Xamarin University

IOS210

# Backgrounding: Running Finite-Length Tasks

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Microsoft or Xamarin may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Microsoft or Xamarin, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

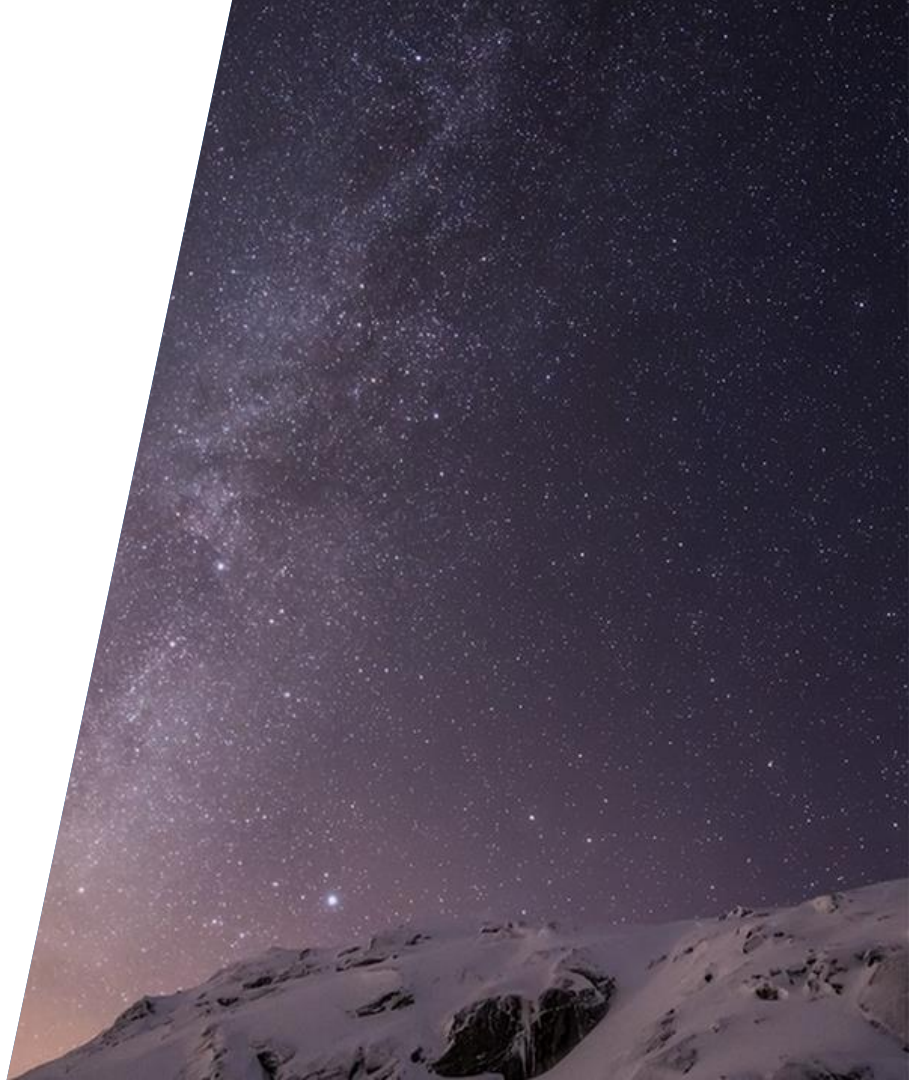


Xamarin, MonoTouch, MonoDroid, Xamarin.iOS, Xamarin.Android, Xamarin Studio, and Visual Studio are either registered trademarks or trademarks of Microsoft in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

# Objectives

1. Understand the iOS Backgrounding Model
2. Work with Finite-Length Tasks



# Understand the iOS Backgrounding Model

Xamarin University

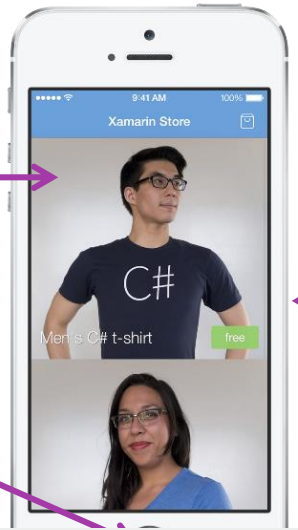
# Tasks

1. What is an application state?
2. Foreground vs. Background
3. Discuss three ways to execute code in the background

# Motivation

- By default, iOS apps do not get CPU time once they are no longer the foreground app

