AshLib Reference Documentation

Documentation for AshLib v3.2.1, by Siljam

https://github.com/siljamdev/AshLib

Documentation for AshLib v3.2.1, by Siljam.
https://github.com/siljamdev/AshLib

Index

Contents

Structs

Color3

Used for holding the values of a RGB color

Namespace

```
AshLib
```

Constructors

Color3(byte, byte, byte)

```
public Color3(byte r, byte g, byte b)
```

Initializes a new color taking in arguments for red, green, and blue channels.

Color3(string)

```
public Color3(string hex)
```

Initializes a new color taking in argument for the color in hex format. As an example of valid strings are:

```
“ffffff”, “#020202”, “#eFE02D”
```

Fields

R

```
public byte R;
```

Red channel of the color. Ranges 0-255

G

```
public byte G;
```

Green channel of the color. Ranges 0-255

B

```
public byte B;
```

Blue channel of the color. Ranges 0-255

Black

```
public static readonly Color3 Black = new Color3(0, 0, 0);
```

A completely black color

White

```
public static readonly Color3 White = new Color3(255, 255, 255);
```

A completely white color

Gray

```
public static readonly Color3 Gray = new Color3(150, 150, 150);
```

A medium gray color

Magenta

```
public static readonly Color3 Magenta = new Color3(255, 0, 255);
```

A magenta color

Cyan

```
public static readonly Color3 Cyan = new Color3(0, 255, 255);
```

A cyan color

Yellow

```csharp
public static readonly Color3 Yellow = new Color3(255, 255, 0);
```

A yellow color

Blue

```csharp
public static readonly Color3 Blue = new Color3(0, 0, 255);
```

A pure blue color

Green

```csharp
public static readonly Color3 Green = new Color3(0, 255, 0);
```

A pure green color

Red

```csharp
public static readonly Color3 Red = new Color3(255, 0, 0);
```

A pure red color

Methods

Parse

```csharp
public static Color3 Parse (string hex)
```

Attempts to parse the string in hex format. Same as constructor. Might throw an Exception

TryParse

```csharp
public static bool TryParse (string hex, out Color3 col)
```

Attempts to parse and returns true if it was possible. The out argument is the parsed Color if successful

Equals

```
public override bool Equals (object obj)
```

Checks if two Color3 are equals

ToString

```
public override string ToString ()
```

Returns the color data in the format:

```
“#RRGGBB”
```

Operators

Operator == (Color3, Color3)

```
public static bool operator == (Color3 a, Color3 b)
```

Checks if two Color3 are equals

Operator != (Color3, Color3)

```
public static bool operator != (Color3 a, Color3 b)
```

Checks if two Color3 are not equals

Implicit Operator Color (Color3)

```
public static implicit operator System.Drawing.Color (Color3 col)
```

Casts into a System.Drawing.Color

Implicit Operator Color3 (Color)

```
public static implicit operator Color3(System.Drawing.Color col)
```

Casts into a Color3

Vec2

Struct for holding two floats, components of a bidimensional vector

Namespace

```
AshLib
```

Constructors

Vec2(float, float)

```
public Vec2 (float x, float y)
```

Initializes with x and y component

Fields

X

```
public float X;
```

X component of the Vector

Y

```
public float Y;
```

Y component of the Vector

Methods

Equals

```
public override bool Equals (object obj)
```

Checks if two Vec2 contain the same data

ToString

```
public override string ToString ()
```

Returns the vector components in the format:

```
"(X, Y)"
```

Operators

Operator == (Vec2, Vec2)

```
public static bool operator == (Vec2 a, Vec2 b)
```

Checks if two Vec2 contain the same data

Operator != (Vec2, Vec2)

```
public static bool operator != (Vec2 a, Vec2 b)
```

Checks if two Vec2 do not contain the same data

Vec3

Struct for holding three floats, components of a tridimensional vector

Namespace

```
AshLib
```

Constructors

Vec3(float, float, float)

```
public Vec3 (float x, float y, float z)
```

Initializes with x, y and z component

Fields

X

```
public float X;
```

X component of the Vector

Y

```
public float Y;
```

Y component of the Vector

Z

```
public float Z;
```

Z component of the Vector

Methods

Equals

```
public override bool Equals (object obj)
```

Checks if two Vec3 contain the same data

ToString

```
public override string ToString ()
```

Returns the vector components in the format:

```
“(X, Y, Z)”
```

Operators

Operator == (Vec3, Vec3)

```
public static bool operator == (Vec3 a, Vec3 b)
```

Checks if two Vec3 contain the same data

Operator != (Vec3, Vec3)

```
public static bool operator != (Vec3 a, Vec3 b)
```

Checks if two Vec3 do not contain the same data

Vec4

Struct for holding four floats, components of a four-dimensional vector

Namespace

```
AshLib
```

Constructors

Vec4(float, float, float, float)

```
public Vec4 (float x, float y, float z, float w)
```

Initializes with x, y, z and w component

Fields

X

```
public float X;
```

X component of the Vector

Y

```
public float Y;
```

Y component of the Vector

Z

```
public float Z;
```

Z component of the Vector

W

```
public float W;
```

W component of the Vector

Methods

Equals

```
public override bool Equals (object obj)
```

Checks if two Vec4 contain the same data

ToString

```
public override string ToString ()
```

Returns the vector components in the format:

```
“(X, Y, Z, W)”
```

Operators

Operator == (Vec4, Vec4)

```
public static bool operator == (Vec4 a, Vec4 b)
```

