CSC351

Writing multithreaded mobile applications

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Using multiple threads in mobile apps
2. Creating threads and tasks
3. Dealing with exceptions
4. Cancelling tasks

Using multiple threads in mobile apps

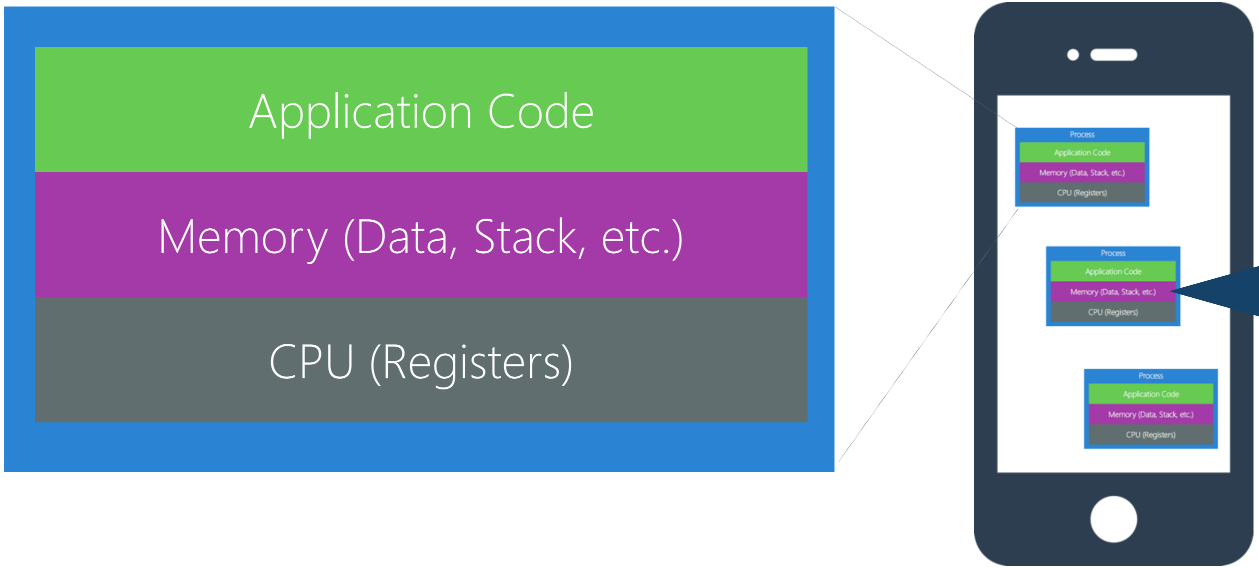
Xamarin
University

Tasks

1. Processes and Threads
2. Mobile app architecture
3. Keeping your app responsive

What is a process?

- A **Process** is an instance of a program that is loaded into memory



Application Code

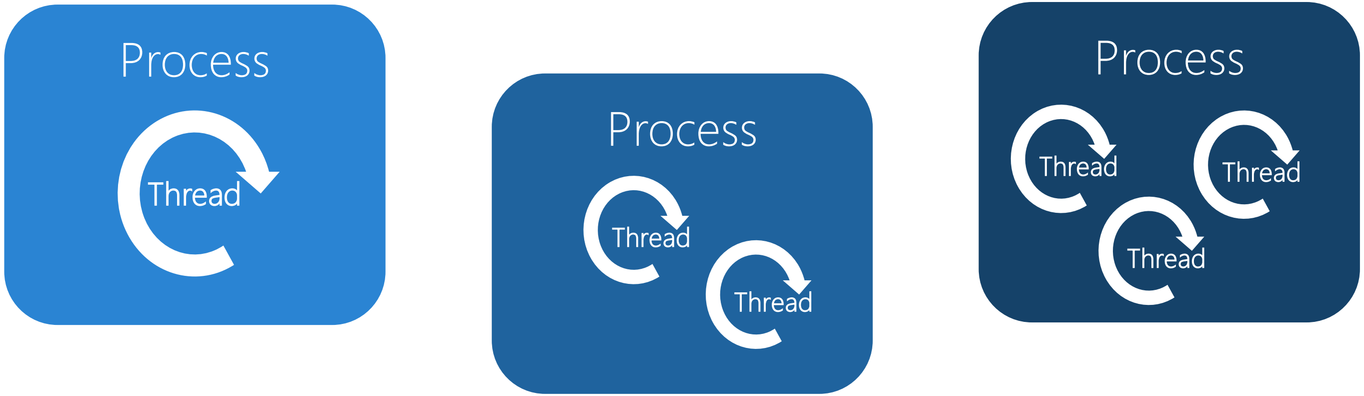
Memory (Data, Stack, etc.)

CPU (Registers)

Most platforms allow multiple processes to run concurrently

What are threads?

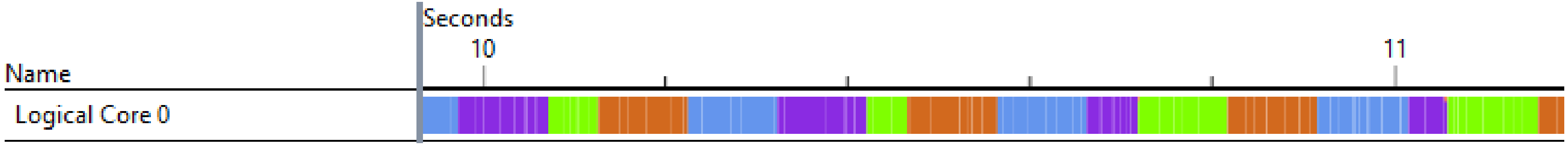
- **Threads** are a software abstraction of the CPU used to enhance the stability of the operating system

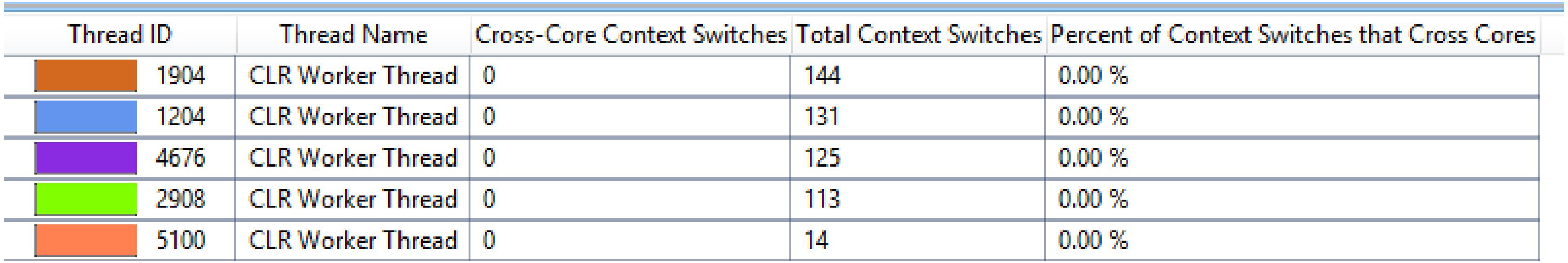


Every active process in iOS, Android and Windows has **at least** one thread

Threads > CPUs

- CPU(s) are **shared** by all running processes and **switch** between threads on given intervals (often called a *time-slice* or *quantum*)





Thread ID	Thread Name	Cross-Core Context Switches	Total Context Switches	Percent of Context Switches that Cross Cores
🟧 1904	CLR Worker Thread	0	144	0.00 %
🟦 1204	CLR Worker Thread	0	131	0.00 %
🟪 4676	CLR Worker Thread	0	125	0.00 %
🟩 2908	CLR Worker Thread	0	113	0.00 %
🟧 5100	CLR Worker Thread	0	14	0.00 %

Reasons to use multiple threads

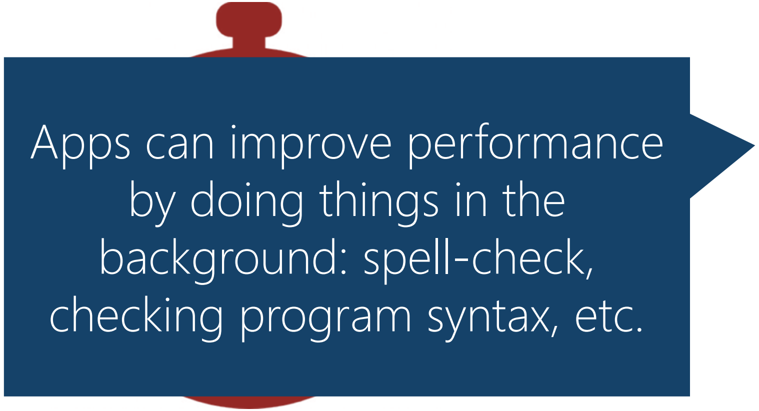
- Two reasons we rely on multiple threads

Users expect apps to respond to interactions immediately; when they don't it is assumed that the app is "broken"

App Responsiveness

Reasons to use multiple threads

- Two reasons we rely on multiple threads



Apps can improve performance by doing things in the background: spell-check, checking program syntax, etc.

