Getting Started

Adding the NuGet package to your project

You need to pull NetAF into your project. The easiest way to do this is to add the NuGet package. The latest package and installation instructions are available here (https://github.com/benpollarduk/netaf/pkgs/nuget/NetAF).

First Game

Once the package has been installed it's time to jump in and start building your first game.

Setup

To start with create a new Console application. Regardless of target framework, it should look something like this:

```
namespace NetAF.GettingStarted
{
    internal class Program
    {
        private static void Main(string[] args)
        {
        }
    }
}
```

Adding a PlayableCharacter

Every game requires a character to play as, lets add that next:

```
private static PlayableCharacter CreatePlayer()
{
    return new PlayableCharacter("Dave", "A young boy on a quest to find the meaning of life.");
}
```

In this example whenever **CreatePlaye**r is called a new **PlayableCharacter** will be created. The character is called "Dave" and has a description that describes him as "A young boy on a quest to find the meaning of life.".

Creating the game world

The game world consists of a hierarchy of three tiers: **Overworld**, **Region** and **Room**. We will create a simple **Region** with two **Rooms**. We can do this directly in the **Main** function for simplicity. To start with lets make the **Rooms**:

```
private static void Main(string[] args)
{
    var cavern = new Room("Cavern", "A dark cavern set in to the base of the mountai
n.", new Exit(Direction.North));

    var tunnel = new Room("Tunnel", "A dark tunnel leading inside the mountain.", ne
w Exit(Direction.South));
}
```


Although the **Rooms** haven't been added to a **Region** yet there are exits in place that will allow the player to move between them.

Games are boring without **Items** to interact with, let's add an item to the tunnel:

```
var holyGrail = new Item("Holy Grail", "A dull golden cup, looks pretty old.", tru
e);
tunnel.AddItem(holyGrail);
```

Looking good, but the **Rooms** need to be contained within a **Region**. **RegionMaker** simplifies this process, but sometimes creating a **Region** directly may be more appropriate if more control is needed. Here we will use **RegionMaker**:

```
var regionMaker = new RegionMaker("Mountain", "An imposing volcano just East of tow
n.")
{
    [0, 0, 0] = cavern,
    [0, 1, 0] = tunnel
};
```

This needs more breaking down. The **RegionMaker** will create a region called "Mountain" with a description of "An imposing volcano just East of town.". The region will contain two rooms, the cavern and the tunnel. The cavern will be added at position *x* 0, *y* 0, *z* 0. The tunnel will be added at position *x* 0, *y* 1, *z* 0, north of the cavern.

The game world is nearly complete, but the **Region** needs to exist within an **Overworld** for it to be finished. We will use **OverworldMaker** to achieve this:

```
var overworldMaker = new OverworldMaker("Daves World", "An ancient kingdom.", region
Maker);
```

This will create an **Overworld** called "Daves World" which is described as "An ancient kingdom" and contains a single **Region**.

All together the code looks like this:

```
var cavern = new Room("Cavern", "A dark cavern set in to the base of the mountain.",
new Exit(Direction.North));

var tunnel = new Room("Tunnel", "A dark tunnel leading inside the mountain.", new Ex
it(Direction.South));

var holyGrail = new Item("Holy Grail", "A dull golden cup, looks pretty old.", tru
e);
tunnel.AddItem(holyGrail);

var regionMaker = new RegionMaker("Mountain", "An imposing volcano just East of tow
n.")
{
    [0, 0, 0] = cavern,
    [0, 1, 0] = tunnel
};

var overworldMaker = new OverworldMaker("Daves World", "An ancient kingdom.", region
Maker);
```


Checking if the game is complete

For a game to come to an end it needs to reach either a game over state or a completion state.

Firstly lets look at the logic that determines if the game is complete. An **EndCheck** is required, which returns an **EndCheckResult** that determines if the game is complete.

In this example lets make a method that determines if the game is complete. The game is complete if the player has the holy grail:

```
private static EndCheckResult IsGameComplete(Game game)
{
    if (!game.Player.FindItem("Holy Grail", out _))
        return EndCheckResult.NotEnded;

    return new EndCheckResult(true, "Game Complete", "You have the Holy Grail!");
}
```

If the player has the holy grail then the **EndCheckResult** will return that the game has ended, and have a title that will read "Game Complete" and a description that reads "You have the Holy Grail!".

A common game over state may be if the player dies:

```
private static EndCheckResult IsGameOver(Game game)
{
    if (game.Player.IsAlive)
        return EndCheckResult.NotEnded;

    return new EndCheckResult(true, "Game Over", "You died!");
}
```


Creating the game

The game now has all the required assets and logic it just needs some boilerplate to tie everything together before it is ready to play.

A **GameCreationCallback** is required to instantiate an instance of a **Game**. This is so that new instances of the **Game** can be created as required.

```
var gameCreator = Game.Create(
    "The Life Of Dave",
    "Dave finds himself in a cavern...",
    "A very low budget adventure.",
    OverworldMaker.Make,
    PlayerMaker.Make,
    IsGameComplete,
    IsGameOver);
```

This requires some breaking down. The **Game** class has a **Create** method that can be used to create instances of **Game**. This takes the following arguments:

- **Name** - the name of the game.
- **Introduction** - an introduction to the game.
- **Description** - a description of the game.
- **OverworldGenerator** - a callback for generating instances of the overworld.
- **PlayerGenerator** - a callback for generating instances of the player.
- **CompletionCondition** - a callback for determining if the game is complete.
- **GameOverCondition** - a callback for determining if the game is over.

