AKKA.NET

BOOTCAMP

Petabridge

Table of Contents

Akka.NET Bootcamp

Welcome to Akka.NET Bootcamp! This is a free, self-directed learning course brought to you by the folks at Petabridge.

GITTER JOIN CHAT →

Over the three units of this bootcamp you will learn how to create fully-functional, real-world programs using Akka.NET actors and many other parts of the core Akka.NET framework!

We will start with some basic actors and have you progressively work your way up to larger, more sophisticated examples.

The course is self-directed learning. You can do it at whatever pace you wish. You can sign up here to have one Akka.NET Bootcamp lesson emailed to you daily if you'd like a little help pacing yourself throughout the course.

> NOTE: F# support is in progress (see the FSharp branch). We will happily accept F# pull requests. Feel free to send them in.

What will you learn?

In Akka.NET Bootcamp you will learn how to use Akka.NET actors to build reactive, concurrent applications.

You will learn how to build types of applications that may have seemed impossible or really, really hard to make prior to learning Akka.NET. You will walk away from this bootcamp with the confidence to handle bigger and harder problems than ever before!

Unit 1

In Unit 1, we will learn the fundamentals of how the actor model and Akka.NET work.

*NIX systems have the `tail` command built-in to monitor changes to a file (such as tailing log files), whereas Windows does not. We will recreate `tail` for Windows and use the process to learn the fundamentals.

In Unit 1 you will learn:

1. How to create your own `ActorSystem` and actors;
2. How to send messages actors and how to handle different types of messages;
3. How to use `Props` and `IActorRef`s to build loosely coupled systems.
4. How to use actor paths, addresses, and `ActorSelection` to send messages to actors.
5. How to create child actors and actor hierarchies, and how to supervise children with `SupervisionStrategy`.
6. How to use the Actor lifecycle to control actor startup, shutdown, and restart behavior.

Begin Unit 1.

Unit 2

In Unit 2, we're going to get into some more of the intermediate Akka.NET features to build a more sophisticated application than what we accomplished at the end of unit 1.

In Unit 2 you will learn:

1. How to use HOCON configuration to configure your actors via App.config and Web.config;
2. How to configure your actor's Dispatcher to run on the Windows Forms UI thread, so actors can make operations directly on UI elements without needing to change contexts;
3. How to handle more sophisticated types of pattern matching using `ReceiveActor`;
4. How to use the `Scheduler` to send recurring messages to actors;
5. How to use the Publish-subscribe (pub-sub) pattern between actors;
6. How and why to switch actor's behavior at run-time; and
7. How to `Stash` messages for deferred processing.

Begin Unit 2.

Unit 3

In Unit 3, we will learn how to use actors for parallelism and scale-out using Octokit and data from Github repos!

In Unit 3 you will learn:

1. How to perform work asynchronously inside your actors using `PipeTo`;
2. How to use `Ask` to wait inline for actors to respond to your messages;
3. How to use `ReceiveTimeout` to time out replies from other actors;
4. How to use `Group` routers to divide work among your actors;
5. How to use `Pool` routers to automatically create and manage pools of actors; and
6. How to use HOCON to configure your routers.

Begin Unit 3.

How to get started

Here's how Akka.NET bootcamp works!

Use Github to Make Life Easy

This Github repository contains Visual Studio solution files and other assets you will need to complete the bootcamp.

Thus, if you want to follow the bootcamp we recommend doing the following:

1. Sign up for Github, if you haven't already.
2. Fork this repository and clone your fork to your local machine.
3. As you go through the project, keep a web browser tab open to the Akka.NET Bootcamp ReadMes so you can read all of the instructions clearly and easily.

Bootcamp Structure

Akka.NET Bootcamp consists of three modules:

- **Unit 1 - Beginning Akka.NET**
- **Unit 2 - Intermediate Akka.NET**

- **Unit 3 - Advanced Akka.NET**

Each module contains the following structure (using **Unit 1** as an example:)

```
src\Unit1\README.MD - table of contents and instructions for the module
src\Unit1\DoThis\ - contains the .SLN and project files that you will use through all lesson
-- lesson 1
src\Unit1\Lesson1\README.MD - README explaining lesson1
src\Unit1\Lesson1\DoThis\ - C# classes, images, text files, and other junk you'll need to co
src\Unit1\Lesson1\Completed\ - Got stuck on lesson1? This folder shows the "expected" output
-- repeat for all lessons
```

Start with the first lesson in each unit and follow the links through their README files on Github. We're going to begin with **Unit 1, Lesson 1**.

Lesson Layout

Each Akka.NET Bootcamp lesson contains a README which explains the following:

1. The Akka.NET concepts and tools you will be applying in the lesson, along with links to any relevant documentation or examples
2. Step-by-step instructions on how to modify the .NET project inside the `Unit-[Num]/DoThis/` to match the expected output at the end of each lesson.
3. If you get stuck following the step-by-step instructions, each lesson contains its own `/Completed/` folder that shows the full source code that will produce the expected output. You can compare this against your own code and see what you need to do differently.

When you're doing the lessons...

A few things to bear in mind when you're following the step-by-step instructions:

1. **Don't just copy and paste the code shown in the lesson's README**. You'll retain and learn all of the built-in Akka.NET functions if you type out the code as it's shown. Kinesthetic learning FTW!
2. **You might be required to fill in some blanks during individual lessons.** Part of helping you learn Akka.NET involves leaving some parts of the exercise up to you - if you ever feel lost, always check the contents of the `/Completed` folder for that

lesson.
3. **Don't be afraid to ask questions**. You can reach the Petabridge and Akka.NET teams in our Gitter chat here.

Docs

We will provide explanations of all key concepts throughout each lesson, but of course, you should bookmark (and feel free to use!) the Akka.NET docs.

Tools / prerequisites

This course expects the following:

- You have some programming experience and familiarity with C#
- A Github account and basic knowledge of Git.
- You are using a version of Visual Studio (it's free now!)
    - We haven't had a chance to test these in Xamarin / on Mono yet, but that will be coming soon. If you try them there, please let us know how it goes! We are planning on having everything on all platforms ASAP.

Enough talk, let's go!

Let's begin!

About Petabridge

Petabridge is a company dedicated to making it easier for .NET developers to build distributed applications.

Petabridge also offers Akka.NET consulting and training - so please sign up for our mailing list!

Akka.NET Bootcamp - Unit 1: Beginning Akka.NET

In Unit 1, we will learn the fundamentals of how the actor model and Akka.NET work.

Concepts you'll learn

*NIX systems have the `tail` command built-in to monitor changes to a file (such as tailing log files), whereas Windows does not. We will recreate `tail` for Windows, and use the process to learn the fundamentals.

In Unit 1 you will learn the following:

1. How to create your own `ActorSystem` and actors;
2. How to send messages actors and how to handle different types of messages;
3. How to use `Props` and `IActorRef`s to build loosely coupled systems.
4. How to use actor paths, addresses, and `ActorSelection` to send messages to actors.
5. How to create child actors and actor hierarchies, and how to supervise children with `SupervisionStrategy`.
6. How to use the Actor lifecycle to control actor startup, shutdown, and restart behavior.

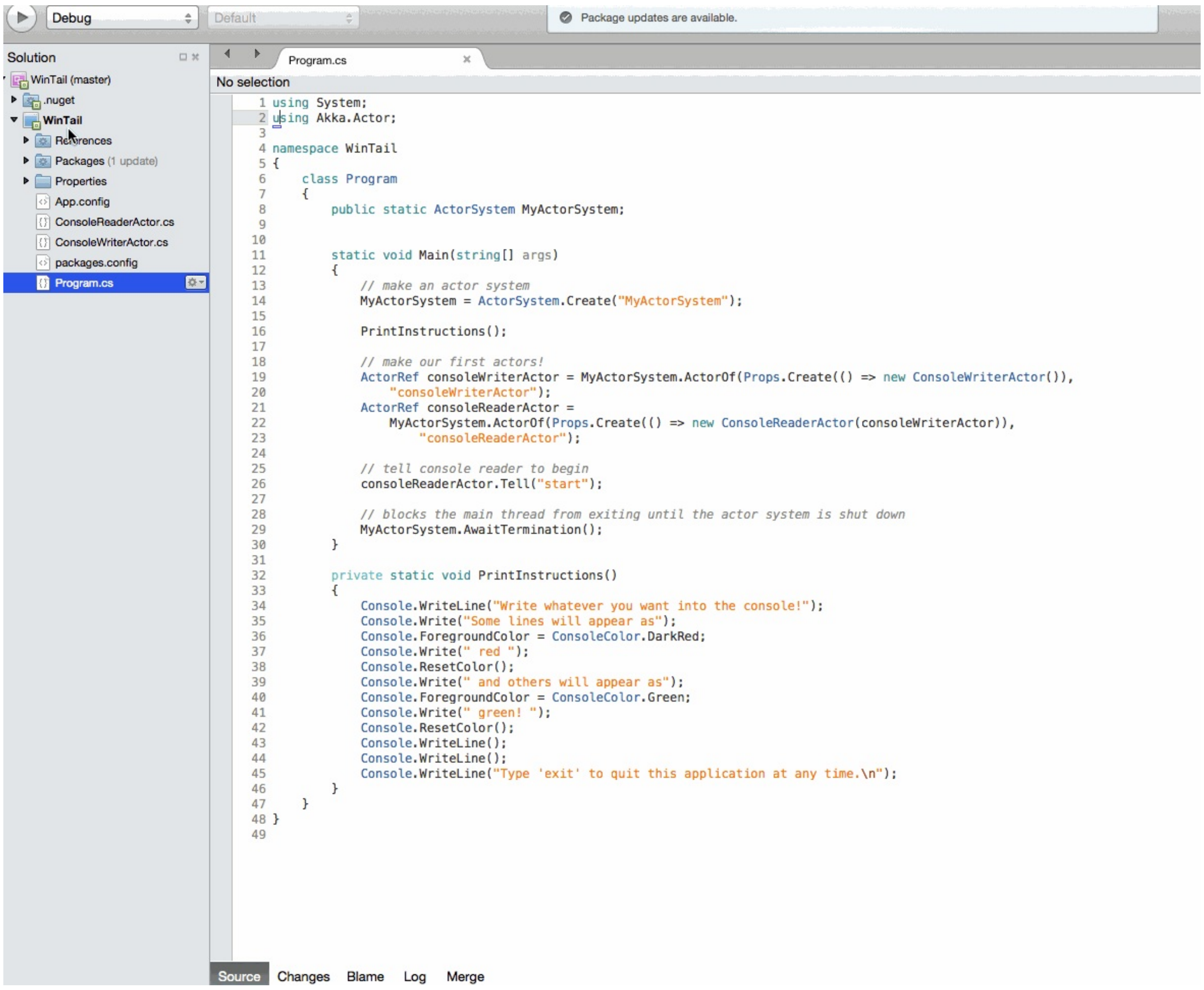
Using Xamarin?

Since Unit 1 relies heavily on the console, you'll need to make a small tweaks before beginning. You need to set up your `WinTail` project file (not the solution) to use an **external console**.

To set this up:

1. Click on the `WinTail` project (not the solution)
2. Navigate to `Project > WinTail Options` in the menu
3. Inside `WinTail Options`, navigate to `Run > General`
4. Select `Run on external console`
5. Click `OK`

Here is a demonstration of how to set it up:



```
using System;
using Akka.Actor;

namespace WinTail
{
    class Program
    {
        public static ActorSystem MyActorSystem;


        static void Main(string[] args)
        {
            // make an actor system
            MyActorSystem = ActorSystem.Create("MyActorSystem");

            PrintInstructions();

            // make our first actors!
            ActorRef consoleWriterActor = MyActorSystem.ActorOf(Props.Create(() => new ConsoleWriterActor()),
                "consoleWriterActor");
            ActorRef consoleReaderActor =
                MyActorSystem.ActorOf(Props.Create(() => new ConsoleReaderActor(consoleWriterActor)),
                    "consoleReaderActor");

            // tell console reader to begin
            consoleReaderActor.Tell("start");

            // blocks the main thread from exiting until the actor system is shut down
            MyActorSystem.AwaitTermination();
        }

        private static void PrintInstructions()
        {
            Console.WriteLine("Write whatever you want into the console!");
            Console.Write("Some lines will appear as");
            Console.ForegroundColor = ConsoleColor.DarkRed;
            Console.Write(" red ");
            Console.ResetColor();
            Console.Write(" and others will appear as");
            Console.ForegroundColor = ConsoleColor.Green;
            Console.Write(" green! ");
            Console.ResetColor();
            Console.WriteLine();
            Console.WriteLine();
            Console.WriteLine("Type 'exit' to quit this application at any time.\n");
        }
    }
}

```

Table of Contents

Get Started

To get started, go to the /DoThis/ folder and open `WinTail.sln`.

And then go to Lesson 1.

Lesson 1.1: Actors and the `ActorSystem`

Here we go! Welcome to lesson 1.

In this lesson, you will make your first actors and be introduced to the fundamentals of Akka.NET.

Key concepts / background

In this first lesson, you will learn the basics by creating a console app with your first actor system and simple actors within it.

We will be creating two actors, one to read from the console, and one to write to it after doing some basic processing.

What is an actor?

An "actor" is really just an analog for a human participant in a system. It's an entity, an object, that can do things and communicate.

> We're going to assume that you're familiar with object-oriented programming (OOP). The actor model is very similar to object-oriented programming (OOP) - just like how everything is an object in OOP, in the actor model ***everything is an actor***.

Repeat this train of thought to yourself: everything is an actor. Everything is an actor. Everything is an actor! Think of designing your system like a hierarchy of people, with tasks being split up and delegated until they become small enough to be handled concisely by one actor.

For now, we suggest you think of it like this: in OOP, you try to give every object a single, well-defined purpose, right? Well, the actor model is no different, except now the objects that you give a clear purpose to just happen to be actors.

Further reading: What is an Akka.NET Actor?

How do actors communicate?

Actors communicate with each other just as humans do, by exchanging messages. These messages are just plain old C# classes.

```
//this is a message!
public class SomeMessage{
    public int SomeValue {get; set}
}
```

We go into messages in detail in the next lesson, so don't worry about it for now. All you need to know is that you send messages by `Tell()` ing them to another actor.

```
//send a string to an actor
someActorRef.Tell("this is a message too!");
```

What can an actor do?

Anything you can code. Really :)

You code actors to handle messages they receive, and actors can do whatever you need them to in order to handle a message. Talk to a database, write to a file, change an internal variable, or anything else you might need.

In addition to processing a message it receives, an actor can:

1. Create other actors
2. Send messages to other actors (such as the `Sender` of the current message)
3. Change its own behavior and process the next message it receives differently

Actors are inherently asynchronous (more on this in a future lesson), and there is nothing about the Actor Model that says which of the above an actor must do, or the order it has to do them in. It's up to you.

What kinds of actors are there?

All types of actors inherit from `UntypedActor`, but don't worry about that now. We'll cover different actor types later.

In Unit 1 all of your actors will inherit from `UntypedActor`.

How do you make an actor?

There are 2 key things to know about creating an actor:

1. All actors are created within a certain context. That is, they are "actor of" a context.
2. Actors need `Props` to be created. A `Props` object is just an object that encapsulates the formula for making a given kind of actor.

We'll be going into `Props` in depth in lesson 3, so for now don't worry about it much. We've provided the `Props` for you in the code, so you just have to figure out how to use `Props` to make an actor.

The hint we'll give you is that your first actors will be created within the context of your actor system itself. See the exercise instructions for more.

What is an `ActorSystem`?

An `ActorSystem` is a reference to the underlying system and Akka.NET framework. All actors live within the context of this actor system. You'll need to create your first actors from the context of this `ActorSystem`.

By the way, the `ActorSystem` is a heavy object: create only one per application.

Aaaaaaand... go! That's enough conceptual stuff for now, so dive right in and make your first actors.

Exercise

Let's dive in!

> Note: Within the sample code there are sections clearly marked `"YOU NEED TO FILL IN HERE"` - find those regions of code and begin filling them in with the appropriate functionality in order to complete your goals.

Launch the fill-in-the-blank sample

Go to the DoThis folder and open WinTail in Visual Studio. The solution consists of a simple console application and only one Visual Studio project file.

You will use this solution file through all of Unit 1.

Install the latest Akka.NET NuGet package

In the Package Manager Console, type the following command:

```
Install-Package Akka
```

This will install the latest Akka.NET binaries, which you will need in order to compile this sample.

Then you'll need to add the `using` namespace to the top of `Program.cs`:

```
// in Program.cs
using Akka.Actor;
```

Make your first `ActorSystem`

Go to `Program.cs` and add this to create your first actor system:

```
MyActorSystem = ActorSystem.Create("MyActorSystem");
```

>

> **NOTE:** When creating `Props`, `ActorSystem`, or `ActorRef` you will very rarely see the `new` keyword. These objects must be created through the factory methods built into Akka.NET. If you're using `new` you might be making a mistake.

Make ConsoleReaderActor & ConsoleWriterActor

The actor classes themselves are already defined, but you will have to make your first actors.

Again, in `Program.cs`, add this just below where you made your `ActorSystem`:

```
var consoleWriterActor = MyActorSystem.ActorOf(Props.Create(() => new ConsoleWriterActor()))
var consoleReaderActor = MyActorSystem.ActorOf(Props.Create(() => new ConsoleReaderActor(con
```

We will get into the details of `Props` and `ActorRef`s in lesson 3, so don't worry about them much for now. Just know that this is how you make an actor.

Have ConsoleReaderActor Send a Message to ConsoleWriterActor

Time to put your first actors to work!

You will need to do the following:

1. ConsoleReaderActor is set up to read from the console. Have it send a message to ConsoleWriterActor containing the content that it just read.

   ```
   // in ConsoleReaderActor.cs
   _consoleWriterActor.Tell(read);
   ```

2. Have ConsoleReaderActor send a message to itself after sending a message to ConsoleWriterActor. This is what keeps the read loop going.