1. User is ordering a T-shirt
2. User presses home button or switches apps
3. App given a short time to finish its work (~5 seconds), then is **suspended** and gets no more CPU time

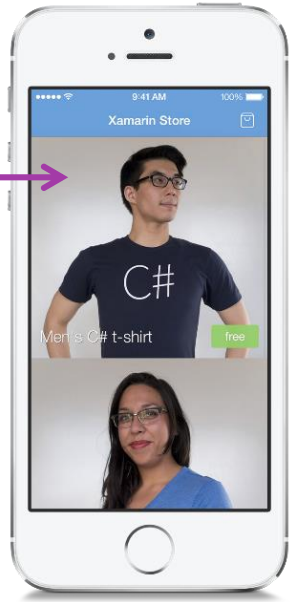



When an app is suspended, not only the UI main thread is affected **but all threads will be suspended**. Creating a thread will not let you execute code while backgrounded.

# What is an application state?

- An *application state* is the set of resources currently granted to an app by iOS (memory, CPU time, event delivery)

Visible app always receives CPU time and user events




In memory, but no CPU time or events

CPU time but no events (e.g. no user interaction)

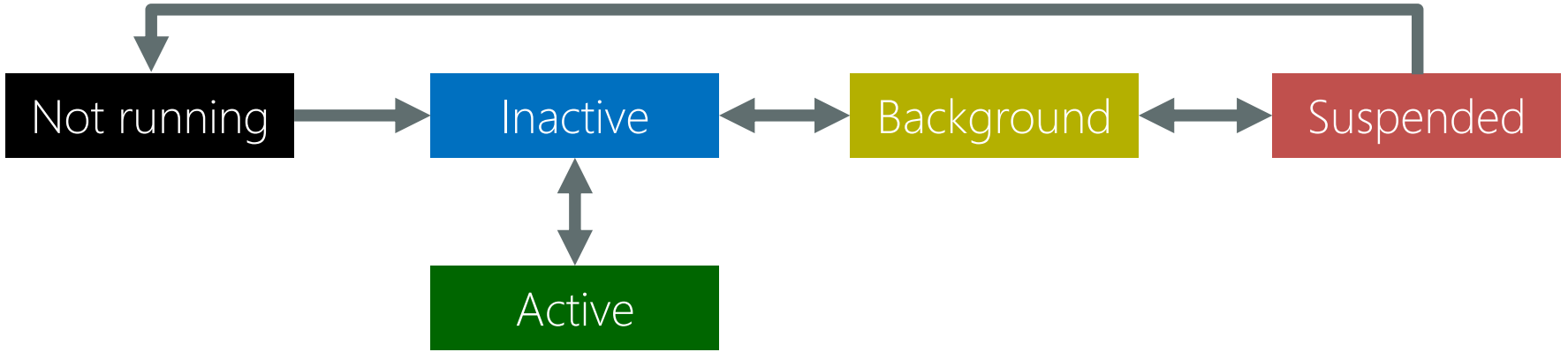
# Application states

- An iOS app can be in one of five states

| | |
|---|---|
| Not running | Not yet launched or already terminated |
| Inactive | Running but not receiving events |
| Active | Running normally |
| Background | Executing code in the background |
| Suspended | In memory but not executing code |