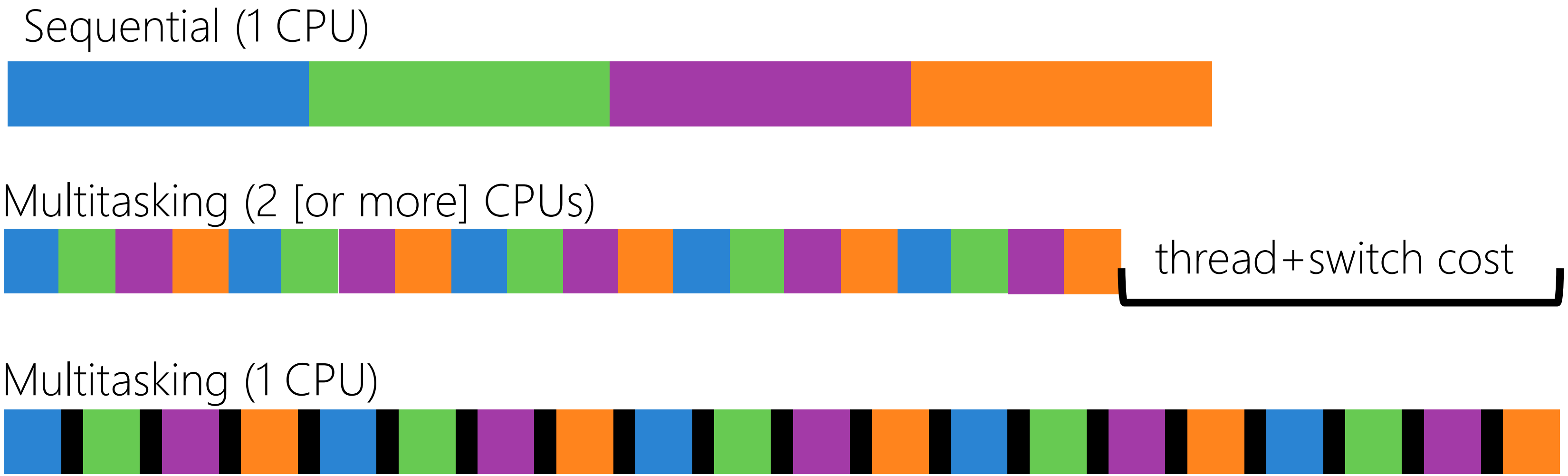
App Responsiveness

Features + Performance

Xamarin University

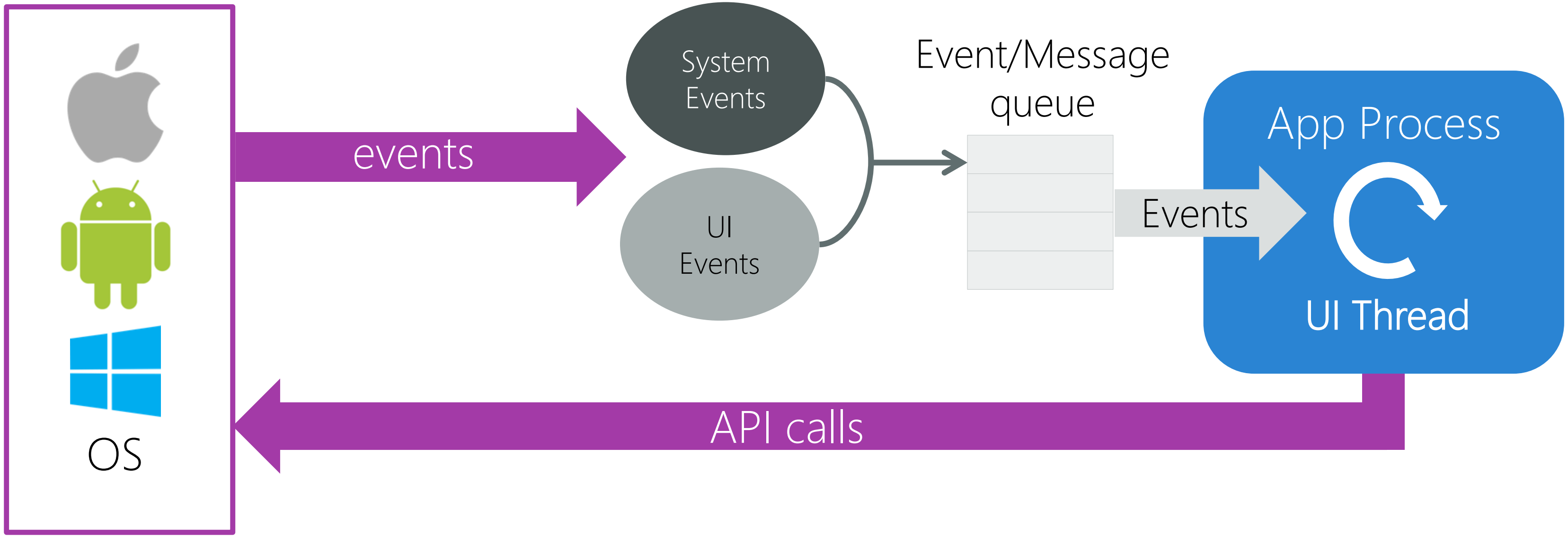
Reasons *not* to use multiple threads

- Processing work that is very short (~30ms or less) might not be worth scheduling multiple threads for; should **profile on device** to determine cost



Mobile app architecture

- All mobile applications start with a **primary** thread, or **UI thread**, which is responsible for managing the UI notifications from the operating system



Xamarin University

Keeping your app responsive

- Blocking or running long-running calculations on the primary thread can cause your application to become sluggish or unresponsive

```
void OnButtonClicked(object sender, EventArgs e)
{
   byte[] audioData = GetAudioDataFromFile("inFile.wav");
   int index = FindStartOfDataChunk(audioData);
   byte[] reversedData = ReverseWavData(audioData, index);
   WriteReversedWavFile(reversedData, "outFile.wav");
}
```

What's wrong with this code?

Keeping your app responsive

- Blocking or running long-running calculations on the primary thread can cause your application to become sluggish or unresponsive

```
void OnButtonClicked(object sender, EventArgs e)
{
    b                        DataFromFile("inFile.wav");
    i                        Chunk(audioData);
    b                        seWavData(audioData, index);
    W                        edData, "outFile.wav");
}
```

Event handlers are almost always called on the UI thread

Keeping your app responsive

- Blocking or running long-running calculations on the primary thread can cause your application to become sluggish or unresponsive

```
void OnButtonClicked(object sender, EventArgs e)
{
   byte[] audioData = GetAudioDataFromFile("inFile.wav");
   int index = FindStartOfDataChunk(audioData);
   byte[] reversedData = ReverseWavData(audioData, index);
   WriteReversedWavFile(reverse      a, "outFile.wav");
}
```

CPU-intensive processing done on the UI thread stops it from reacting to UI activity

Keeping your app responsive

- Blocking or running long-running calculations on the primary thread can cause your application to become sluggish or unresponsive

```
void OnButtonClicked(object sender, EventArgs e)
{
   byte[] audioData = GetAudioDataFromFile("inFile.wav");
   int index = FindStartOfDataChunk(audioData);
   byte[] reversedData = ReverseWavData(audioData, index);
   WriteReversedWavFile(reversedData, "outFile.wav");
}
```

Performing I/O will force the UI thread to *wait* for the response

Never execute long operations or call methods that block or wait when on the UI Thread

Group Exercise

Explore an unresponsive app

Xamarin University

Tasks

1. Processes and Threads
2. Mobile app architecture
3. Keeping your app responsive

Creating threads and tasks

Xamarin University

Tasks

1. Creating Threads and Tasks
2. Waiting for tasks to finish
3. Processing results from tasks

Creating threads

- ❖ Each mobile platform has a dedicated API to create and manage threads – can utilize these in platform-specific code

```
DispatchQueue.DefaultGlobalQueue.DispatchAsync(() => {
    ... // Stuff to do on background thread
});
```

Grand Central Dispatch allows you to pass work off onto a set of threads managed natively by iOS; Xamarin makes this task painless