   ```
   // in ConsoleReaderActor.cs
   Self.Tell("continue");
   ```

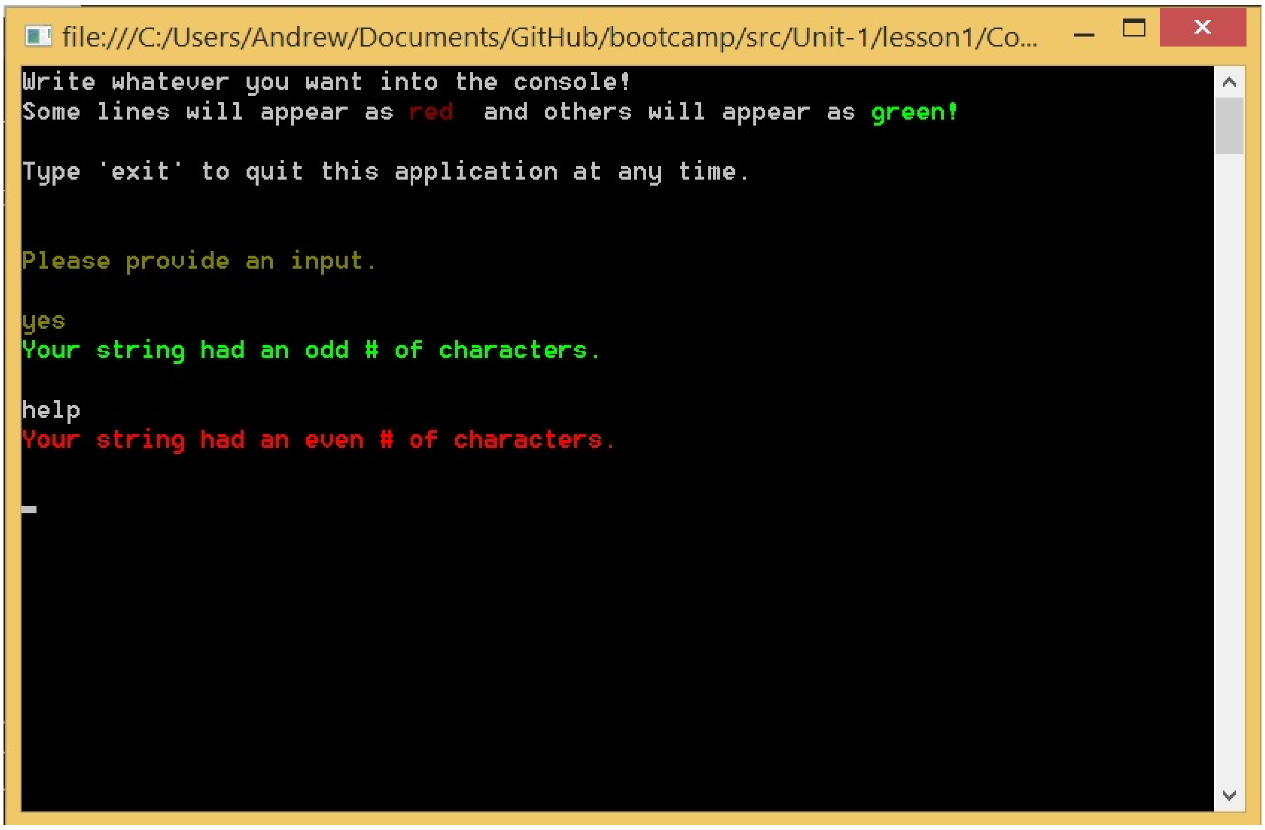
3. Send an initial message to ConsoleReaderActor in order to get it to start reading from the console.

   ```
   // in Program.cs
   consoleReaderActor.Tell("start");
   ```

Step 5: Build and Run!

Once you've made your edits, press `F5` to compile and run the sample in Visual Studio.

You should see something like this, when it is working correctly:


file:///C:/Users/Andrew/Documents/GitHub/bootcamp/src/Unit-1/lesson1/Co...

```
Write whatever you want into the console!
Some lines will appear as red  and others will appear as green!

Type 'exit' to quit this application at any time.


Please provide an input.

yes
Your string had an odd # of characters.

help
Your string had an even # of characters.
```

Once you're done

Compare your code to the code in the Completed folder to see what the instructors included in their samples.

Great job! Onto Lesson 2!

Awesome work! Well done on completing your first lesson.

Let's move onto Lesson 2 - Defining and Handling Different Types of Messages.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 1.2: Defining and Handling Messages

In this lesson, you will make your own message types and use learn how to control processing flow within your actors based on your custom messages. Doing so will teach you the fundamentals of communicating in a message- and event-driven manner within your actor system.

This lesson picks up right where Lesson 1 left off, and continues extending our budding systems of console actors. In addition to defining our own messages, we'll also add some simple validation for the input we enter and take action based on the results of that validation.

Key concepts / background

What is a message?

Any POCO can be a message. A message can be a `string`, a value like `int`, a type, an object that implements an interface... whatever you want.

That being said, the recommended approach is to make your own custom messages into semantically named classes, and to encapsulate any state you want inside those classes (e.g. store a `Reason` inside a `ValidationFailed` class... hint, hint...).

How do I send an actor a message?

As you saw in the first lesson, you `Tell()` the actor the message.

How do I handle a message?

This is entirely up to you, and doesn't really have much to do with Akka.NET. You can handle (or not handle) a message as you choose within an actor.

What happens if my actor receives a message it doesn't

know how to handle?

Actors ignore messages they don't know how to handle. Whether or not this ignored message is logged as such depends on the type of actor.

With an `UntypedActor`, unhandled messages are not logged as unhandled unless you manually mark them as such, like so:

```
class MyActor : UntypedActor
{
    protected override void OnReceive(object message)
    {
         if (message is Messages.InputError)
        {
            var msg = message as Messages.InputError;
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(msg.Reason);
        }
        else
        {
            Unhandled(message);
        }
    }
}
```

However, in a `ReceiveActor` —which we cover in Unit 2—unhandled messages are automatically sent to `Unhandled` so the logging is done for you.

How do my actors respond to messages?

This is up to you - you can respond by simply processing the message, replying to the `Sender`, forwarding the message onto another actor, or doing nothing at all.

> **NOTE:** Whenever your actor receives a message, it will always have the sender of the current message available via the `Sender` property inside your actor.

Exercise

In this exercise, we will introduce some basic validation into our system. We will then use custom message types to signal the results of that validation back to the user.

Phase 1: Define your own message types

Add a new class called `Messages` and the corresponding file, `Messages.cs`.

This is the class we'll use to define system-level messages that we can use to signal events. The pattern we'll be using is to turn events into messages. That is, when an event occurs, we will send an appropriate message class to the actor(s) that need to know about it, and then listen for / respond to that message as needed in the receiving actors.

Add regions for each message type

Add three regions for different types of messages to the file. Next we'll be creating our own message classes that we'll use to signify events.

```
// in Messages.cs
#region Neutral/system messages
#endregion

#region Success messages
#endregion

#region Error messages
#endregion
```

In these regions we will define custom message types to signal these situations:

```
- user provided blank input
- user provided invalid input
- user provided valid input
```

Make `ContinueProcessing` message

Define a marker message class in the `Neutral/system messages` region that we'll use to signal to continue processing (the "blank input" case):

```
// in Messages.cs
```

```
#region Neutral/system messages
/// <summary>
/// Marker class to continue processing.
/// </summary>
public class ContinueProcessing { }
#endregion
```

Make `InputSuccess` message

Define an `InputSuccess` class in the `Success messages` region. We'll use this to signal that the user's input was good and passed validation (the "valid input" case):

```
#region Success messages
// in Messages.cs
/// <summary>
/// Base class for signalling that user input was valid.
/// </summary>
public class InputSuccess
{
    public InputSuccess(string reason)
    {
        Reason = reason;
    }

    public string Reason { get; private set; }
}
#endregion
```

Make `InputError` messages

Define the following `InputError` classes in the `Error messages` region. We'll use these messages to signal invalid input occurring (the "invalid input" cases):

```
// in Messages.cs
#region Error messages
/// <summary>
/// Base class for signalling that user input was invalid.
/// </summary>
public class InputError
{
    public InputError(string reason)
    {
        Reason = reason;
    }

    public string Reason { get; private set; }
}

/// <summary>
/// User provided blank input.
/// </summary>
public class NullInputError : InputError
{
    public NullInputError(string reason) : base(reason) { }
}

/// <summary>
/// User provided invalid input (currently, input w/ odd # chars)
/// </summary>
public class ValidationError : InputError
{
    public ValidationError(string reason) : base(reason) { }
}
#endregion
```

> **NOTE:** You can compare your final `Messages.cs` to Messages.cs to make sure you're set up right before we go on.

Phase 2: Turn events into messages and send them

Great! Now that we've got messages classes set up to wrap our events, let's use them in `ConsoleReaderActor` and `ConsoleWriterActor`.

Update `ConsoleReaderActor`

Add the following internal message type to `ConsoleReaderActor`:

```
// in ConsoleReaderActor
public const string StartCommand = "start";
```

Update the `Main` method to use `ConsoleReaderActor.StartCommand`:

Replace this:

```
// in Program.cs
// tell console reader to begin
consoleReaderActor.Tell("start");
```

with this:

```
// in Program.cs
// tell console reader to begin
consoleReaderActor.Tell(ConsoleReaderActor.StartCommand);
```

Replace the `OnReceive` method of `ConsoleReaderActor` as follows. Notice that we're now listening for our custom `InputError` messages, and taking action when we get an error.

```
// in ConsoleReaderActor
protected override void OnReceive(object message)
{
    if (message.Equals(StartCommand))
    {
        DoPrintInstructions();
    }
    else if (message is Messages.InputError)
    {
        _consoleWriterActor.Tell(message as Messages.InputError);
    }

    GetAndValidateInput();
}
```

While we're at it, let's add `DoPrintInstructions()`, `GetAndValidateInput()`, `IsValid()` to `ConsoleReaderActor`. These are internal methods that our `ConsoleReaderActor` will use to get input from the console and determine if it is valid. (Currently, "valid" just means that the input had an even number of characters. It's an arbitrary placeholder.)

```
// in ConsoleReaderActor, after OnReceive()
#region Internal methods
private void DoPrintInstructions()
{
    Console.WriteLine("Write whatever you want into the console!");
    Console.WriteLine("Some entries will pass validation, and some won't...\n\n");
    Console.WriteLine("Type 'exit' to quit this application at any time.\n");
}

/// <summary>
/// Reads input from console, validates it, then signals appropriate response
/// (continue processing, error, success, etc.).
/// </summary>
private void GetAndValidateInput()
{
    var message = Console.ReadLine();
    if (string.IsNullOrEmpty(message))
    {
        // signal that the user needs to supply an input, as previously
        // received input was blank
        Self.Tell(new Messages.NullInputError("No input received."));
    }
    else if (String.Equals(message, ExitCommand, StringComparison.OrdinalIgnoreCase))
    {
        // shut down the entire actor system (allows the process to exit)
        Context.System.Shutdown();
    }
    else
    {
        var valid = IsValid(message);
        if (valid)
        {
            _consoleWriterActor.Tell(new Messages.InputSuccess("Thank you! Message was valid

            // continue reading messages from console
            Self.Tell(new Messages.ContinueProcessing());
        }
        else
        {
            Self.Tell(new Messages.ValidationError("Invalid: input had odd number of characte
        }
    }
}

/// <summary>
/// Validates <see cref="message"/>.
/// Currently says messages are valid if contain even number of characters.
/// </summary>
/// <param name="message"></param>
/// <returns></returns>
private static bool IsValid(string message)
{
    var valid = message.Length % 2 == 0;
    return valid;
}
#endregion
```

Update `Program`

First, remove the definition and call to `PrintInstructions()` from `Program.cs`.

Now that `ConsoleReaderActor` has its own well-defined `StartCommand`, let's go ahead and use that instead of hardcoding the string "start" into the message.

As a quick checkpoint, your `Main()` should now look like this:

```
static void Main(string[] args)
{
    // initialize MyActorSystem
    MyActorSystem = ActorSystem.Create("MyActorSystem");

    var consoleWriterActor = MyActorSystem.ActorOf(Props.Create(() => new ConsoleWriterActor
    var consoleReaderActor = MyActorSystem.ActorOf(Props.Create(() => new ConsoleReaderActor

    // tell console reader to begin
    consoleReaderActor.Tell(ConsoleReaderActor.StartCommand);

    // blocks the main thread from exiting until the actor system is shut down
    MyActorSystem.AwaitTermination();
}
```

Not much has changed here, just a bit of cleanup.

Update `ConsoleWriterActor`

Now, let's get `ConsoleWriterActor` to handle these new types of messages.

Change the `OnReceive` method of `ConsoleWriterActor` as follows:

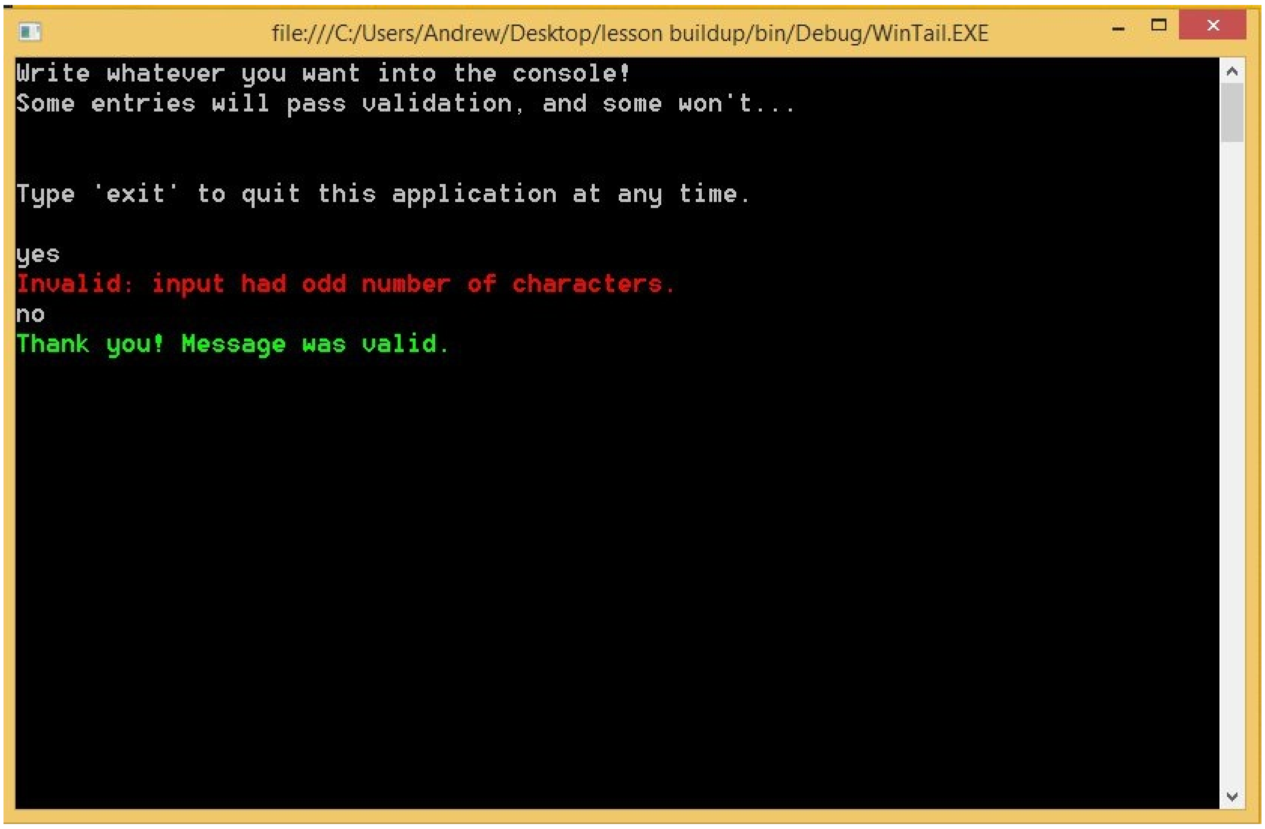
```
// in ConsoleWriterActor.cs
protected override void OnReceive(object message)
{
    if (message is Messages.InputError)
    {
        var msg = message as Messages.InputError;
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(msg.Reason);
    }
    else if (message is Messages.InputSuccess)
    {
        var msg = message as Messages.InputSuccess;
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine(msg.Reason);
    }
    else
    {
        Console.WriteLine(message);
    }

    Console.ResetColor();
}
```

As you can see here, we are making `ConsoleWriterActor` pattern match against the type of message it receives, and take different actions according to what type of message it receives.

Phase 3: Build and run!

You should now have everything you need in place to be able to build and run. Give it a try!

If everything is working as it should, you should see an output like this:


file:///C:/Users/Andrew/Desktop/lesson buildup/bin/Debug/WinTail.EXE

```
Write whatever you want into the console!
Some entries will pass validation, and some won't...


Type 'exit' to quit this application at any time.

yes
Invalid: input had odd number of characters.
no
Thank you! Message was valid.
```

Once you're done

Compare your code to the solution in the Completed folder to see what the instructors included in their samples.

Great job! Onto Lesson 3!

Awesome work! Well done on completing this lesson.

Let's move onto Lesson 3 - `Props` and `IActorRef` s.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 1.3: `Props` and `IActorRef` s

In this lesson, we will review/reinforce the different ways you can create actors and send them messages. This lesson is more conceptual and has less coding for you to do, but it's an essential foundation and key to understanding the code you will see down the line.

In this lesson, the code has changed a bit. The change is that the `ConsoleReaderActor` no longer does any validation work, but instead, just passes off the messages it receives from the console to another actor for validation (the `ValidationActor`).

Key concepts / background

`IActorRef` s

What is an `IActorRef` ?

An `IActorRef` is a reference or handle to an actor. The purpose of an `IActorRef` is to support sending messages to an actor through the `ActorSystem` . You never talk directly to an actor—you send messages to its `IActorRef` and the `ActorSystem` takes care of delivering those messages for you.

