

# Asynchronous Programming Advanced Topics

---



**Filip Ekberg**

Principal Consultant & CEO

@fekberg   fekberg.com



# Report on the Progress of a Task

---



**Out of the box** the **Task**  
**does not** automatically **report**  
the **progress**



# What Determines the Progress?

**Is it how much of the data that  
has been loaded?**

**Is it how much of the data that  
is completely processed?**



# Progress<T>

“Provides an **IProgress<T>** that invokes **callbacks** for **each reported progress value**.”



# Progress<T>

```
var progress = new Progress<string>();  
  
progress.ProgressChanged = (_, stringValue) => {  
    // Use the "stringValue" here!  
};
```



**Progress reporting** can be  
**complex** and **diffucult** but it's  
**made easier** with  
**IProgress<T>**



There is **no way** for a **task** to  
**automatically figure** out its  
**own progress**

We have to **introduce**  
something like **Progress<T>**





# Using Task Completion Source

---



# How Would You Use This with Async & Await?

**Event-based asynchronous  
pattern**

**Manually queue work on the  
thread pool**



# Event-based Asynchronous Pattern

```
var worker = new BackgroundWorker();  
  
worker.DoWork += (sender, e) => {  
    // Runs work on a different thread  
};  
  
worker.RunWorkerCompleted += (sender, e) => {  
    // Event triggered when work is done  
};
```



# Manually Queue Work on the Thread Pool

```
ThreadPool.QueueUserWorkItem(_ => {  
    // Run work on a different thread  
});
```



# TaskCompletionSource<T>

“Represents the producer side of a **Task<T>** unbound to a delegate, providing access to the consumer side **through the Task property**.”



# Task Completion Source

```
var tcs = new TaskCompletionSource<string>();
```

```
Task<string> task = tcs.Task;
```



Use **TaskCompletionSource** to  
create **awaitables** out of **legacy**  
**code** that **don't use** the **TPL**



# Working with Attached and Detached Tasks

---





# Nested / Child Tasks

```
Task.Run(() => {
```

```
    Task.Run(() => {});
```

```
    Task.Run(() => {});
```

← These are child tasks  
←

```
});
```



# Task.Factory.StartNew Overloads

```
StartNew(Action)  
StartNew(Action, CancellationToken)  
StartNew(Action, TaskCreationOptions)  
StartNew(Action, CancellationToken, TaskCreationOptions, TaskScheduler)
```

```
StartNew(Action<Object>, Object)  
StartNew(Action<Object>, Object, CancellationToken)  
StartNew(Action<Object>, Object, TaskCreationOptions)  
StartNew(Action<Object>, Object, CancellationToken,  
TaskCreationOptions, TaskScheduler)
```

+ 8 more



# Task.Factory.StartNew Overloads

**StartNew**(Action)

StartNew(Action, CancellationToken)

StartNew(Action, TaskCreationOptions)

StartNew(Action, CancellationToken, TaskCreationOptions, TaskScheduler)

StartNew(Action<Object>, Object)

StartNew(Action<Object>, Object, CancellationToken)

StartNew(Action<Object>, Object, TaskCreationOptions)

StartNew(Action<Object>, Object, CancellationToken,  
TaskCreationOptions, TaskScheduler)

+ 8 more



# Task.Factory.StartNew Overloads

StartNew(Action)

**StartNew**(Action, CancellationToken)

StartNew(Action, TaskCreationOptions)

StartNew(Action, CancellationToken, TaskCreationOptions, TaskScheduler)

StartNew(Action<Object>, Object)

StartNew(Action<Object>, Object, CancellationToken)

StartNew(Action<Object>, Object, TaskCreationOptions)

StartNew(Action<Object>, Object, CancellationToken,  
TaskCreationOptions, TaskScheduler)

+ 8 more



# Task.Factory.StartNew Overloads

StartNew(Action)

StartNew(Action, CancellationToken)

**StartNew**(Action, TaskCreationOptions)

StartNew(Action, CancellationToken, TaskCreationOptions, TaskScheduler)

StartNew(Action<Object>, Object)

StartNew(Action<Object>, Object, CancellationToken)

StartNew(Action<Object>, Object, TaskCreationOptions)

StartNew(Action<Object>, Object, CancellationToken,  
TaskCreationOptions, TaskScheduler)

+ 8 more



# Task.Factory.StartNew Overloads

StartNew(Action)

StartNew(Action, CancellationToken)

StartNew(Action, TaskCreationOptions)

StartNew(Action, CancellationToken, TaskCreationOptions, TaskScheduler)

**StartNew**(Action<Object>, Object)

StartNew(Action<Object>, Object, CancellationToken)

StartNew(Action<Object>, Object, TaskCreationOptions)

StartNew(Action<Object>, Object, CancellationToken,  
TaskCreationOptions, TaskScheduler)

+ 8 more



Using **Task.Run** is in most situations  
the **best option**



# AttachedToParent

“Specifies that a **task** is **attached** to a **parent** in the task hierarchy. **By default**, a **child task** (that is, an inner task created by an outer task) **executes independently** of its parent.

You can use the **AttachedToParent** option so that the **parent** and **child tasks** are **synchronized**.

Note that if a **parent task** is **configured** with the **DenyChildAttach** option, the **AttachedToParent** option in the child task **has no effect**, and the child task will execute as a detached child task.”





If a **parent task** is **configured** with  
the **DenyChildAttach** option

**AttachedToParent** option in the  
child task **has no effect**



# Task.Run Automatically Unwraps!

```
Task<string> task = Task.Run(async () => {  
    await Task.Delay(1000);  
  
    return "Pluralsight";  
});
```

```
Task<Task<string>> taskFromFactory = Task.Factory.StartNew(async () => {  
    await Task.Delay(1000);  
  
    return "Pluralsight";  
});
```

```
Task<string> unwrappedTask = taskFromFactory.Unwrap();
```



# Passing a Value to Task.Factory.StartNew

```
IEnumerable<StockPrice> stocks = ...  
Task.Factory.StartNew((state) => {  
    // Cast the state to the correct type  
    var items = state as IEnumerable<StockPrice>  
}, stocks);
```



# Passing a Value to Task.Factory.StartNew

```
IEnumerable<StockPrice> stocks = ...  
Task.Factory.StartNew((state) => {
```

```
    // Cast the state to the correct type  
    var items = state as IEnumerable<StockPrice>
```

```
}, stocks);
```

**Using “stocks” directly in the anonymous method would introduce a closure**



# Passing a Value to Task.Factory.StartNew

```
IEnumerable<StockPrice> stocks = ...  
Task.Factory.StartNew((state) => {  
  
    // Cast the state to the correct type  
    var items = state as IEnumerable<StockPrice>  
  
}, stocks);
```

**You can pass a reference to the object  
which will be used by the asynchronous  
operation**



# Passing a Value to Task.Factory.StartNew

```
IEnumerable<StockPrice> stocks = ...  
Task.Factory.StartNew((state) => {  
  
    // Cast the state to the correct type  
    var items = state as IEnumerable<StockPrice>  
  
}, stocks);
```



# Task.Run

```
Task.Run(() => {});
```



**Internally uses the factory with these default values**

```
Task.Factory.StartNew(  
    () => {},  
    CancellationToken.None,  
    TaskCreationOptions.DenyChildAttach,  
    TaskScheduler.Default  
);
```