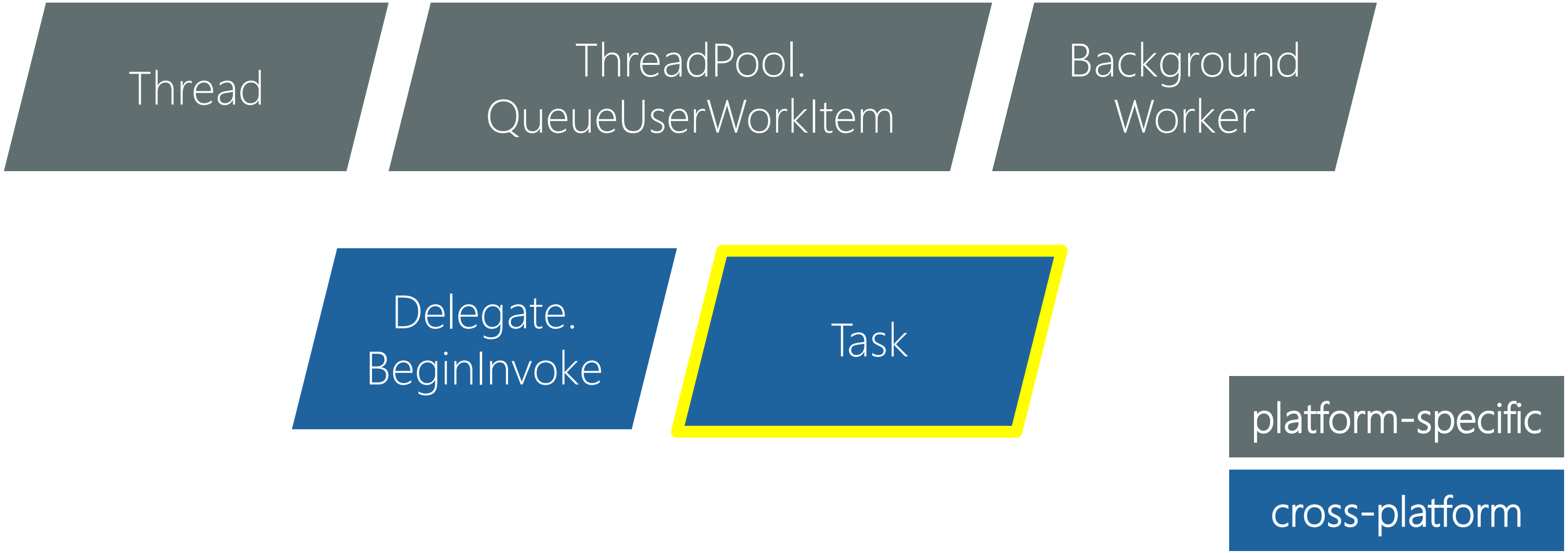
Creating threads

- Each mobile platform has a dedicated API to create and manage threads – can utilize these in platform-specific code

```
class MyWork : Java.Lang.Object, IRunnable
{
    // Helper method to start our work
    static public void Start() {
        AsyncTask.Execute(new MyWork());
    }

    void IRunnable.Run() {
        ... // Stuff to do on a background thread
    }
}
```

Creating threads in Xamarin

- .NET/Mono provide a variety of ways to create and start threads to perform **CPU bound work**



Thread

ThreadPool.
QueueUserWorkItem

Background
Worker

Delegate.
BeginInvoke

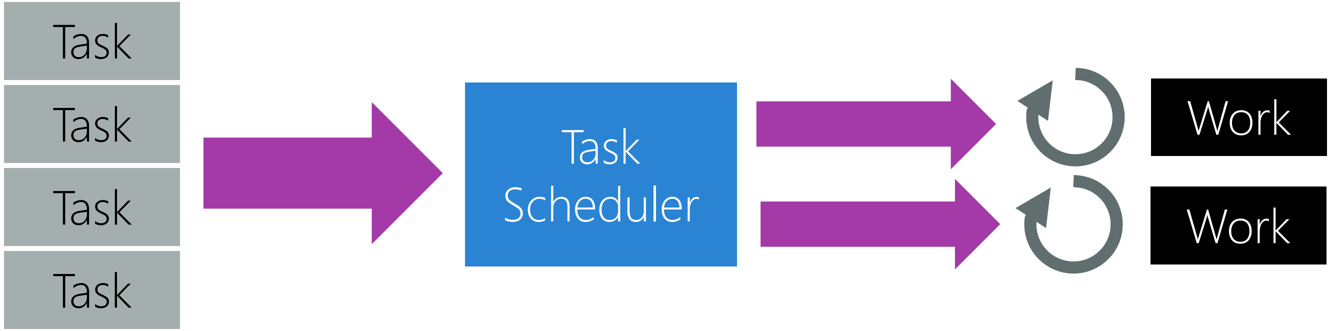
Task

platform-specific

cross-platform

What is a Task?

- A **Task** represents an asynchronous operation and includes an associated delegate that defines the work to perform



User initiated tasks are queued to a scheduler

... and executed on pool of available threads (the "thread pool")

Task vs. Thread

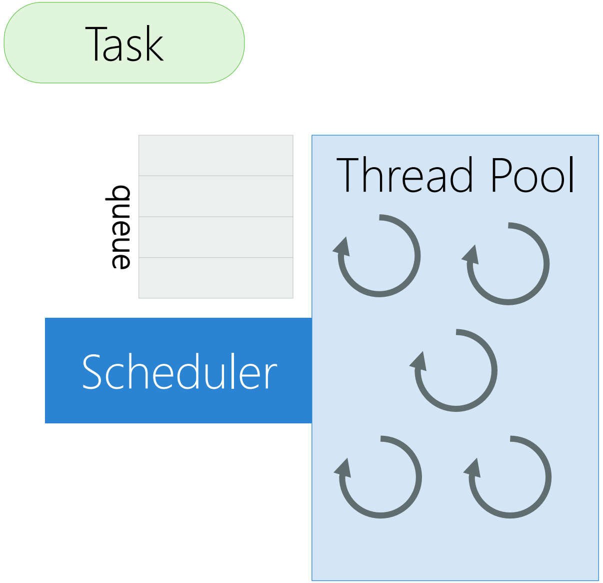
- A task is considered a "future" or "promise" of something that will execute and possibly return some value at a later point in time

- A thread is a way to execute a task, but **not every task requires a thread;** non-CPU bound work can often be performed through some other mechanism (e.g. I/O, timers, etc.)

What if the Task needs a thread?

- To save time and memory, the runtime creates a dynamic collection of *worker threads* (called a **Thread Pool**) that are used to execute CPU-bound work such as **`Tasks`**

- Tasks are handled in a FIFO fashion and mapped to a thread by a managed thread scheduler



Starting a Task

- Running CPU-bound work in a **Task** is a two-step process – create the **Task** object and then call **Start** to get it scheduled on a thread

```
using System.Threading.Tasks;
```

```
Task theTask = new Task(new Action(SomeLongOperation) );
...
```

```
void SomeLongOperation ()
{ ... }
```

Takes an **Action** delegate to define the work to perform on a background thread

Starting a Task

- Running CPU-bound work in a **Task** is a two-step process – create the **Task** object and then call **Start** to get it scheduled on a thread

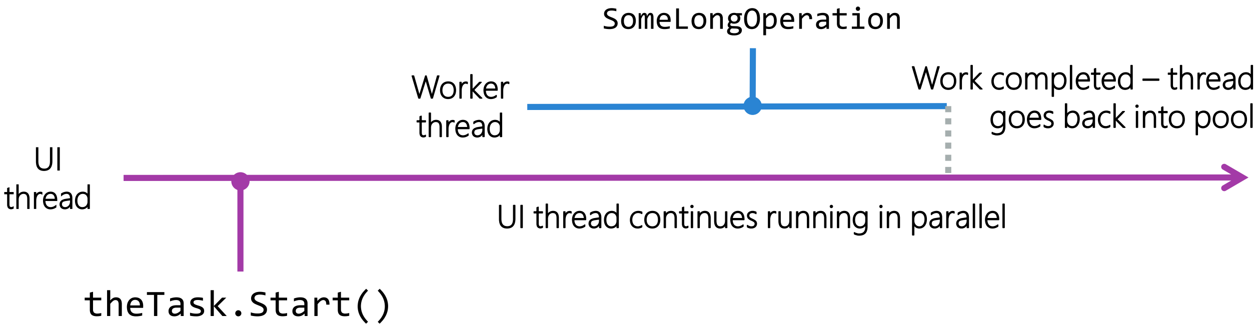
```
using System.Threading.Tasks;
```

```
Task theTask = new Task(new Action(SomeLongOperation));
...
theTask.Start();
```

Start passes the task to the scheduler; it is at this point that thread-resources are assigned and (eventually) the work is actually performed

Task execution progress

- Starting a Task will cause the worker delegate to be executed on a thread pool thread – this runs in parallel to the UI thread


SomeLongOperation
Worker thread
Work completed – thread goes back into pool
UI thread
UI thread continues running in parallel
theTask.Start()

Starting a long-running task

- If the work you want to execute will run for the life of the application, then you can ask the scheduler to use a **dedicated thread**

