

# 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)

## Setup

+ Characters

Conditional Descriptions

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

(conditional-

descriptions.html)

```
namespace NetAF.GettingStarted
```

Attributes (attributes.html)

```
internal class Program
```

Commands

(commands.html) static void Main(string[] args)

```
{
```

Frame Builders (frame-

builders.html)

```
}
```

End Conditions (end-

conditions.html)

Persistence

## Adding a PlayableCharacter

(persistence.html)

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)

## 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
```

## Persistence (persistence.html)

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)**

```
{
    [0, 0] = cavern,
    [0, 1, 0] = tunnel
}
```

## + Characters

**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:

**Persistence**

**(persistence.html)**

```
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-

- **GameInfo** - information about the game.

- **Introduction** - an introduction to the game.

- **AssetGenerator** - a generator for game assets.

### (persistence.html)

- **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.Logic.Coniguration;
using NetAF.Utilities;

namespace NetAF.GettingStarted
{
    Getting Started (getting-started.html)
    internal class Program
    {
        + Locations
        private static EndCheckResult IsGameComplete(Game game)
        {
            Items (items.html)
            if (!game.Player.FindItem("Holy Grail", out _))
            {
                + Characters
                return EndCheckResult.NotEnded;
            }
            Conditional Descriptions (conditional-descriptions.html)
            return EndCheckResult(true, "Game Complete", "You have the Holy Grai
            l.");
        }
        Attributes (attributes.html)
        private static EndCheckResult IsGameOver(Game game)
        {
            Commands (commands.html)
            if (game.Player.IsAlive)
            {
                return EndCheckResult.NotEnded;
            }
            Frame Builders (frame-builders.html)
            return EndCheckResult(true, "Game Over", "You died!");
        }
        End Conditions (end-conditions.html)
        private static PlayableCharacter CreatePlayer()
        {
            return new PlayableCharacter("Dave", "A young boy on a quest to find the
            Holy Grail.");
        }
        Persistence (persistence.html)
        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.", "Ben Pollard"),  
    "Dave awakes to find himself in a cavern...",  
    AssetGenerator.Custom(overworldMaker.Make, CreatePlayer),  
    new GameEndConditions(IsGameComplete, IsGameOver),  
    ConsoleGameConfiguration.Default);
```

## Getting Started (getting-started.html)

### + Locations

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

## Items (items.html)

### + Characters

## Conditional Descriptions

(conditional-descriptions.html)  
Simply build and run the application and congratulations, you have a working NetAF game!

## Attributes (attributes.html)

## Commands (commands.html)

## Frame Builders (frame-builders.html)

## End Conditions (end-conditions.html)

## Persistence (persistence.html)

# Overworld

## Overview

Filter by title

An Overworld is the top level location in a game. A game can only contain a single Overworld. An Overworld can contain multiple Regions.

## Getting Started (getting-started.html)

## Overworld

- **Locations**

## Locations

Overworld Room  
Overworld (overworld.html)

|Region (region.html)

```
Room (room.html)
```

Exit (exit.html)

Room

## Items (items.html)

## + Characters

## Conditional Descriptions

(conditional-

descriptions.html)

## Attributes (attributes.html)

```
var overworld = new Overworld("Name", "Description.");
```

## Commands

**(commands.html)**

Regions can be added to the Overworld with the **AddRegion** method.

## Frame Builders (frame-builders.html)

```
overworld.AddRegion(region);
```

## End Conditions (end-conditions.html)

Regions can be removed from an Overworld with the **RemoveRegion** method.

## Persistence

```
(persistentEntity)Region(region);
```

The Overworld can be traversed with the **Move** method.

```
overworld.Move(region);
```

# OverworldMaker

The OverworldMaker simplifies the creation of the Overworld, when used in conjunction with RegionMakers.

```
var overworldMaker = new OverworldMaker("Name", "Description.", regionMakers);
```

However, the main benefit of using an OverworldMaker is that it allows multiple instances of an Overworld to be created from a single definition of an Overworld.