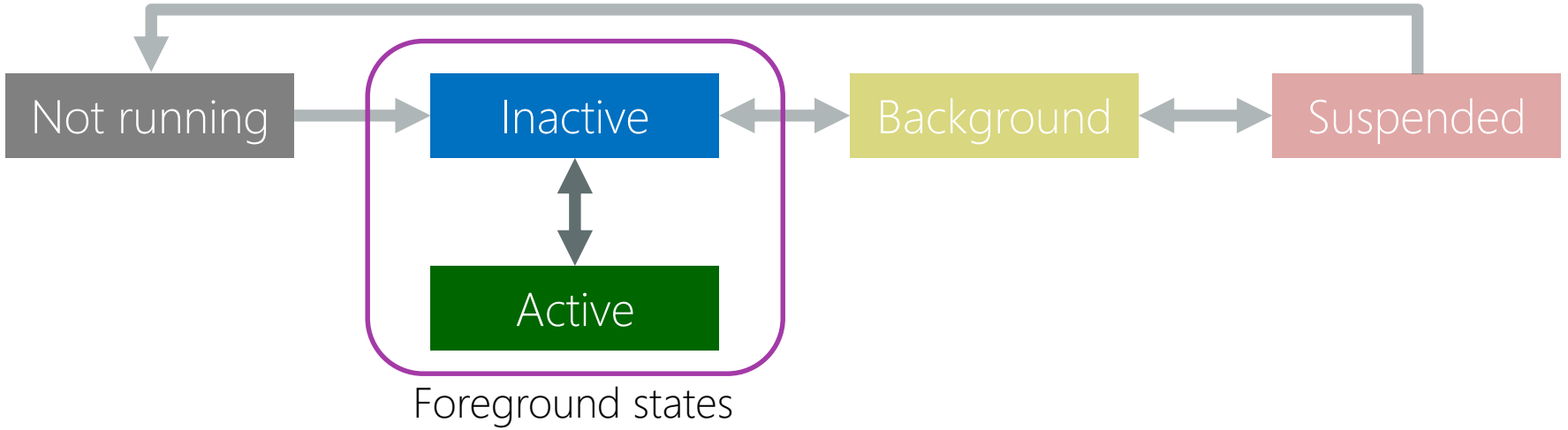
# What is the application lifecycle?

- The *application lifecycle* defines how an iOS app transitions between states in response to user actions and system events



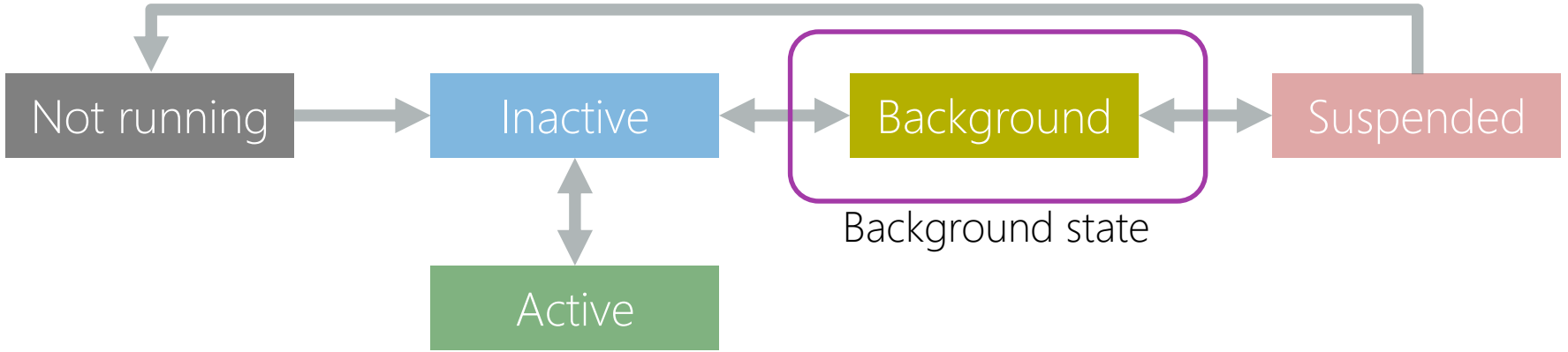
# What is the foreground app?

- The *foreground app* is the app the user is currently working with. Only one app will be in the foreground at any time.



# What is a backgrounded app?

- A *backgrounded app* runs code while in the background state. Multiple applications can be backgrounded at the same time.



