

# Getting Started

## Adding the NuGet package to your project

▼ Filter by title

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>).

### Getting Started (getting-started.html)

### First Game

+ Locations

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

### Items (items.html)

### Characters

### Setup

### Conditional Descriptions (conditional-descriptions.html)

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

```
namespace NetAF.GettingStarted
{
    internal class Program
    {
        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 **CreatePlayer** 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 mountain.", new Exit(Direction.North));
```

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

## Getting Started (getting-started.html)

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:

## + Characters

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

## Conditional Descriptions (conditional-descriptions.html)

```
holyGrail.AddDescription(holyGrail);
```

## Attributes (attributes.html)

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**.

## Frame Builders (frame-builders.html)

```
var regionMaker = new RegionMaker("Mountain", "An imposing volcano just East of town.")
```

## End Conditions (end-conditions.html)

```
if (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.", regionMaker);
```

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 Exit(Direction.South));
```



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

**Getting Started (getting-started.html)**

```
tunnel.AddItem(holyGrail);
```

## + Locations

```
var regionMaker = new RegionMaker("Mountain", "An imposing volcano just East of town");
```

**Items (items.html)**

```
{
```

## + Characters

```
{0, 0} = cavern,
```

```
[0, 1, 0] = tunnel
```

**Conditional Descriptions (conditional-**

**descriptions.html)**

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

**Attributes (attributes.html)**

**Commands**

## Checking if the game is complete (commands.html)

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

Firstly let's 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.

**End Conditions (end-**

**conditions.html)** In this example let's 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!");
}
```

## Getting Started (getting-started.html)

## Creating the game

### + Locations

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.

### Items (items.html)

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.

### Conditional Descriptions

#### (conditional-

### descriptions.html)

### Attributes (attributes.html)

### Commands

### (commands.html)

### Frame Builders (frame-

### builders.html)

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:

### End Conditions (end-

### conditions.html)

- **GameInfo** - information about the game.
- **Introduction** - an introduction to the game.
- **AssetGenerator** - a generator for game assets.
- **GameEndConditions** - conditions for determining if the game has been completed or otherwise ended.
- **GameConfiguration** - a configuration for the game, including display size, error prefix and other elements.

## 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
{
Getting Started (getting-started.html)
    internal class Program
    {
        private static EndCheckResult IsGameComplete(Game game)
+ Locations
        {
            if (!game.Player.FindItem("Holy Grail", out _))
Items (items.html)
                return EndCheckResult.NotEnded;

+ Characters
            return new EndCheckResult(true, "Game Complete", "You have the Holy Grai
Conditional Descriptions
(conditional-descriptions.html)
            private static EndCheckResult IsGameOver(Game game)
Attributes (attributes.html)
            {
                if (game.Player.IsAlive)
Commands
                return EndCheckResult.NotEnded;
(commands.html)
                return new EndCheckResult(true, "Game Over", "You died!");

Frame Builders (frame-builders.html)
            private static PlayableCharacter CreatePlayer()
End Conditions (end-conditions.html)
            {
                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 kingdom.", regionMaker);
```

```
var gameCreator = Game.Create(  
    new GameInfo("The Life of Dave", "A very low budget adventure.", "Be  
    n Pollard"),  
    "Dave awakes to find himself in a cavern...",  
    AssetGenerator.Custom(overworldMaker.Make, CreatePlayer),  
    new GameEndConditions(IsGameComplete, IsGameOver),  
    GameConfiguration.Default);
```

**Getting Started (getting-started.html)**

+ **Locations**      Game.Execute(gameCreator);

**Items (items.html)**

+ **Characters**

**Conditional Descriptions**

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

**(conditional-descriptions.html)**

**Attributes (attributes.html)**

**Commands**  
**(commands.html)**

**Frame Builders (frame-builders.html)**

**End Conditions (end-conditions.html)**

# Namespace NetAF.Adapters

## Classes

▼ Filter by title

### SystemConsoleAdapter (NetAF.Adapters.SystemConsoleAdapter.html)

Provides an adapter for the System.Console.

### IIOAdapter (NetAF.Adapters.IIOAdapter.html)

### Interfaces

### IIOAdapter (NetAF.Adapters.IIOAdapter.html)

+ NetAF.Assets  
Represents any object that provides an adapter for input.  
(NetAF.Assets.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

+ NetAF.Extensions  
(NetAF.Extensions.html)

+ NetAF.Interpretation  
(NetAF.Interpretation.html)

+ NetAF.Logic  
(NetAF.Logic.html)