Checks if two Vec4 contain the same data

Operator != (Vec4, Vec4)

```
public static bool operator != (Vec4 a, Vec4 b)
```

Checks if two Vec4 do not contain the same data

Date

Represents a time between the year 1488 and 2511, down to seconds. It's most important characteristic is CPTF(Compressed printable date format), which will transform the date into a 6 characters long string in base64

Namespace

```
AshLib.Dates
```

Constructors

Date(byte, byte, byte, byte, byte, ushort)

```
public Date(byte s, byte m, byte h, byte d, byte mo, ushort y)
```

Initializes the Date with seconds, minutes, hours, days, months, and years(in that order). Numbers will be cropped (seconds 0-59...)

Date(string)

```
public Date(string cptf)
```

Initializes the Date directly from CPTF format.

Properties

seconds

```
public byte seconds {get;}
```

The seconds of the Date(0-59)

minutes

```
public byte minutes {get;}
```

The minutes of the Date(0-59)

hours

```
public byte hours {get;}
```

The hours of the Date(0-23)

days

```
public byte days {get;}
```

The day of the Date(1-31)

months

```
public byte months {get;}
```

The month of the Date(1-12)

years

```
public ushort years {get;}
```

The year of the Date(1488-2511)

Methods

ToCPTF

```
public string ToCPTF ()
```

Transforms the Date instance into the CPTF format

FromCPTF

```
public static Date FromCPTF (string cptf)
```

Transforms a date in CPTF format into an instance. Does the same as the constructor

Equals

```
public override bool Equals (object obj)
```

Checks if two Dates are the same

ToString

```
public override string ToString ()
```

Returns the Date in the format:

```
“Day/Month/Year Hour:Minute:Second”
```

Operators

Operator == (Date, Date)

```
public static bool operator == (Date a, Date b)
```

Checks if two Dates are equals

Operator != (Date, Date)

```
public static bool operator != (Date a, Date b)
```

Checks if two Dates are not equals

Explicit Operator DateTime (Date)

```
public static explicit operator DateTime (Date d)
```

Casts to DateTime

Explicit Operator Date (DateTime)

```
public static explicit operator Date (DateTime d)
```

Casts to Date from DateTime

AshFileFormatConfig

Represents the config of the AshFile byte format

Namespace

```
AshLib.AshFiles
```

Constructors

AshFileFormatConfig(bool, bool, bool)

```
public AshFileFormatConfig(bool compactBoo, bool maskCampNam, bool maskStr)
```

Creates a new instance with the three bools

AshFileFormatConfig(byte)

```
public AshFileFormatConfig(byte compactByte)
```

Creates a new instance from a compact bool. Used in the byte format

Fields

compactBools

```
public bool compactBools;
```

If the bools and bool arrays will compact to occupy 8 bools per byte instead of 1 bool per byte

maskCampNames

```
public bool maskCampNames;
```

If the camp names will be masked so it is impossible to read them when viewed as raw text

maskStrings

```
public bool maskStrings;
```

If the string values will be masked so it is impossible to read them when viewed as raw text

Default

```
public static readonly AshFileFormatConfig Default = new AshFileFormatConfig(true,
true, true);
```