# iOS backgrounding options

- iOS has three ways for apps to do work in the background

| Finite-Length Tasks | Long-Running Tasks | Background Transfers |
| --- | --- | --- |
| Any code, but time limited | Only for specific tasks, not time limited | Data transfer, not time limited |

# iOS backgrounding philosophy

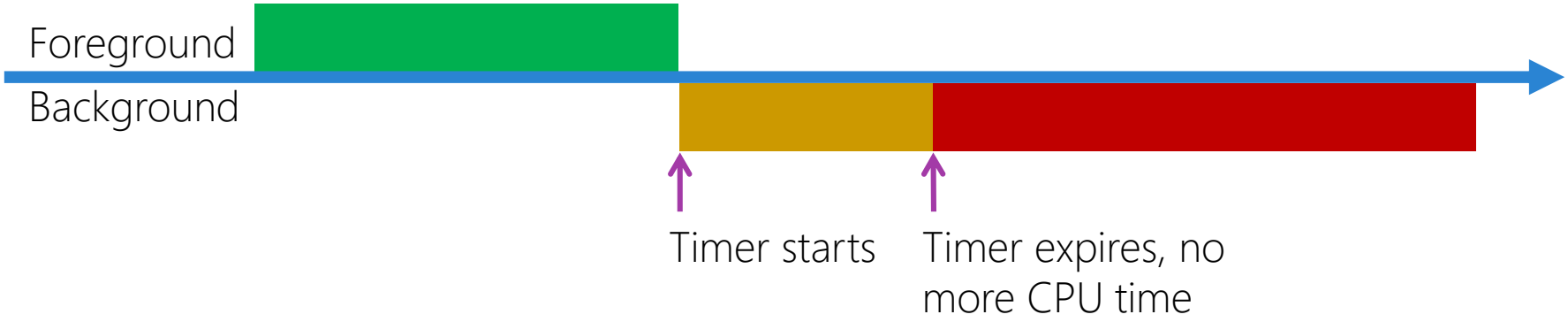
- iOS limits what an app can do in the background in order to conserve battery and dedicate CPU time to the foreground app

iOS tries to conserve resources since mobile hardware is optimized for low power use, not raw performance:

- Constrained memory
- No swap file
- Limited CPU performance

# Finite-Length Tasks

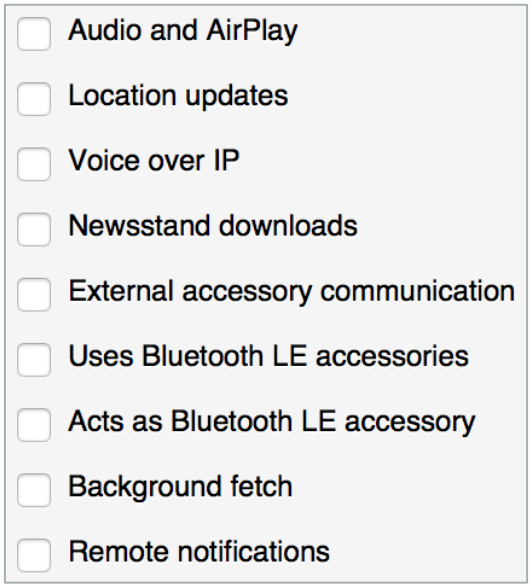
- Finite-Length Tasks let you run arbitrary code (i.e. they are not limited to specific operation types), but you have limited time



# Long-Running Tasks

- Long-Running Tasks let you execute specific operations without time restrictions even if the app is backgrounded

Only allowed for these operations →



# Background Transfer

- *Background Transfer* lets you transfer files in a separate process that continues not only if your app is suspended, but also if it is terminated

App notified when transfer completes, even if it was terminated

# Flash Quiz

Xamarin University

# Flash Quiz

① Which iOS background technique should you use to transfer the contents of a catalogue of a shopping app in order to support offline usage?

a) Finite-Length Task
b) Background Transfer
c) Long-Running Task

# Flash Quiz

① Which iOS background technique should you use to transfer the contents of a catalogue of a shopping app in order to support offline usage?

a) Finite-Length Task

b) Background Transfer

c) Long-Running Task

# Flash Quiz

② Which iOS background technique should you use to encrypt and store user settings when the app gets backgrounded?

a) Finite-Length Task

b) Background Download

c) Long-Running Task

# Flash Quiz

② Which iOS background technique should you use to encrypt and store user settings when the app gets backgrounded?

a) Finite-Length Task

b) Background Download

c) Long-Running Task

# Flash Quiz

③ Which iOS background technique should you use to track the device's current location?

a) Finite-Length Task

b) Background Download

c) Long-Running Task

# Flash Quiz

③ Which iOS background technique should you use to track the device's current location?