```
var overworld = overworldMaker.Make();
```

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

### **- Locations**

Overworld (overworld.html)

Region (region.html)

Room (room.html)

Exit (exit.html)

## **Items (items.html)**

### **+ Characters**

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

## **Attributes (attributes.html)**

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

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

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

## **Persistence (persistence.html)**



# Region

## Overview

A Region is the intermediate level location in a game. An Overworld can contain multiple Regions. A Region can contain multiple Rooms.

## Getting Started (getting-started.html)

# Overworld

## - Locations

## Localities

Overworld (overworld.html)

|Region (region.html)

Room (room.html)

Exit (exit.html)

Room

## Items (items.html)

## + Characters

A Region represents a 3D space.

## Conditional Descriptions

The x location always refers to the horizontal axis, with lower values being west and higher values being east.

cast.  
descriptions.html)

- The **y** location always refers to the vertical axis, with lower values being south and higher values being

## Attributes (attributes.html)

- The **z** location always refers to the depth axis, with lower values being down and higher values being up.

# Commands

**(commands.html)**

## Use

## Frame Builders (frame-

A Region can be simply instantiated with a name and description.

## End Conditions (end-

```
var region = new Region("Name", "Description.");
```

## Persistence

Rooms can be added to the Region with the **AddRoom** method. The x, y and z location within the Region must be specified. ([persistence.html](#))

```
region.AddRoom(room, 0, 0, 0);
```

Rooms can be removed from a Region with the **RemoveRoom** method.

```
region.RemoveRoom(room);
```

The Region can be traversed with the **Move** method.

```
region.Move(Direction.North);
```

The start position, that is the position that the Player will start in when entering a Region, can be specified with **SetStartPosition**.



```
region.SetStartPosition(0, 0, 0);
```

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

The **UnlockDoorPair** method can be used to unlock an **Exit** in the current Room, which will also unlock the corresponding Exit in the adjoining Room.

Overworld (overworld.html)

Region (region.html)

```
region.UnlockDoorPair(Direction.East);
```

Room (room.html)

Exit (exit.html)

Like all Examinable objects, Regions can be assigned custom commands.

### Items (items.html)

### + Characters

```
Region region = null;
```

```
region = new("Woodlands", "A quiet woodland.", commands:
```

### Conditional Descriptions

(conditional-

descriptions.html)

```
{
```

### Attributes (attributes.html)

```
region.JumpToRoom(0, 0, 0);
```

```
return new Reaction(ReactionResult.OK, "You warped to the start.");
```

### Commands

(commands.html)

### Frame Builders (frame-

builders.html)

## RegionMaker

End Conditions (end-

conditions.html)

The RegionMaker simplifies the creation of a Region. Rooms are added to the Region with a specified **x**, **y** and **z** position within the Region.

### Persistence

(persistence.html)

```
var regionMaker = new RegionMaker("Region", "Description.")
```

```
{
```

```
[0, 0, 0] = new Room("Room 1", "Description of room 1."),
```

```
[1, 0, 0] = new Room("Room 2", "Description of room 2."),
```

```
};
```

The main benefit of using a RegionMaker is that it allows multiple instances of a Region to be created from a single definition of a Region.

```
var region = regionMaker.Make();
```



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

### **- Locations**

Overworld (overworld.html)

Region (region.html)

Room (room.html)

Exit (exit.html)

## **Items (items.html)**

### **+ Characters**

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

## **Attributes (attributes.html)**

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

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

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

## **Persistence (persistence.html)**

# Room

## Overview

Filter by title

A Room is the lowest level location in a game. A Region can contain multiple Rooms.

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



### Items (items.html)

A Room can contain up to six Exits, one for each of the directions **north**, **east**, **south**, **west**, **up** and **down**.

### Conditional Descriptions

## Use

### Conditional-descriptions.html)

A Region can be simply instantiated with a name and description.

### Attributes (attributes.html)

```
var room = new Room("Name", "Description.");
```

### Commands

### (commands.html)

Items can be added to the Room with the **AddItem** method.

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

```
room.AddItem(new Item("Name", "Description."));
```

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

Items can be removed from a Room with the **RemoveItem** method.

### Persistence

### (persistence.html)

```
region.RemoveItem(item);
```

Characters can be added to the Room with the **AddCharacter** method.