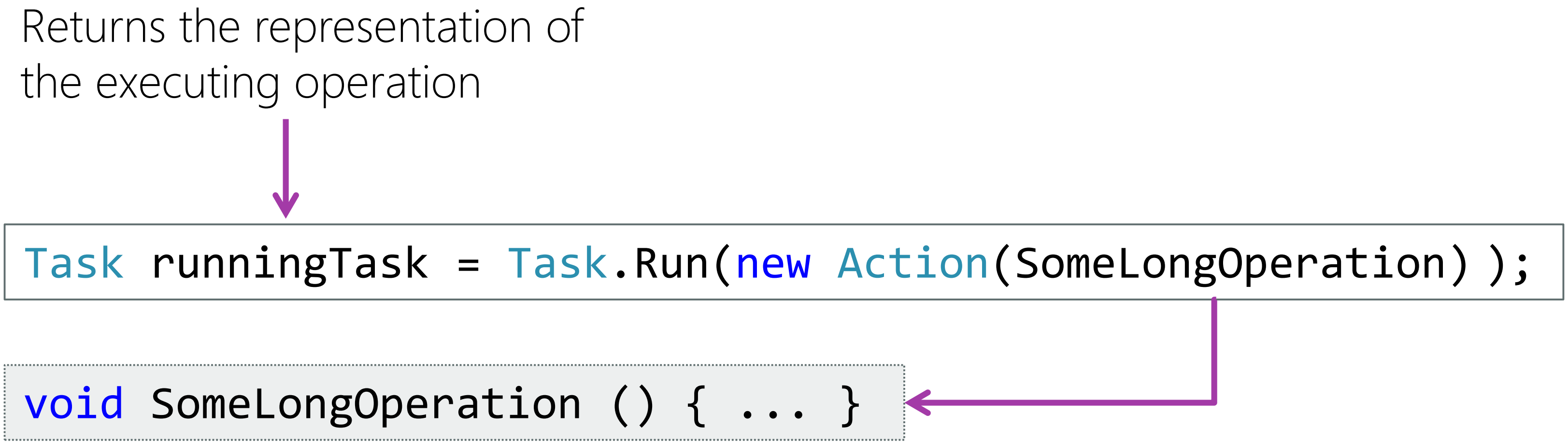
```
Task nwTask = new Task(new Action(CheckCellNetwork),
    TaskCreationOptions.LongRunning);
nwTask.Start();
```

Can pass options ("hints") as part of the task creation – in this case to ask for a dedicated thread vs. a thread pool thread

```
void CheckCellNetwork () {
   while (true) { ... }
}
```

Starting a Task in one step

- Can also use **`Task.Run`** to create and start a new CPU-bound task in one step; provides a more streamlined approach for common case



Returns the representation of the executing operation

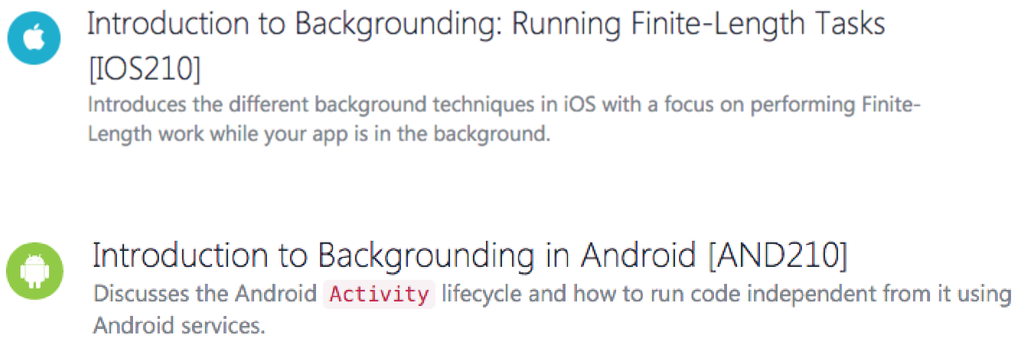
```
Task runningTask = Task.Run(new Action(SomeLongOperation) );
```

```
void SomeLongOperation () { ... }
```

Threads vs. App lifetime

- Task API will start an asynchronous operation, however you might still need to register with the OS to allow background tasks to continue to run when your app is not visible

Makes sure to check out the appropriate courses to get more information on a per-platform basis



Introduction to Backgrounding: Running Finite-Length Tasks [IOS210]

Introduces the different background techniques in iOS with a focus on performing Finite-Length work while your app is in the background.

Introduction to Backgrounding in Android [AND210]

Discusses the Android `Activity` lifecycle and how to run code independent from it using Android services.

Flash Quiz

Flash Quiz

① When you start a **`Task`**, it always *creates* a thread to execute your code

a) True

b) False

Flash Quiz

① When you start a `Task`, it always *creates* a thread to execute your code

a) True

b) False

Flash Quiz

② The Task API guarantees that the Task will be running after calling **Start**, or using **Task.Run**

a) True

b) False

Flash Quiz

② The Task API guarantees that the Task will be running after calling **Start**, or using **Task.Run**

a) True

b) False

Task activities

- Starting a **Task** is the first step – now we need to:

Pass parameters (optional)

Know when it's finished

Process result(s) (optional)

Passing parameters to a Task

- Can use the constructor to pass a single **`object`** parameter to the task – then cast to known type in the delegate

```
Task theTask = new Task(new Action<object>(WriteToLog), "Hello");
theTask.Start();
```

```
void AddToLog(object parameter)
{
   string text = (parameter ?? "").ToString();
   ... // Do something with the parameter
}
```

Hint: There is also a `Task.Factory.StartNew` method which can create and start the task in one step with an optional parameter

Other ways to pass parameters

- Can also use class fields to provide access to parameters

```
class LogEntry
{
   public string Text;
   ...
   public void WriteEntry() { ... }
}
```

Here we put the worker code into a class which manages the instance-specific data

```
LogEntry le = new LogEntry() { Text = "Hello" };
Task theTask = Task.Run(le.WriteEntry);
```

... and have the task execute that code where it has access to the data

Other ways to pass parameters

- Another option is to use lambda expressions to provide parameters inline

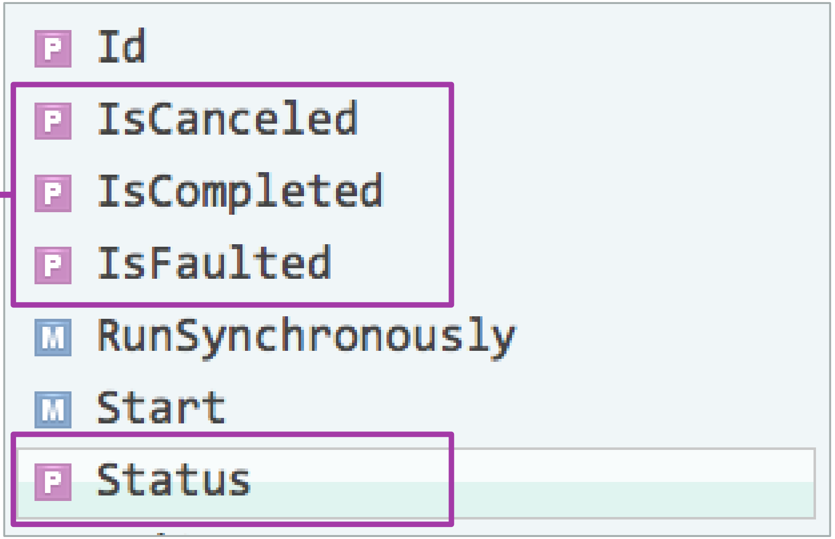
```
string greeting = editText.Text;

Task theTask = Task.Run(() => {
    // Can access local variables inli
    WriteToLog(greeting);
}
```

Be aware that this will *capture these variables* into a compiler-generated class to ensure they stay alive throughout the whole task's lifetime

Detecting when the task is finished

- Often need to know when the task work is finished in order to update the UI or use data produced by the task; **`Task`** class has properties to indicate the final completion state

```
Task work = Task.Run(DoWork);
// ... Time passes
if (work.IsCompleted) {
   // ... Report that work
   // is complete
}
```



P Id
P IsCanceled
P IsCompleted
P IsFaulted
M RunSynchronously
M Start
P Status

Waiting for a Task to finish

- Can use **Wait** method to block our current thread until the **Task** is done

```
Task funTask = Task.Run(DoSnipeHunt) ;

... // Do other stuff until we've caught a snipe!

funTask.Wait();

// Task is complete, display results to user
```

Warning: this waits indefinitely for the task to finish .. If the task never finishes, the user will be very unhappy with your app

Waiting for a Task to finish

- Can provide a timeout to **`Wait`**; returns **true** if task completed within the timeout period, **false** if the task did not finish in time

```
Task funTask = Task.Run(DoSnipeHunt) ;

... // Do other stuff until we've caught a snipe!

bool taskDone = funTask.Wait(5000);
if (taskDone) {
   // Task is complete, display results to user
}
```