a) Finite-Length Task

b) Background Download

c) Long-Running Task

# Summary

1. Decide which of the three iOS backgrounding techniques is appropriate for your app

# Work with Finite-Length Tasks

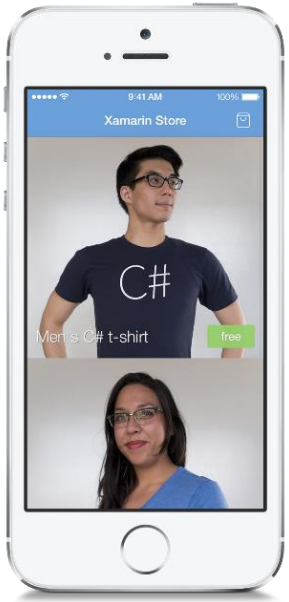
# Tasks

1. Integrate a Finite-Length Task with the Application Lifecycle
2. Wrap a critical operation in a Finite-Length Task

# Motivation: Saving State

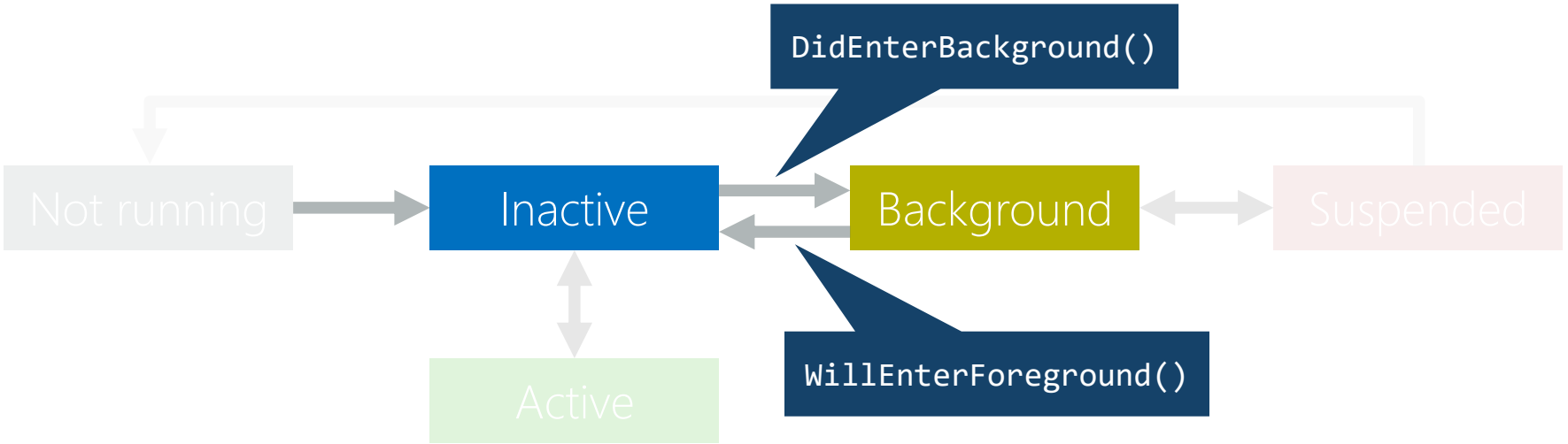
- Apps often need to **save state** when entering the background to preserve user choices, release expensive resources, etc.

User is ordering a T-shirt; when the app moves to the background, it should save their choices



# Lifecycle methods

- **`AppDelegate`** notifies you when transitioning between application states - two methods can be useful for background execution