The default configuration

Methods

ToByte

```
public byte ToByte ()
```

Transforms into a compact bool. Used in the byte format

Enums

AshFileTypeOld

Represents the type of value in an AshFile camp. No longer used in AshFiles V3

Namespace

```
AshLib.AshFiles
```

Fields

ByteArray

```
ByteArray = 0;
```

Array of bytes type

String

```
String = 1;
```

String type

Byte

```
Byte = 2;
```

Byte type. 1 byte unsigned integer

Ushort

```
Ushort = 3;
```

Unsigned short type. 2 byte unsigned integer

Uint

```
Uint = 4;
```

Unsigned int type. 4 byte unsigned integer

Ulong

```
Ulong = 5;
```

Unsigned long type. 8 byte unsigned integer

Sbyte

```
Sbyte = 6;
```

Signed byte type. 1 byte signed integer

Short

```
Short = 7;
```

Signed short type. 2 byte signed integer

Int

```
Int = 8;
```

Signed int type. 4 byte signed integer

Long

```
Long = 9;
```

Signed long type. 8 byte signed integer

Color

```
Color = 10;
```

Color3 type. Struct defined earlier

Float

```
Float = 11;
```

Float type. 4 byte floating point number

Double

```
Double = 12;
```

Double type. 8 byte floating point number

Vec2

```
Vec2 = 13;
```

Vec2 type. 2 component vector, struct defined earlier

Vec3

```
Vec3 = 14;
```

Vec3 type. 3 component vector, struct defined earlier

Vec4

```
Vec4 = 15;
```

Vec4 type. 4 component vector, struct defined earlier

Bool

```
Bool = 16;
```

Boolean type

UbyteArray

```
UbyteArray = 17;
```

Array of bytes type. 1 byte unsigned integer array

UshortArray

```
UshortArray = 18;
```

Array of ushorts type. 2 byte unsigned integer array

UintArray

```
UintArray = 19;
```

Array of uints type. 4 byte unsigned integer array

UlongArray

```
UshortArray = 20;
```

Array of ulongs type. 4 byte unsigned integer array

SbyteArray

```
SbyteArray = 21;
```

Array of sbytes type. 1 byte signed integer array

ShortArray

```
ShortArray = 22;
```

Array of shorts type. 2 byte signed integer array

IntArray

```
IntArray = 23;
```

Array of ints type. 4 byte signed integer array

LongArray

```
LongArray = 24;
```

Array of longs type. 8 byte signed integer array

FloatArray

```
FloatArray = 25;
```

Array of floats type. 4 byte floating point number array

DoubleArray

```
DoubleArray = 26;
```

Array of doubles type. 8 byte floating point number array

Date

```
Date = 27;
```

Date type. Struct defined earlier

AshFileType

Represents the type of value in an AshFile camp. Used in AshFiles V3

Namespace

```
AshLib.AshFiles
```

Fields

Default

```
Default = 0;
```

Default type, often used for invalid types and error checking.

String

```
String = 1;
```

String type

Byte

```
Byte = 2;
```

Byte type

Ushort

```
Ushort = 3;
```

Unsigned short type

Uint

```
Uint = 4;
```

Unsigned int type

Ulong

```
Ulong = 5;
```

Unsigned long type

Sbyte

```
Sbyte = 6;
```

Signed byte type

Short

```
Short = 7;
```

Short type

Int

```
Int = 8;
```

Int type

Long

```
Long = 9;
```

Long type

Color3

```
Color3 = 10;
```

Color3 type. Struct defined earlier

Float

```
Float = 11;
```

Float type

Double

```
Double = 12;
```

Double type

Vec2

```
Vec2 = 13;
```

Two component vector type. Struct defined earlier

Vec3

```
Vec3 = 14;
```

Three component vector type. Struct defined earlier

Vec4

```
Vec4 = 15;
```

Four component vector type. Struct defined earlier.

Bool

```
Bool = 16;
```

Boolean type

Date

```
Date = 17;
```

Date type. Struct defined earlier

ModelInstanceOperation

Determines the operation that an instance of an AshFile model will do

Namespace

```
AshLib.AshFiles
```

Fields

Delete

```
Delete
```

The name mentioned will be deleted

Exists

```
Exists
```

Ensures the camp exists

Type

```
Type
```

Ensures that a camp is of a determined type

TypeCast

```
TypeCast
```

Ensures that a camp is of a determined type of one that can be casted

Value

```
Value
```

Ensures that a camp has a specific value

None

```
None
```

Does nothing. Mainly used to stop a camp being deleted by deleteNotMentioned

Classes

DeltaHelper

Class used for easily calculating fps and deltaTime

Namespace

```
AshLib.Time
```

Properties

deltaTime

```
public double deltaTime {get;}
```

Time that passes between frames, in milliseconds

fps

```
public double fps {get;}
```

Frames per second. Is calculated each frame, and so it fluctuates rapidly

stableFps

```
public double stableFps {get;}
```

Frames per second. Will update once every second(default behavior, can be changed). Thought for displaying current fps

Methods

Start

```
public void Start ()
```

Has to be called to start the utility. Will initialize the internal clock. Call it once at the start of the application

Frame

```
public void Frame ()
```

Call it at the very end of each frame, every frame. Updates all the values

Target

```
public void Target (double FPS)
```

Call it at the end of each frame, but before Frame(), to achieve the desired fps(the argument)

SetStableUpdateTime

```
public void SetStableUpdateTime (double milliseconds)
```

Changes how often will the stableFps update

GetTime

```
public double GetTime ()
```

Returns the whole time since start, in seconds

TimeTool

Class used for calculating the time things take, and calculating what percentage of time. Each loop(tick, frame...) is divided into categories. Useful for debugging what is holding fps down

Namespace

```
AshLib.Time
```

Constructors

TimeTool(params string[])

```
public TimeTool(params string[] categNames)
```

Starts a new instance with an empty history and the names of the categories

Fields

maxHistory

```
public int maxHistory
```

The maximum number of loops stored, used for calculating the mean. The default value is 1000

Properties

categoryNames

```
public string[] categoryNames {get;}
```

The category names

Methods

Reset

```
public void Reset ()
```

Resets the instance and deletes history

LoopStart

```
public void LoopStart ()
```

Meant to be executed at the start of each loop

CategoryEnd()

```
public void CategoryEnd ()
```

Ends the next category, in order. When executed after LoopStart, ends recording for the category in index 0. Next call for the one in index 1, etc.

CategoryEnd(int)

```
public void CategoryEnd (int i)
```

Ends a category specifying its index. The index will be saved and used for next. Useful for categories with parted times. For example:

```
LoopStart();
CategoryEnd(); //Ends 0
CategoryEnd(); //Ends 1
CategoryEnd(0); //Ends 0 again, times added
CategoryEnd(); //Ends 1 again
```

LoopEnd

```
public void LoopEnd ()
```

Ends the loop

LastLoopInfo

```
public double[] LastLoopInfo ()
```

Returns a double array with how much time each category took in the last loop. They are ordered according to index, and the last extra element is the total loop time

LastLoopString

```
public string LastLoopString ()
```

Returns a nicely formatted string with all the useful information about last loop

MeanInfo

```
public double[] MeanInfo ()
```

Returns a double array with how much time each category took on average in all recorded loops. They are ordered according to index, and the last extra element is the average loop time