This is better, but still blocks the current thread for up to 5 seconds

Recall: async and await

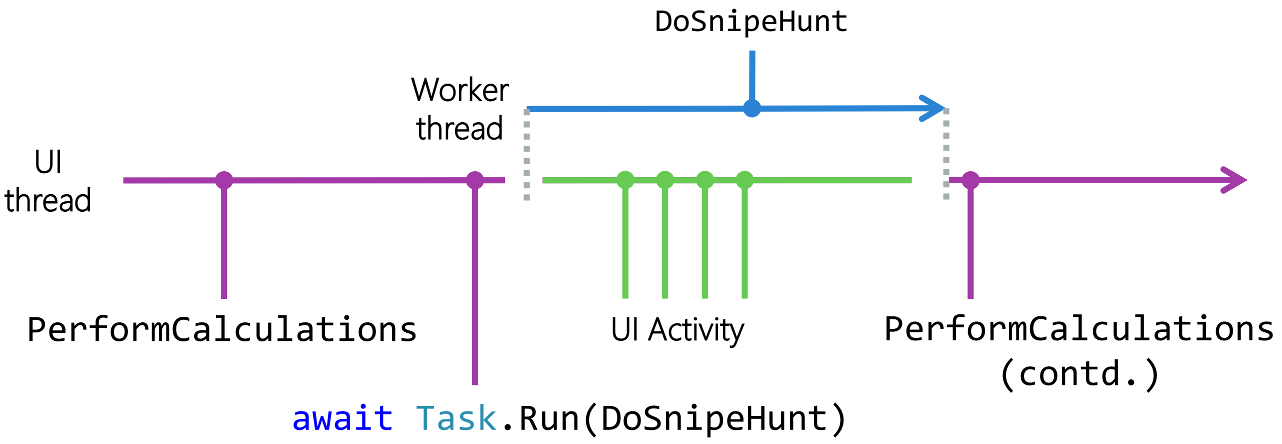
- Should prefer to use the **async** and **await** keywords to keep the current (UI) thread responsive to input

```
async Task FunActivityForKids()
{
   Task funTask = Task.Run(DoSnipeHunt);
   ... // Do other stuff until we've caught a snipe!
   await funTask;
   // Task is finished..
}
```

This stops forward execution of *this* method, but allows the current thread to return back to the caller until the **Task** is finished

Execution progress for await

- At runtime, the **`await`** keyword starts the **`Task`** and then **returns to the caller** … then when the **`Task`** is finished, the runtime will **return to the method where it left off** to continue execution



The `async` and `await` keywords are covered in CSC350

Getting results from a Task

- Tasks have access to the memory of your process – can alter data as they run which can then be displayed once the task is finished

```
class Program
{
   public string Digits;
   public void CalcPiFor10KDigits()
   {
      ...
      Digits = result;
   }
}
```

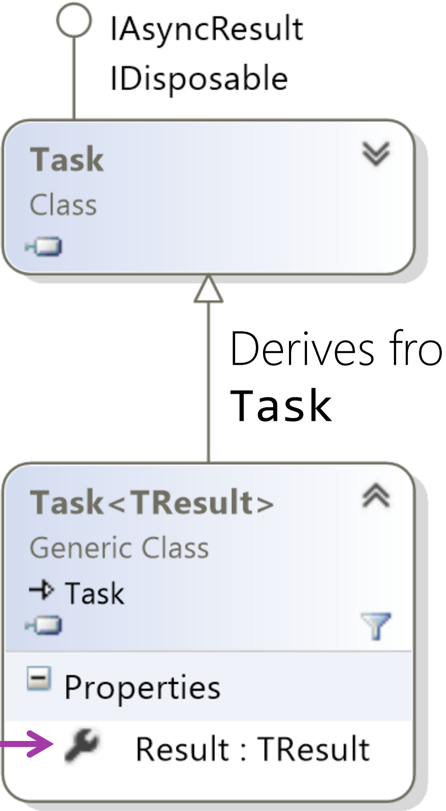
The downside to this approach is that it requires two threads to share this value which must be synchronized

```
await Task.Run(CalcPiFor10kDigits);
string result = this.Digits;
```

Getting a result from a Task

- **`Task<T>`** supports returning a single value, called a *future* or *promise*, from the delegate worker

- This is the preferred approach because it does not require any synchronization



Adds a **`Result`** property of type **`<T>`** to retrieve a single result value from the task

Getting a result from a Task

- `Task.Run` has overload that uses **`Func<TR>`** as the worker delegate

```
Task<byte[]> t1 = Task.Run(new Func<byte[]>(ReverseWavFile));
...
byte[] reversedWavData = t1.Result;
```

Get the return value using the **`Result`** property; this will *block* until the task completes and the result is available

```
byte[] ReverseWavFile() { ... }
```

Getting a result from a Task

- **`Task.Run`** has overload that uses **`Func<TR>`** as the worker delegate

```
byte[] reversedWavData = await Task.Run(ReverseWavFile);
```

Can use `await` to efficiently wait without blocking for the task to produce a value

```
byte[] ReverseWavFile() { ... }
```

Getting a result from a Task

- **Task.Run** has overload that uses **Func<TR>** as the worker delegate

```
byte[] reversedWavData = await Task.Run(ReverseWavFile);
```

await also unpacks the result – so what we get back is the value from the **Result** property, not the task itself

```
byte[] ReverseWavFile() { ... }
```

Individual Exercise

Partition work over multiple threads

Summary

1. Creating Threads and Tasks
2. Waiting for tasks to finish
3. Processing results from tasks

Dealing with exceptions

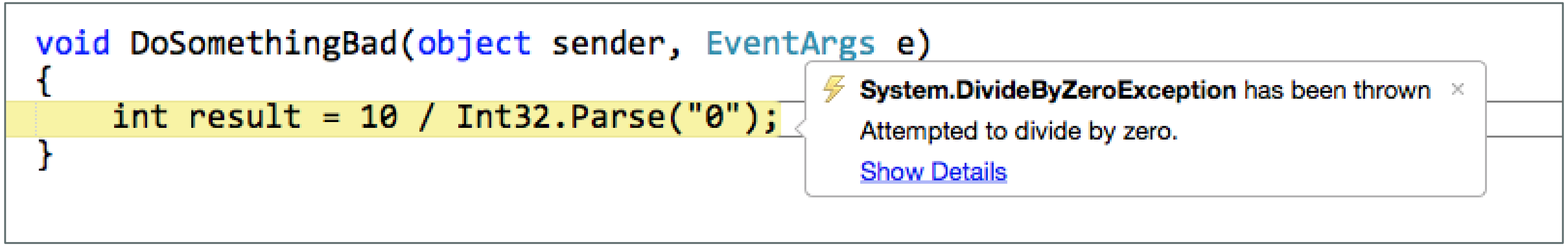
Xamarin University

Tasks

1. Dealing with exceptions
2. What is an `AggregateException`?
3. Catching task exceptions

Uncaught exceptions

- Normally, if an exception is not caught by your code, it will terminate the application immediately – this is the most common cause for a "crash"



```
void DoSomethingBad(object sender, EventArgs e)
{
    int result = 10 / Int32.Parse("0");
}
```

System.DivideByZeroException has been thrown

Attempted to divide by zero.

Debuggers are good at identifying these when they occur

Exceptions in Tasks

- When an uncaught exception occurs in a **Task**, something interesting happens

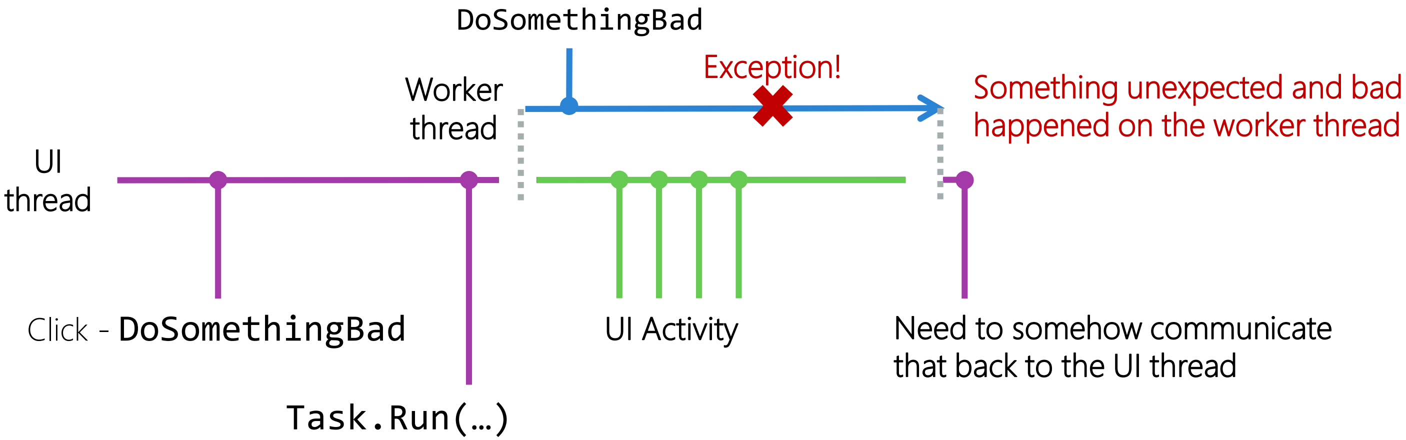
```
void DoSomethingBad(object sender, EventArgs e)
{
    Task.Run(() => {
        int result = 10 / Int32.Parse("0");
    });
}
```

Same code .. no **try**/**catch** ..