WTF? I don't actually talk to my actors? Why not?

You do talk to them, just not directly :) You have to talk to them via the intermediary of the `ActorSystem` .

Here are two of the reasons why it is an advantage to send messages to an `IActorRef` and let the underlying `ActorSystem` do the work of getting the messages to the actual actor.

- It gives you better information to work with and messaging semantics. The `ActorSystem` wraps all messages in an `Envelope` that contains metadata about the message. This metadata is automatically unpacked and made available in the context of your actor.
- It allows "location transparency": this is a fancy way of saying that you don't have to

worry about which process or machine your actor lives in. Keeping track of all this is the system's job. This is essential for allowing remote actors, which is how you can scale an actor system up to handle massive amounts of data (e.g. have it work on multiple machines in a cluster). More on this later.

How do I know my message got delivered to the actor?

For now, this is not something you should worry about. The underlying `ActorSystem` of Akka.NET itself provides mechanisms to guarantee this, but `GuaranteedDeliveryActors` are an advanced topic.

For now, just trust that delivering messages is the `ActorSystem` s job, not yours. Trust, baby. :)

Okay, fine, I'll let the system deliver my messages. So how do I get an `IActorRef` ?

There are two ways to get an `IActorRef` .

1) Create the actor

Actors form intrinsic supervision hierarchies (we cover in detail in lesson 5). This means there are "top level" actors, which essentially report directly to the `ActorSystem` itself, and there are "child" actors, which report to other actors.

To make an actor, you have to create it from its context. And **you've already done this!** Remember this?

```
// assume we have an existing actor system, "MyActorSystem"
IActorRef myFirstActor = MyActorSystem.ActorOf(Props.Create(() => new MyActorClass()), "myFi
```

As shown in the above example, you create an actor in the context of the actor that will supervise it (almost always). When you create the actor on the `ActorSystem` directly (as above), it is a top-level actor.

You make child actors the same way, except you create them from another actor, like so:

```
// have to create the child actor somewhere inside myFirstActor
// usually happens inside OnReceive or PreStart
class MyActorClass : UntypedActor{
    protected override void PreStart(){
        IActorRef myFirstChildActor = Context.ActorOf(Props.Create(() => new MyChildActorCla
    }
}
```

CAUTION: Do NOT call `new MyActorClass()` outside of `Props` and the `ActorSystem` to make an actor. We can't go into all the details here, but essentially, by doing so you're trying to create an actor outside the context of the `ActorSystem`. This will produce a completely unusable, undesirable object.

2) Look up the actor

All actors have an address (technically, an `ActorPath`) which represents where they are in the system hierarchy, and you can get a handle to them (get their `IActorRef`) by looking them up by their address.

We will cover this in much more detail in the next lesson.

Do I have to name my actors?

You may have noticed that we passed names into the `ActorSystem` when we were creating the above actors:

```
// last arg to the call to ActorOf() if a name
IActorRef myFirstActor = MyActorSystem.ActorOf(Props.Create(() => new MyActorClass()), "myFi
```

This name is not required. It is perfectly valid to create an actor without a name, like so:

```
// last arg to the call to ActorOf() if a name
IActorRef myFirstActor = MyActorSystem.ActorOf(Props.Create(() => new MyActorClass()))
```

That said, **the best practice is to name your actors**. Why? Because the name of your actor is used in log messages and in identifying actors. Get in the habit, and your future self will thank you when you have to debug something and it has a nice label on it :)

Are there different types of `IActorRef` s?

Actually, yes. The most common, by far, is just a plain-old `IActorRef` or handle to an actor, as above.

However, there are also some other `IActorRef` s available to you within the context of an actor. As we said, all actors have a context. That context holds metadata, which includes information about the current message being processed. That information includes things like the `Parent` or `Children` of the current actor, as well as the `Sender` of the current message.

`Parent` , `Children` , and `Sender` all provide `IActorRef` s that you can use.

Props

What are `Props` ?

Think of `Props` as a recipe for making an actor. Technically, `Props` is a configuration class that encapsulates all the information needed to make an instance of a given type of actor.

Why do we need `Props` ?

`Props` objects are shareable recipes for creating an instance of an actor. `Props` get passed to the `ActorSystem` to generate an actor for your use.

Right now, `Props` probably feels like overkill. (If so, no worries.) But here's the deal.

The most basic `Props` , like we've seen, seem to only include the ingredients needed to make an actor—it's class and required args to its constructor.

BUT, what you haven't seen yet is that `Props` get extended to contain deployment information and other configuration details that are needed to do remote work. For example, `Props` are serializable, so they can be used to remotely create and deploy entire groups of actors on another machine somewhere on the network!

That's getting way ahead of ourselves though, but the short answer is that we need

`Props` to support a lot of the advanced features (clustering, remote actors, etc) that give Akka.NET the serious horsepower which makes it interesting.

How do I make `Props`?

Before we tell you how to make `Props`, let me tell you what NOT to do.

DO NOT TRY TO MAKE PROPS BY CALLING `new Props(...)`. Similar to trying to make an actor by calling `new MyActorClass()`, this is fighting the framework and not letting Akka's `ActorSystem` do its work under the hood to provide safe guarantees about actor restarts and lifecycle management.

There are 3 ways to properly create `Props`, and they all involve a call to `Props.Create()`.

1. **The `typeof` syntax:**

   ```
   Props props1 = Props.Create(typeof(MyActor));
   ```

   While it looks simple, **we recommend that you do not use this approach.** Why? *Because it has no type safety and can easily introduce bugs where everything compiles fine, and then blows up at runtime*.

2. **The lambda syntax**:

   ```
   Props props2 = Props.Create(() => new MyActor(..), "...");
   ```

   This is a mighty fine syntax, and our favorite. You can pass in the arguments required by the constructor of your actor class inline, along with a name.

3. **The generic syntax**:

   ```
   Props props3 = Props.Create<MyActor>();
   ```

   Another fine syntax that we whole-heartedly recommend.

How do I use `Props`?

You actually already know this, and have done it. You pass the `Props` —the actor recipe —to the call to `Context.ActorOf()` and the underlying `ActorSystem` reads the recipe, et voila! Whips you up a fresh new actor.

Enough of this conceptual business. Let's get to it!

Exercise

Before we can get into the meat of this lesson (`Props` and `IActorRef` s), we have to do a bit of cleanup.

Phase 1: Move validation into its own actor

We're going to move all our validation code into its own actor. It really doesn't belong in the `ConsoleReaderActor` . Validation deserves to have its own actor (similar to how you want single-purpose objects in OOP).

Create `ValidationActor` class

Make a new class called `ValidationActor` and put it into its own file. Fill it with all the validation logic that is currently in `ConsoleReaderActor` :

```
// ValidationActor.cs
using Akka.Actor;

namespace WinTail
{
    /// <summary>
    /// Actor that validates user input and signals result to others.
    /// </summary>
    public class ValidationActor : UntypedActor
    {
        private readonly IActorRef _consoleWriterActor;

        public ValidationActor(IActorRef consoleWriterActor)
        {
            _consoleWriterActor = consoleWriterActor;
        }

        protected override void OnReceive(object message)
        {
            var msg = message as string;
            if (string.IsNullOrEmpty(msg))
            {
                // signal that the user needs to supply an input
                _consoleWriterActor.Tell(new Messages.NullInputError("No input received."));
            }
            else
            {
                var valid = IsValid(msg);
                if (valid)
                {
                    // send success to console writer
                    _consoleWriterActor.Tell(new Messages.InputSuccess("Thank you! Message wa
                }
                else
                {
                    // signal that input was bad
                    _consoleWriterActor.Tell(new Messages.ValidationError("Invalid: input had
                }
            }

            // tell sender to continue doing its thing (whatever that may be, this actor doe
            Sender.Tell(new Messages.ContinueProcessing());

        }

        /// <summary>
        /// Determines if the message received is valid.
        /// Currently, arbitrarily checks if number of chars in message received is even.
        /// </summary>
        /// <param name="msg"></param>
        /// <returns></returns>
        private static bool IsValid(string msg)
        {
            var valid = msg.Length % 2 == 0;
            return valid;
        }
    }
}
```

Phase 2: Making `Props`, our actor recipes

Okay, now we can get to the good stuff! We are going to use what we've learned about `Props` and tweak the way we make our actors.

Again, we do not recommend using the `typeof` syntax. For practice, use both of the lambda and generic syntax!

> **Remember**: do NOT try to create `Props` by calling `new Props(...)`.

> When you do that, kittens die, unicorns vanish, Mordor wins and all manner of badness happens. Let's just not.

In this section, we're going to split out the `Props` objects onto their own lines for easier reading. In practice, we usually inline them into the call to `ActorOf`.

Delete existing `Props` and `IActorRef` s

In `Main()`, remove your existing actor declarations so we have a clean slate.

Your code should look like this right now:

```
// Program.cs
static void Main(string[] args)
{
    // initialize MyActorSystem
    MyActorSystem = ActorSystem.Create("MyActorSystem");

    // nothing here where our actors used to be!

    // tell console reader to begin
    consoleReaderActor.Tell(ConsoleReaderActor.StartCommand);

    // blocks the main thread from exiting until the actor system is shut down
    MyActorSystem.AwaitTermination();
}
```

Make `consoleWriterProps`

Go to `Program.cs`. Inside of `Main()`, split out `consoleWriterProps` onto its own line like so:

```
// Program.cs
Props consoleWriterProps = Props.Create(typeof (ConsoleWriterActor));
```

Here you can see we're using the typeof syntax, just to show you what it's like. But again, *we do not recommend using the `typeof` syntax in practice*.

Going forward, we'll only use the lambda and generic syntaxes for `Props`.

Make `validationActorProps`

Add this just to `Main()` also:

```
// Program.cs
Props validationActorProps = Props.Create(() => new ValidationActor(consoleWriterActor));
```

As you can see, here we're using the lambda syntax.

Make `consoleReaderProps`

Add this just to `Main()` also:

```
// Program.cs
Props consoleReaderProps = Props.Create<ConsoleReaderActor>(validationActor);
```

This is the generic syntax. `Props` accepts the actor class as a generic type argument, and then we pass in whatever the actor's constructor needs.

Phase 3: Making `IActorRef`s using various `Props`

Great! Now that we've got `Props` for all the actors we want, let's go make some actors!

Remember: do not try to make an actor by calling `new Actor()` outside of a `Props` object and/or outside the context of the `ActorSystem` or another `IActorRef`. Mordor and all that, remember?

Make a new `IActorRef` for `consoleWriterActor`

Add this to `Main()` on the line after `consoleWriterProps`:

```
// Program.cs
IActorRef consoleWriterActor = MyActorSystem.ActorOf(consoleWriterProps, "consoleWriterActor
```

Make a new `IActorRef` for `validationActor`

Add this to `Main()` on the line after `validationActorProps`:

```
// Program.cs
IActorRef validationActor = MyActorSystem.ActorOf(validationActorProps, "validationActor");
```

Make a new `IActorRef` for `consoleReaderActor`

Add this to `Main()` on the line after `consoleReaderProps`:

```
// Program.cs
IActorRef consoleReaderActor = MyActorSystem.ActorOf(consoleReaderProps, "consoleReaderActor
```

Calling out a special `IActorRef`: `Sender`

You may not have noticed it, but we actually are using a special `IActorRef` now: `Sender`. Go look for this in `ValidationActor.cs`:

```
// tell sender to continue doing its thing (whatever that may be, this actor doesn't care)
Sender.Tell(new Messages.ContinueProcessing());
```

This is the special `Sender` handle that is made available within an actors `Context` when it is processing a message. The `Context` always makes this reference available, along with some other metadata (more on that later).

Phase 4: A bit of cleanup

Just a bit of cleanup since we've changed our class structure. Then we can run our app again!

Update `ConsoleReaderActor`

Now that `ValidationActor` is doing our validation work, we should really slim down `ConsoleReaderActor`. Let's clean it up and have it just hand the message off to the

`ValidationActor` for validation.

We'll also need to store a reference to `ValidationActor` inside the `ConsoleReaderActor`, and we don't need a reference to the the `ConsoleWriterActor` anymore, so let's do some cleanup.

Modify your version of `ConsoleReaderActor` to match the below:

```
// ConsoleReaderActor.cs
// removing validation logic and changing store actor references
using System;
using Akka.Actor;

namespace WinTail
{
    /// <summary>
    /// Actor responsible for reading FROM the console.
    /// Also responsible for calling <see cref="ActorSystem.Shutdown"/>.
    /// </summary>
    class ConsoleReaderActor : UntypedActor
    {
        public const string StartCommand = "start";
        public const string ExitCommand = "exit";
        private readonly IActorRef _validationActor;

        public ConsoleReaderActor(IActorRef validationActor)
        {
            _validationActor = validationActor;
        }

        protected override void OnReceive(object message)
        {
            if (message.Equals(StartCommand))
            {
                DoPrintInstructions();
            }

            GetAndValidateInput();
        }


        #region Internal methods
        private void DoPrintInstructions()
        {
            Console.WriteLine("Write whatever you want into the console!");
            Console.WriteLine("Some entries will pass validation, and some won't...\n\n");
            Console.WriteLine("Type 'exit' to quit this application at any time.\n");
        }


        /// <summary>
        /// Reads input from console, validates it, then signals appropriate response
        /// (continue processing, error, success, etc.).
        /// </summary>
        private void GetAndValidateInput()
        {
            var message = Console.ReadLine();
            if (!string.IsNullOrEmpty(message) && String.Equals(message, ExitCommand, String
            {
                // if user typed ExitCommand, shut down the entire actor system (allows the
                Context.System.Shutdown();
                return;
            }

            // otherwise, just hand message off to validation actor (by telling its actor re
            _validationActor.Tell(message);
        }
        #endregion
    }
}
```

As you can see, we're now handing off the input from the console to the `ValidationActor` for validation and decisions. `ConsoleReaderActor` is now only responsible for reading from the console and handing the data off to another more sophisticated actor.

Fix that first `Props` call...

We can't very well recommend you not use the `typeof` syntax and then let it stay there. Real quick, go back to `Main()` and update `consoleWriterProps` to be use the generic syntax.

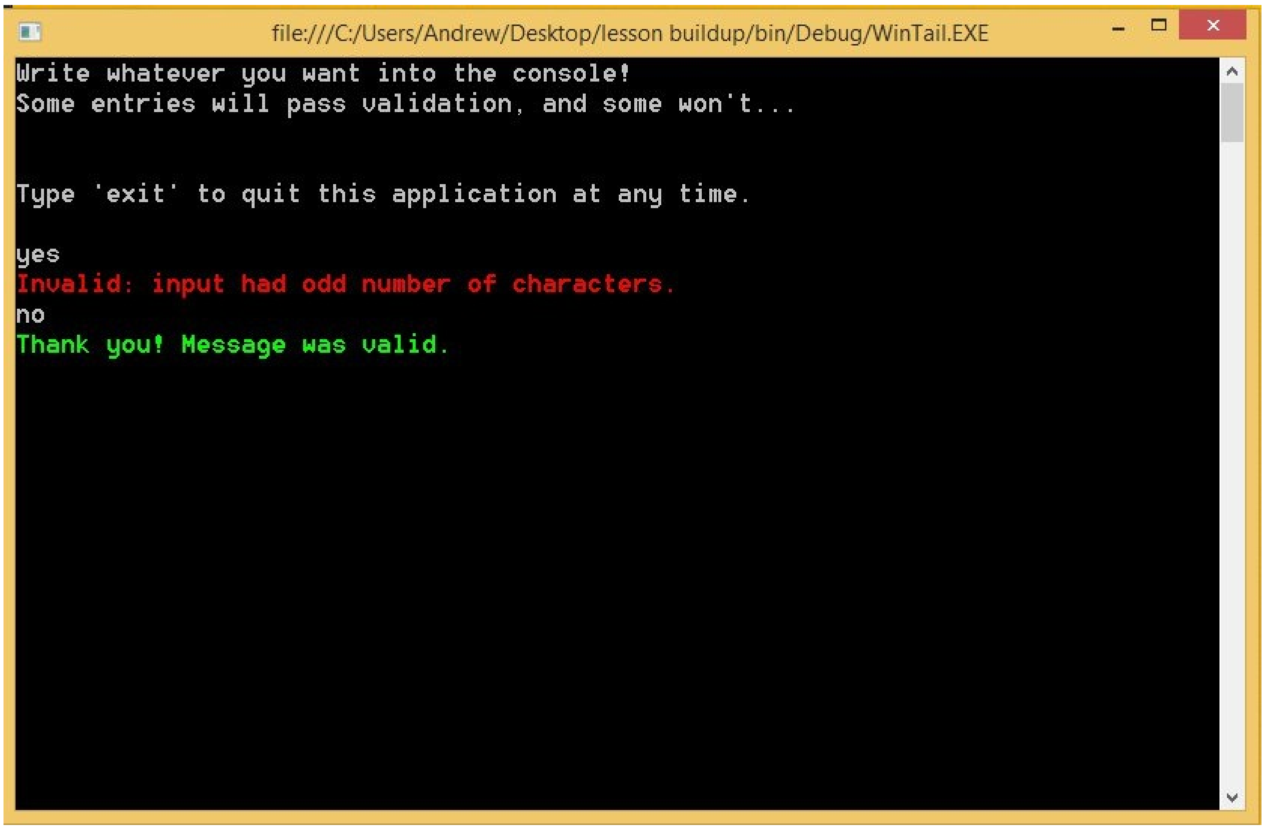
```
Props consoleWriterProps = Props.Create<ConsoleWriterActor>();
```

There. That's better.

Once you're done

Compare your code to the solution in the Completed folder to see what the instructors included in their samples.