MaxInfo

```
public double[] MaxInfo ()
```

Returns a double array with the maximum time each category took in all recorded loops. They are ordered according to index, and the last extra element is the maximum loop time

MeanString

```
public string MeanString ()
```

Generates a nicely formatted string with all the mean and max information of the recorded loops

TreeNode<T>

A class that represents a node of a tree graph. It can also represent a tree in itself with its children

Namespace

```
AshLib.Trees
```

Constructors

TreeNode(T)

```
public TreeNode (T val)
```

Initializes the node with its value

TreeNode(T, List<TreeNode<T>>)

```
public TreeNode (T val, List<TreeNode<T>> childlist)
```

Initializes the node with its value and the list of children nodes

Fields

value

```
public T value;
```

The value of the node

Properties

children

```
public List<TreeNode<T>> children {get;}
```

The list of child nodes

parent

```
public TreeNode<T>? parent {get;}
```

The parent node, if it exists

isLeaf

```
public bool isLeaf {get;}
```

A leaf node is a node with no children

isRoot

```
public bool isRoot {get;}
```

A Root node is the node that has no parent

Methods

AddChild

```
public void AddChild (TreeNode<T> node)
```

Adds a child node. Please use this method instead of directly adding to the children list, because this sets the parent too

RemoveChild

```
public bool RemoveChild (TreeNode<T> node)
```

Attempts to remove a child node and returns true if it was successful

ClearChildren

```
public void ClearChildren ()
```

Clears all children nodes

CountChildren

```
public int CountChildren ()
```

Returns the number of direct children

Clone

```
public TreeNode<T> Clone ()
```

Clones the tree

FindRoot

```
public TreeNode<T> FindRoot ()
```

Searches for the root of the tree

CountDescendants

```
public int CountDescendants ()
```

Counts the number of children and their children, recursively, giving the number of nodes of the tree downwards

CountLeafs

```
public int CountLeafs ()
```

Counts the number of leaf nodes in the descendants

DetermineTreeDepth

```
public int DetermineTreeDepth ()
```

Returns the max levels that the tree has until its deepest node. A single root node has a depth of 1

GetAllNodes

```
public List<TreeNode<T>> GetAllNodes ()
```

Returns all nodes of the tree in a flat list

GetAllLeafNodes

```
public List<TreeNode<T>> GetLeafNodes ()
```

Returns all leaf nodes in a flat list

TraversePreOrder

```
public void TraversePreOrder (Action<TreeNode<T>> action)
```

Traverses the tree, calling the action on each node, first calling the node and then its children, recursively

TraversePostOrder

```
public void TraversePostOrder (Action<TreeNode<T>> action)
```

Traverses the tree, calling the action on each node, first calling its children, and then the node itself, recursively

TraverseLevelOrder

```
public void TraverseLevelOrder (Action<TreeNode<T>> action)
```

Traverses the tree, calling the action on each node, calling first all nodes of the higher levels and then going down level by level

TraverseLeafsOnly

```
public void TraverseLeafsOnly (Action<TreeNode<T>> a)
```

Traverses the tree, calling the action on each leaf node

GetDepth

```
public int GetDepth ()
```

Gets the depth of a node respecting its general root. Root itself has depth 0

GetPathToRoot

```
public List<TreeNode<T>> GetPathToRoot ()
```

Gets the Node Path that is followed from the general root to the current node

FindNode(T)

```
public TreeNode<T>? FindNode (T target)
```

Searches for a node in all the descendants with a value that matches the target

FindNode(Func<TreeNode<T>, bool>)

```
public TreeNode<T>? FindNode (Func<TreeNode<T>, bool> condition)
```

Searches for a node in all the descendants based on a function that returns a boolean in function of each node searched

FindNode(Func<T, bool>)

```
public TreeNode<T>? FindNode(Func<T, bool> condition)
```

Searches for a node in all the descendants based on a function that returns a boolean in function of the value of the node searched

FindChildNode(T)

```
public TreeNode<T>? FindChildNode (T target)
```

Searches for a node in the direct children with a value that matches the target

FindChildNode(Func<TreeNode<T>, bool>)

```
public TreeNode<T>? FindChildNode (Func<TreeNode<T>, bool> condition)
```

Searches for a node in the direct children based on a function that returns a boolean in function of each node searched

FindChildNode(Func<T, bool>)

```
public TreeNode<T>? FindChildNode(Func<T, bool> condition)
```

Searches for a node in the direct children based on a function that returns a boolean in function of the value of the node searched

ToStringFormat

```
public string ToStringFormat ()
```

Transforms the tree into a string format that looks like this:

```
>R:1,2:3,4;5;
```

The tree always starts with > and the root element. Then, ":" is used for opening the list of child nodes, "," is used for specifying the next parallel child node, and ";" is used for closing the list of child nodes

ParseStringFormat(string)

```
public static TreeNode<string> ParseStringFormat (string s)
```

Parses the string format to a string tree, because it is just text. Might throw an Exception

ParseStringFormat(string, Func<string, T>)

```
public static TreeNode<T> ParseStringFormat (string s, Func<string, T> parseFunc)
```

Parses a tree into any type, providing a parsing function for that type

TryParseStringFormat(string, out TreeNode<string>)

```
public static bool TryParseStringFormat (string s, out TreeNode<string> tree)
```