What should happen here

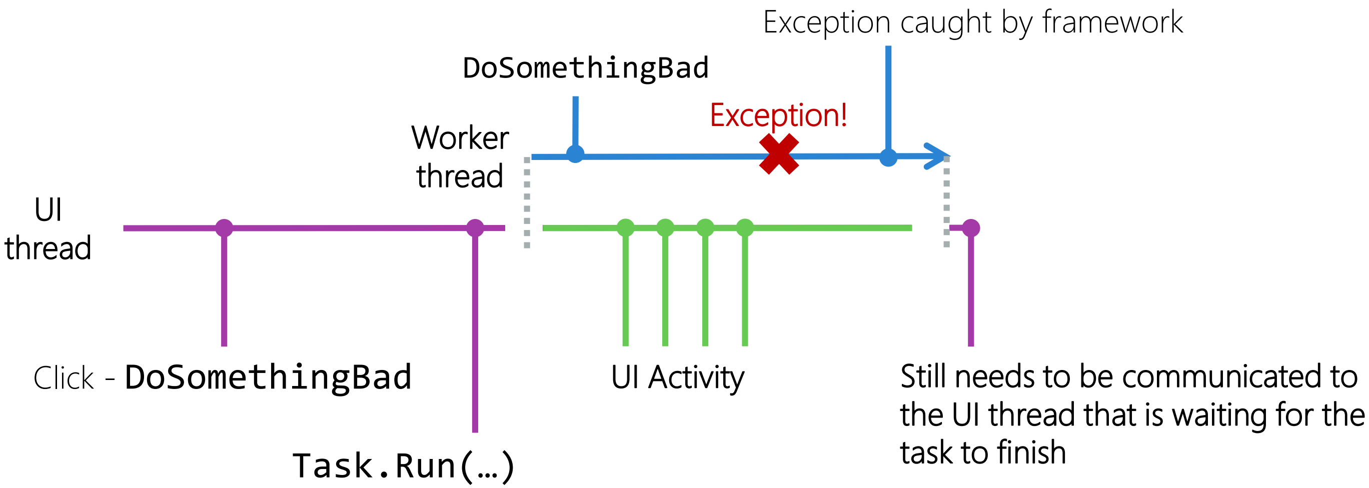
Flowing exceptions across threads

- Unhandled exceptions that occur in tasks present a unique challenge for processing – how do we communicate unexpected things properly?


DoSomethingBad
Worker thread
Exception!
Something unexpected and bad happened on the worker thread
UI thread
Click - DoSomethingBad
Task.Run(…)
UI Activity
Need to somehow communicate that back to the UI thread

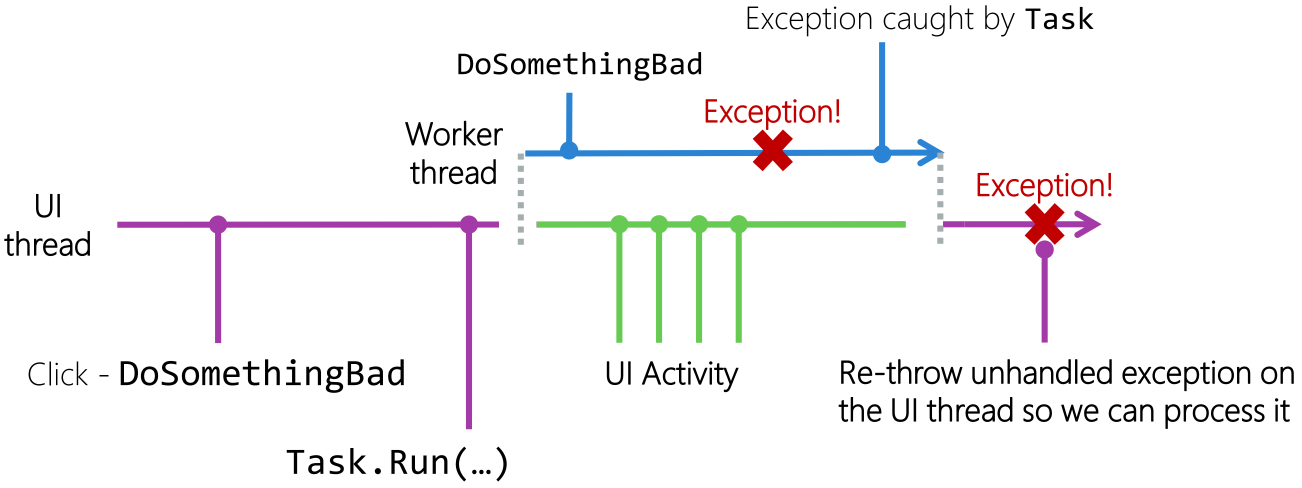
Flowing exceptions across threads

- To provide for this case, uncaught exceptions inside a **`Task`** are *always* caught by the framework


Exception caught by framework
DoSomethingBad
Exception!
Worker thread
UI thread
Click - DoSomethingBad
Task.Run(…)
UI Activity
Still needs to be communicated to the UI thread that is waiting for the task to finish

Exceptions

- To provide for this case, uncaught exceptions inside a **`Task`** are *always* caught by the framework; which will then *re-throw* the exception when the result is accessed



Handling task-based exceptions

- Should wrap waits/results in **try/catch** to ensure exception doesn't turn into an uncaught case that terminates your application

```
Task<double> badTask = Task.Run<double>(DoSomethingBad);
double calculatedValue = 0;
try {
   calculatedValue = badTask.Result;
}
catch (DivideByZeroException ex)
{
   ...
}
```

But this won't quite work the way you expect

Catching task exceptions

- Should wrap waits/results in try/catch to ensure exception doesn't turn into an uncaught case that terminates your application

```
Task<double> badTask = Task.Run<double>(DoSomethingBad);
double calculatedValue = 0;
try {
   calculatedValue = badTask.Result;
}
catch (AggregateException ex)
{
   ...
}
```

Tasks always throw **`AggregateException`** type which holds the "real" exception that was thrown

What is an AggregateException?

- **`AggregateException`** is an exception type that represents one or more errors that occurred in one or more tasks

```
try {
   ...
}
catch (AggregateException x)
{
    foreach (var ex in x.InnerExceptions)
    {
        // Process each exception
    }
}
```

Property exposes collection of exceptions with at least one entry

Getting the real exception

- For most cases, there will only be a single entry in the collection and you can simply pull it out by index to process

```
catch (AggregateException x)
{
    var ex = x.InnerExceptions[0];
    if (ex is DivideByZeroException)
    {
        // ...
    }
}
```

Catching task exceptions

- Again, **async** and **await** simplify asynchronous code by re-throwing the single exception directly

```
double calculatedValue = 0;
try {
   calculatedValue = await Task.Run<double>(DoSomethingBad);
}
catch (DivideByZeroException ex)
{
   ... // Caught!
}
```

Await automatically unwraps the first exception in the `AggregateException`

Xamarin University

Fire and forget Tasks

- Sometimes it's useful to start tasks which you don't need to wait on, or collect any result from – often referred to as "fire-and-forget" tasks

```
Task.Run(() => DoBackgroundSpellcheck);
```

```
void DoBackgroundSpellcheck()
{
    foreach (var line in Lines)
    {
        int start, end;
        if (CheckSpelling(line, out start, out end))
        {
            HighlightLine(start, end);
        }
    }
}
```

What if an exception happens in this code?

Fire and forget Tasks

- Exceptions are re-thrown when you call **`await`**, **`Wait`** or access the **`Result`** property on a **`Task`** – if you never do any of these, the exception is **unobserved by your code!**

```
void DoBackgroundSpellcheck()
{
   try {
      foreach (var line in Lines) {
          int start, end;
          if (CheckSpelling(line, out start, out end))
              HighlightLine(start, end);
      }
   }
   catch (Exception ex) { ... }
}
```

Should always catch all exceptions in "fire-and-forget" methods

Detect unobserved exceptions

- To make sure you don't have unobserved exceptions, can wire up to a static event handler that is raised by the GC when a `Task` is collected with an unobserved exception

```
#if DEBUG
TaskScheduler.UnobservedTaskException += (sender, e) => {
   Debug.WriteLine(
       $"Warning: unobserved Task exception - {e.Exception}");
};
#endif
```