```
room.AddCharacter(new NonPlayableCharacter("Name", "Description."));
```

Characters can be removed from a Room with the **RemoveCharacter** method.

```
region.RemoveCharacter(character);
```

Rooms can contain custom commands that allow the user to directly interact with the Room.

```

Room room = null;
room = new("Dungeon", "A grim dungeon.", commands:
[
    new CustomCommand(new CommandHelp("Pull lever", "Pull the lever."), true, (game,
args) =>
    {
        room.FindExit(Direction.East, true, out var exit);
        exit.Unlock();
        return new Reaction(ReactionResult.OK, "The exit was unlocked.");
    }
]);

```

## Getting Started (getting-started.html)

### - Locations

Overworld (overworld.html)  
 Region (region.html)  
 Room (room.html)  
 Exit (exit.html)

## Items (items.html)

### + Characters

## Conditional Descriptions (conditional-descriptions.html)

## Attributes (attributes.html)

## Commands (commands.html)

## Frame Builders (frame-builders.html)

## End Conditions (end-conditions.html)

## Persistence (persistence.html)

# Exit

## Overview

Filter by title

An Exit is essentially a connector between to adjoining rooms.

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

#### Locations

An Exit can be simply instantiated with a direction.

Overworld (overworld.html)

Region (region.html)

```
var exit = new Exit(Direction.North);
```

Room (room.html)

Exit (exit.html)

An Exit can be hidden from the player by setting its **IsPlayerVisible** property to false, this can be set in the constructor.

#### Items (items.html)

#### + Characters

#### Conditional Descriptions

(conditional-

descriptions.html)

Ordering

#### Attributes (attributes.html)

```
exit.IsPlayerVisible = false;
```

#### Commands

(commands.html)

Optionally, a description of the Exit can be specified.

#### Frame Builders (frame-

builders.html)

```
var exit = new Exit(Direction.North, true, new Description("A door covered in iv
```

#### End Conditions (end-

conditions.html)

This will be returned if the player examines the Exit.

#### Persistence

Like all other objects, an Exit can be assigned custom commands.

(persistence.html)

```
Exit exit = null;
exit = new(Direction.North, commands:
[
    new CustomCommand(new CommandHelp("Shove", "Shove the door."), true, true, (game, args) =>
    {
        exit.Unlock();
        return new Reaction(ReactionResult.OK, "The door swung open.");
    })
]);
```



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

### **- Locations**

Overworld (overworld.html)

Region (region.html)

Room (room.html)

Exit (exit.html)

## **Items (items.html)**

### **+ Characters**

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

## **Attributes (attributes.html)**

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

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

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

## **Persistence (persistence.html)**

# Item

## Overview

▼ Filter by title

Items can be used to add interactivity with a game. Items can be something that a player can take with them, or they may be static in a Room.

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

## Use

### + Locations

An Item can be simply instantiated with a name and description.

### Items (items.html)

### + Characters

```
var sword = new Item("Sword", "A heroes sword.");
```

### Conditional Descriptions

By default, an item is not takeable and is tied to a Room. If it is takeable this can be specified in the constructor.

### (conditional-descriptions.html)

### Attributes (attributes.html)

```
var sword = new Item("Sword", "A heroes sword.", true);
```

### Commands

Like all other objects, an Item can be assigned custom commands.

### (commands.html)

### Frame Builders (frame-

```
Item bomb = new("Bomb", "A bomb", commands:
```

```
[
```

```
    new CustomCommand(new CommandHelp("Cut wire", "Cut the red wire."), true, (game,
```

```
    args) =>
```

```
    {
```

```
        game.Player.Kill();
```

```
        return new Reaction(ReactionResult.Fatal, "Boom!");
```

```
    })
```

```
]);
```

### Persistence

### (persistence.html)

## Interaction

Interactions can be set up between different assets in the game. The **InteractionResult** contains the result of the interaction, and allows the game to react to the interaction.



