикл. мат. та інф. Випуск 23, 2015. – С. 125-137.
3. Task.Run vs BackgroundWorker [Електронний ресурс] / Stephen Cleary – Режим доступу: https://blog.stephencleary.com/2013/05/taskrun-vs-backgroundworker-intro.html – Назва з екрану.
4. Ярошко С. А. Побудова багатопотокових програм засобами платформи .NET / Сергій Ярошко, Світлана Ярошко // Вісник Львівського університету. Серія прикл. мат. та інф. Випуск 27, 2019. – С. 154-165.
5. Ярошко С.А. Налаштування функціональності багатопотокової аплікації на етапі виконання / С.А. Ярошко, С.М. Ярошко // XXII Всеукраїнська наукова конференція Сучасні проблеми прикладної математики та інформатики (APAMCS-2016) : Збірник наукових праць, 5 7 жовтня 2016 р. – Львів: ЛНУ імені Івана Франка, 2016. – С. 200-201.

6. Sorting Threads by Tasks [Електронний ресурс] / Serhiy Yaroshko // github, 2021. – Режим доступу: https://github.com/mr-Serg/ReflactionSortingTasks – Назва з екрану.

COMPARISON OF Task.Run AND BackgroundWorker CAPABILITIES FOR BUILDING MULTITHREADED PROGRAMS IN C#

Svitlana Yaroshko, Serhii Yaroshko

Ivan Franko National University of Lviv,
Universytetska str, 1, Lviv, 79000, e-mail: serhiy.yaroshko@lnu.edu.ua

We compare the capabilities of instances of the classes *BackgroundWorker* and *Task* using the example of building an application in C#.

In a C# program, a single execution thread can be run in many ways: by asynchronously calling a delegate, by an instance of class *Thread*, by asynchronous event-based template (component *BackgroundWorker*), by an instance of class *Task* [1]. The developers of the .Net platform advise using the last one. In a previous publication [2] we described in detail the use of *BackgroundWorker* in *Windows Forms*, which visualizes the process of executing different algorithms for ordering a sequence of values running in individual threads. The *BackgroundWorker* component was very progressive at the time of its creation, but today more advanced technologies have appeared. We compare its capabilities and methods of use with the capabilities of the instance *Task*, in particular, obtained by the method *Task.Run()* on the example of the same application. A similar comparison with an invented example can be found in [3].

Recall that the application runs on execution in parallel threads of several methods of sorting an array of integers. Thanks to the use of reflection tools in the program, the user can choose the sorting methods and test them on different input data sets. The application allows the user to stop each of the methods if the execution takes too long. Each of the arrays is represented on a separate graphics panel by a set of segments of appropriate length. Sort threads report the application of exchanges of sorted array elements, which causes redrawing of the corresponding pair of segments. Each thread announces its completion. Therefore, the following requirements apply to the thread creation tool: convenient start of the ordering method; receiving information about the progress and the ability to display it in the application window; the ability to respond in the application thread to the end of the sorting thread; the ability to stop the thread early.

Both the *BackgroundWorker* and the *Task* meet these requirements, each in its own way. The startup code *Task.Run ()* is much simpler and clearer, similar in appearance to a normal method call. To return the result, *Task.Run()* uses the *return* statement as a normal method. The result is strictly typed. The *Task* instance is not involved in retrieving or stopping asynchronous thread execution by using separate entities: *Progress*<*T*> and *CancellationTokenSource*, respectively. Progress information is strictly typed. At the same time, *BackgroundWorker* handles data of unknown type *object*, mixing the responsibility of one method for different tasks and several settings for one purpose. For example, the *BackgroundWorker.RunWorkerCompleted* event handler recognizes execution errors, retrieves the result, and sets the next ones after the action is completed. We haven't covered some of the benefits of *Task* yet. In particular, it allows you to seamlessly interact with nested asynchronous calls, correctly recognize and handle exceptions that occur in asynchronous threads. The *Task* class provides ready-made tools for synchronizing multiple asynchronous operations. The reader will be able to get acquainted with these and other advantages of *Task* in [3].

Key words: thread, multithread application, class Task, cancellation of a thread execution, representation of a calculation progress, sorting algorithm, reflection.