Individual Exercise

Handle Exceptions that occur in Tasks

Xamarin University

Summary

1. Dealing with exceptions
2. What is an **`AggregateException`**?
3. Catching task exceptions

Cancelling tasks

Xamarin University

Tasks

1. Task completion states
2. Using a cancellation token
3. Signaling cancellation
4. Detecting cancellation

Task completion state

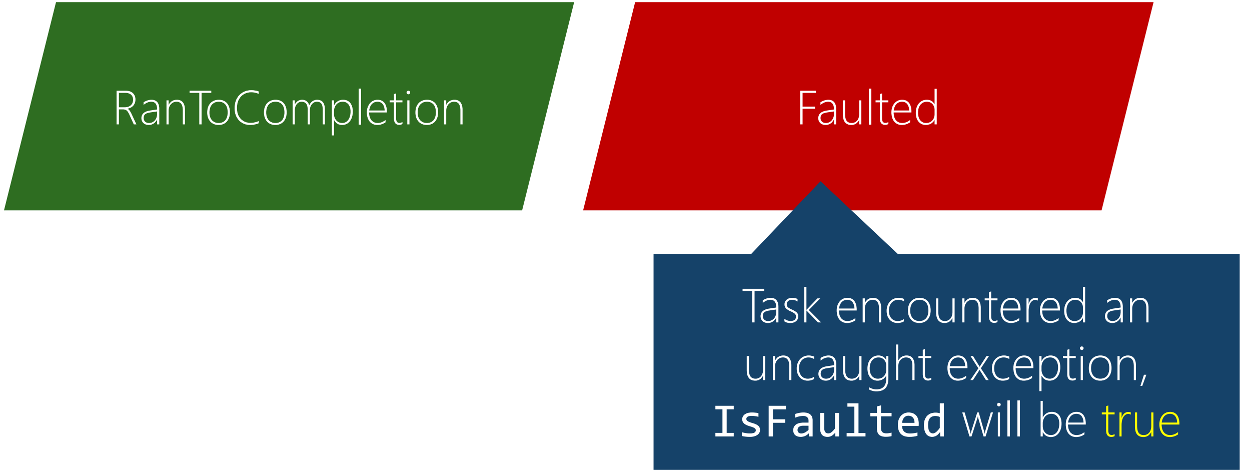
- Tasks can end in one of three different states; specific flags on the **`Task`** indicate the final completion status

RanToCompletion

Task completed normally, **`IsCompleted`** will be true

Task completion state

- Tasks can end in one of three different states; specific flags on the **`Task`** indicate the final completion status



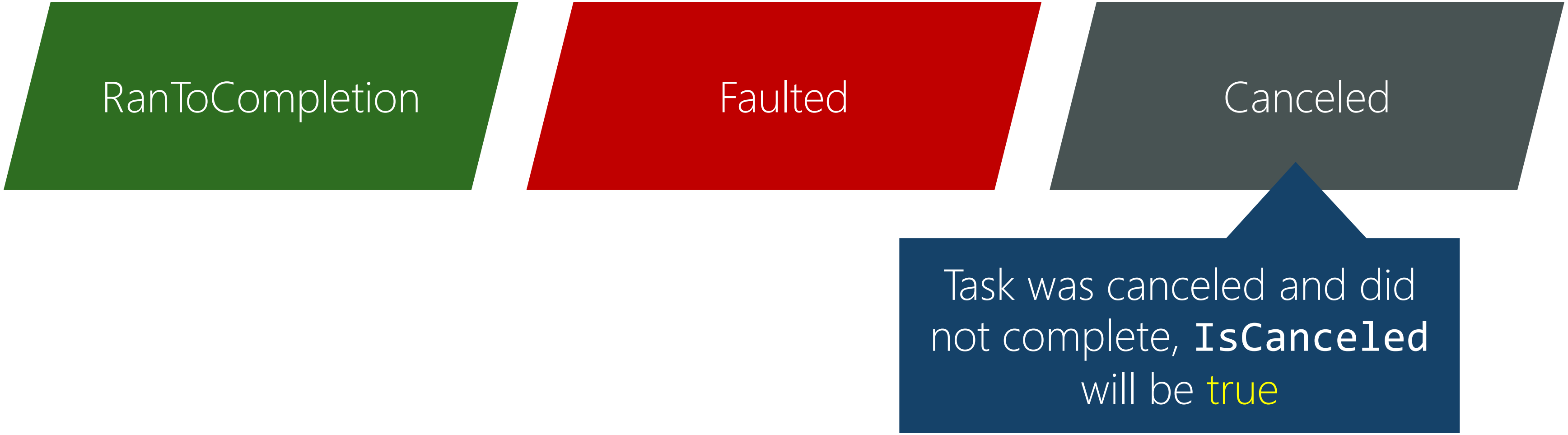
RanToCompletion

Faulted

Task encountered an uncaught exception, **`IsFaulted`** will be true

Task completion state

- Tasks can end in one of three different states; specific flags on the `Task` indicate the final completion status



RanToCompletion

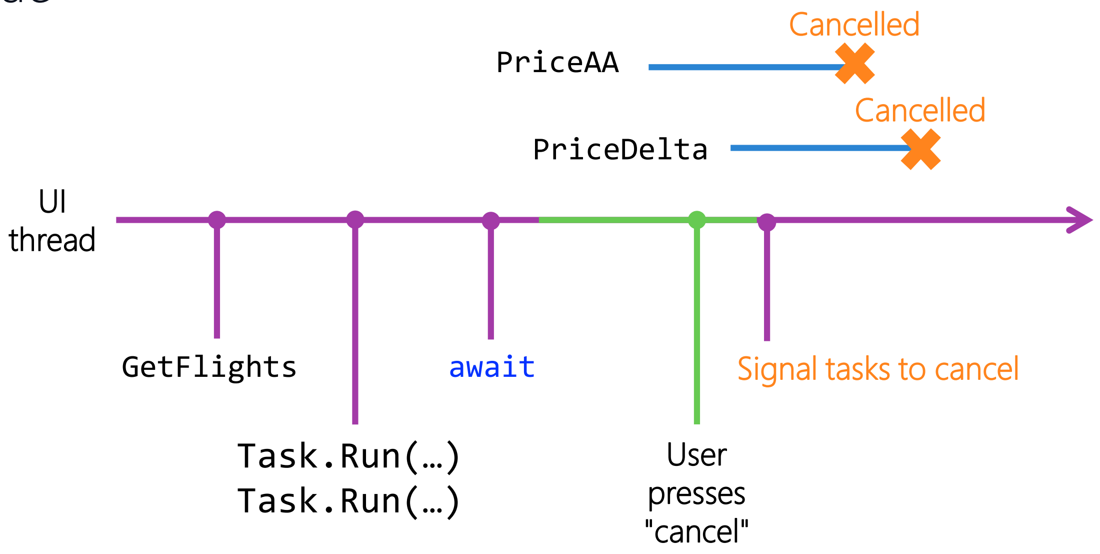
Faulted

Canceled

Task was canceled and did not complete, **`IsCanceled`** will be true

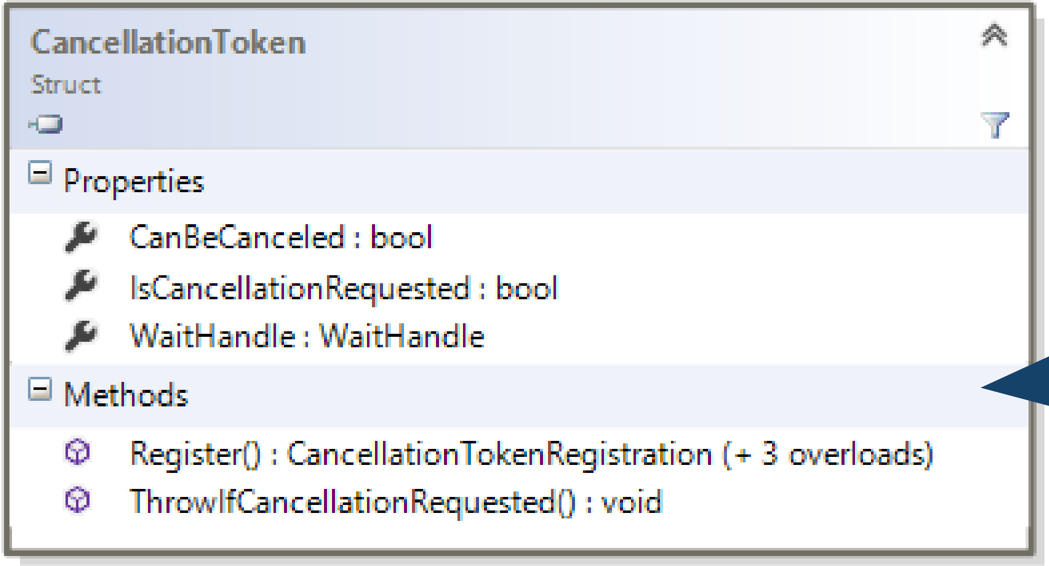
What is cancellation?