# Strict time limits

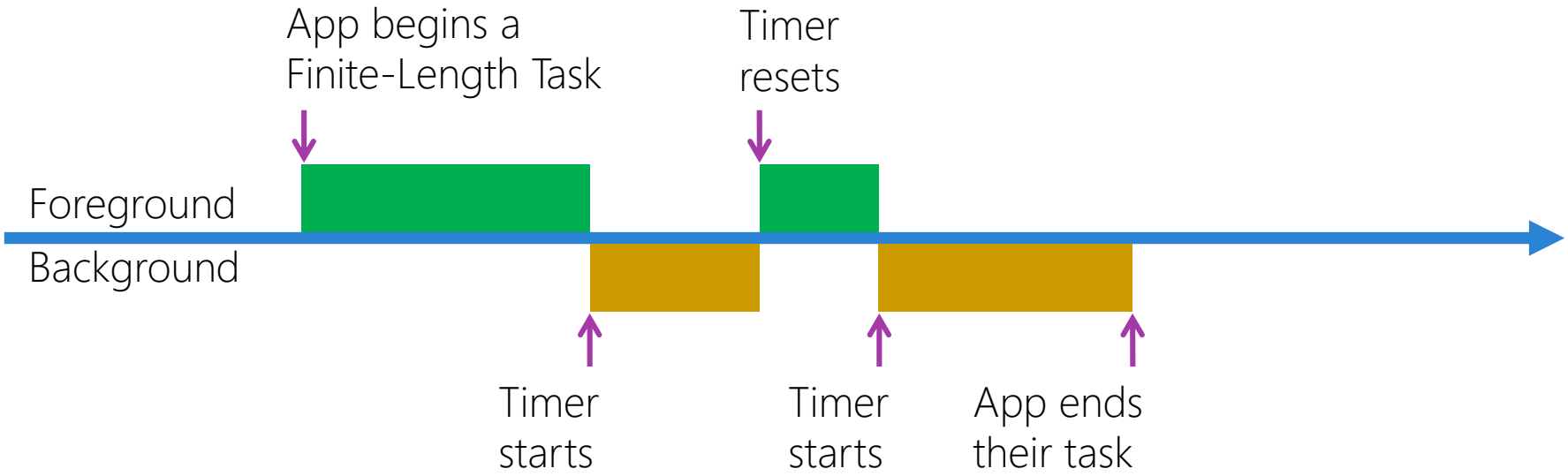
- `DidEnterBackground` must return within ~5 seconds – if it takes any longer, iOS will terminate the app

```
public override void DidEnterBackground(UIApplication app)
{
    // Save app state
    SaveUserChoices();
    ...
}
```

Risky to save app state like this – what if it takes longer than expected?

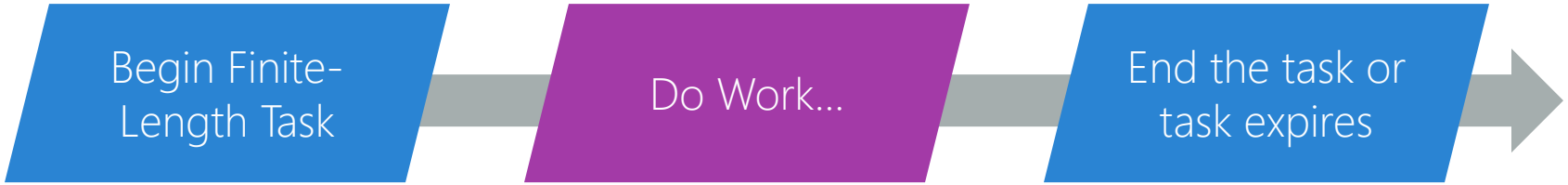
# What is a Finite-Length Task?

- A *Finite-Length Task* lets you do arbitrary work for a **limited time** after your app enters the background (time limit may differ in iOS versions)



# Save state with a Finite-Length Task

- Using a Finite-Length Task gives more time (up to a maximum limit) to complete the operation and involves three stages



# Finite-Length Task API

- API to manage Finite-Length Tasks is provided by **`UIApplication`**

Called by iOS just before timer expires,
this is not the work you need to do

```
nint BeginBackgroundTask(Action expirationHandler);

void EndBackgroundTask(nint taskId);

double BackgroundTimeRemaining { get; }
```



Note that even though we are using the term *task* here, it is not related to the .NET **`System.Threading.Task`** in any way!