```
var dartsBoard = new Item("Darts board", "A darts board.");
```

```
var dart = new Item("Dart", "A dart", interaction: item =>  
{
```

```
    if (item == dartsBoard)
```

```
    return new InteractionResult(InteractionEffect.SelfContained, item, "The dart  
    stuck in the darts board.");
```

```
    return new InteractionResult(InteractionEffect.NoEffect, item);  
},
```

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

## + Locations

**Items (items.html)**

## + Characters

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

**Attributes (attributes.html)**

**Commands  
(commands.html)**

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

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

**Persistence  
(persistence.html)**

# PlayableCharacter

## Overview

▼ Filter by title

A PlayableCharacter represents the character that the player plays as throughout the game. Each game has only a single PlayableCharacter.

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

## Use

### + Locations

A PlayableCharacter can be simply instantiated with a name and description.

### Items (items.html)

### - Characters

```
var player = new PlayableCharacter("Ben", "A 39 year old man.");
PlayableCharacter (playable-character.html)
```

A PlayableCharacter can be also be instantiated with a list of Items.

```
NonPlayableCharacter (non-playable-character.html)
```

### Conditional Descriptions

### (conditional-

### descriptions.html)

```
new Item("Guitar", "A PRS Custom 22, in whale blue, of course."),
new Item("Mallet", "An empty wallet, of course.")
});
```

### Attributes (attributes.html)

### Commands

A PlayableCharacter can be given items with the **AddItem** method.

### (commands.html)

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

```
player.AddItem(new Item("Mallet", "A large mallet."));
```

A PlayableCharacter can use an item with the **RemoveItem** method.

### conditions.html)

```
player.RemoveItem(mallet);
```

### Persistence

### (persistence.html)

A PlayableCharacter can use an item on another asset:

```
var trapDoor = new Exit(Direction.Down);
var mallet = new Item("Mallet", "A large mallet.");
player.UseItem(mallet, trapDoor);
```

A PlayableCharacter can give an item to a non-playable character.

```

var goblin = new NonPlayableCharacter("Goblin", "A vile goblin.");
var daisy = new Item("Daisy", "A beautiful daisy that is sure to cheer up even the most miserable creature.");
player.Give(daisy, goblin);

```



PlayableCharacters can contain custom commands that allow the user to directly interact with the character or other assets.

## Getting Started (getting-started.html)

```

PlayableCharacter player = new("Daisy", "A beautiful daisy that is sure to cheer up
most miserable creature.", commands:

```

### + Locations

```

[
new CustomCommand(new CommandHelp("Punch wall", "Punch the wall."), true, (game,
args) =>

```

### Items (items.html)

### - Characters

```

{
    PlayableCharacter playableCharacter = new("Daisy", "A beautiful daisy that is sure to cheer up
    character.html)
    NonPlayableCharacter(non-
    playable-character.html)
}

```

## Conditional Descriptions (conditional-descriptions.html)

## Attributes (attributes.html)

## Commands (commands.html)

## Frame Builders (frame-builders.html)

## End Conditions (end-conditions.html)

## Persistence (persistence.html)

# NonPlayableCharacter

## Overview

Filter by title

A NonPlayableCharacter represents any character that the player may meet throughout the game.

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

#### Locations

A NonPlayableCharacter can be simply instantiated with a name and description.

#### Items (items.html)

```
var goblin = new NonPlayableCharacter("Goblin", "A vile goblin.");
```

#### Characters

A NonPlayableCharacter can give an item to another NonPlayableCharacter.