- Cancelling a task indicates that the consumer of the task is no longer interested in the results and therefore does not want the task to continue



Cancelling Tasks

- The **Task** API has explicit opt-in cancellation support provided by a separate cancellation API controlled by two classes



CancellationToken
Struct

Properties

- CanBeCanceled : bool
- IsCancellationRequested : bool
- WaitHandle : WaitHandle

Methods

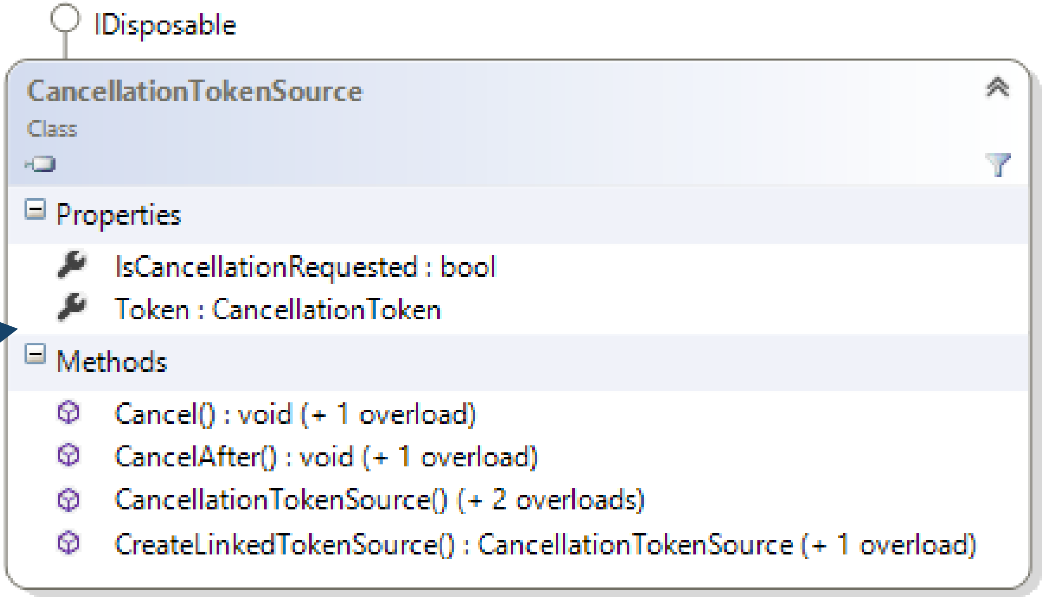
- Register() : CancellationTokenRegistration (+ 3 overloads)
- ThrowIfCancellationRequested() : void

A structure that represents a "potential" request for cancellation

Cancelling Tasks

- The **`Task`** API has explicit opt-in cancellation support provided by a separate cancellation API controlled by two classes

Provides the mechanism for initiating a cancellation request and signaling any monitoring party



IDisposable

CancellationTokenSource
Class

⊟ Properties

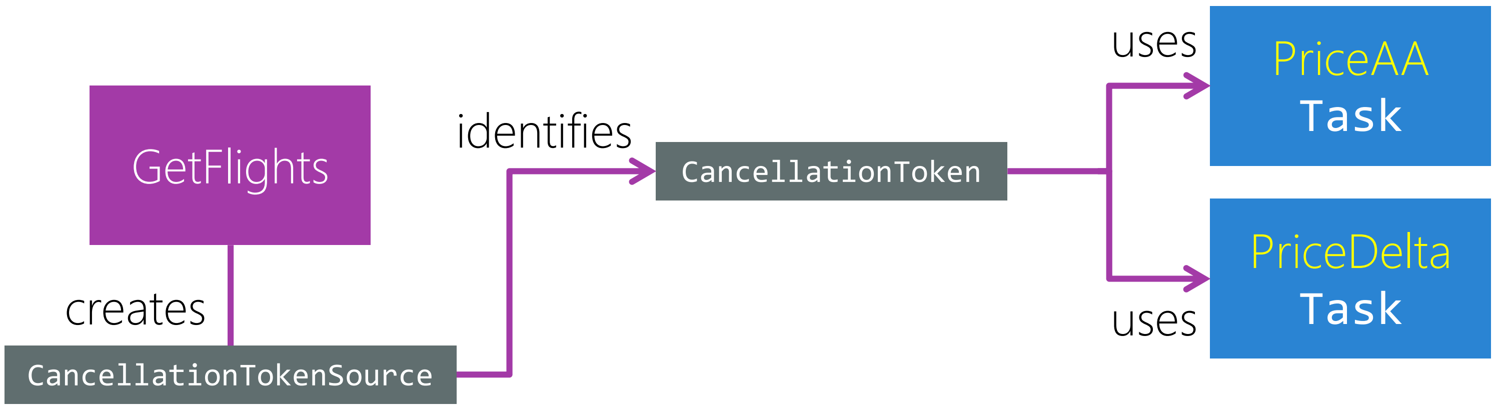
IsCancellationRequested : bool
Token : CancellationToken

⊟ Methods

Cancel() : void (+ 1 overload)
CancelAfter() : void (+ 1 overload)
CancellationTokenSource() (+ 2 overloads)
CreateLinkedTokenSource() : CancellationTokenSource (+ 1 overload)

What is a Cancellation Token?

- **`CancellationToken`** is created by the **`CancellationTokenSource`** and is then *shared with each task* that will support cancellation requests



Using a CancellationToken

- All of the task-creation mechanisms have an override that takes a `CancellationToken`

```
CancellationTokenSource cts = new CancellationTokenSource();

var t1 = Task.Run(DoCalculation, cts.Token);
...
var t2 = new Task(DoCalculation, cts.Token);
```

Signaling cancelation

- Use the **`Cancel`** method on the **`CancellationTokenSource`** to trigger a cancelation request

```
CancellationTokenSource cts = new CancellationTokenSource();

var t1 = Task.Run(DoCalculation, cts.Token);

...

cts.Cancel();
```

If the token is signaled *before* the task is scheduled, then it is immediately canceled without execution

Canceling a running Task

- If the token is signaled *while* the task is running, then your worker code must **opt-in to the cancelation support**

```
double DoCalculation(object parameter) {
    CancellationToken tok = (CancellationToken)parameter;
    for (int i = 0; i < 1000000; i++)
    {
        ...
    }
}
```

Must provide access to the token somehow – common to pass into the method

Canceling a running Task

- If the token is signaled *while* the task is running, then your worker code must **opt-in to the cancelation support**

```
double DoCalculation(object parameter) {
   CancellationToken tok = (CancellationToken)parameter;
   for (int i = 0; i < 1000000; i++)
   {
       ...
   }
}
```

Must somehow check if cancellation has been requested

Canceling a running Task

- If the token is signaled *while* the task is running, then your worker code must **opt-in to the cancelation support**

```
double DoCalculation(object parameter) {
   CancellationToken tok = (CancellationToken)parameter;
   for (int i = 0; i < 1000000; i++)
   {
      if (tok.IsCancellationRequested) {
         ... // What do we do now?
      }
      ...
   }
```

Running tasks are **never** canceled automatically, only direct user code in the worker can cause the task to be canceled

Canceling a running Task

- If the token is signaled *while* the task is running, then your worker code must **opt-in to the cancelation support**

```
double DoCalculation(object parameter) {
    CancellationToken tok = (CancellationToken)parameter;
    for (int i = 0; i < 1000000; i++)
    {
        if (tok.IsCancell
            return -1;
        }
        ...
    }
}
```

Could return an illegal value – but does this really say it was cancelled?

Canceling a running Task

- If the token is signaled *while* the task is running, then your worker code must **opt-in to the cancelation support**

```
double DoCalculation(object parameter) {
   CancellationToken tok = (CancellationToken)parameter;
   for (int i = 0; i < 1000000; i++)
   {
      if (tok.IsCancellationRequested) {
          throw new TaskCanceledException();
      }
      ...
   }
}
```

Must throw special exception which is treated as a cancellation by the task

Canceling a running Task

- If the token is signaled *while* the task is running, then your worker code must **opt-in to the cancelation support**

```
double DoCalculation(object parameter) {
   CancellationToken tok = (CancellationToken)parameter;
   for (int i = 0; i < 1000000; i++)
   {
      tok.ThrowIfCancellationRequested();
      ...
   }
}
```

Helper method does the check *and* throw if cancellation is requested

How does the client detect cancellation?

- Any **await** on the task will be notified through a specific exception to distinguish success vs. failure vs. cancellation

```
try
{
   calculatedValue =  await Task.Run(...);
   ... // Process results
}
catch (TaskCanceledException ex)
{
   ... // Task was cancelled
}
```

How does the client detect cancellation?

- Alternatively, if the task was not awaited, the client can detect that the task was cancelled successfully through a **task property**

Only consume the result if the task was successful

```
Task theTask = ...;
if (!theTask.IsCanceled && !theTask.IsFaulted)
{
    calculatedValue = t1.Result;
}
```

Individual Exercise

Use cancellation tokens to stop long running operations

Summary

1. Task completion states
2. Using a cancellation token
3. Signaling cancellation
4. Detecting cancellation

Where are we going from here?

- The Task Parallel Library combined with the async and await keywords makes writing asynchronous code much easier

- However, there are potential problems we introduce into our applications when using threads – we'll look at those next in CSC352!

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile

Microsoft