[Task](https://msdn.microsoft.com/en-us/library/system.threading.tasks.task(v=vs.110).aspx) and [Task<TResult>](https://msdn.microsoft.com/en-us/library/dd321424(v=vs.110).aspx) each expose a static [Factory](https://msdn.microsoft.com/en-us/library/system.threading.tasks.task.factory(v=vs.110).aspx) property that returns a default instance of [TaskFactory](https://msdn.microsoft.com/en-us/library/system.threading.tasks.taskfactory(v=vs.110).aspx), so that you can call the method asTask.Factory.StartNew(). Also, in the following example, because the tasks are of type [System.Threading.Tasks.Task<TResult>](https://msdn.microsoft.com/en-us/library/dd321424(v=vs.110).aspx), they each have a public [Task<TResult>.Result](https://msdn.microsoft.com/en-us/library/dd321468(v=vs.110).aspx) property that contains the result of the computation. The tasks run asynchronously and may complete in any order. If the [Result](https://msdn.microsoft.com/en-us/library/dd321468(v=vs.110).aspx) property is accessed before the computation finishes, the property blocks the calling thread until the value is available.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace \_01\_console

{

class Program

{

static void Main(string[] args)

{

Task<Double>[] taskArray = { Task<Double>.Factory.StartNew(() => DoComputation(5)),

Task<Double>.Factory.StartNew(() => DoComputation(4)),

Task<Double>.Factory.StartNew(() => DoComputation(3)) };

var results = new Double[taskArray.Length];

for (int i = 0; i < taskArray.Length; i++)

{

results[i] = taskArray[i].Result;

Console.WriteLine("{0} ", results[i]);

}

Console.ReadLine();

}

private static Double DoComputation(Double start)

{

Double f= 1;

for (var value = 2; value <= start ; value ++)

f \*= value;

return f;

}

}

}

Problem

When you use a lambda expression to create a delegate, you have access to all the variables that are visible at that point in your source code. However, in some cases, most notably within loops, a lambda **doesn't capture the variable as expected**. It only captures the final value, not the value as it mutates after each iteration. The following example illustrates the problem. It passes a loop counter to a lambda expression that instantiates a CustomData object and uses the loop counter as the object's identifier. As the output from the example shows, each CustomDataobject has an identical identifier.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace \_01\_console

{

class CustomData

{

public long CreationTime;

public int Name;

public int ThreadNum;

}

class Program

{

static void Main(string[] args)

{

// Create the task object by using an Action(Of Object) to pass in the loop

// counter. This produces an unexpected result.

Task[] taskArray = new Task[10];

for (int i = 0; i < taskArray.Length; i++)

{

taskArray[i] = Task.Factory.StartNew((Object obj) =>

{

var data = new CustomData() { Name = i, CreationTime = DateTime.Now.Ticks };

data.ThreadNum = Thread.CurrentThread.ManagedThreadId;

Console.WriteLine("Task #{0} created at {1} on thread #{2}.", data.Name, data.CreationTime, data.ThreadNum);

},

i);

}

Task.WaitAll(taskArray);

Console.ReadLine();

}

}

}

Solution

This state is passed as an argument to the task delegate, and it can be accessed from the task object by using the [Task.AsyncState](https://msdn.microsoft.com/en-us/library/system.threading.tasks.task.asyncstate(v=vs.110).aspx) property. The following example is a variation on the previous example. It uses the [AsyncState](https://msdn.microsoft.com/en-us/library/system.threading.tasks.task.asyncstate(v=vs.110).aspx) property to display information about the CustomData objects passed to the lambda expression.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace \_01\_console

{

class CustomData

{

public long CreationTime;

public int Name;

public int ThreadNum;

}

class Program

{

static void Main(string[] args)

{

Task[] taskArray = new Task[10];

for (int i = 0; i < taskArray.Length; i++)

{

taskArray[i] = Task.Factory.StartNew((Object obj) =>

{

CustomData data = obj as CustomData;

if (data == null)

return;

data.ThreadNum = Thread.CurrentThread.ManagedThreadId;

},

new CustomData() { Name = i, CreationTime = DateTime.Now.Ticks });

}

Task.WaitAll(taskArray);

foreach (var task in taskArray)

{

var data = task.AsyncState as CustomData;

if (data != null)

Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",

data.Name, data.CreationTime, data.ThreadNum);

}

Console.ReadLine();

}

}

}