# Starting a Finite-Length Task

- Start a task using **`UIApplication.BeginBackgroundTask`** method – this returns a unique task identifier, the timer starts once the app is backgrounded

```
nint taskId = -1;

public override void DidEnterBackground(UIApplication app)
{
  taskId = app.BeginBackgroundTask(OnExpiration);
  // Save app state
  SaveUserChoices();
}
```

# Doing work in the background

- Beginning a Finite-Length Task does not create a thread, app must create a non UI thread to run the operation

```
nint taskId = -1;
Task myWork;

public override void DidEnterBackground(UIApplication app)
{
  taskId = app.BeginBackgroundTask(OnExpiration);
  myWork = Task.Run(SaveUserChoices);
}
```

Start a new thread, then return from **DidEnterBackground**

# Checking the remaining time

- Can use the **`BackgroundTimeRemaining`** property to get the available time *in seconds* once a background task has been registered and the app is in the background

```
if (UIApplication.SharedApplication.BackgroundTimeRemaining < 10)
{
    ... // Not enough time – just end task and return
}
```

# Ending a Finite-Length Task

- You should end a finite-length task when it has completed or expired

App moves to foreground

Task completes

Task expires

# Ending a Finite-Length Task

- Should end the task if the **application transitions to the foreground**; if app is backgrounded again, task should be registered again

```
nint taskId = -1; // -1 means not running (our convention here)
public override void WillEnterForeground(UIApplication app)
{
  if (taskId != -1)
  {
     app.EndBackgroundTask(taskId);
     taskId = -1;
  }
}
```

Must pass in the unique task identifier returned from the call to **BeginBackgroundTask**

# Ending a Finite-Length Task

- If the **work completes while the app is backgrounded**, it should end the registered task to turn off the associated iOS timer

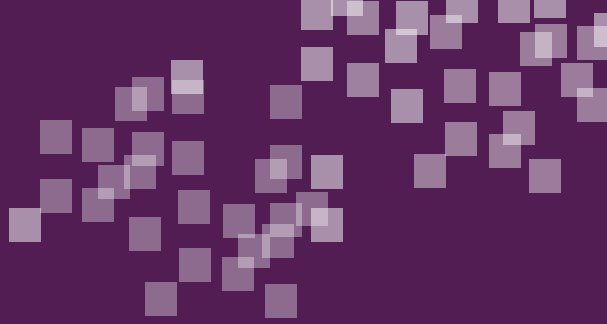
```
nint taskId = -1;
private void SaveUserChoices()
{
   ... // persist state (not shown)
   if (taskId != -1) {
      UIApplication.SharedApplication.EndBackgroundTask(taskId);
      taskId = -1;
   }
}
```

# Ending a Finite-Length Task

- When the **task timer expires**, iOS invokes the expiration callback – this *must* end the background task and signals that the application threads are about to be suspended

```
taskId = app.BeginBackgroundTask(OnExpiration);
```

```
nint taskId = -1;
void OnExpiration() {
    UIApplication.SharedApplication.EndBackgroundTask(taskId);
    taskId = -1;
    ...
}
```



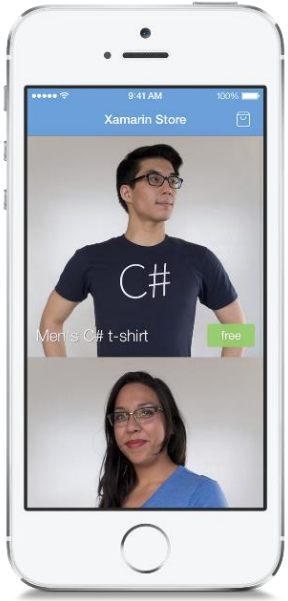
# Individual Exercise

Integrate a Finite-Length Task with the application lifecycle

Xamarin University

# Motivation: Finish application work

- May need to ensure operations complete even if app gets suspended




User clicks on "Buy", should try to finish the transaction even if the app moves to background

# Wrap code in a Finite-Length Task

- Typical to wrap a critical operation in a **Finite-Length Task** to ensure it will complete even if the app leaves the foreground

Begin →
End (success) →

```
async Task OnBuyShirtAsync()
{
  nint taskId = UIApplication.SharedApplication.BeginBackgroundTask(OnExpiration);

  await Task.Run(() => SubmitOrder());

  UIApplication.SharedApplication.EndBackgroundTask(taskId);
}
```

End (fail, out of time) →

```
void OnExpiration()
{
  UIApplication.SharedApplication.EndBackgroundTask(taskId);
  taskId = -1;
}
```