```
NonPlayableCharacter (non-playable-character.html)
```

```
var daisy = new Item("Daisy", "A beautiful daisy that is sure to cheer up even the most miserable creature.");
npc.Give(daisy, goblin);
```

#### Conditional Descriptions

#### (conditional-descriptions.html)

NonPlayableCharacters can contain custom commands that allow the user to directly interact with the character or other assets.

#### Commands

#### (commands.html)

```
NonPlayableCharacter goblin = new("Goblin", "A vile goblin", commands:
[
    new CustomCommand(new CommandHelp("Smile", "Crack a smile."), true, (game, args)
    {
        return new Reaction(ReactionResult.OK, "Well that felt weird.");
    })
]);
```

#### Persistence

#### (persistence.html)

## Conversations

A NonPlayableCharacter can hold a conversation with the player.

- A Conversation contains **Paragraphs**.
- A Paragraph can contain one or more **Responses**.
- A **Response** can contain a delta or other implementation of **IEndOfParagraphInstruction** to shift the conversation by, which will cause the conversation to jump paragraphs by the specified value.
- A **Response** can also contain a callback to perform some action when the player selects that option.

```

NonPlayableCharacter goblin = new("Goblin", "A vile goblin", conversation: new(
    new Paragraph("This is a question.")
    {
        Responses =
            [
                new Response("This is the first response.", new Jump(1)),
                new Response("This is the second response.", new Jump(2)),
                new Response("This is the third response.", new Jump(3))
            ]
    }
),

```

## Getting Started (getting-started.html)

```

    new Paragraph("You picked first response, return to start of conversation.", new
    goto(1)),
    new Paragraph("You picked second response, return to start of conversation.", ne
    w goto(1)),
    new Paragraph("You picked third response, you are dead.", game => game.Player.Ki
    ll()),
    PlayableCharacter (playable-
    character.html)
    NonPlayableCharacter (non-
    playable-character.html)

```

## Conditional Descriptions (conditional-descriptions.html)

## Attributes (attributes.html)

## Commands (commands.html)

## Frame Builders (frame-builders.html)

## End Conditions (end-conditions.html)

## Persistence (persistence.html)

# Conditional Descriptions

## Overview

Filter by title

Normally assets are assigned a **Description** during the constructor. This is what is returned when the asset is examined.

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

Descriptions are usually specified as a string.

#### + Locations

```
var item = new Item("The items name", "The items description.");
```

#### Items (items.html)

#### + Characters

They can also be specified as a **Description**.

### Conditional Descriptions

```
(conditional = new Item(new Identifier("The items name"), new Description("The items description"));
```

#### descriptions.html)

### Attributes (attributes.html)

However, sometimes it may be desirable to have a conditional description that can change based on the state of the asset.

### Commands

#### (commands.html)

Conditional descriptions can be specified with **ConditionalDescription** and contain a lambda which determines which one of two strings are returned when the asset is examined.

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

```
// the player, just for demo purposes
```

```
var player = new PlayableCharacter("Ben", "A man.");
```

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

```
// the description to use when the condition is true
```

```
var trueString = "A gleaming sword, owned by Ben.";
```

#### Persistence (persistence.html)

```
// the string to use when the condition is false
```

```
var falseString = "A gleaming sword, without an owner.";
```

```
// a lambda that determines which string is returned
```

```
Condition condition = () => player.FindItem("Sword", out _);
```

```
// the conditional description itself
```

```
var conditionalDescription = new ConditionalDescription(trueString, falseString, condition);
```

