PFx: Tasks

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Agenda

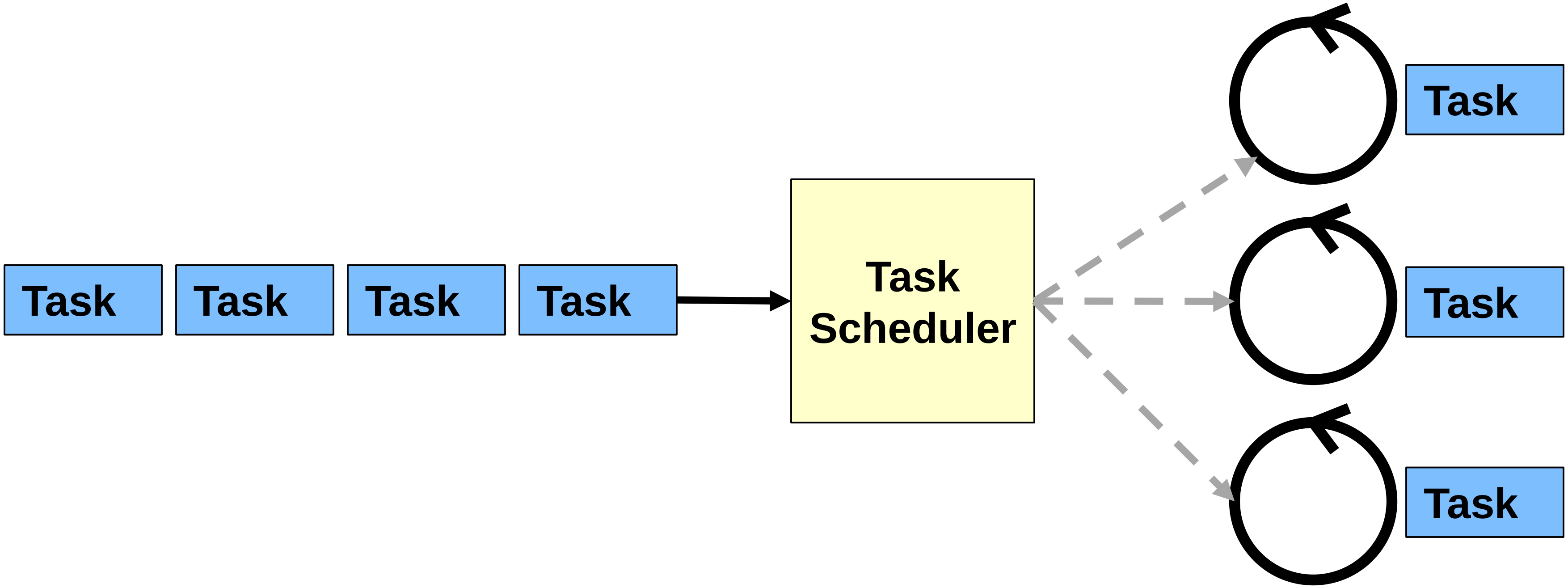
- What is PFx
- The Task abstraction
- Creating Tasks
- Passing data into tasks and retrieving results
- Cancellation
- Task dependency
- Task Scheduling

Parallel Framework Extensions (PFx)

- **Part of `mscorlib`**
  - `System.Threading.Tasks`
- **Originally designed to aid parallelizing CPU intensive algorithms**
  - CTP for VS 2008
- **Now general purpose library for asynchronous work**
  - Tasks
  - Concurrent data structures
  - Synchronization primitives
  - Parallelization

What is a Task?

- **A Task is a schedulable Unit of Work**
  - Wraps a delegate that is the actual work
- **Task is enqueued to a TaskScheduler**



Creating a Task

- **Tasks are created by passing a delegate to the constructor**
  - Call **`Start`** to queue the task to the scheduler
  - Can also use a factory

```
Action a = delegate
{
  Console.WriteLine("Hello from task");
};
Task tsk = new Task(a);
tsk.Start();
```

```
Action a = delegate
{
  Console.WriteLine("Hello from task");
};
Task tsk = Task.Factory.StartNew(a);
```

A Unifying API

- **.NET 3.5 had separate APIs for short and long running async work**
  - Short running work put on ThreadPool
  - Long running spawn new thread using Thread class
- **Task unifies API by hinting to scheduler that task is long running**
  - Current implementation long running tasks spawn own thread otherwise ThreadPool used

```
Task t1 = new Task( DoWork, TaskCreationOptions.LongRunning );
```