If everything is working as it should, the output you see should be identical to last time:


file:///C:/Users/Andrew/Desktop/lesson buildup/bin/Debug/WinTail.EXE

```
Write whatever you want into the console!
Some entries will pass validation, and some won't...


Type 'exit' to quit this application at any time.

yes
Invalid: input had odd number of characters.
no
Thank you! Message was valid.
```

Experience the danger of the `typeof` syntax for `Props` yourself

Since we harped on it earlier, let's illustrate the risk of using the `typeof` `Props` syntax and why we avoid it.

We've left a little landmine as a demonstration. You should blow it up just to see what happens.

1. Open up Completed/WinTail.sln.
2. Find the lines containing `fakeActorProps` and `fakeActor` (should be around line 18).
3. Uncomment these lines.
    - Look at what we're doing here—intentionally substituting a non-actor class into a `Props` object! Ridiculous! Terrible!
    - While this is an unlikely and frankly ridiculous example, that is exactly the point. It's just leaving open the door for mistakes, even with good intentions.
4. Build the solution. Watch with horror as this ridiculous piece of code *compiles without error!*
5. Run the solution.
6. Try to shield yourself from everything melting down when your program reaches that line of code.

Okay, so what was the point of that? Contrived as that example was, it should show you that *using the `typeof` syntax for `Props` has no type safety and is best avoided unless you have a damn good reason to use it.*

Great job! Onto Lesson 4!

Awesome work! Well done on completing your this lesson. It was a big one.

Let's move onto Lesson 4 - Child Actors, Actor Hierarchies, and Supervision.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 1.4: Child Actors, Actor Hierarchies, and Supervision

This lesson will make a big jump forward in both the capabilities of our codebase, and in your understanding of how the actor model works.

This lesson is our most challenging one yet, so let's get right to it!

Key concepts / background

Before we get into the details of the actor hierarchy itself, let's stop and ask: why do we need a hierarchy at all?

There are two key reasons actors exist in a hierarchy:

1. To atomize work and turn massive amounts of data into manageable chunks
2. To contain errors and make the system resilient

Hierarchies atomize work

Having a hierarchy helps our system to break down work into smaller and smaller pieces, and to allow for different skill specializations at different levels of the hierarchy.

A common way this is realized in an actor systems is that large data streams get atomized, broken down over and over again until they are small and can easily be dealt with by a small code footprint.

Let's take Twitter as an example (users of JVM Akka). Using Akka, Twitter is able to break up their massive ingestion of data into small, manageable streams of information that they can react to. For instance - Twitter can break up their giant firehose of tweets into individual streams for the timeline of each user currently on the site, and they can use Akka to push messages that have arrived for that user into their stream via websocket / etc.

What's the pattern? Take a lot of work. Break it down recursively until it is easily dealt with. Respond as needed.

Hierarchies enable resilient systems

A hierarchy allows for different levels of risk and specialization to exist that could not otherwise.

Think of how an army works. An army has a general setting strategy and overseeing everything, but she is usually not going to be on the front line of the battle where there is the most risk. However, she has wide leverage and guides everything. At the same time, there are lower-ranking soldiers who are on the front lines, doing risky operations and carrying out the orders that they receive.

This is exactly how an actor system operates.

Higher-level actors are more supervisional in nature, and this allows the actor system to push risk down and to the edges. By pushing risky operations to the edges of the hierarchy, the system can isolate risk and recover from errors without the whole system crashing.

Both of these concepts are important, but for the rest of this lesson we'll put our emphasis on how actor systems use hierarchies to be resilient.

How is this achieved? **Supervision.**

What is supervision? Why should I care?

Supervision is the basic concept that allows your actor system to quickly isolate and recover from failures.

Every actor has another actor that supervises it, and helps it recover when errors occur. This is true from the top all the way to the bottom of the hierarchy.

This supervision ensures that when part of your application encounters an unexpected failure (unhandled exception, network timeout, etc.), that failure will be contained to only the affected part of your actor hierarchy.

All other actors will keep on working as though nothing happened. We call this "failure isolation" or "containment."

How is this accomplished? Let's find out…

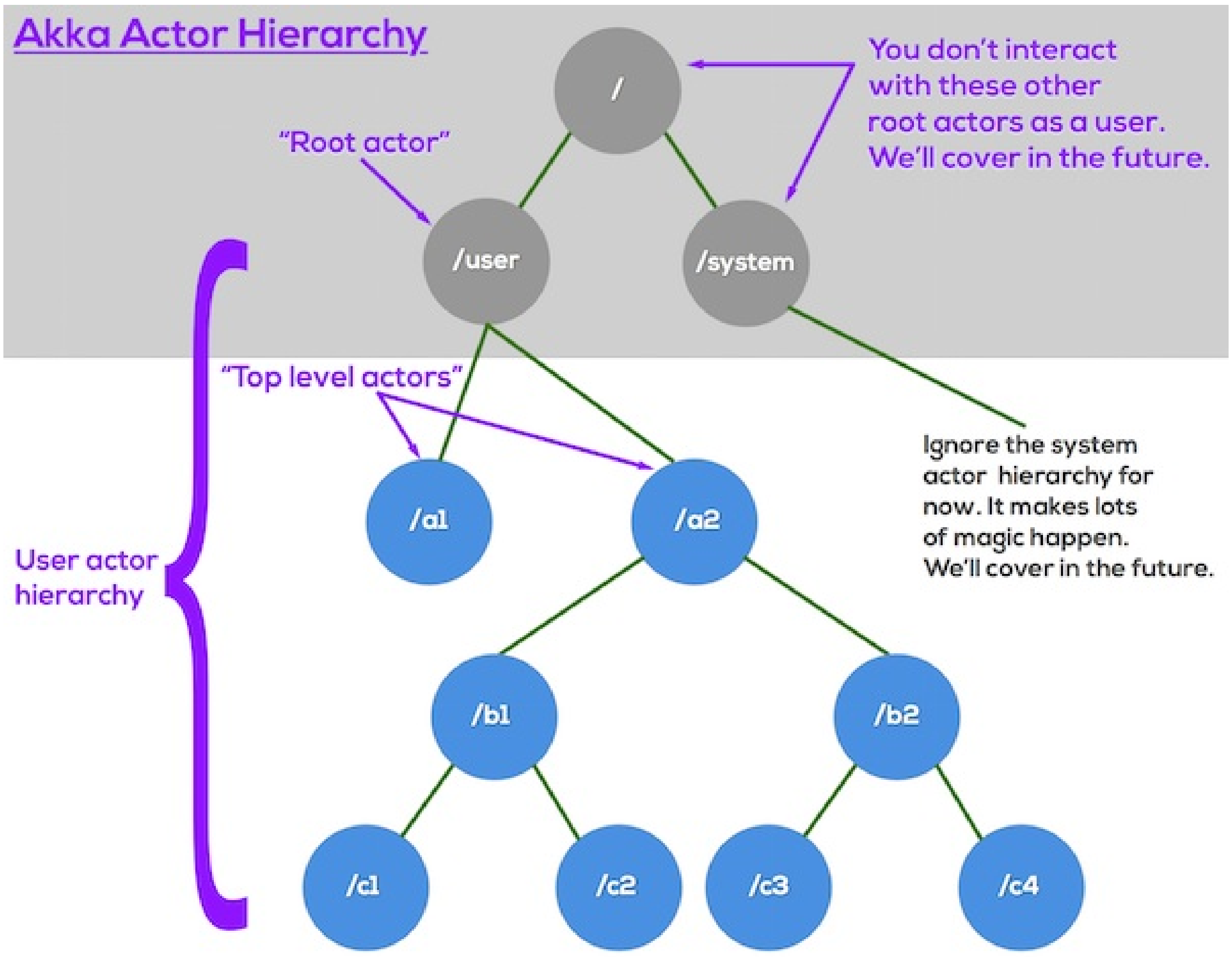
Actor Hierarchies

First, a key point: Every actor has a parent, and some actors have children. Parents supervise their children.

Since parents supervise their children, this means that ***every actor has a supervisor, and every actor can also BE a supervisor.***

Within your actor system, actors are arranged into a hierarchy. This means there are "top level" actors, which essentially report directly to the `ActorSystem` itself, and there are "child" actors, which report to other actors.

The overall hierarchy looks like this (we'll go through piece by piece in a moment):

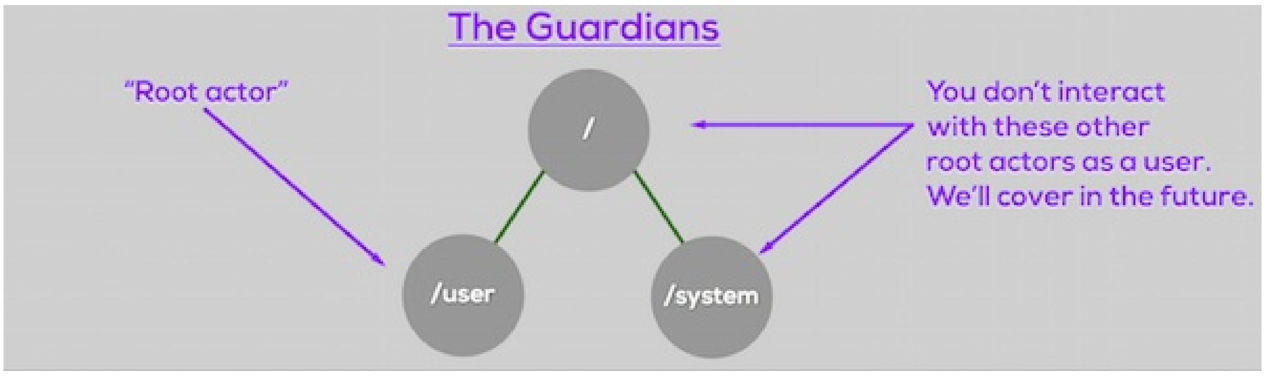

Akka Actor Hierarchy
/
"Root actor"
You don't interact with these other root actors as a user. We'll cover in the future.
/user
/system
"Top level actors"
User actor hierarchy
/a1
/a2
Ignore the system actor hierarchy for now. It makes lots of magic happen. We'll cover in the future.
/b1
/b2
/c1
/c2
/c3
/c4

What are the levels of the hierarchy?

The base of it all: The "Guardians"

The "guardians" are the root actors of the entire system.

I'm referring to these three actors at the very top of the hierarchy:


The Guardians
"Root actor"
/
/user
/system
You don't interact with these other root actors as a user. We'll cover in the future.

The `/` actor

The `/` actor is the base actor of the entire actor system, and may also be referred to as "The Root Guardian." This actor supervises the `/system` and `/user` actors (the other "Guardians").

All actors require another actor as their parent, except this one. This actor is also sometimes called the "bubble-walker" since it is "out of the bubble" of the normal actor system. For now, don't worry about this actor.

The `/system` actor

The `/system` actor may also be referred to as "The System Guardian". The main job of this actor is to ensure that the system shuts down in an orderly manner, and to maintain/supervise other system actors which implement framework level features and utilities (logging, etc). We'll discuss the system guardian and the system actor hierarchy in a future post.

The `/user` actor

This is where the party starts! And this is where you'll be spending all your time as a developer.

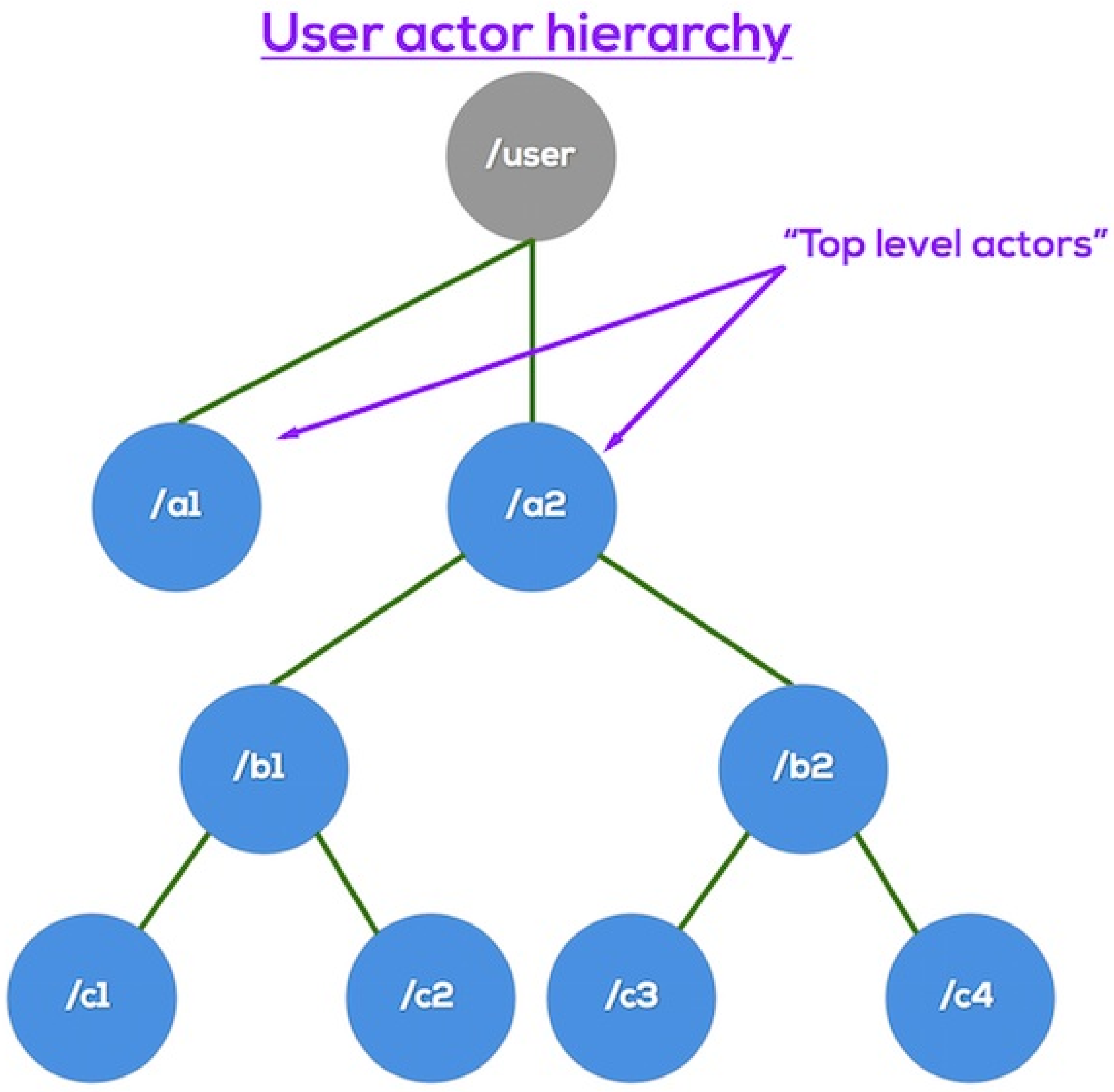
The `/user` actor may also be referred to as "The Guardian Actor". But from a user perspective, `/user` is the root of your actor system and is usually just called the "root actor."

> Generally, "root actor" refers to the `/user` actor.

As a user, you don't really need to worry too much about the Guardians. We just have to make sure that we use supervision properly under `/user` so that no exception can bubble up to the Guardians and crash the whole system.

The `/user` actor hierarchy

This is the meat and potatoes of the actor hierarchy: all of the actors you define in your applications.



> The direct children of the `/user` actor are called "top level actors."

Actors are always created as a child of some other actor.

Whenever you make an actor directly from the context of the actor system itself, that new actor is a top level actor, like so:

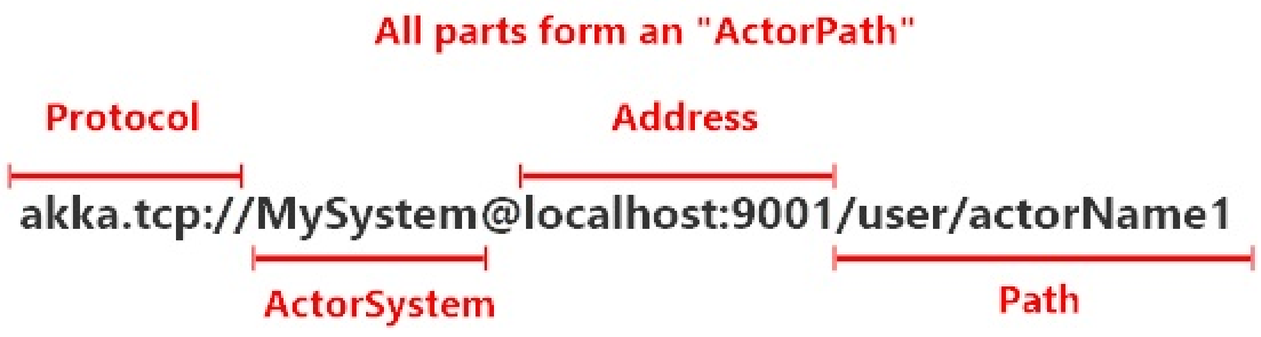
```
// create the top level actors from above diagram
IActorRef a1 = MyActorSystem.ActorOf(Props.Create<BasicActor>(), "a1");
IActorRef a2 = MyActorSystem.ActorOf(Props.Create<BasicActor>(), "a2");
```

Now, let's make child actors for `a2` by creating them inside the context of `a2`, our parent-to-be:

```
// create the children of actor a2
// this is inside actor a2
IActorRef b1 = Context.ActorOf(Props.Create<BasicActor>(), "b1");
IActorRef b2 = Context.ActorOf(Props.Create<BasicActor>(), "b2");
```

Actor path == actor position in hierarchy

Every actor has an address. To send a message from one actor to another, you just have to know it's address (AKA its "ActorPath"). This is what a full actor address looks like:


All parts form an "ActorPath"
Protocol
Address
akka.tcp://MySystem@localhost:9001/user/actorName1
ActorSystem
Path

> *The "Path" portion of an actor address is just a description of where that actor is in your actor hierarchy. Each level of the hierarchy is separated by a single slash ('/').*

For example, if we were running on `localhost`, the full address of actor `b2` would be `akka.tcp://MyActorSystem@localhost:9001/user/a2/b2`.