```
// create the item with the conditional description
```

```
var sword = new Item(new Identifier("Sword"), conditionalDescription);
```



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

**+ Locations**

**Items (items.html)**

**+ Characters**

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

**Attributes (attributes.html)**

**Commands  
(commands.html)**

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

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

**Persistence  
(persistence.html)**

# Attributes

## Overview

Filter by title

All examinable objects can have attributes. Attributes provide a way of adding a lot of depth to games. For example, attributes could be used to buy and sell items, contain a characters XP or HP or even provide a way to add durability to items.

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

## Use Locations

### Use

**Items (items.html)**

To add to an existing attribute or to create a new one use the **Add** method.

**+ Characters**

**Conditional Descriptions**

**(conditional-descriptions.html)**

**Commands**

To subtract from an existing attribute use the **Subtract** method.

**Attributes (attributes.html)**

**Frame Builders**

**(frame-builders.html)**

Attributes values can be capped. In this example the \$ attribute is limited to a range of 0 - 100. Adding or subtracting will cause the value of the attribute to change outside of this range.

**End Conditions**

**(end-conditions.html)**

```
var cappedAttribute = new Attribute("$", "Dollars.", 0, 100);
```

```
player.Attributes.Add(cappedAttribute, 50);
```

**Persistence**

**(persistence.html)**

## An example - buying an Item from a NonPlayableCharacter.

The following is an example of buying an Item from NonPlayableCharacter. Here a trader has a spade. The player can only buy the spade if they have at least \$5. The conversation will jump to the correct paragraph based on if they choose to buy the spade or not. If the player chooses to buy the spade and has enough \$ the transaction is made and the spade changes hands.



```
const string currency = "$";
```

```
var player = new PlayableCharacter("Player", string.Empty);  
player.Attributes.Add(currency, 10);
```

```
var trader = new NonPlayableCharacter("Trader", string.Empty);  
var spade = new Item("Spade", string.Empty);  
trader.AcquireItem(spade);
```

## Getting Started (getting-

### started.html)

```
trader.Conversation = new Conversation(  
    new Paragraph("What will you buy?")
```

## + Locations

```
{  
    Responses =  
    Items (items.html)
```

```
+ Characters    new Response("Spade", new ByCallback(() =>  
                player.Attributes.GetValue(currency) >= 5
```

```
                : new ToName("BoughtSpade")  
                : new ToName("NotEnough"))),
```

```
                new Response("Nothing", new Last())  
            ]
```

## Attributes (attributes.html)

```
    new Paragraph("Here it is.", _ =>
```

## Commands

```
(commands.html)    player.Attributes.Subtract(currency, 5);  
                    trader.Attributes.Add(currency, 5);
```

## Frame Builders (frame-

```
builders.html)    trader.Offer(spade, player);  
                    new First(), "BoughtSpade"),
```

```
                    new Paragraph("You don't have enough money.", new First(), "NotEnough"),  
                    new Paragraph("Thank you very much, fine.")
```

## End Conditions (end-

### conditions.html)

## Persistence

This is just one example of using attributes to add depth to a game.

## (persistence.html)

# Commands

## Overview

▼ Filter by title

There are three main types of Command.

**Getting Started (getting-started.html)** Game Commands are used to interact with the game.

**Conditional Commands** are used to interact with the program running the game.

- **Custom Commands** allow developers to add custom commands to the game without having to worry about extended the games interpreters.

## + Locations

**Items (items.html)**

## + Characters

**Conditional Descriptions**

## Drop

**(conditional-**

**descriptions.html)** Allows players to drop an item. **R** can be used as a shortcut.

**Attributes (attributes.html)**

drop sword

**Commands**

**(commands.html)**

The player can also drop **all** items.

**Frame Builders (frame-**

**builders.html)**

drop all

**End Conditions (end-**

**conditions.html)**

## Examine

**Persistence**

**(persistence.html)**

Allows players to examine any asset. **X** can be used as a shortcut.

Examine will examine the current room.

```
examine
```

The player themselves can be examined with **me** or the players name.

```
examine me
```

or

```
examine ben
```

The same is true for Regions, Overworlds, Items and Exits.

# Take

Allows the player to take an Item. **T** can be used as a shortcut.