# Setting up for cancellation

- Can use .NET **CancellationToken** to signal that an asynchronous operation must be terminated

```
CancellationTokenSource cts = new CancellationTokenSource();
nint taskId = -1;

async Task OnBuyShirtAsync()
{
  this.taskId = UIApplication.SharedApplicat
    OnExpiration);

  await Task.Run(() => SubmitOrder(cts.Token), cts.Token);
  UIApplication.SharedApplication.EndBackgroundTask(taskId);
}
```

Tokens are created by a TokenSource which can be used to signal the token

# Setting up for cancellation

- Can use .NET **`CancellationToken`** to signal that an asynchronous operation must be terminated

```
CancellationTokenSource cts = new CancellationTokenSource();
nint taskId = -1;

async Task OnBuyShirtAsync()
{
  this.taskId = UIApplication.SharedApplicatio
    OnExpiration);

  await Task.Run(() => SubmitOrder(cts.Token), cts.Token);
  UIApplication.SharedApplication.EndBackgroundTask(taskId);
}
```

Provide access to the token so the async worker can watch for cancellation

# Setting up for cancellation

- Can use .NET **CancellationToken** to signal that an asynchronous operation must be terminated

```
CancellationTokenSource cts = new CancellationTokenSource();
nint taskId = -1;

async Task OnBuyShirtAsync()
{
  this.taskId = UIApplication.SharedApp                    sk(
    OnExpiration);

  await Task.Run(() => SubmitOrder(cts.Token), cts.Token);
  UIApplication.SharedApplication.EndBackgroundTask(taskId);
}
```

Should also pass the token to **Task.Run** – this allows the task to be cancelled if it has not been started yet

# Detecting cancellation

- Async code must check the passed token to see if it should stop

```
void SubmitOrder(CancellationToken token)
{
   ...
   if (token.IsCancellationRequested)
   {
      // Do cleanup
      return;
   }
   ...
   token.ThrowIfCancellationRequested();
}
```

Can check a boolean flag periodically in your code and cleanup + exit, the Task will finish successfully in this case

# Detecting cancellation

- Async code must check the passed token to see if it should stop

```
void SubmitOrder(CancellationToken token)
{
   ...
   if (token.IsCancellationRequested)
   {
      // Do cleanup
      return;
   }
   ...
   token.ThrowIfCancellationRequested();
}
```

can record the task itself as *cancelled* by throwing a special exception – this is raised at the call site as a `OperationCanceledException`

# Signaling cancellation

- Operation can be cancelled by the user, but also by the Finite-Length Task expiration handler

```
CancellationTokenSource cts = new CancellationTokenSource();
nint taskId = -1;
...

void OnExpiration()
{
   cts.Cancel();
   UIApplication.SharedApplication.EndBackgroundTask(taskId);
   taskId = -1;
}
```

# Individual Exercise

Code a cancellable Finite-Length Task

Xamarin University

# Flash Quiz

Xamarin University

# Flash Quiz

① The remaining time available for a Finite-Length Task to finish is always _____.

a) 10 minutes
b) 3 minutes
c) different
d) available through `UIApplication.BackgroundTimeRemaining` property

# Flash Quiz

① The remaining time available for a Finite-Length Task to finish is always _____.

a) 10 minutes
b) 3 minutes
c) different
d) available through **`UIApplication.BackgroundTimeRemaining`** property

# Flash Quiz

② Calling `UIApplication.BeginBackgroundTask(OnDone)`…
[select all that apply]

a) starts a thread and executes `OnDone()` asynchronously

b) signals the operating system that it should not suspend currently running threads when backgrounded

c) triggers `OnDone()` if the available time for background operations is about to expire

# Flash Quiz

② Calling **`UIApplication.BeginBackgroundTask(OnDone)`**…
[select all that apply]

a) starts a thread and executes **`OnDone()`** asynchronously

b) signals the operating system that it should not suspend currently running threads when backgrounded

c) triggers **`OnDone()`** if the available time for background operations is about to expire

# Summary

1. Integrate a Finite-Length Task with the Application Lifecycle
2. Wrap a critical operation in a Finite-Length Task

# More backgrounding is iOS

- iOS211 continues this topic by exploring:
  - Long-Running Tasks without time limits with Background Modes
  - Transferring files in the background

# Thank You!

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