Tries parsing from the string, and if it was possible, returns true and the parsed tree in the out argument

TryParseStringFormat(string, Func<string, T>, out TreeNode<string>)

```
public static bool TryParseStringFormat (string s, Func<string, T> parseFunc, out
TreeNode<string> tree)
```

Tries parsing from the string, and if it was possible, returns true and the parsed tree in the out argument

ToString

```
public override string ToString ()
```

Generates a nicely formatted way to visualize the tree idented. Do not mistake this with the string format

Dependencies

Used for handling files in a central folder of the application. Reminiscent of ".minecraft" folder

Namespace

```
AshLib.Folders
```

Constructors

Dependencies(string, bool, string[], string[])

```
public Dependencies (string path, bool config, string[] directories, string[]
files)
```

Initializes the utility. The first argument is the main path, the second is if you want the config AshFile to be created, the directories array specifies folders inside the main path that will be created, and the files array specifies files that will be created inside the main path or subfolders, will be created empty.

Fields

path

```
public string path;
```

The master path. For example:

```
"C://Users/user22/AppData/Roaming/HelloWorld"
```

config

```
public AshFile config;
```

The main configuration, in an AshFile. The path to it is “mainpath/config.ash”

Methods

ReadFileText

```
public string ReadFileText (string p)
```

Will read the contents of a file as text of the file in masterpath + argument

ReadAshFile

```
public AshFile ReadAshFile (string p)
```

Will read the contents of a file as an AshFile of the file in masterpath + argument

SaveFileText

```
public void SaveFileText (string p, string t)
```

Will save the text(argument t) in a file in masterpath + argument p

SaveAshFile

```
public void SaveAshFile (string p, AshFile a)
```

Will save the text(argument t) in a file in masterpath + argument p

CreateDir

```
public void CreateDir (string p)
```

Will create a new directory in masterpath + argument

TreeLog

Utility that helps you create a collapsible and indented, easy to read log with a tree structure. If you use notepad++ there is a custom language available for it

Namespace

```
AshLib.Logging
```

Methods

GetLog

```
public string GetLog ()
```

Returns the whole string of the log

Reset

```
public void Reset ()
```

Resets the content and level of the log

Deep

```
public void Deep (string s)
```

Increases the level of the log. You can include a message

Shallow

```
public void Shallow ()
```

Decreases the level of the log

Write(string)

```
public void Write (string s)
```

Writes content to the log

Write(object)

```
public void Write (object s)
```

Writes content to the log

AshFileException

Used internally for all exceptions that happen surrounding AshFiles

```
[Serializable]
internal class AshFileException : Exception
```

Namespace

```
AshLib.AshFiles
```

Properties

errorCode

```
public int errorCode {get;}
```

Different errors give different codes

Methods

GetFullMessage

```
public string GetFullMessage ()
```

Returns the whole information needed about the exception(error code, message, stack trace...) in a nice format

ToString

```
public override string ToString ()
```

Returns the same as GetFullMessage

GetObjectData

```
public override void GetObjectData(SerializationInfo info, StreamingContext
context)
```

Does something about Serialization. Honestly no idea

CharFormat

Used to determine the format of a character

Namespace

```
AshLib.Formatting
```

Constructors

CharFormat(byte?, bool?, byte?, bool?, Color3?, bool?, Color3?, bool?)

```
public CharFormat(byte? dens, bool? ital, byte? uline, bool? sthrough, Color3?
fgcolor, bool? fgreset, Color3? bgcolor, bool? bgreset)
```

Initializes with density, italic, underlined, strikethrough, foreground color, foreground reset, background color, background reset

CharFormat(Color3?, bool, Color3?, bool)

```
public CharFormat(Color3? fgcolor, bool fgreset, Color3? bgcolor, bool bgreset)
```

Initializes with foreground color, foreground reset, background color, background reset. The rest of the formatting properties are initialized to same as last

CharFormat(Color3?, bool)

```
public CharFormat(Color3? fgcolor, bool fgreset)
```

Initializes with foreground color and foreground reset. The rest of the formatting properties are initialized to same as last

CharFormat()

```
public CharFormat()
```

Initializes all formatting properties to same as last

Properties

density

```
public byte? density {get;}
```

Font density. 0 is normal, 1 is bold, 2 is thin. Null is same as last character format

italic

```
public bool? Italic {get;}
```

If the font is in italic or not. Null is same as last character format

underline

```
public byte? underline {get;}
```

0 is not underlined, 1 is single underline, 2 is double underline. Null is same as last character format

strikeThrough

```
public bool? strikeThrough {get;}
```

If the has a strike through or not. Null is same as last character format

foreground

```
public Color3? foreground {get;}
```

Foreground (character) color. Null is same as last character format

foregroundReset

```
public bool? foregroundReset {get;}
```

If the foreground color is reset to the terminal default

background

```
public Color3? background {get;}
```

Background color. Null is same as last character format

backgroundReset

```
public bool? backgroundReset {get;}
```

If the background color is reset to the terminal default

Methods

ToString

```
public override string ToString ()
```

Represents all the class properties in a nice format

Equals

```
public override bool Equals (object obj)
```

Checks if two formats are the same

Operators

Operator ==(CharFormat, CharFormat)

```
public static bool operator == (CharFormat a, CharFormat b)
```

Checks if two formats are the same

Operator !=(CharFormat, CharFormat)

```
public static bool operator != (CharFormat a, CharFormat b)
```

Checks if two formats are not the same

FormatString

A formatted string with support for colors and many more. (Note that windows terminal only supports colors and not bold and other options). It uses ANSI escape sequences

Namespace

```
AshLib.Formatting
```

Constructors

FormatString()

```
public FormatString ()
```

Initializes an empty string.