One question that comes up a lot is, "Do my actor classes have to live at a certain point in the hierarchy?" For example, if I have an actor class, `FooActor`—can I only deploy that actor as a child of `BarActor` on the hierarchy? Or can I deploy it anywhere?

The answer is **any actor may be placed anywhere in your actor hierarchy**.

> *Any actor may be placed anywhere in your actor hierarchy.*

Okay, now that we've got this hierarchy business down, let's do something interesting with it. Like supervising!

How supervision works in the actor hierarchy

Now that you know how actors are organized, know this: actors supervise their children. *But, they only supervise the level that is immediately below them in the hierarchy (actors do not supervise their grandchildren, great-grandchildren, etc).*

> Actors only supervise their children, the level immediately below them in the hierarchy.

When does supervision come into play? Errors!

When things go wrong, that's when! Whenever a child actor has an unhandled exception and is crashing, it reaches out to its parent for help and to tell it what to do.

Specifically, the child will send its parent a message that is of the `Failure` class. Then it's up to the parent to decide what to do.

How can the parent resolve the error?

There are two factors that determine how a failure is resolved:

1. How the child failed (what type of `Exception` did the child include in its `Failure` message to its parent.)
2. What Directive the parent actor executes in response to a child `Failure`. This is determined by the parent's `SupervisionStrategy`.

Here's the sequence of events when an error occurs:

1. Unhandled exception occurs in child actor (`c1`), which is supervised by its parent (`p1`).
2. `c1` suspends operations.
3. The system sends a `Failure` message from `c1` to `p1`, with the `Exception` that was raised.
4. `p1` issues a directive to `c1` telling it what to do.

5. Life goes on, and the affected part of the system heals itself without burning down the whole house. Kittens and unicorns, handing out free ice cream and coffee to be enjoyed while relaxing on a pillowy rainbow. Yay!

Supervision directives

When it receives an error from its child, a parent can take one of the following actions ("directives"). The supervision strategy maps different exception types to these directives, allowing you to handle different types of errors as appropriate.

Types of supervision directives (i.e. what decisions a supervisor can make):

- **Restart** the child (default): this is the common case, and the default.
- **Stop** the child: this permanently terminates the child actor.
- **Escalate** the error (and stop itself): this is the parent saying "I don't know what to do! I'm gonna stop everything and ask MY parent!"
- **Resume** processing (ignores the error): you generally won't use this. Ignore it for now.

> *The critical thing to know here is that* ** whatever action is taken on a parent propagates to its children ** *. If a parent is halted, all its children halt. If it is restarted, all its children restart.*

Supervision strategies

There are two built-in supervision strategies:

1. One-For-One Strategy (default)
2. All-For-One Strategy

   The basic difference between these is how widespread the effects of the error-resolution directive will be.

One-For-One says that that the directive issued by the parent only applies to the failing child actor. It has no effect on the siblings of the failing child. This is the default strategy if you don't specify one. (You can also define your own custom supervision strategy.)

All-For-One says that that the directive issued by the parent applies to the failing child actor AND all of its siblings.

The other important choice you make in a supervision strategy is how many times a child can fail within a given period of time before it is shut down (e.g. "no more than 10 errors within 60 seconds, or you're shut down").

Here's an example supervision strategy:

```
public class MyActor : UntypedActor
{
    // if any child of MyActor throws an exception, apply the rules below
    // e.g. Restart the child, if 10 exceptions occur in 30 seconds or
    // less, then stop the actor
    protected override SupervisorStrategy SupervisorStrategy()
    {
        return new OneForOneStrategy(// or AllForOneStrategy
            maxNrOfRetries: 10,
            withinTimeRange: TimeSpan.FromSeconds(30),
            localOnlyDecider: x =>
            {
                // Maybe ArithmeticException is not application critical
                // so we just ignore the error and keep going.
                if (x is ArithmeticException) return Directive.Resume;

                // Error that we have no idea what to do with
                else if (x is InsanelyBadException) return Directive.Escalate;

                // Error that we can't recover from, stop the failing child
                else if (x is NotSupportedException) return Directive.Stop;

                // otherwise restart the failing child
                else return Directive.Restart;
            });
    }

    ...
}
```

What's the point? Containment.

The whole point of supervision strategies and directives is to contain failure within the system and self-heal, so the whole system doesn't crash. How do we do this?

We push potentially-dangerous operations from a parent to a child, whose only job is to carry out the dangerous task.

For example, let's say we're running a stats system during the World Cup, that keeps scores and player statistics from a bunch of games in the World Cup.

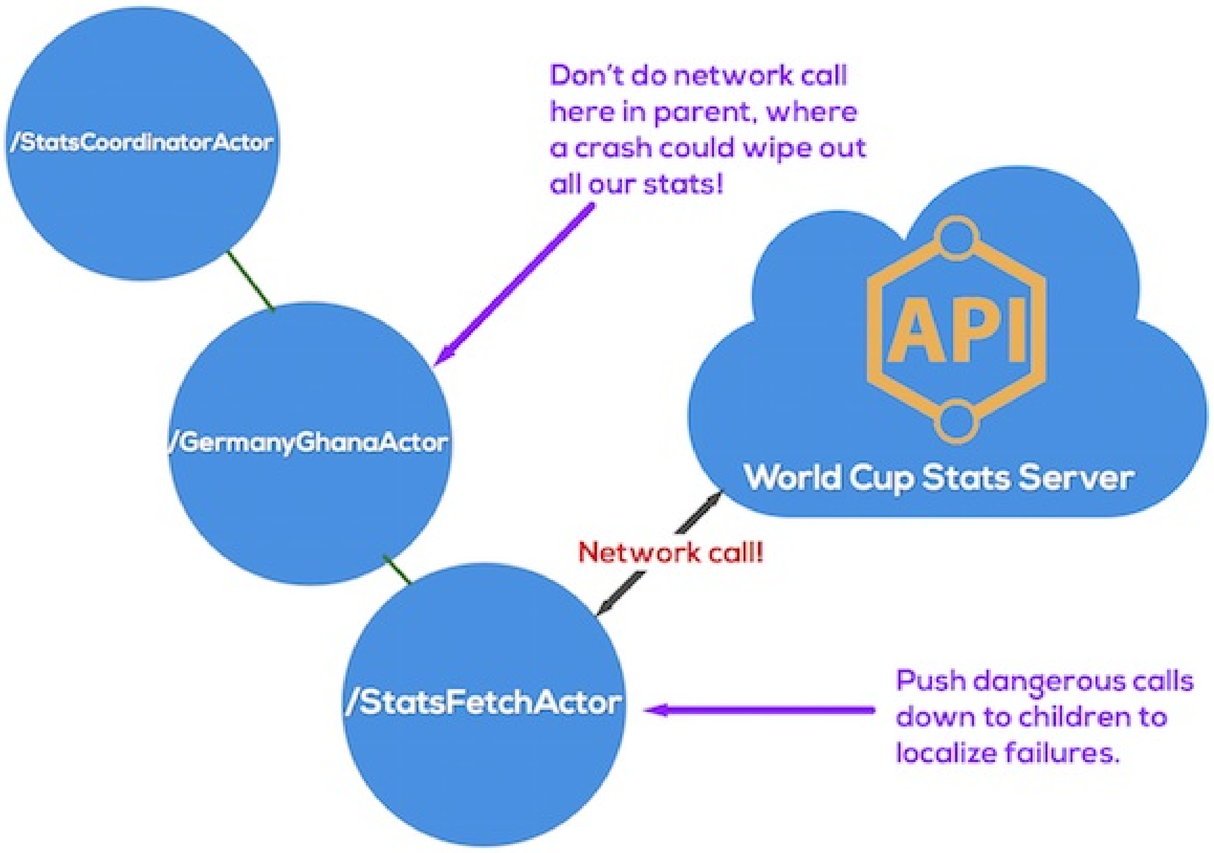
Now, being the World Cup, there could be huge demand on that API and it could get throttled, start rate-limiting, or just plain crash (no offense FIFA, I love you guys and the Cup). We'll use the epic Germany-Ghana match as an example.

But our scorekeeper has to periodically update its data as the game progresses. Let's assume it has to call to an external API maintained by FIFA to get the data it needs.

This network call is dangerous! If the request raises an error, it will crash the actor that started the call. So how do we protect ourselves?

We keep the stats in a parent actor, and push that nasty network call down into a child actor. That way, if the child crashes, it doesn't affect the parent, which is holding on to all the important data. By doing this, we are **localizing the failure** and keeping it from spreading throughout the system.

Here's an example of how we could structure the actor hierarchy to safely accomplish the goal:


/StatsCoordinatorActor
Don't do network call here in parent, where a crash could wipe out all our stats!
API
/GermanyGhanaActor
World Cup Stats Server
Network call!
/StatsFetchActor
Push dangerous calls down to children to localize failures.

Recall that we could have many clones of this exact structure working in parallel, with one clone per game we are tracking. **And we wouldn't have to write any new code to scale it out!** Beautiful.

> You may also hear people use the term "error kernel," which refers to how much of the system is affected by the failure. You may also hear "error kernel pattern," which is just fancy shorthand for the approach I just explained where we push dangerous behavior to child actors to isolate/protect the parent.

Exercise

To start off, we need to do some upgrading of our system. We are going to add in the components which will enable our actor system to actually monitor a file for changes. We have most of the classes we need, but there are a few pieces of utility code that we need to add.

We're almost done! We really just need to add the `TailCoordinatorActor`, `TailActor`, and the `FileObserver`.

The goal of this exercise is to show you how to make a parent/child actor relationship.

Phase 1: A quick bit of prep

Replace `ValidationActor` with `FileValidatorActor`

Since we're shifting to actually looking at files now, go ahead and replace `ValidationActor` with `FileValidatorActor`.

Add a new class, `FileValidatorActor`, with this code:

```
// FileValidatorActor.cs
using System.IO;
using Akka.Actor;

namespace WinTail
{
    /// <summary>
    /// Actor that validates user input and signals result to others.
    /// </summary>
    public class FileValidatorActor : UntypedActor
    {
        private readonly IActorRef _consoleWriterActor;
        private readonly IActorRef _tailCoordinatorActor;

        public FileValidatorActor(IActorRef consoleWriterActor, IActorRef tailCoordinatorAct
        {
            _consoleWriterActor = consoleWriterActor;
            _tailCoordinatorActor = tailCoordinatorActor;
        }

        protected override void OnReceive(object message)
        {
            var msg = message as string;
            if (string.IsNullOrEmpty(msg))
            {
                // signal that the user needs to supply an input
                _consoleWriterActor.Tell(new Messages.NullInputError("Input was blank. Pleas

                // tell sender to continue doing its thing (whatever that may be, this actor
                Sender.Tell(new Messages.ContinueProcessing());
            }
            else
            {
                var valid = IsFileUri(msg);
                if (valid)
                {
                    // signal successful input
                    _consoleWriterActor.Tell(new Messages.InputSuccess(string.Format("Starti

                    // start coordinator
                    _tailCoordinatorActor.Tell(new TailCoordinatorActor.StartTail(msg, _cons
                }
                else
                {
                    // signal that input was bad
                    _consoleWriterActor.Tell(new Messages.ValidationError(string.Format("{0}

                    // tell sender to continue doing its thing (whatever that may be, this a
                    Sender.Tell(new Messages.ContinueProcessing());
                }
            }



        }

        /// <summary>
        /// Checks if file exists at path provided by user.
        /// </summary>
        /// <param name="path"></param>
        /// <returns></returns>
        private static bool IsFileUri(string path)
        {
            return File.Exists(path);
        }
    }
}
```

You'll also want to make sure to update the `Props` instance in `Main` that references the class:

```
// Program.cs
Props validationActorProps = Props.Create(() => new FileValidatorActor(consoleWriterActor));
```

Update `DoPrintInstructions`

Just making a slight tweak to our instructions here, since we'll be using a text file on disk going forward instead of prompting the user for input.

Update `DoPrintInstructions()` to this:

```
// ConsoleReaderActor.cs
private void DoPrintInstructions()
{
    Console.WriteLine("Please provide the URI of a log file on disk.\n");
}
```

Add `FileObserver`

This is a utility class that we're providing for you to use. It does the low-level work of actually watching a file for changes.

Create a new class called `FileObserver` and type in the code for FileObserver.cs. If you're running this on Mono, note the extra environment variable that has to be uncommented in the `Start()` method:

```
// FileObserver.cs
using System;
using System.IO;
using Akka.Actor;

namespace WinTail
{
    /// <summary>
    /// Turns <see cref="FileSystemWatcher"/> events about a specific file into messages for
    /// </summary>
    public class FileObserver : IDisposable
    {
        private readonly IActorRef _tailActor;
        private readonly string _absoluteFilePath;
        private FileSystemWatcher _watcher;
        private readonly string _fileDir;
        private readonly string _fileNameOnly;

        public FileObserver(IActorRef tailActor, string absoluteFilePath)
        {
            _tailActor = tailActor;
            _absoluteFilePath = absoluteFilePath;
            _fileDir = Path.GetDirectoryName(absoluteFilePath);
            _fileNameOnly = Path.GetFileName(absoluteFilePath);
        }

        /// <summary>
        /// Begin monitoring file.
        /// </summary>
        public void Start()
        {
            // Need this for Mono 3.12.0 workaround
            // Environment.SetEnvironmentVariable("MONO_MANAGED_WATCHER", "enabled"); // unc

            // make watcher to observe our specific file
            _watcher = new FileSystemWatcher(_fileDir, _fileNameOnly);

            // watch our file for changes to the file name, or new messages being written to
            _watcher.NotifyFilter = NotifyFilters.FileName | NotifyFilters.LastWrite;

            // assign callbacks for event types
            _watcher.Changed += OnFileChanged;
            _watcher.Error += OnFileError;

            // start watching
            _watcher.EnableRaisingEvents = true;

        }

        /// <summary>
        /// Stop monitoring file.
        /// </summary>
        public void Dispose()
        {
            _watcher.Dispose();
        }

        /// <summary>
        /// Callback for <see cref="FileSystemWatcher"/> file error events.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        void OnFileError(object sender, ErrorEventArgs e)
        {
            _tailActor.Tell(new TailActor.FileError(_fileNameOnly, e.GetException().Message)
        }

        /// <summary>
        /// Callback for <see cref="FileSystemWatcher"/> file change events.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        void OnFileChanged(object sender, FileSystemEventArgs e)
        {
            if (e.ChangeType == WatcherChangeTypes.Changed)
            {
                // here we use a special ActorRefs.NoSender
                // since this event can happen many times, this is a little microoptimizatio
                _tailActor.Tell(new TailActor.FileWrite(e.Name), ActorRefs.NoSender);
            }

        }

    }
}
```

Phase 2: Make your first parent/child actors!

Great! Now we're ready to create our actor classes that will form a parent/child relationship.

Recall that in the hierarchy we're going for, there is a `TailCoordinatorActor` that coordinates child actors to actually monitor and tail files. For now it will only supervise one child, `TailActor`, but in the future it can easily expand to have many children, each observing/tailing a different file.

Add `TailCoordinatorActor`

Create a new class called `TailCoordinatorActor` in a file of the same name.

Add the following code, which defines our coordinator actor (which will soon be our first parent actor).

```
// TailCoordinatorActor.cs
using System;
using Akka.Actor;

namespace WinTail
{
    public class TailCoordinatorActor : UntypedActor
    {
        #region Message types
        /// <summary>
        /// Start tailing the file at user-specified path.
        /// </summary>
        public class StartTail
        {
            public StartTail(string filePath, IActorRef reporterActor)
            {
                FilePath = filePath;
                ReporterActor = reporterActor;
            }

            public string FilePath { get; private set; }

            public IActorRef ReporterActor { get; private set; }
        }

        /// <summary>
        /// Stop tailing the file at user-specified path.
        /// </summary>
        public class StopTail
        {
            public StopTail(string filePath)
            {
                FilePath = filePath;
            }

            public string FilePath { get; private set; }
        }

        #endregion

        protected override void OnReceive(object message)
        {
            if (message is StartTail)
            {
                var msg = message as StartTail;
                // YOU NEED TO FILL IN HERE
            }

        }
    }
}
```

Create `IActorRef` for `TailCoordinatorActor`

In `Main()`, create a new `IActorRef` for `TailCoordinatorActor` and then pass it into `fileValidatorActorProps`, like so:

```
// Program.Main
// make tailCoordinatorActor
Props tailCoordinatorProps = Props.Create(() => new TailCoordinatorActor());
IActorRef tailCoordinatorActor = MyActorSystem.ActorOf(tailCoordinatorProps, "tailCoordinato
// pass tailCoordinatorActor to fileValidatorActorProps (just adding one extra arg)
Props fileValidatorActorProps = Props.Create(() => new FileValidatorActor(consoleWriterActor
IActorRef validationActor = MyActorSystem.ActorOf(fileValidatorActorProps, "validationActor"
```

Add `TailActor`

Now, add a class called `TailActor` in its own file. This actor is the actor that is actually responsible for tailing a given file. `TailActor` will be created and supervised by `TailCoordinatorActor` in a moment.