Passing Data to a Task

- Data passed explicitly using `Action<object>`
- Data can be passed implicitly using anonymous delegate rules

```
Guid jobId = Guid.NewGuid();

Action<object> a = delegate(object state)
{
  Console.WriteLine("{0}: Hello {1}", jobId, state);
};

Task tsk = new Task(a, "World");
tsk.Start();
```

Returning Data from a Task

- Generic version of Task available
  - `T` is return type
  - Accessed from the task `Result`
- Takes a Func delegate as a constructor parameter
  - `Func<T>`
  - `Func<object, T>`

```
Func<object, int> f = delegate(object state)
{
  Console.WriteLine("Hello {0}", state);
  return 42;
};
Task<int> tsk = new Task<int>(f, "World");
tsk.Start();
//...
Console.WriteLine("Task return is: {0}", tsk.Result);
```

Waiting for a Task to End

- Can wait for one or more tasks to end using `Wait`, `WaitAll` or `WaitAny`
- Can pass timeout for wait

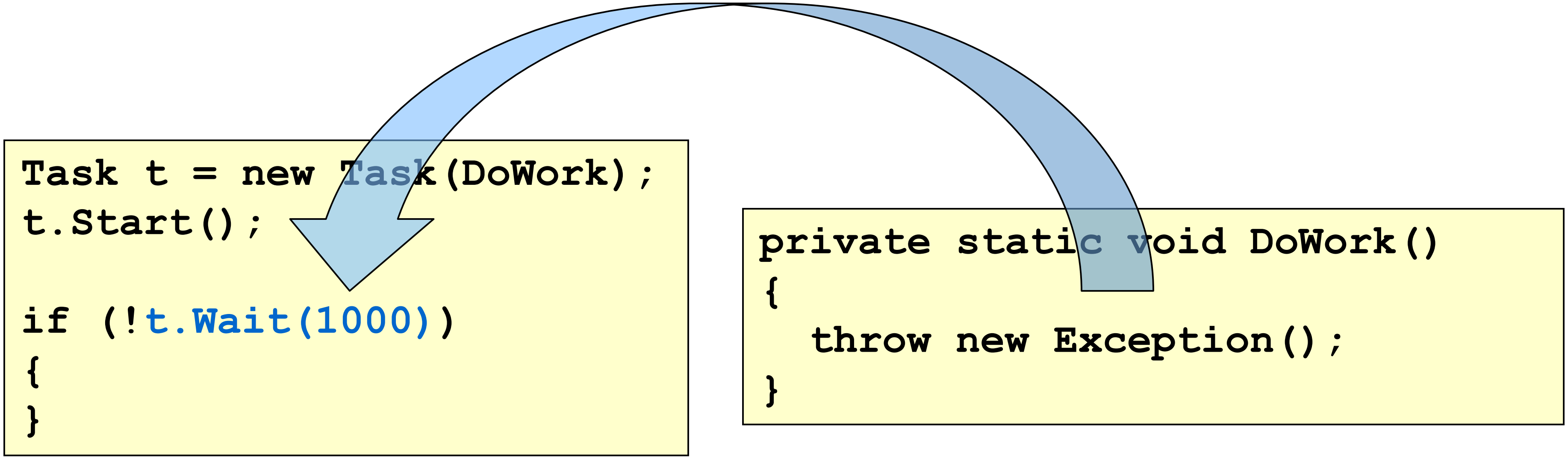
```
Task t = new Task( DoWork );
t.Start();
t.Wait();
```

```
Task t1 = new Task( DoWork );
t1.Start();
Task t2 = new Task( DoOtherWork );
t2.Start();

if (!Task.WaitAll(new Task[]{t1, t2}, 2000))
{
  Console.WriteLine( "wait timed out" );
}
```

Task Completion States

- **Tasks can end in one of three states**
  - RanToCompletion: everything completed normally
  - Canceled: task was cancelled
  - Faulted: an unhandled exception occurred on the task
- **Unhandled Exceptions get thrown when waiting on a task**



```
Task t = new Task(DoWork);
t.Start();

if (!t.Wait(1000))
{
}
```

```
private static void DoWork()
{
   throw new Exception();
}
```

Exceptions

- **In .NET 4 …**
  - Any Task’s unobserved exceptions re-thrown in Finalizer
    - Results in process being torn down
    - This is a GOOD THING
- **In .NET 4.5 …**
  - Unobserved exceptions are simply ignored by default
  - Can reintroduce the .NET 4 behaviour through configuration

```
<configuration>
    <runtime>
        <ThrowUnobservedTaskExceptions enabled="true"/>
    </runtime>
</configuration>
```

Cancellation

- **Tasks support cancellation**
  - Modelled by `CancellationToken`
- **Token can be passed into many APIs**
  - Task creation
  - Waiting

```
CancellationTokenSource source =
                           new CancellationTokenSource();

Task t1 = new Task( DoWork, source.Token );
t1.Start();

t1.Wait(source.Token);
```