FormatString(string)

```
public FormatString (string s)
```

Initializes an string with content. Here, you can use the string formatting, explained later

Fields

addFinalReset

```
public bool addFinalReset;
```

If the built string will reset back to the terminal default at the end. Default is true

Properties

content

```
public string content {get;}
```

Gets the text content

format

```
public List<CharFormat?> format {get;}
```

Gets the format in the form of CharFormat, class defined earlier

built

```
public string built {get;}
```

The built string with the format applied

length

```
public int length {get;}
```

The length of the string

Methods

Clear

```
public void Clear ()
```

Clears the content of the string

Append(string, CharFormat?[])

```
public void Append (string s, CharFormat?[] f)
```

Appends text with a specific format. Each character has its own format

Append(string, CharFormat?)

```
public void Append (string s, CharFormat? f)
```

Appends text with a specific format. Each character has the same format

Append(object, CharFormat?)

```
public void Append (object s, CharFormat? f)
```

The object is transformed to string and the text is text is appended. Each character has the same format

Append(string)

```
public void Append (string s)
```

Text is appended, and text formatting can be used. Text formatting determined the format of the next characters. It is started with / and enclosed in []. Inside, the different values and options are separated by commas. This are all the options as examples:

```
"/[B]Hello World!" //Bold text
"/[T]Hello World!" //Thin text
"/[D]Hello World!" //Normal density text
"/[RT]Hello World!" //Normal density text
"/[RD]Hello World!" //Normal density text

"/[I]Hello World!" //Italic text
"/[RI]Hello World!" //Not italic text
```

```
"/[U]Hello World!" //Underlined text
"/[DU]Hello World!" //Double underlined text
"/[RU]Hello World!" //Not underlined text

"/[S]Hello World!" //Strike-through text
"/[RS]Hello World!" //No strike-through text

"/[C,255,0,0]Hello World!" //Red colored text(RGB)
"/[F,255,0,0]Hello World!" //Red colored text(RGB)
"/[C#,ff0000]Hello World!" //Red colored text(hex)
"/[F#,ff0000]Hello World!" //Red colored text(hex)
"/[RC]Hello World!" //Reset color to the terminal default text
"/[RF]Hello World!" //Reset color to the terminal default text

"/[BG,255,0,0]Hello World!" //Red background text(RGB)
"/[BG#,ff0000]Hello World!" //Red background text(hex)
"/[RB]Hello World!" //Reset background color to the terminal default text

"/[0]Hello World!" //All properties reset to the terminal default text

"/[C#,10ff30,B#,ffffff]Hello World!" //Green text over white background text.
                                     //Multiple properties can be set at once
```

Append(string, params object[])

```
public void Append (string s, params object[] objs)
```

Appends text but replaces {$X} formats. For example:

```
Append("Hello {$0}", "World"); //Results in "Hello World"
```

DeleteStart

```
public void DeleteStart (int n)
```

Deletes n chars from the start of the content

DeleteEnd

```
public void DeleteEnd (int n)
```

Deletes n chars from the end of the content

Delete

```
public void Delete(int si, int n)
```

Deletes n chars using as starting index si (0 indexing)

Equals

```
public override bool Equals (object obj)
```

Checks if two FormatStrings are equal

ToString

```
public override string ToString ()
```

Returns the text with the format applied. Same as built property

Operators

Operator +(FormatString, FormatString)

```
public static FormatString operator + (FormatString a, FormatString b)
```

Appends together two FormatStrings

Operator +(string, FormatString)

```
public static FormatString operator + (string a, FormatString b)
```

Appends together a string and a FormatString

Operator +(FormatString, string)

```
public static FormatString operator + (FormatString a, string b)
```

Appends together a FormatString and a string

Operator +(char, FormatString)

```
public static FormatString operator + (char a, FormatString b)
```

Appends together a char and a FormatString

Operator +(FormatString, char)

```
public static FormatString operator + (FormatString a, char b)
```

Appends together a FormatString and a char

Operator ==(FormatString, FormatString)

```
public static bool operator == (FormatString a, FormatString b)
```

Checks if two FormatStrings are equal

Operator !=(FormatString, FormatString)

```
public static bool operator != (FormatString a, FormatString b)
```

Checks if two FormatStrings are not equal

Implicit Operator FormatString(string)

```
public static implicit operator FormatString (string s)
```

Casts a string to a FormatString

AshFile

A Data structure made up from camps, that each have a name and a value(represented by an object because it can be of many types).
This structure makes it easy to be saved into a file in a new file format (.ash).
It also allow to be easily shared as text with its human-readable string format.