For now, add the following code in `TailActor.cs`:

```
// TailActor.cs
using System.IO;
using System.Text;
using Akka.Actor;

namespace WinTail
{
    /// <summary>
    /// Monitors the file at <see cref="_filePath"/> for changes and sends file updates to c
    /// </summary>
    public class TailActor : UntypedActor
    {
        #region Message types

        /// <summary>
        /// Signal that the file has changed, and we need to read the next line of the file.
        /// </summary>
        public class FileWrite
        {
            public FileWrite(string fileName)
            {
                FileName = fileName;
            }

            public string FileName { get; private set; }
        }

        /// <summary>
        /// Signal that the OS had an error accessing the file.
        /// </summary>
        public class FileError
        {
            public FileError(string fileName, string reason)
            {
                FileName = fileName;
                Reason = reason;
            }

            public string FileName { get; private set; }

            public string Reason { get; private set; }
        }

        /// <summary>
        /// Signal to read the initial contents of the file at actor startup.
        /// </summary>
        public class InitialRead
        {
            public InitialRead(string fileName, string text)
            {
                FileName = fileName;
                Text = text;
            }

            public string FileName { get; private set; }
            public string Text { get; private set; }
        }

        #endregion

        private readonly string _filePath;
        private readonly IActorRef _reporterActor;
        private readonly FileObserver _observer;
        private readonly Stream _fileStream;
        private readonly StreamReader _fileStreamReader;

        public TailActor(IActorRef reporterActor, string filePath)
        {
            _reporterActor = reporterActor;
            _filePath = filePath;

            // start watching file for changes
            _observer = new FileObserver(Self, Path.GetFullPath(_filePath));
            _observer.Start();

            // open the file stream with shared read/write permissions (so file can be writt
            _fileStream = new FileStream(Path.GetFullPath(_filePath), FileMode.Open, FileAcc
                FileShare.ReadWrite);
            _fileStreamReader = new StreamReader(_fileStream, Encoding.UTF8);

            // read the initial contents of the file and send it to console as first message
            var text = _fileStreamReader.ReadToEnd();
            Self.Tell(new InitialRead(_filePath, text));
        }

        protected override void OnReceive(object message)
        {
            if (message is FileWrite)
            {
                // move file cursor forward
                // pull results from cursor to end of file and write to output
                // (this is assuming a log file type format that is append-only)
                var text = _fileStreamReader.ReadToEnd();
                if (!string.IsNullOrEmpty(text))
                {
                    _reporterActor.Tell(text);
                }

            }
            else if (message is FileError)
            {
                var fe = message as FileError;
                _reporterActor.Tell(string.Format("Tail error: {0}", fe.Reason));
            }
            else if (message is InitialRead)
            {
                var ir = message as InitialRead;
                _reporterActor.Tell(ir.Text);
            }
        }
    }
}
```

Add `TailActor` as a child of `TailCoordinatorActor`

Quick review: `TailActor` is to be a child of `TailCoordinatorActor` and will therefore be supervised by `TailCoordinatorActor`.

This also means that `TailActor` must be created in the context of `TailCoordinatorActor`.

Go to `TailCoordinatorActor.cs` and replace `OnReceive()` with the following code to create your first child actor!

```
// TailCoordinatorActor.OnReceive
protected override void OnReceive(object message)
{
    if (message is StartTail)
    {
        var msg = message as StartTail;
        // here we are creating our first parent/child relationship!
        // the TailActor instance created here is a child
        // of this instance of TailCoordinatorActor
        Context.ActorOf(Props.Create(() => new TailActor(msg.ReporterActor, msg.FilePath)));
    }

}
```

BAM!

You have just established your first parent/child actor relationship!

Phase 3: Implement a `SupervisorStrategy`

Now it's time to add a supervision strategy to your new parent, `TailCoordinatorActor`.

The default `SupervisorStrategy` is a One-For-One strategy (docs) w/ a Restart directive (docs).

Add this code to the bottom of `TailCoordinatorActor`:

```
// TailCoordinatorActor.cs
protected override SupervisorStrategy SupervisorStrategy()
{
    return new OneForOneStrategy (
        10, // maxNumberOfRetries
        TimeSpan.FromSeconds(30), // withinTimeRange
        x => // localOnlyDecider
        {
            //Maybe we consider ArithmeticException to not be application critical
            //so we just ignore the error and keep going.
            if (x is ArithmeticException) return Directive.Resume;

            //Error that we cannot recover from, stop the failing actor
            else if (x is NotSupportedException) return Directive.Stop;

            //In all other cases, just restart the failing actor
            else return Directive.Restart;
        });
}
```

Phase 4: Build and Run!

Awesome! It's time to fire this baby up and see it in action.

Get a text file you can tail

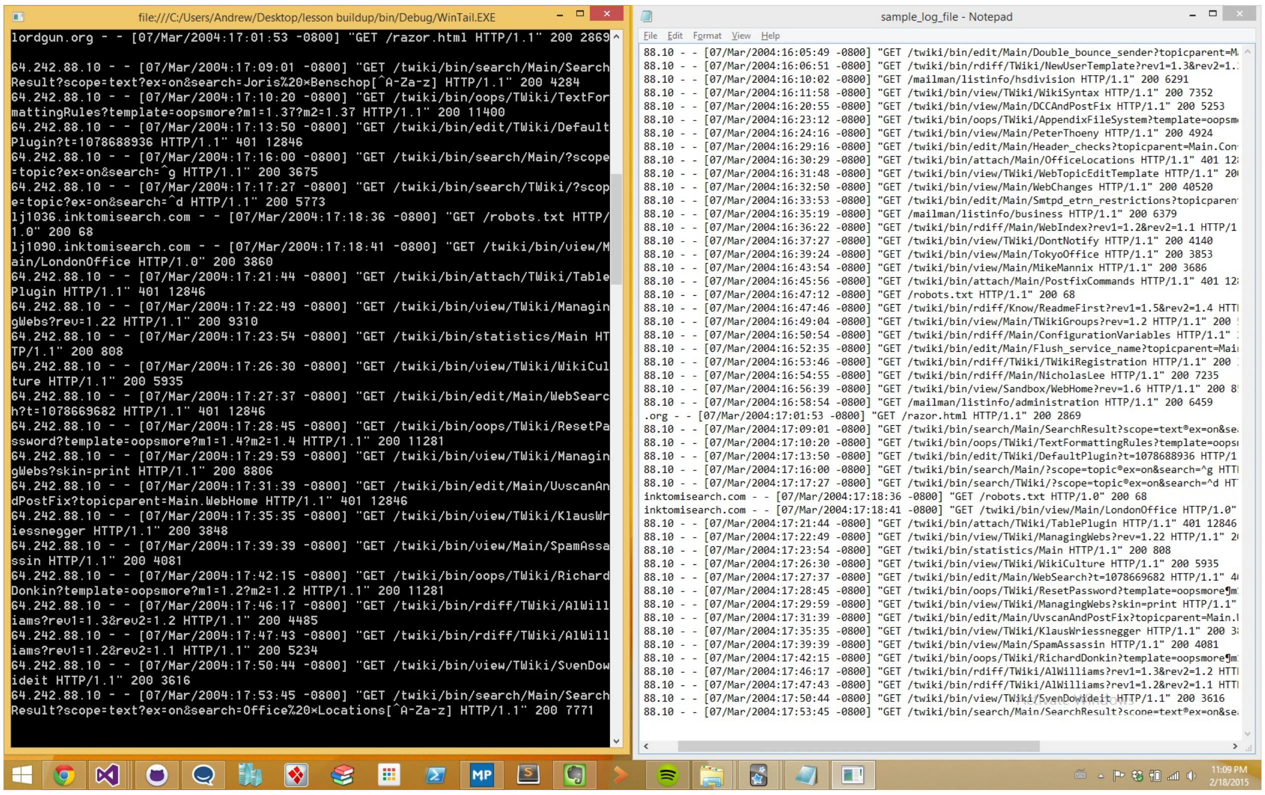
We recommend a log file like this sample one, but you can also just make a plain text file and fill it with whatever you want.

Open the text file up and put it on one side of your screen.

Fire it up

Check the starting output

Run the application and you should see a console window open up and print out the starting contents of your log file. The starting state should look like this if you're using the sample log file we provided:


file:///C:/Users/Andrew/Desktop/lesson buildup/bin/Debug/WinTail.EXE
sample_log_file - Notepad
File Edit Format View Help
11:09 PM
2/18/2015

```
lordgun.org - - [07/Mar/2004:17:01:53 -0800] "GET /razor.html HTTP/1.1" 200 2869

64.242.88.10 - - [07/Mar/2004:17:09:01 -0800] "GET /twiki/bin/search/Main/Search
Result?scope=text?ex=on&search=Joris%20*Benschop[^A-Za-z] HTTP/1.1" 200 4284
64.242.88.10 - - [07/Mar/2004:17:10:20 -0800] "GET /twiki/bin/oops/TWiki/TextFor
mattingRules?template=oopsmore?m1=1.37?m2=1.37 HTTP/1.1" 200 11400
64.242.88.10 - - [07/Mar/2004:17:13:50 -0800] "GET /twiki/bin/edit/TWiki/Default
Plugin?t=1078688936 HTTP/1.1" 401 12846
64.242.88.10 - - [07/Mar/2004:17:16:00 -0800] "GET /twiki/bin/search/Main/?scope
=topic?ex=on&search=^g HTTP/1.1" 200 3675
64.242.88.10 - - [07/Mar/2004:17:17:27 -0800] "GET /twiki/bin/search/TWiki/?scop
e=topic?ex=on&search=^d HTTP/1.1" 200 5773
lj1036.inktomisearch.com - - [07/Mar/2004:17:18:36 -0800] "GET /robots.txt HTTP/
1.0" 200 68
lj1090.inktomisearch.com - - [07/Mar/2004:17:18:41 -0800] "GET /twiki/bin/view/M
ain/LondonOffice HTTP/1.0" 200 3860
64.242.88.10 - - [07/Mar/2004:17:21:44 -0800] "GET /twiki/bin/attach/TWiki/Table
Plugin HTTP/1.1" 401 12846
64.242.88.10 - - [07/Mar/2004:17:22:49 -0800] "GET /twiki/bin/view/TWiki/Managin
gWebs?rev=1.22 HTTP/1.1" 200 9310
64.242.88.10 - - [07/Mar/2004:17:23:54 -0800] "GET /twiki/bin/statistics/Main HT
TP/1.1" 200 808
64.242.88.10 - - [07/Mar/2004:17:26:30 -0800] "GET /twiki/bin/view/TWiki/WikiCul
ture HTTP/1.1" 200 5935
64.242.88.10 - - [07/Mar/2004:17:27:37 -0800] "GET /twiki/bin/edit/Main/WebSearc
h?t=1078669682 HTTP/1.1" 401 12846
64.242.88.10 - - [07/Mar/2004:17:28:45 -0800] "GET /twiki/bin/oops/TWiki/ResetPa
ssword?template=oopsmore?m1=1.4?m2=1.4 HTTP/1.1" 200 11281
64.242.88.10 - - [07/Mar/2004:17:29:59 -0800] "GET /twiki/bin/view/TWiki/Managin
gWebs?skin=print HTTP/1.1" 200 8806
64.242.88.10 - - [07/Mar/2004:17:31:39 -0800] "GET /twiki/bin/edit/Main/UvscanAn
dPostFix?topicparent=Main.WebHome HTTP/1.1" 401 12846
64.242.88.10 - - [07/Mar/2004:17:35:35 -0800] "GET /twiki/bin/view/TWiki/KlausWr
iessnegger HTTP/1.1" 200 3848
64.242.88.10 - - [07/Mar/2004:17:39:39 -0800] "GET /twiki/bin/view/Main/SpamAssa
ssin HTTP/1.1" 200 4081
64.242.88.10 - - [07/Mar/2004:17:42:15 -0800] "GET /twiki/bin/oops/TWiki/Richard
Donkin?template=oopsmore?m1=1.2?m2=1.2 HTTP/1.1" 200 11281
64.242.88.10 - - [07/Mar/2004:17:46:17 -0800] "GET /twiki/bin/rdiff/TWiki/AlWill
iams?rev1=1.3&rev2=1.2 HTTP/1.1" 200 4485
64.242.88.10 - - [07/Mar/2004:17:47:43 -0800] "GET /twiki/bin/rdiff/TWiki/AlWill
iams?rev1=1.2&rev2=1.1 HTTP/1.1" 200 5234
64.242.88.10 - - [07/Mar/2004:17:50:44 -0800] "GET /twiki/bin/view/TWiki/SvenDow
ideit HTTP/1.1" 200 3616
64.242.88.10 - - [07/Mar/2004:17:53:45 -0800] "GET /twiki/bin/search/Main/Search
Result?scope=text?ex=on&search=Office%20*Locations[^A-Za-z] HTTP/1.1" 200 7771
```

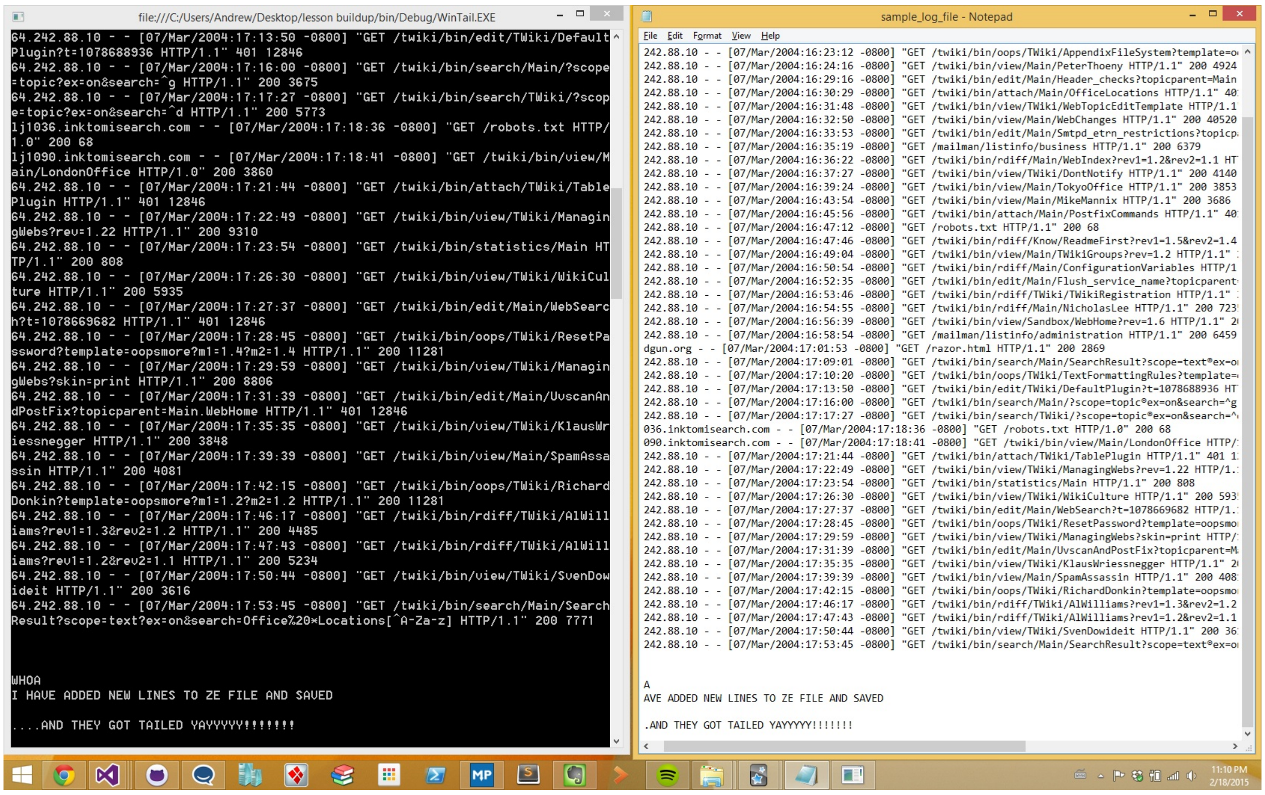
```
88.10 - - [07/Mar/2004:16:05:49 -0800] "GET /twiki/bin/edit/Main/Double_bounce_sender?topicparent=M
88.10 - - [07/Mar/2004:16:06:51 -0800] "GET /twiki/bin/rdiff/TWiki/NewUserTemplate?rev1=1.3&rev2=1.
88.10 - - [07/Mar/2004:16:10:02 -0800] "GET /mailman/listinfo/hsdivision HTTP/1.1" 200 6291
88.10 - - [07/Mar/2004:16:11:58 -0800] "GET /twiki/bin/view/TWiki/WikiSyntax HTTP/1.1" 200 7352
88.10 - - [07/Mar/2004:16:20:55 -0800] "GET /twiki/bin/view/Main/DCCAndPostFix HTTP/1.1" 200 5253
88.10 - - [07/Mar/2004:16:23:12 -0800] "GET /twiki/bin/oops/TWiki/AppendixFileSystem?template=oopsm
88.10 - - [07/Mar/2004:16:24:16 -0800] "GET /twiki/bin/view/Main/PeterThoeny HTTP/1.1" 200 4924
88.10 - - [07/Mar/2004:16:29:16 -0800] "GET /twiki/bin/edit/Main/Header_checks?topicparent=Main.Con
88.10 - - [07/Mar/2004:16:30:29 -0800] "GET /twiki/bin/attach/Main/OfficeLocations HTTP/1.1" 401 12
88.10 - - [07/Mar/2004:16:31:48 -0800] "GET /twiki/bin/view/TWiki/WebTopicEditTemplate HTTP/1.1" 20
88.10 - - [07/Mar/2004:16:32:50 -0800] "GET /twiki/bin/view/Main/WebChanges HTTP/1.1" 200 40520
88.10 - - [07/Mar/2004:16:33:53 -0800] "GET /twiki/bin/edit/Main/Smtpd_etrn_restrictions?topicparen
88.10 - - [07/Mar/2004:16:35:19 -0800] "GET /mailman/listinfo/business HTTP/1.1" 200 6379
88.10 - - [07/Mar/2004:16:36:22 -0800] "GET /twiki/bin/rdiff/Main/WebIndex?rev1=1.2&rev2=1.1 HTTP/1
88.10 - - [07/Mar/2004:16:37:27 -0800] "GET /twiki/bin/view/TWiki/DontNotify HTTP/1.1" 200 4140
88.10 - - [07/Mar/2004:16:39:24 -0800] "GET /twiki/bin/view/Main/TokyoOffice HTTP/1.1" 200 3853
88.10 - - [07/Mar/2004:16:43:54 -0800] "GET /twiki/bin/view/Main/MikeMannix HTTP/1.1" 200 3686
88.10 - - [07/Mar/2004:16:45:56 -0800] "GET /twiki/bin/attach/Main/PostfixCommands HTTP/1.1" 401 12
88.10 - - [07/Mar/2004:16:47:12 -0800] "GET /robots.txt HTTP/1.1" 200 68
88.10 - - [07/Mar/2004:16:47:46 -0800] "GET /twiki/bin/rdiff/Know/ReadmeFirst?rev1=1.5&rev2=1.4 HTT
88.10 - - [07/Mar/2004:16:49:04 -0800] "GET /twiki/bin/view/Main/TWikiGroups?rev=1.2 HTTP/1.1" 200
88.10 - - [07/Mar/2004:16:50:54 -0800] "GET /twiki/bin/rdiff/Main/ConfigurationVariables HTTP/1.1"
88.10 - - [07/Mar/2004:16:52:35 -0800] "GET /twiki/bin/edit/Main/Flush_service_name?topicparent=Mai
88.10 - - [07/Mar/2004:16:53:46 -0800] "GET /twiki/bin/rdiff/TWiki/TWikiRegistration HTTP/1.1" 200
88.10 - - [07/Mar/2004:16:54:55 -0800] "GET /twiki/bin/rdiff/Main/NicholasLee HTTP/1.1" 200 7235
88.10 - - [07/Mar/2004:16:56:39 -0800] "GET /twiki/bin/view/Sandbox/WebHome?rev=1.6 HTTP/1.1" 200 8
88.10 - - [07/Mar/2004:16:58:54 -0800] "GET /mailman/listinfo/administration HTTP/1.1" 200 6459
.org - - [07/Mar/2004:17:01:53 -0800] "GET /razor.html HTTP/1.1" 200 2869
88.10 - - [07/Mar/2004:17:09:01 -0800] "GET /twiki/bin/search/Main/SearchResult?scope=text®ex=on&se
88.10 - - [07/Mar/2004:17:10:20 -0800] "GET /twiki/bin/oops/TWiki/TextFormattingRules?template=oops
88.10 - - [07/Mar/2004:17:13:50 -0800] "GET /twiki/bin/edit/TWiki/DefaultPlugin?t=1078688936 HTTP/1
88.10 - - [07/Mar/2004:17:16:00 -0800] "GET /twiki/bin/search/Main/?scope=topic®ex=on&search=^g HTT
88.10 - - [07/Mar/2004:17:17:27 -0800] "GET /twiki/bin/search/TWiki/?scope=topic®ex=on&search=^d HT
inktomisearch.com - - [07/Mar/2004:17:18:36 -0800] "GET /robots.txt HTTP/1.0" 200 68
inktomisearch.com - - [07/Mar/2004:17:18:41 -0800] "GET /twiki/bin/view/Main/LondonOffice HTTP/1.0"
88.10 - - [07/Mar/2004:17:21:44 -0800] "GET /twiki/bin/attach/TWiki/TablePlugin HTTP/1.1" 401 12846
88.10 - - [07/Mar/2004:17:22:49 -0800] "GET /twiki/bin/view/TWiki/ManagingWebs?rev=1.22 HTTP/1.1" 2
88.10 - - [07/Mar/2004:17:23:54 -0800] "GET /twiki/bin/statistics/Main HTTP/1.1" 200 808
88.10 - - [07/Mar/2004:17:26:30 -0800] "GET /twiki/bin/view/TWiki/WikiCulture HTTP/1.1" 200 5935
88.10 - - [07/Mar/2004:17:27:37 -0800] "GET /twiki/bin/edit/Main/WebSearch?t=1078669682 HTTP/1.1" 4
88.10 - - [07/Mar/2004:17:28:45 -0800] "GET /twiki/bin/oops/TWiki/ResetPassword?template=oopsmore¶m
88.10 - - [07/Mar/2004:17:29:59 -0800] "GET /twiki/bin/view/TWiki/ManagingWebs?skin=print HTTP/1.1"
88.10 - - [07/Mar/2004:17:31:39 -0800] "GET /twiki/bin/edit/Main/UvscanAndPostFix?topicparent=Main.
88.10 - - [07/Mar/2004:17:35:35 -0800] "GET /twiki/bin/view/TWiki/KlausWriessnegger HTTP/1.1" 200 3
88.10 - - [07/Mar/2004:17:39:39 -0800] "GET /twiki/bin/view/Main/SpamAssassin HTTP/1.1" 200 4081
88.10 - - [07/Mar/2004:17:42:15 -0800] "GET /twiki/bin/oops/TWiki/RichardDonkin?template=oopsmore¶m
88.10 - - [07/Mar/2004:17:46:17 -0800] "GET /twiki/bin/rdiff/TWiki/AlWilliams?rev1=1.3&rev2=1.2 HTT
88.10 - - [07/Mar/2004:17:47:43 -0800] "GET /twiki/bin/rdiff/TWiki/AlWilliams?rev1=1.2&rev2=1.1 HTT
88.10 - - [07/Mar/2004:17:50:44 -0800] "GET /twiki/bin/view/TWiki/SvenDowideit HTTP/1.1" 200 3616
88.10 - - [07/Mar/2004:17:53:45 -0800] "GET /twiki/bin/search/Main/SearchResult?scope=text®ex=on&se
```

