

# Namespace Refinity.Benchmark

## Classes

[BenchmarkUtility](#)


# Class BenchmarkUtility

Namespace: [Refinity.Benchmark](#)








Assembly: Refinity.dll

```
public static class BenchmarkUtility
```

## Inheritance

[object](#)  ← BenchmarkUtility

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) ,  
[object.GetType\(\)](#) , [object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) ,  
[object.ToString\(\)](#) 

## Methods

### RunCodeBenchmark(Action, int)

Run a benchmark on an action.

```
public static BenchmarkModels RunCodeBenchmark(Action testMethod, int iterations  
= 1)
```

## Parameters

**testMethod** [Action](#) 

The method to test.

**iterations** [int](#) 

The number of iterations to run.

## Returns

[BenchmarkModels](#)

A BenchmarkModels object containing the benchmark results.

# RunCodeBenchmark<T>(Func<T>, int)

Run a benchmark on a method.

```
public static BenchmarkModels RunCodeBenchmark<T>(Func<T> testMethod, int iterations  
= 1)
```

## Parameters

**testMethod** [Func<T>](#)

The method to test.

**iterations** [int](#)

The number of iterations to run.

## Returns

[BenchmarkModels](#)

A BenchmarkModels object containing the benchmark results.

## Type Parameters

**T**

The type of the method's return value.

# Namespace Refinity.Benchmark.Enums

## Enums

[BenchmarkResult](#)

# Enum BenchmarkResult

Namespace: [Refinity.Benchmark.Enums](#)

Assembly: Refinity.dll

```
public enum BenchmarkResult
```

## Fields

Failure = 1

Success = 0

# Namespace Refinity.Benchmark.Models

## Classes

[BenchmarkModels](#)


# Class BenchmarkModels

Namespace: [Refinity.Benchmark.Models](#)








Assembly: Refinity.dll

```
public class BenchmarkModels
```

## Inheritance

[object](#)  ← BenchmarkModels

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) ,  
[object.GetType\(\)](#) , [object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) ,  
[object.ToString\(\)](#) 

## Constructors

### BenchmarkModels()

```
public BenchmarkModels()
```

## Properties

### ElapsedTimeMs

```
public double ElapsedTimeMs { get; set; }
```

### Property Value

[double](#) 

## Exception

```
public Exception? Exception { get; set; }
```

Property Value

[Exception](#)

## Iterations

```
public int Iterations { get; set; }
```

Property Value

[int](#)

## Method

```
public string Method { get; set; }
```

Property Value

[string](#)

## Result

```
public BenchmarkResult Result { get; set; }
```

Property Value

[BenchmarkResult](#)



# Namespace Refinity.Conversion

## Classes

[ConvertUtility](#)

[LogEntry](#)

# Class ConvertUtility

Namespace: [Refinity.Conversion](#)








Assembly: Refinity.dll

```
public static class ConvertUtility
```

## Inheritance

[object](#)  ← ConvertUtility

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) ,  
[object.GetType\(\)](#) , [object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) ,  
[object.ToString\(\)](#) 

## Methods

### ConvertCsvToDataTable(string, char)

Converts a CSV file to a DataTable.

```
public static DataTable ConvertCsvToDataTable(string path, char delimiter = ',')
```

## Parameters

**path** [string](#) 

The path of the CSV file.

**delimiter** [char](#) 

The delimiter used in the CSV file. Default is ','.

## Returns

[DataTable](#) 

A DataTable containing the data from the CSV file.

## ConvertCsvToObject<T>(Stream, char, bool)

Converts a CSV file to a list of objects of type T.

```
public static List<T> ConvertCsvToObject<T>(Stream stream, char delimiter = ',',  
bool ignoreCaseHeader = false)
```

### Parameters

**stream** [Stream](#)

The stream containing the CSV data.

**delimiter** [char](#)

The delimiter used to separate values in the CSV file. Default is ','.

**ignoreCaseHeader** [bool](#)

If true, the header is case insensitive. Default is false.

### Returns

[List](#)<T>

A list of objects of type T.

### Type Parameters

**T**

The type of objects to convert to.

## ConvertTextToCSV(string, char, char, string[]?)

Converts a text file to a CSV file.

```
public static byte[] ConvertTextToCSV(string pathToTxt, char delimiter = ',', char  
saveDelimiter = ',', string[]? headers = null)
```

### Parameters

`pathToTxt` [string](#)

The path to the input text file.

`delimiter` [char](#)

The character used to separate values in the text file. Default is ','.

`saveDelimiter` [char](#)

The character used to separate values in the CSV file. Default is ','.

`headers` [string](#) []

Returns

[byte](#) []

True if the conversion is successful, false otherwise.

## ConvertToBase64(string)

Converts a file to a Base64 string representation.

```
public static string ConvertToBase64(string path)
```

Parameters

`path` [string](#)

The path of the file to convert.

Returns

[string](#)

The Base64 string representation of the file.