Namespace

```
AshLib.AshFiles
```

Constructors

AshFile(Dictionary<string, object>)

```
public AshFile(Dictionary<string, object> d)
```

Initializes a new AshFile using an existing Dictionary, the structure used internally

AshFile(string)

```
public AshFile(string path)
```

Loads an AshFile from the specified path

AshFile()

```
public AshFile()
```

Initializes a new empty AshFile

Fields

path

```
public string? path;
```

The path of the file, if it has any

format

```
public byte format;
```

Specifies the format in which the file will be saved. Latest format is 3

compactBools

```
public bool compactBools;
```

Config value for the file format. If set to true, bools will be compacted together using less space. Default is true

maskCampNames

```
public bool maskCampNames;
```

Config value for the file format. If set to true, camp names will be masked so they are not readable when the file is viewed as raw text. Default is true

maskStrings

```
public bool maskStrings;
```

Config value for the file format. If set to true, string values will be masked so they are not readable when the file is viewed as raw tex. Default is true

formatConfig

```
public AshFileFormatConfig formatConfig {get;}
```

Gets the file format config as a AshFileFormatConfig struct, defined earlier

numberOfCamps

```
public int numberOfcamps {get;}
```

Returns the number of camps

DefaultSeparator

```
public const string DefaultSeparator = ".";
```

The default separator for tree structures

Properties

data

```
public Dictionary<string, CampValue> data {get;}
```

The actual structure holding the data

Methods

Visualize

```
public string Visualize ()
```

Produces a list of all the elements. Arrays show all elements. The format is:

```
“Name1: Value1
 Name2: Value2”
```

ImportFormatConfig

```
public void ImportFormatConfig (AshFileFormatConfig conf)
```

Loads the format config from a struct

Clear

```
public void Clear ()
```

Deletes all camps

Load(string)

```
public void Load (string path)
```

Loads the file from the path and save the path internally

Load()

```
public void Load ()
```

Loads the file from the path saved internally

Save(string)

```
public void Save (string path)
```

Saves the file from the path and saves the path internally

Save()

```
public void Save ()
```

Saves the file from the internal path

ReadFromFile

```
public static Dictionary<string, object> ReadFromFile(string path, out byte f, out
AshFileFormatConfig conf)
```

Reads the file from the path argument and returns it. Outputs the format in the f argument and the format config in conf

ReadFromBytes

```
public static Dictionary<string, object> ReadFromFile(byte[] fileBytes, out byte
f, out AshFileFormatConfig conf)
```

Reads the file from the byte array and returns it. Outputs the format in the f argument

WriteToFile

```
public static void WriteToFile (string path, Dictionary<string, object>
dictionary, byte format, AshFileFormatConfig conf)
```

Writes the data into a file in the path argument. Format and config can be specified

WriteToBytes(Dictionary<string, CampValue>, byte)

```
public static byte[] WriteToBytes (Dictionary<string, object> dictionary, byte
format, AshFileFormatConfig conf)
```

Transforms the data into the byte representation. Format and config can be specified

WriteToBytes()

```
public byte[] WriteToBytes ()
```

Transforms the current instance into the byte representation.

GetErrorCount

```
public static ulong GetErrorCount ()
```

Gets the number of errors that occurred. These errors occur while converting to/from file

GetErrorLog

```
public static string GetErrorLog ()
```

Gets the whole log of errors that occurred, nicely formatted. These errors occur while converting to/from file

EmptyErrors

```
public static void EmptyErrors ()
```

Empties all the errors and sets the count to zero.

DeepCopy

```
public static AshFile DeepCopy (AshFile a)
```

Copies all the camps from one AshFile to a new one, keeping the references different

Merge

```
public static AshFile Merge (AshFile a1, AshFile a2)
```

Merges two AshFiles into 1. The second AshFile has priority if two camps are named the same

ApplyModel

```
public static AshFile ApplyModel (AshFile a, AshFileModel m)
```

Applies a model to an AshFile

GetCampTree

```
public TreeNode<string> GetCampTree (string separator)
```

Gets a string tree with the camp structure, using a string as separator

GetValueTree

```
public TreeNode<string> GetValueTree (string separator)
```

Gets a string with all the camps visualized nicely in a tree structure using a separator. Arrays show all elements too

VisualizeAsTree()

```
public string VisualizeAsTree ()
```

Returns the tree gotten with GetValueTree as a string, a nicely formatted tree structure. As separator, the DefaultSeparator is used

VisualizeAsTree(string)

```
public string VisualizeAsTree (string separator)
```

Returns the tree gotten with GetValueTree as a string, a nicely formatted tree structure

ExistsCamp

```
public bool ExistsCamp (string name)
```

Checks if a specific camp exists, returns true if it does.

SetCamp(string, object)

```
public void SetCamp (string name, object val)
```

Sets the camp with the argument name to the object. It can be of any type

InitializeCamp(string, object)

```
public void InitializeCamp (string name, object val)
```

If the camp with that name exists, nothing will happen. If the camp with that name doesn't exists, it will set it to that object

GetCamp

```
public object GetCamp (string name)
```

Returns the value of a camp. If the camp doesn't exists, a null object will be returned

CanGetCamp

```
public bool CanGetCamp (string name, out object val)
```

If the camp doesn't exists, returns false. If it is possible to get the value, it will return true and output the value in the out argument

GetCamp<T>

```
public T GetCamp (string name)
```

If the camp doesn't exist or isn't of the type T, it will return the default value of the T type. Else it will return the camp value directly cast

CanGetCamp<T>

```
public bool CanGetCamp<T> (string name, out T val)
```

If the camp doesn't exist or isn't of the type T, it will return false. Else it will return true and the camp value directly cast in the out argument