Leave both the console and the file open, and then...

Add text and see if the `tail` works!

Add some lines of text to the text file, save it, and watch it show up in the `tail`!

It should look something like this:


file:///C:/Users/Andrew/Desktop/lesson buildup/bin/Debug/WinTail.EXE
sample_log_file - Notepad
File Edit Format View Help
11:10 PM
2/18/2015

```
64.242.88.10 - - [07/Mar/2004:17:13:50 -0800] "GET /twiki/bin/edit/TWiki/Default
Plugin?t=1078688936 HTTP/1.1" 401 12846
64.242.88.10 - - [07/Mar/2004:17:16:00 -0800] "GET /twiki/bin/search/Main/?scope
=topic?ex=on&search=^g HTTP/1.1" 200 3675
64.242.88.10 - - [07/Mar/2004:17:17:27 -0800] "GET /twiki/bin/search/TWiki/?scop
e=topic?ex=on&search=^d HTTP/1.1" 200 5773
lj1036.inktomisearch.com - - [07/Mar/2004:17:18:36 -0800] "GET /robots.txt HTTP/
1.0" 200 68
lj1090.inktomisearch.com - - [07/Mar/2004:17:18:41 -0800] "GET /twiki/bin/view/M
ain/LondonOffice HTTP/1.0" 200 3860
64.242.88.10 - - [07/Mar/2004:17:21:44 -0800] "GET /twiki/bin/attach/TWiki/Table
Plugin HTTP/1.1" 401 12846
64.242.88.10 - - [07/Mar/2004:17:22:49 -0800] "GET /twiki/bin/view/TWiki/Managin
gWebs?rev=1.22 HTTP/1.1" 200 9310
64.242.88.10 - - [07/Mar/2004:17:23:54 -0800] "GET /twiki/bin/statistics/Main HT
TP/1.1" 200 808
64.242.88.10 - - [07/Mar/2004:17:26:30 -0800] "GET /twiki/bin/view/TWiki/WikiCul
ture HTTP/1.1" 200 5935
64.242.88.10 - - [07/Mar/2004:17:27:37 -0800] "GET /twiki/bin/edit/Main/WebSearc
h?t=1078669682 HTTP/1.1" 401 12846
64.242.88.10 - - [07/Mar/2004:17:28:45 -0800] "GET /twiki/bin/oops/TWiki/ResetPa
ssword?template=oopsmore?m1=1.4?m2=1.4 HTTP/1.1" 200 11281
64.242.88.10 - - [07/Mar/2004:17:29:59 -0800] "GET /twiki/bin/view/TWiki/Managin
gWebs?skin=print HTTP/1.1" 200 8806
64.242.88.10 - - [07/Mar/2004:17:31:39 -0800] "GET /twiki/bin/edit/Main/UvscanAn
dPostFix?topicparent=Main.WebHome HTTP/1.1" 401 12846
64.242.88.10 - - [07/Mar/2004:17:35:35 -0800] "GET /twiki/bin/view/TWiki/KlausWr
iessnegger HTTP/1.1" 200 3848
64.242.88.10 - - [07/Mar/2004:17:39:39 -0800] "GET /twiki/bin/view/Main/SpamAssa
ssin HTTP/1.1" 200 4081
64.242.88.10 - - [07/Mar/2004:17:42:15 -0800] "GET /twiki/bin/oops/TWiki/Richard
Donkin?template=oopsmore?m1=1.2?m2=1.2 HTTP/1.1" 200 11281
64.242.88.10 - - [07/Mar/2004:17:46:17 -0800] "GET /twiki/bin/rdiff/TWiki/AlWill
iams?rev1=1.3&rev2=1.2 HTTP/1.1" 200 4485
64.242.88.10 - - [07/Mar/2004:17:47:43 -0800] "GET /twiki/bin/rdiff/TWiki/AlWill
iams?rev1=1.2&rev2=1.1 HTTP/1.1" 200 5234
64.242.88.10 - - [07/Mar/2004:17:50:44 -0800] "GET /twiki/bin/view/TWiki/SvenDow
ideit HTTP/1.1" 200 3616
64.242.88.10 - - [07/Mar/2004:17:53:45 -0800] "GET /twiki/bin/search/Main/Search
Result?scope=text?ex=on&search=Office%20*Locations[^A-Za-z] HTTP/1.1" 200 7771



WHOA
I HAVE ADDED NEW LINES TO ZE FILE AND SAVED

....AND THEY GOT TAILED YAYYYYY!!!!!!!!
```

```
242.88.10 - - [07/Mar/2004:16:23:12 -0800] "GET /twiki/bin/oops/TWiki/AppendixFileSystem?template=o
242.88.10 - - [07/Mar/2004:16:24:16 -0800] "GET /twiki/bin/view/Main/PeterThoeny HTTP/1.1" 200 4924
242.88.10 - - [07/Mar/2004:16:29:16 -0800] "GET /twiki/bin/edit/Main/Header_checks?topicparent=Main
242.88.10 - - [07/Mar/2004:16:30:29 -0800] "GET /twiki/bin/attach/Main/OfficeLocations HTTP/1.1" 40
242.88.10 - - [07/Mar/2004:16:31:48 -0800] "GET /twiki/bin/view/TWiki/WebTopicEditTemplate HTTP/1.1
242.88.10 - - [07/Mar/2004:16:32:50 -0800] "GET /twiki/bin/view/Main/WebChanges HTTP/1.1" 200 40520
242.88.10 - - [07/Mar/2004:16:33:53 -0800] "GET /twiki/bin/edit/Main/Smtpd_etrn_restrictions?topicp
242.88.10 - - [07/Mar/2004:16:35:19 -0800] "GET /mailman/listinfo/business HTTP/1.1" 200 6379
242.88.10 - - [07/Mar/2004:16:36:22 -0800] "GET /twiki/bin/rdiff/Main/WebIndex?rev1=1.2&rev2=1.1 HT
242.88.10 - - [07/Mar/2004:16:37:27 -0800] "GET /twiki/bin/view/TWiki/DontNotify HTTP/1.1" 200 4140
242.88.10 - - [07/Mar/2004:16:39:24 -0800] "GET /twiki/bin/view/Main/TokyoOffice HTTP/1.1" 200 3853
242.88.10 - - [07/Mar/2004:16:43:54 -0800] "GET /twiki/bin/view/Main/MikeMannix HTTP/1.1" 200 3686
242.88.10 - - [07/Mar/2004:16:45:56 -0800] "GET /twiki/bin/attach/Main/PostfixCommands HTTP/1.1" 40
242.88.10 - - [07/Mar/2004:16:47:12 -0800] "GET /robots.txt HTTP/1.1" 200 68
242.88.10 - - [07/Mar/2004:16:47:46 -0800] "GET /twiki/bin/rdiff/Know/ReadmeFirst?rev1=1.5&rev2=1.4
242.88.10 - - [07/Mar/2004:16:49:04 -0800] "GET /twiki/bin/view/Main/TWikiGroups?rev=1.2 HTTP/1.1"
242.88.10 - - [07/Mar/2004:16:50:54 -0800] "GET /twiki/bin/rdiff/Main/ConfigurationVariables HTTP/1
242.88.10 - - [07/Mar/2004:16:52:35 -0800] "GET /twiki/bin/edit/Main/Flush_service_name?topicparent
242.88.10 - - [07/Mar/2004:16:53:46 -0800] "GET /twiki/bin/rdiff/TWiki/TWikiRegistration HTTP/1.1"
242.88.10 - - [07/Mar/2004:16:54:55 -0800] "GET /twiki/bin/rdiff/Main/NicholasLee HTTP/1.1" 200 723
242.88.10 - - [07/Mar/2004:16:56:39 -0800] "GET /twiki/bin/view/Sandbox/WebHome?rev=1.6 HTTP/1.1" 2
242.88.10 - - [07/Mar/2004:16:58:54 -0800] "GET /mailman/listinfo/administration HTTP/1.1" 200 6459
dgun.org - - [07/Mar/2004:17:01:53 -0800] "GET /razor.html HTTP/1.1" 200 2869
242.88.10 - - [07/Mar/2004:17:09:01 -0800] "GET /twiki/bin/search/Main/SearchResult?scope=text®ex=o
242.88.10 - - [07/Mar/2004:17:10:20 -0800] "GET /twiki/bin/oops/TWiki/TextFormattingRules?template=
242.88.10 - - [07/Mar/2004:17:13:50 -0800] "GET /twiki/bin/edit/TWiki/DefaultPlugin?t=1078688936 HT
242.88.10 - - [07/Mar/2004:17:16:00 -0800] "GET /twiki/bin/search/Main/?scope=topic®ex=on&search=^g
242.88.10 - - [07/Mar/2004:17:17:27 -0800] "GET /twiki/bin/search/TWiki/?scope=topic®ex=on&search=^
036.inktomisearch.com - - [07/Mar/2004:17:18:36 -0800] "GET /robots.txt HTTP/1.0" 200 68
090.inktomisearch.com - - [07/Mar/2004:17:18:41 -0800] "GET /twiki/bin/view/Main/LondonOffice HTTP/
242.88.10 - - [07/Mar/2004:17:21:44 -0800] "GET /twiki/bin/attach/TWiki/TablePlugin HTTP/1.1" 401 1
242.88.10 - - [07/Mar/2004:17:22:49 -0800] "GET /twiki/bin/view/TWiki/ManagingWebs?rev=1.22 HTTP/1.
242.88.10 - - [07/Mar/2004:17:23:54 -0800] "GET /twiki/bin/statistics/Main HTTP/1.1" 200 808
242.88.10 - - [07/Mar/2004:17:26:30 -0800] "GET /twiki/bin/view/TWiki/WikiCulture HTTP/1.1" 200 593
242.88.10 - - [07/Mar/2004:17:27:37 -0800] "GET /twiki/bin/edit/Main/WebSearch?t=1078669682 HTTP/1.
242.88.10 - - [07/Mar/2004:17:28:45 -0800] "GET /twiki/bin/oops/TWiki/ResetPassword?template=oopsmo
242.88.10 - - [07/Mar/2004:17:29:59 -0800] "GET /twiki/bin/view/TWiki/ManagingWebs?skin=print HTTP/
242.88.10 - - [07/Mar/2004:17:31:39 -0800] "GET /twiki/bin/edit/Main/UvscanAndPostFix?topicparent=M
242.88.10 - - [07/Mar/2004:17:35:35 -0800] "GET /twiki/bin/view/TWiki/KlausWriessnegger HTTP/1.1" 2
242.88.10 - - [07/Mar/2004:17:39:39 -0800] "GET /twiki/bin/view/Main/SpamAssassin HTTP/1.1" 200 408
242.88.10 - - [07/Mar/2004:17:42:15 -0800] "GET /twiki/bin/oops/TWiki/RichardDonkin?template=oopsmo
242.88.10 - - [07/Mar/2004:17:46:17 -0800] "GET /twiki/bin/rdiff/TWiki/AlWilliams?rev1=1.3&rev2=1.2
242.88.10 - - [07/Mar/2004:17:47:43 -0800] "GET /twiki/bin/rdiff/TWiki/AlWilliams?rev1=1.2&rev2=1.1
242.88.10 - - [07/Mar/2004:17:50:44 -0800] "GET /twiki/bin/view/TWiki/SvenDowideit HTTP/1.1" 200 36
242.88.10 - - [07/Mar/2004:17:53:45 -0800] "GET /twiki/bin/search/Main/SearchResult?scope=text®ex=o



A
AVE ADDED NEW LINES TO ZE FILE AND SAVED

.AND THEY GOT TAILED YAYYYYY!!!!!!!
```

Congrats! YOU HAVE JUST MADE A PORT OF `tail` IN .NET!