Executing the game

The game is executed simply by calling the static **Execute** method on **Game** and passing in the game creation callback.

```
Game.Execute(gameCreator);
```

Bringing it all together

The full example code should look like this:

```
using NetAF.Assets;
using NetAF.Assets.Characters;
using NetAF.Assets.Locations;
using NetAF.Logic;
using NetAF.Utilities;

namespace NetAF.GettingStarted
{
    internal class Program
    {
        private static EndCheckResult IsGameComplete(Game game)
        {
            if (!game.Player.FindItem("Holy Grail", out _))
                return EndCheckResult.NotEnded;

            return new EndCheckResult(true, "Game Complete", "You have the Holy Grai
l!");
        }

        private static EndCheckResult IsGameOver(Game game)
        {
            if (game.Player.IsAlive)
                return EndCheckResult.NotEnded;

            return new EndCheckResult(true, "Game Over", "You died!");
        }

        private static PlayableCharacter CreatePlayer()
        {
            return new PlayableCharacter("Dave", "A young boy on a quest to find the
 meaning of life.");
        }

        private static void Main(string[] args)
        {
            var cavern = new Room("Cavern", "A dark cavern set in to the base of the
 mountain.", new Exit(Direction.North));

            var tunnel = new Room("Tunnel", "A dark tunnel leading inside the mounta
in.", new Exit(Direction.South));

            var holyGrail = new Item("Holy Grail", "A dull golden cup, looks pretty
 old.", true);

            tunnel.AddItem(holyGrail);

            var regionMaker = new RegionMaker("Mountain", "An imposing volcano just
 East of town.")
            {
                [0, 0, 0] = cavern,
                [0, 1, 0] = tunnel
            };
```

```
            var overworldMaker = new OverworldMaker("Daves World", "An ancient kingd
om.", regionMaker);

            var gameCreator = Game.Create(
                "The Life Of Dave",
                "Dave awakes to find himself in a cavern...",
                "A very low budget adventure.",
                overworldMaker.Make,
                CreatePlayer,
                IsGameComplete,
                IsGameOver);

            Game.Execute(gameCreator);
        }
    }
}
```

Simply build and run the application and congratulations, you have a working NetAF game!

Namespace NetAF.Assets

Classes

Filter by title

ConditionalDescription (NetAF.Assets.ConditionalDescription.html)

Represents a conditional description of an object.

Description (NetAF.Assets.Description.html)

Represents a description of an object.

ExaminableObject (NetAF.Assets.ExaminableObject.html)

Represents an object that can be examined.

ExaminationResult (NetAF.Assets.ExaminationResult.html)

Represents the result of an examination.

Identifier (NetAF.Assets.Identifier.html)

Provides a class that can be used as an identifier.

Item (NetAF.Assets.Item.html)

Represents an item that can be used within the game.

Structs

Size (NetAF.Assets.Size.html)

Interfaces

IExaminable (NetAF.Assets.IExaminable.html)

Represents any object that is examinable.

IPlayerVisible (NetAF.Assets.IPlayerVisible.html)

Represents any object that is visible to a player.

- NetAF.Assets (NetAF.Assets.html)
  - ConditionalDescription (NetAF.Assets.ConditionalDescriptio
  - Description (NetAF.Assets.Description.html)
  - ExaminableObject (NetAF.Assets.ExaminableObject.h
  - ExaminationCallback (NetAF.Assets.ExaminationCallbac
  - ExaminationResult (NetAF.Assets.ExaminationResult.h
  - IExaminable (NetAF.Assets.IExaminable.html)
  - IPlayerVisible (NetAF.Assets.IPlayerVisible.html)
  - Identifier (NetAF.Assets.Identifier.html)
  - Item (NetAF.Assets.Item.html)
  - Size (NetAF.Assets.Size.html)
- + NetAF.Assets.Attributes (NetAF.Assets.Attributes.html)
- + NetAF.Assets.Characters (NetAF.Assets.Characters.htm
- + NetAF.Assets.Interaction (NetAF.Assets.Interaction.html
- + NetAF.Assets.Locations (NetAF.Assets.Locations.html)
- + NetAF.Commands (NetAF.Commands.html)
- + NetAF.Conversations (NetAF.Conversations.html)
- + NetAF.Conversations.Instructions (NetAF.Conversations.Instruct

Delegates

ExaminationCallback (NetAF.Assets.ExaminationCallback.html)

Represents the callback for examinations.

- **NetAF.Assets (NetAF.Assets.html)**
  - ConditionalDescription (NetAF.Assets.ConditionalDescripti
  - Description (NetAF.Assets.Description.html)
  - ExaminableObject (NetAF.Assets.ExaminableObject.h
  - ExaminationCallback (NetAF.Assets.ExaminationCallbac
  - ExaminationResult (NetAF.Assets.ExaminationResult.h
  - IExaminable (NetAF.Assets.IExaminable.html)
  - IPlayerVisible (NetAF.Assets.IPlayerVisible.html)
  - Identifier (NetAF.Assets.Identifier.html)
  - Item (NetAF.Assets.Item.html)
  - Size (NetAF.Assets.Size.html)
- **+ NetAF.Assets.Attributes (NetAF.Assets.Attributes.html)**
- **+ NetAF.Assets.Characters (NetAF.Assets.Characters.htm**
- **+ NetAF.Assets.Interaction (NetAF.Assets.Interaction.html**
- **+ NetAF.Assets.Locations (NetAF.Assets.Locations.html)**
- **+ NetAF.Commands (NetAF.Commands.html)**
- **+ NetAF.Conversations (NetAF.Conversations.html)**
- **+ NetAF.Conversations.Instructions (NetAF.Conversations.Instruct**