GetCampOrDefault<T>

```
public T GetCampOrDefault<T> (string name, T def)
```

If the camp doesn't exist or isn't of the type T, it will return the default value given in the arguments. Else it will return the camp value directly cast

GetCampType

```
public Type GetCampType (string name)
```

Returns the Type of value of a camp. If the camp doesn’t exist, null will be returned

CanGetCampType

```
public bool CanGetCampType (string name, out Type t)
```

If the camp doesn’t exists, returns false. If it is possible to get the type of value, it will return true and output the type in the out argument

DeleteCamp

```
public void DeleteCamp (string name)
```

Deletes the camp with that name if it exists

CanDeleteCamp

```
public bool CanDeleteCamp (string name)
```

Deletes the camp with that name if it exists and outputs true, else will output false

RenameCamp

```
public void RenameCamp (string oldName, string newName)
```

Renames the camp with the new name if it exists

CanRenameCamp

```
public bool CanRenameCamp (string oldName, string newName)
```

Renames the camp with the new name if it exists and output true, else will output false

Equals

```
public override bool Equals (object obj)
```

Checks if the contents of two AshFiles are the same

Parse

```
public static AshFile Parse (string s)
```

Parses an AshFile from a string in the AshFile string format. This is its structure of a camp:

```
<name> : type : value ; nextCamp…

<name> : type : [value1; value2; value3]; //For arrays
```

Note that the spaces can be any whitespace of any length. Here is the list of all types(same as supported by the file format):

```
@: string, values must be enclosed with “”
ub: byte
us: ushort
ui: uint
ul: ulong
sb: sbyte
s: short
n: int
i: int
l: long
#: Color3, struct defined earlier, values must be in hex
f: float
d: double
v2: Vec2, struct defined earlier, 2 float values are expected serparated by commas
v3: Vec3, struct defined earlier, 3 float values are expected serparated by commas
v4: Vec4, struct defined earlier, 4 float values are expected serparated by commas
b: boolean
dt: Date, struct defined earlier, 6 numbers are expected separated by /, day, month,
year, minute, second
```

Here is an example of an AshFile in this string format:

```
<names> :@: [“Julian”; “George”; “Donald”; “Viktor”; “Theodore”];
<todayDate>:dt:    13/4/2006/12/31/9; <message>:@:”Hello! Good morning!”;
    <emailID> :ul: 1298908;
```

TryParse

```
public static bool TryParse (string s, out AshFile a)
```

Tries parsing and returns true if its possible along with the parsed object in the out argument

ToString

```
public override string ToString ()
```

Returns the string representation of the AshFile

Operators

Operator == (AshFile, AshFile)

```
public static bool operator == (AshFile a1, AshFile a2)
```

Checks if the contents of two AshFiles are the same

Operator != (AshFile, AshFile)

```
public static bool operator != (AshFile a1, AshFile a2)
```

Checks if the contents of two AshFiles are not the same

Operator + (AshFile, AshFile)

```
public static AshFile operator + (AshFile a1, AshFile a2)
```

Will merge the two data structures. If both contain a camp with the same name, the second operand (a2) will have priority and put its camps over

Operator * (AshFile, AshFileModel)

```
public static AshFile operator * (AshFile b, AshFileModel m)
```

Will apply a model to an AshFile

Explicit Operator Dictionary<string, CampValue> (AshFile)

```
public static explicit operator Dictionary<string, object> (AshFile af)
```

Casts to a dictionary. Will just return the AshFile's data

Implicit Operator AshFile (Dictionary<string, CampValue>)

```
public static implicit operator AshFile (Dictionary<string, object> d)
```

Casts a dictionary into an AshFile. Create a new AshFile with the Dictionary as data

ModelInstance

An instance for the AshFileModel

Namespace

```
AshLib.AshFiles
```

Constructors

ModelInstance(ModelInstanceOperation, string, object)

```
public ModelInstance (ModelInstanceOperation o, string nam, object val)
```

Initlaizes a new ModelInstance with its type of operation, the target camp(name), and the value of the camp

Fields

operation

```
public ModelInstanceOperation operation;
```

The mode of operation of the instance

name

```
public string name;
```

The target camp name

value

```
public object value;
```

The value of the camp, its use depends on the operation mode

AshFileModel

Helps putting an AshFile into a correct format for easy use. It has instances, that will do a different number of things to camps, and actions, that are functions that will execute for all camps

Namespace

```
AshLib.AshFiles
```

Constructors

AshFileModel(params ModelInstance[])

```
public AshFileModel (params ModelInstance[] insArray)
```

Initializes a new model with the instances passed as arguments

Fields

allowUnsupportedTypes

```
public bool allowUnsupportedTypes;
```

If object types not supported by the file format of string format(they are the same) will be deleted. Default is true

deleteNotMentioned

```
public bool deleteNotMentioned;
```

If the camps not mentioned in the instance will be deleted. Default is false

DeleteUnsupportedTypes

```
public static readonly AshFileModel DeleteUnsupportedTypes;
```

A model that will delete all camps with types not supported

Properties

instances

```
public List<ModelInstance> instances {get;}
```

The instances of the model

actions

```
public List<Action<AshFile, string, object>> actions {get;}
```

The list of actions of the model that will be executed for all camps after the instances