```
take sword
```



Take **all** allows the player to take all takeables Items in the current Room.

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

```
take all
```

+ **Locations**

**Items (items.html)**

## Talk

+ **Characters**

Talk allows the player to start a conversation with a NonPlayableCharacter. **L** can be used as a shortcut.

**Conditional Descriptions**

If only a single NonPlayableCharacter is in the current Room no argument needs to be specified.

**(conditional-**

**descriptions.html)**

```
talk
```

**Attributes (attributes.html)**

**Commands**

However, if the current Room contains two or more NonPlayableCharacters then **to** and the

NonPlayableCharacter name must be specified.

**(command.html)**

**Frame Builders (frame-**

**builders.html)**

```
talk to dave
```

**End Conditions (end-**

**conditions.html)**

## Use

**Persistence**

Use allows the player to use the Items that the player has or that are in the current Room.

**(persistence.html)**

```
use sword
```

Items can be used on the Player, the Room, an Exit, a NonPlayableCharacter or another Item. The target must be specified with the **on** keyword.

```
use sword on me
```

Or

```
use sword on bush
```

# Move

Regions are traversed with direction commands.

- **North** or **N** moves north.
- **East** or **E** moves east.
- **South** or **S** moves south.
- **West** or **W** moves west.
- **Down** or **D** moves down.
- **Up** or **U** moves up.

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

## End

On the `End` conversation with a `NonPlayableCharacter`, the `End` command will end the conversation.

**Items (items.html)**

## + Characters

end

## Conditional Descriptions

(conditional-

descriptions.html)

## Global Commands

**Attributes (attributes.html)**

## About

**Commands (commands.html)**

Displays a screen containing information about the game.

## Frame Builders (frame-

builders.html)

## End Conditions (end-

conditions.html)

## CommandsOn / CommandsOff

Persistence

To persist the contextual commands on the screen on and off.

**(persistence.html)**

commandson

Or

commandsoff

## Exit

Exit the current game.

exit

# Help

Displays a Help screen listing all available commands.

help



## KeyOn / KeyOff (keyonstarted.html)

Toggles the display of the map key on and off.

### + Locations

keyon  
Items (items.html)

### + Characters

Or

Conditional Descriptions  
(conditional-  
keyoff  
descriptions.html)

Attributes (attributes.html)

## Map

Commands

(commands.html)  
Displays the regular map screen.

Frame Builders (frame-  
map  
builders.html)

End Conditions (end-  
conditions.html)

## New

Persistence  
Starts a new game.  
(persistence.html)

new

# Custom Commands

Custom commands can be added to many of the assets, including Room, PlayableCharacter, NonPlayableCharacter, Item and Exit.

# Overview

In BP.AdventureFramework output is handled using the **FrameBuilders**. A **FrameBuilder** is essentially a class that builds a **Frame** that can render a specific state in the game. This **Frame** can then be rendered on a **TextWriter** by calling its **Render** method. Think of the **FrameBuilder** as the instructions that build the output display and the **Frame** as the output itself.

There are a few types of **FrameBuilder**, each responsible for rendering a specific game state.

- **SceneFrameBuilder** is responsible for building frames that render the scenes in a game.
- + **TitleFrameBuilder** is responsible for building the title screen frame.
- **RegionMapFrameBuilder** is responsible for building a frame that displays a map of a Region.
- + **Items (items.html)**
  - **TransitionFrameBuilder** is responsible for building frames that display transitions.
- + **Characters**
  - **AboutFrameBuilder** is responsible for building a frame to display the about information.
  - **HelpFrameBuilder** is responsible for building frames to display the help.
- + **Conditional Descriptions (conditional-descriptions.html)**
  - **GameOverFrameBuilder** is responsible for building a frame to display the game over screen.
  - **CompletionFrameBuilder** is responsible for building a frame to display the completion screen.
  - **ConversationFrameBuilder** is responsible for building a frame that can render a conversation.

A game accepts a **FrameBuilderCollection**. A **FrameBuilderCollection** is a collection of all the different **FrameBuilders** required to render a game. All **FrameBuilders** are extensible, so the output for all parts of the game can be easily customised.

**Attributes (attributes.html)**

**Commands (commands.html)**

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

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

**Persistence (persistence.html)**

# End Conditions

## Overview

▼ Filter by title

The **EndCheck** class allows the game to determine if it has come to an end. Each game has two end conditions

**GameOverCondition** when the game is over, but has not been won.

**CompletionCondition** when the game is over because it has been won.

+ Locations

## Use

Items (items.html)

When an **EndCheck** is invoked it returns an **EndCheckResult**. The **EndCheckResult** details the result of the check to see if the game has ended.

+ Characters

## Conditional Descriptions

(conditional-  
descriptions.html)