Triggering Cancellation

- `CancellationTokenSource` has `Cancel` method to trigger the cancellation of tasks and blocking APIs

```
CancellationTokenSource source =
                           new CancellationTokenSource();

Task t1 = new Task( DoWork, source.Token );
t1.Start();

source.Cancel();
```

The Effects of Cancellation

- **Cancellation has different effects depending on state of task**
  - Unscheduled tasks are never run
  - Scheduled tasks must cooperate to end. Requires access to CancellationToken

```
private static void DoWork(object o)
{
  CancellationToken tok = (CancellationToken)o;

  while (true)
  {
    Console.WriteLine("Working ...");
    Thread.Sleep(1000);
    tok.ThrowIfCancellationRequested();
  }
}
```

Cancellation of Blocking Operations

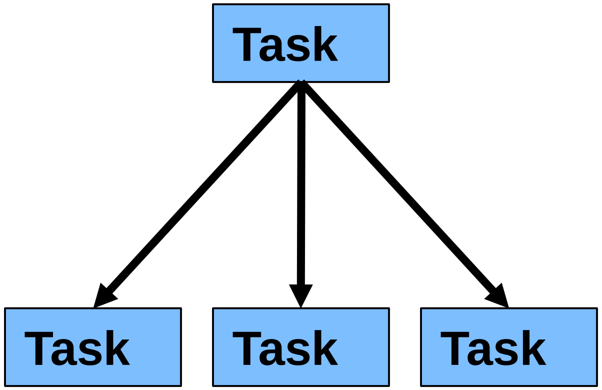
- **Blocking operations throw exceptions when cancelled**
  - OperationCancelledException
  - AggregateException (when more than one exception could have occurred)

```
try
{
  t1.Wait(1000, source.Token);
}
catch (AggregateException x)
{
  foreach (var item in x.Flatten().InnerExceptions)
  {
    Console.WriteLine(item.Message);
  }
}
```

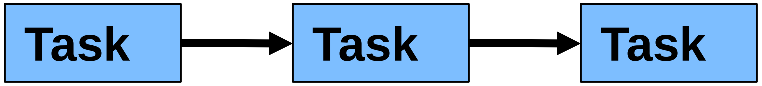
Dependent Tasks

- **Tasks can be dependent on other tasks**
- **Two models of dependency**
  - Child tasks: parent only complete when all children complete
  - Chained tasks: scheduled when previous task finishes