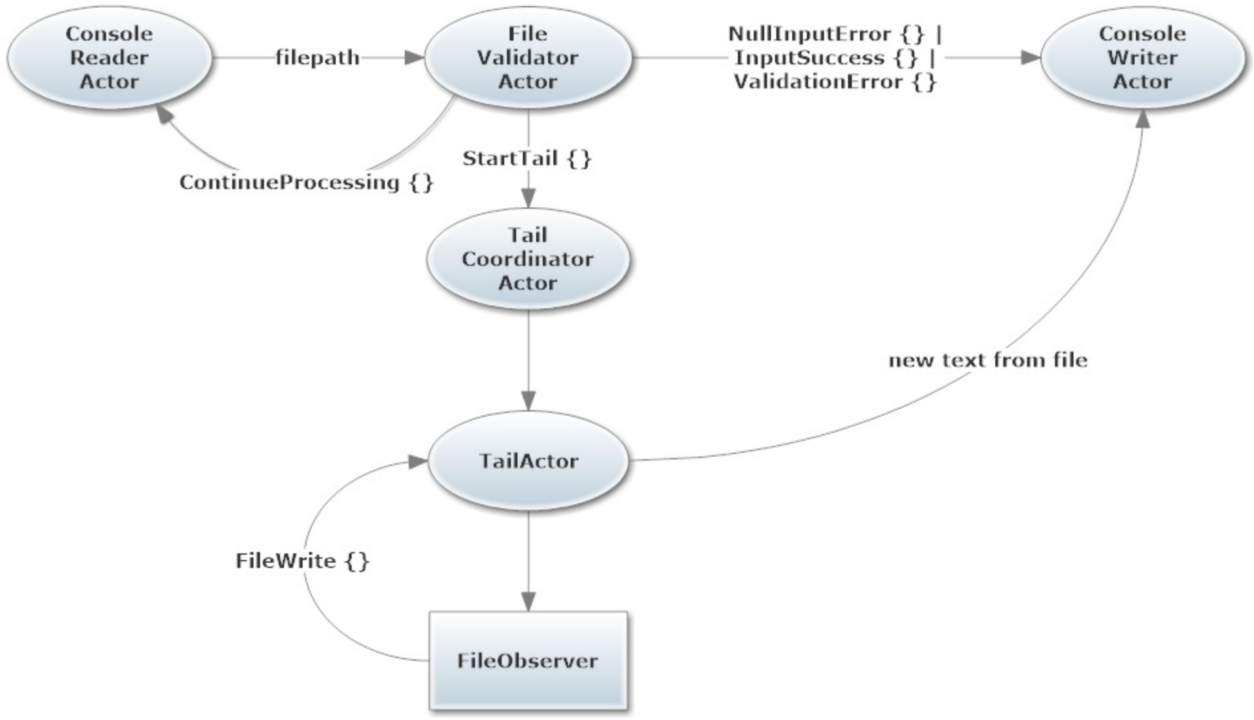
Once you're done

Compare your code to the solution in the Completed folder to see what the instructors included in their samples.

Great job! Onto Lesson 5!

Awesome work! Well done on completing this lesson, we know it was a bear! It was a big jump forward for our system and in your understanding.

Here is a high-level overview of our working system!



Let's move onto Lesson 5 - Looking up Actors by Address with `ActorSelection`.

Supervision FAQ

How long do child actors have to wait for their supervisor?

This is a common question we get: What if there are a bunch of messages already in the supervisor's mailbox waiting to be processed when a child reports an error? Won't the crashing child actor have to wait until those are processed until it gets a response?

Actually, no. When an actor reports an error to its supervisor, it is sent as a special type of "system message." *System messages jump to the front of the supervisor's mailbox and are processed before the supervisor returns to its normal processing.*

> *System messages jump to the front of the supervisor's mailbox and are processed before the supervisor returns to its normal processing.*

Parents come with a default SupervisorStrategy object (or you can provide a custom one) that makes decisions on how to handle failures with their child actors.

But what happens to the current message when an actor fails?

The current message being processed by an actor when it is halted (regardless of whether the failure happened to it or its parent) can be saved and re-processed after restarting. There are several ways to do this. The most common approach used is during `preRestart()`, the actor can stash the message (if it has a stash) or it can send the message to another actor that will send it back once restarted. (Note: If the actor has a stash, it will automatically unstash the message once it successfully restarts.)

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 1.6: The Actor Lifecycle

Wow! Look at you--made it all the way to the end of Unit 1! Congratulations. Seriously. We appreciate and commend you on your dedication to learning and growing as a developer.

This last lesson will wrap up our "fundamentals series" on working with actors, and it ends with a critical concept: actor life cycle.

Key concepts / background

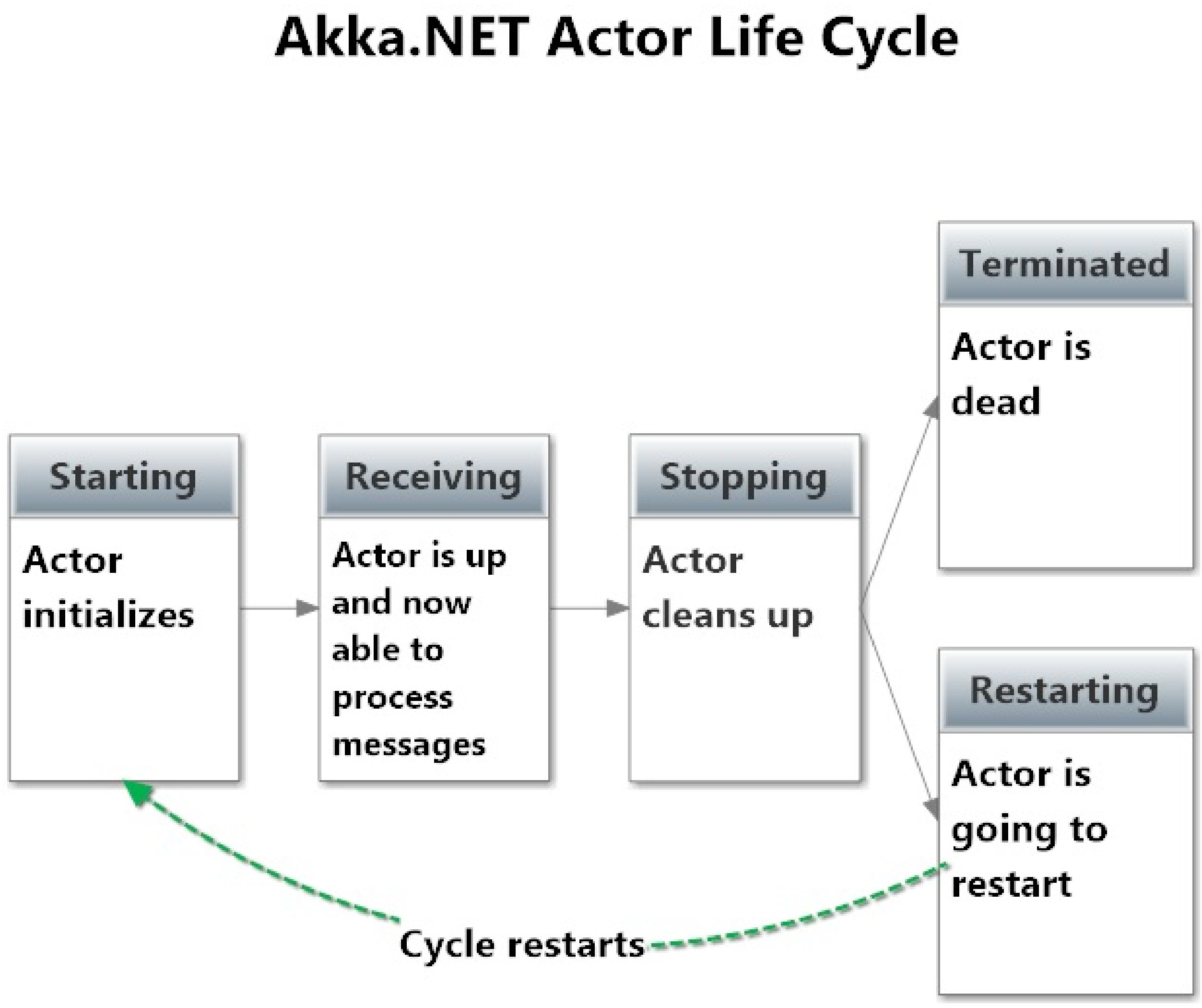
What is the actor life cycle?

Actors have a well-defined life cycle. Actors are created and started, and then they spend most of their lives receiving messages. In the event that you no longer need an actor, you can terminate or "stop" an actor.

What are the stages of the actor life cycle?

There are 5 stages of the actor life cycle in Akka.NET:

1. `Starting`
2. `Receiving`
3. `Stopping`
4. `Terminated`, or
5. `Restarting`



Let's take them in turn.

`Starting`

The actor is waking up! This is the initial state of the actor, when it is being initialized by the `ActorSystem`.

`Receiving`

The actor is now available to process messages. Its `Mailbox` (more on that later) will begin delivering messages into the `OnReceive` method of the actor for processing.

`Stopping`

During this phase, the actor is cleaning up its state. What happens during this phase depends on whether the actor is being terminated, or restarted.

If the actor is being restarted, it's common to save state or messages during this phase to be processed once the actor is back in its Receiving state after the restart.

If the actor is being terminated, all the messages in its `Mailbox` will be sent to the `DeadLetters` mailbox of the `ActorSystem`. `DeadLetters` is a store of undeliverable messages, usually undeliverable because an actor is dead.

`Terminated`

The actor is dead. Any messages sent to its former `IActorRef` will now go to `DeadLetters` instead. The actor cannot be restarted, but a new actor can be created at its former address (which will have a new `IActorRef` but an identical `ActorPath`).

`Restarting`

The actor is about to restart and go back into a `Starting` state.

Life cycle hook methods

So, how can you link into the actor life cycle? Here are the 4 places you can hook in.

`PreStart`

`PreStart` logic gets run before the actor can begin receiving messages and is a good place to put initialization logic. Gets called during restarts too.

`PreRestart`

If your actor accidentally fails (i.e. throws an unhandled Exception) the actor's parent will restart the actor. `PreRestart` is where you can hook in to do cleanup before the actor restarts, or to save the current message for reprocessing later.

`PostStop`

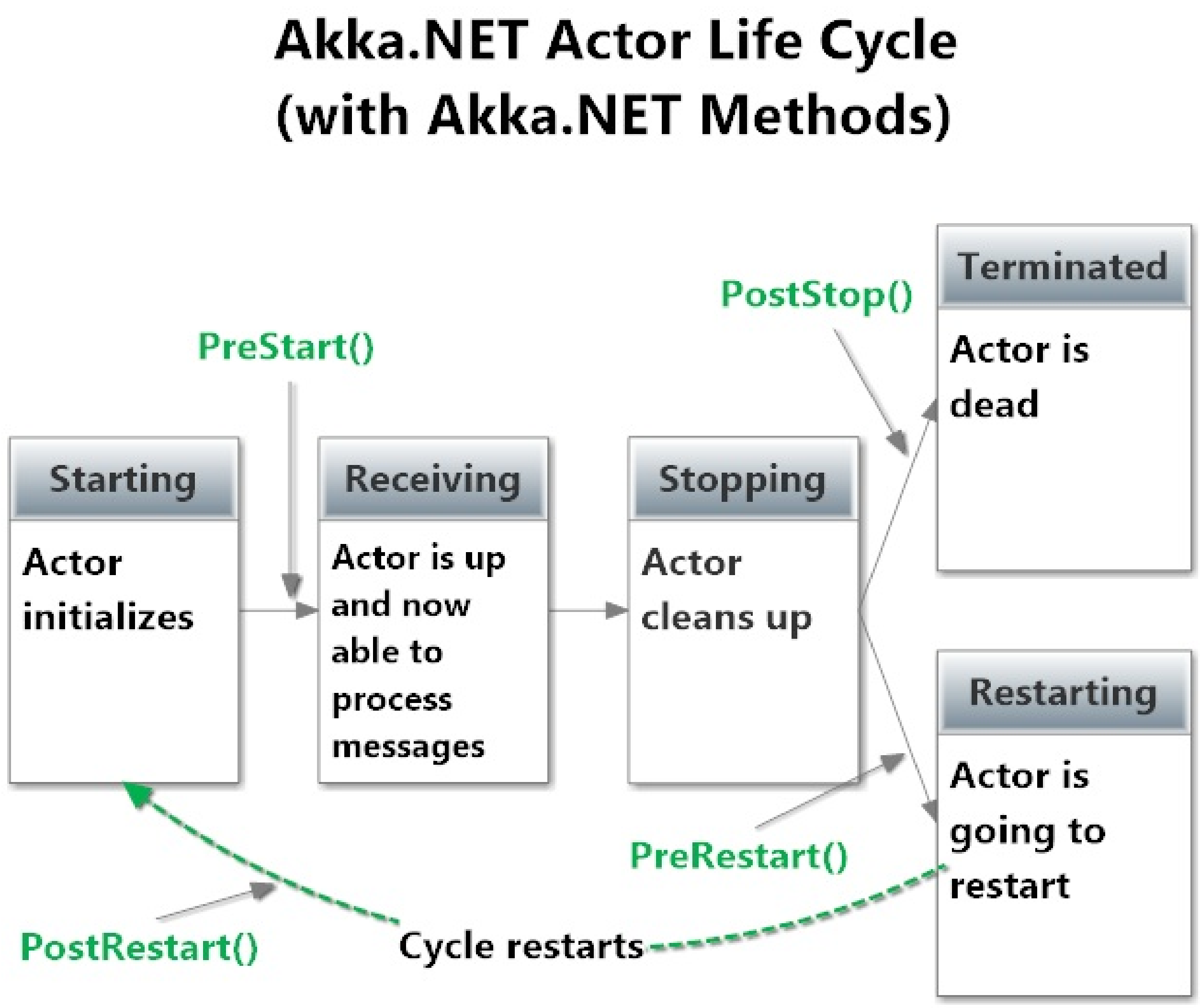
`PostStop` is called once the actor has stopped and is no longer receiving messages. This is a good place to include clean-up logic. PostStop does not get called during actor restarts - only when an actor is being terminated.

`DeathWatch` is also when an actor notifies any other actors that have subscribed to be alerted when it terminates. `DeathWatch` is just a pub/sub system built into framework for any actor to be alerted to the termination of any other actor.

`PostRestart`

`PostRestart` is called during restarts after PreRestart but before PreStart. This is a good place to do any additional reporting or diagnosis on the error that caused the actor to crash, beyond what Akka.NET already does for you.

Here's where the hook methods fit into the stages of the life cycle:



How do I hook into the life cycle?

To hook in, you just override the method you want to hook into, like this:

```
 /// <summary>
/// Initialization logic for actor
/// </summary>
```

```
protected override void PreStart()
{
    // do whatever you need to here
}
```

Which are the most commonly used life cycle methods?

`PreStart`

`PreStart` is far and away the most common hook method used. It is used to set up initial state for the actor and run any custom initialization logic your actor needs.

`PostStop`

The second most common place to hook into the life cycle is in `PostStop`, to do custom cleanup logic. For example, you may want to make sure your actor releases file system handles or any other resources it is consuming from the system before it terminates.

`PreRestart`

`PreRestart` is in a distant third to the above methods, but you will occasionally use it. What you use it for is highly dependent on what the actor does, but one common case is to stash a message or otherwise take steps to get it back for reprocessing once the actor restarts.

How does this relate to supervision?

In the event that an actor accidentally crashes (i.e. throws an unhandled Exception,) the actor's supervisor will automatically restart the actor's lifecycle from scratch - without losing any of the remaining messages still in the actor's mailbox.

As we covered in lesson 4 on the actor hierarchy/supervision, what occurs in the case of an unhandled error is determined by the `SupervisionDirective` of the parent. That parent can instruct the child to terminate, restart, or ignore the error and pick up where it left off. The default is to restart, so that any bad state is blown away and the actor starts clean. Restarts are cheap.

Exercise

This final exercise is very short, as our system is already complete. We're just going to use it to optimize the initialization and shutdown of `TailActor`.

Move initialization logic from `TailActor` constructor to `PreStart()`

See all this in the constructor of `TailActor`?

```
// TailActor.cs constructor
// start watching file for changes
_observer = new FileObserver(Self, Path.GetFullPath(_filePath));
_observer.Start();

// open the file stream with shared read/write permissions (so file can be written to while
_fileStream = new FileStream(Path.GetFullPath(_filePath), FileMode.Open, FileAccess.Read,
    FileShare.ReadWrite);
_fileStreamReader = new StreamReader(_fileStream, Encoding.UTF8);

// read the initial contents of the file and send it to console as first message
var text = _fileStreamReader.ReadToEnd();
Self.Tell(new InitialRead(_filePath, text));
```

While it works, initialization logic really belongs in the `PreStart()` method.

Time to use your first life cycle method!

Pull all of the above init logic out of the `TailActor` constructor and move it into `PreStart()`. We'll also need to change `_observer`, `_fileStream`, and `_fileStreamReader` to non-readonly fields since they're moving out of the constructor.

The top of `TailActor.cs` should now look like this

```
// TailActor.cs
private FileObserver _observer;
private Stream _fileStream;
private StreamReader _fileStreamReader;

public TailActor(IActorRef reporterActor, string filePath)
{
    _reporterActor = reporterActor;
    _filePath = filePath;
}
```

```
// we moved all the initialization logic from the constructor
// down below to PreStart!

/// <summary>
/// Initialization logic for actor that will tail changes to a file.
/// </summary>
protected override void PreStart()
{
    // start watching file for changes
    _observer = new FileObserver(Self, Path.GetFullPath(_filePath));
    _observer.Start();

    // open the file stream with shared read/write permissions (so file can be written to wh
    _fileStream = new FileStream(Path.GetFullPath(_filePath), FileMode.Open, FileAccess.Read
        FileShare.ReadWrite);
    _fileStreamReader = new StreamReader(_fileStream, Encoding.UTF8);

    // read the initial contents of the file and send it to console as first message
    var text = _fileStreamReader.ReadToEnd();
    Self.Tell(new InitialRead(_filePath, text));
}
```

Much better! Okay, what's next?

Let's clean up and take good care of our `FileSystem` resources

`TailActor` instances are each storing OS handles in `_fileStreamReader` and `FileObserver`. Let's use `PostStop()` to make sure those handles are cleaned up and we are releasing all our resources back to the OS.

Add this to `TailActor`:

```
// TailActor.cs
/// <summary>
/// Cleanup OS handles for <see cref="_fileStreamReader"/> and <see cref="FileObserver"/>.
/// </summary>
protected override void PostStop()
{
    _observer.Dispose();
    _observer = null;
    _fileStreamReader.Close();
    _fileStreamReader.Dispose();
    base.PostStop();
}
```

Phase 4: Build and Run!

That's it! Hit `F5` to run the solution and it should work exactly the same as before, albeit a little more optimized. :)

Once you're done

Compare your code to the solution in the Completed folder to see what the instructors included in their samples.

Great job!

WOW! YOU WIN! Phenomenal work finishing Unit 1.

Ready for more? Start Unit 2 now.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.