```
private static EndCheckResult IsGameOver(Game game)
```

```
    if (game.Player.IsAlive)
```

Attributes (attributes.html)

```
        return EndCheckResult.NotEnded;
```

## Commands

(commands.html)

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

## Frame Builders (frame-

builders.html)

The **GameOverCondition** is used as an **EndCheck**:

## End Conditions (end-

conditions.html)

```
    EndCheck gameOverCheck = IsGameOver;
```

## Persistence

(persistence.html)

The **GameOverCondition** and **CompletionCondition** are passed in to the game as arguments when a game is created.

# Persistence

## Overview

Filter by title

NetAF provides persistence for game states. This allows a game to be saved and restored at any time.

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

#### + Locations

### Creating a restore point

Items (items.html)

A named restore point can be created at any time by using the RestorePoint class:

#### + Characters

```
var restorePoint = RestorePoint.Create("File 1", game);
```

### Conditional Descriptions

(conditional-

RestorePoint will serialize the state of the game so that it can be restored later. However it still needs to be saved to file if persistence is required.

descriptions.html)

### Attributes (attributes.html)

### Saving a restore point to file

Commands

(commands.html)

NetAF uses JsonAF for persistence as default, provided by the JsonSave class:

### Frame Builders (frame-

```
JsonSave.ToFile(@"C:\save.json", restorePoint, out _);
```

builders.html)

### End Conditions (end-

conditions.html)

**Note:** The extension is not important. Any extension can be used.

### Persistence

(persistence.html)

## Loading

### Loading a restore point from file

A restore point can be loaded from file using the JsonSave class:

```
JsonSave.FromFile(@"C:\save.json", out var restorePoint, out _);
```

### Applying a restore point to a game

The restore point can then be applied to a game to restore the state of the game:

```
restorePoint.Game.Restore(game);
```



# Commands

To simplify adding persistence to your game the *Save* and *Load* commands have been added.

Both *Save* and *Load* are custom commands, which means they can be applied to any *IExaminable*. This gives you the power to choose how you implement saving and loading. Add these commands to the *PlayableCharacter* or *Overworld* to have them always available, or add them to items, rooms or any other in game object.

```
Item typewriter = new("Typewriter", "An old style typewriter")
{
    new Save(),
    new Load()
}
+ Locations
+ Characters
```

## Conditional Descriptions

## Registering dynamically loaded content

It is recommended to *add* and *remove* content in the game by using the *IsPlayerVisible* property. This allows all content to be loaded when a game is created, which in turn allows serialization to function correctly. When a game is instantiated an instance of *AssetCatalog* is created which is a key component of serialization. If you choose to dynamically add an *Item*, *NonPlayableCharacter* or *Room* while a game is executing it may not serialize correctly as it will not be in the games *AssetCatalog*. This may be acceptable, however if you wish for the *IExaminable* to be serialized you need to register the instance of the *IExaminable* with the games *AssetCatalog*.

```
Item sword = new("Sword", "The sword of destiny");
game.Catalog.Register(sword);
```

## End Conditions

By registering with the *AssetCatalog* the game will be able to resolve references to the object when deserialization is applied.

## Persistence