Child tasks

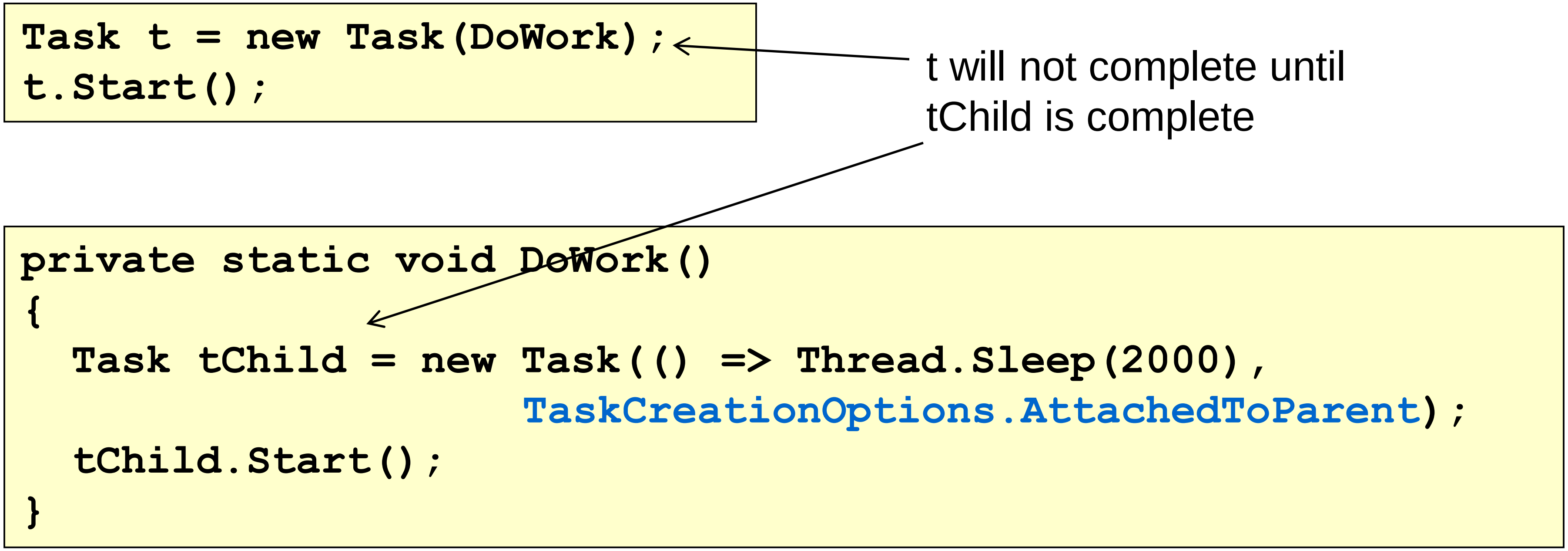


Chained tasks



Creating Child Tasks

- **When one task creates another it can optionally declare it as a child task**
  - Default is to spawn an independent task
- **Parent task will not complete until all children are complete**



```
Task t = new Task(DoWork);
t.Start();
```

t will not complete until tChild is complete

```
private static void DoWork()
{
  Task tChild = new Task(() => Thread.Sleep(2000),
                         TaskCreationOptions.AttachedToParent);
  tChild.Start();
}
```

Creating Chained Tasks

- **Tasks are chained using the ContinuesWith**
  - New task will be scheduled when previous one finishes

```
Task t = new Task(DoWork);

t.ContinueWith(tPrev => Console.WriteLine(tPrev.Status));
t.Start();
```

Flowing Data to Chained Tasks

- **Often you need the chained task to work on the results of the previous task**
  - Tasks returning results modelled on Task<T>
  - Chained task takes an Action<Task<T>>

```
Task<int> t = new Task<int>(GetData);

t.ContinueWith(ProcessData);

t.Start();
```

```
static void ProcessData(Task<int> prevTask)
{
  Console.WriteLine(prevTask.Result);
}
```

Chaining Tasks Based on Outcome

- **Tasks can be chained depending on the outcome of the previous task**
  - RunToCompletion
  - Canceled
  - Faulted
- **TaskContinuationOptions flags passed to ContinuesWith**

```
Task<int> t = new Task<int>(GetData);

t.ContinueWith(ProcessData,
               TaskContinuationOptions.OnlyOnRanToCompletion);

t.Start();
```

Integrating with Async APIs

- **.NET has Async Pattern baked into many APIs**
  - WebRequest: BeginGetResponse / EndGetResponse
  - SqlCommand: BeginExecuteReader / EndExecuteReader
- **Tasks integrate with async APIs using FromAsync on Factory**
  - Takes `IAsyncResult` as first parameter
  - Takes `Func<IAsyncResult, T>` as second parameter

```
WebRequest req = WebRequest.Create("http://www.develop.com");

Task<WebResponse> t = Task<WebResponse>.Factory
             .FromAsync(req.BeginGetResponse(null, null),
                        req.EndGetResponse);

t.Wait();

Console.WriteLine(t.Result.ContentLength);
```

Custom Integration with Tasks

- **TaskCompletionSource**
- **Used to bridge between other apis and task based apis**
- **TaskCompletionSource.Task provides a task**
- **New Task state signalled via**
  - TaskCompletionSource.SetResult
- **Used to build adapters for legacy apis**

Fairness

- **By default task scheduling makes no guarantee about order of task execution**
  - Allows scheduler flexibility over caching for related tasks
- **Sometimes work should be processed in creation order**
  - Can pass `TaskCreationOptions.PreferFairness` on task creation
  - Tells scheduler to schedule it in order with other tasks created with same flag (other tasks could still run out of order)
- **Flag should be viewed as hint to scheduler**
  - Not a guarantee

Scheduling

- **TaskScheduler is an extensible abstract class**
- **Two implementations come with .NET 4**
  - ThreadPoolTaskScheduler (Default)
  - SynchronizationContextTaskScheduler
- **Can pass scheduler when starting a task**

```
TaskScheduler scheduler = GetScheduler();

Task t = new Task(DoWork);

t.Start(scheduler);
```

Integrating with SynchronizationContext

- GUI applications need UI updates marshalled to the UI thread
- SynchronizationContext is abstraction that wraps specific technology
- Can get scheduler associated with SynchronizationContext

On UI Thread

```
TaskScheduler scheduler =
      TaskScheduler.FromCurrentSynchronizationContext();
```

On Background Thread

```
private void DoAsyncWork()
{
  Task t = new Task(() => label1.Text = "TADA!!");
  t.Start(scheduler);
}
```

Summary

- Tasks unify the async API
- Tasks support cancellation
- Tasks support dependency
- Task integrate with Async APIs
- Task Scheduling is an extensible model