# Class LogEntry

Namespace: [Refinity.Conversion](#)








Assembly: Refinity.dll

```
public class LogEntry
```

## Inheritance

[object](#)  ← LogEntry

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) ,  
[object.GetType\(\)](#) , [object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) ,  
[object.ToString\(\)](#) 

## Properties

### LogLevel

```
public string? LogLevel { get; set; }
```

### Property Value

[string](#) 

### Message

```
public string? Message { get; set; }
```

### Property Value

[string](#) 

### Severity

```
public int Severity { get; set; }
```

Property Value

[int](#)

Time

```
public DateTime Time { get; set; }
```

Property Value

[DateTime](#)

# Namespace Refinity.Date

## Classes

[DateDifference](#)

[DateUtility](#)

# Class DateDifference

Namespace: [Refinity.Date](#)








Assembly: Refinity.dll

```
public class DateDifference
```

## Inheritance

[object](#)  ← DateDifference

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) ,  
[object.GetType\(\)](#) , [object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) ,  
[object.ToString\(\)](#) 

## Properties

### Days

```
public int Days { get; set; }
```

### Property Value

[int](#) 

### Hours

```
public int Hours { get; set; }
```

### Property Value

[int](#) 

### Minutes



```
public int Minutes { get; set; }
```

Property Value

[int](#)

## Months

```
public int Months { get; set; }
```

Property Value

[int](#)

## Seconds

```
public int Seconds { get; set; }
```

Property Value

[int](#)

## Years

```
public int Years { get; set; }
```

Property Value

[int](#)

# Class DateUtility

Namespace: [Refinity.Date](#)








Assembly: Refinity.dll

```
public static class DateUtility
```

## Inheritance

[object](#)  ← DateUtility

## Inherited Members

[object.Equals\(object\)](#)  , [object.Equals\(object, object\)](#)  , [object.GetHashCode\(\)](#)  ,  
[object.GetType\(\)](#)  , [object.MemberwiseClone\(\)](#)  , [object.ReferenceEquals\(object, object\)](#)  ,  
[object.ToString\(\)](#) 

## Methods

### Add(DateTime, int, int, int, int, int, int)

Adds a specified number of years, months, days, hours, minutes, and seconds to the given DateTime value. TEST TEST TEST

```
public static DateTime Add(this DateTime value, int years = 0, int months = 0, int  
days = 0, int hours = 0, int minutes = 0, int seconds = 0)
```

## Parameters

value [DateTime](#) 

The DateTime value to which the specified time interval should be added.

years [int](#) 

The number of years to add. The default value is 0.

months [int](#) 

The number of months to add. The default value is 0.

**days** [int](#)

The number of days to add. The default value is 0.

**hours** [int](#)

The number of hours to add. The default value is 0.

**minutes** [int](#)

The number of minutes to add. The default value is 0.

**seconds** [int](#)

The number of seconds to add. The default value is 0.

## Returns

[DateTime](#)

A new DateTime value that is the result of adding the specified time interval to the original DateTime value.

## CalculateAge(DateTime)

Calculates the age based on the provided birth date.

```
public static int CalculateAge(this DateTime birthDate)
```

## Parameters

**birthDate** [DateTime](#)

The birth date.

## Returns

[int](#)

The calculated age.

## DeserializeDateTime(string)

Deserializes a string representation of a DateTime object.

```
public static DateTime DeserializeDateTime(this string dateTimeString)
```

### Parameters

**dateTimeString** [string](#)<sup>↗</sup>

The string representation of the DateTime object.

### Returns

[DateTime](#)<sup>↗</sup>

The deserialized DateTime object.

## FirstDayOfMonth(DateTime)

Returns the first day of the month for the specified DateTime value.

```
public static DateTime FirstDayOfMonth(this DateTime value)
```

### Parameters

**value** [DateTime](#)<sup>↗</sup>

The DateTime value.

### Returns

[DateTime](#)<sup>↗</sup>

The first day of the month.


## GetDateRange(DateTime, int, bool)

```
public static DateRangeModel GetDateRange(this DateTime startDate, int
numberOfMonths, bool startToFirst = false)
```

## Parameters

startDate [DateTime](#) 

numberOfMonths [int](#) 

startToFirst [bool](#) 

## Returns

[DateRangeModel](#)

## GetDateRangeFromWeekNumber(int)

Gets the date range (start and end dates) for a given week number.

```
public static DateTime[] GetDateRangeFromWeekNumber(int weekNumber)
```

## Parameters

weekNumber [int](#) 

The week number.

## Returns

[DateTime](#)  []

An array of DateTime objects representing the start and end dates of the week.

## GetDifference(DateTime, DateTime)

```
public static DateDifference GetDifference(this DateTime from, DateTime to)
```

## Parameters

from [DateTime](#)

to [DateTime](#)

Returns

[DateDifference](#)

## GetWeekNumber(DateTime)

Gets the week number of the specified date.

```
public static int GetWeekNumber(this DateTime value)
```

Parameters

value [DateTime](#)

The date value.

Returns

[int](#)

The week number of the specified date.

## LastDayOfMonth(DateTime)

Returns the last day of the month for the specified DateTime value.

```
public static DateTime LastDayOfMonth(this DateTime value)
```

Parameters

value [DateTime](#)

The DateTime value.

Returns

[DateTime](#)<sup>↗</sup>

The last day of the month.

## QuarterFromMonth(Months)

Calculates the quarter from a given month.

```
public static int QuarterFromMonth(Months value)
```

### Parameters

value [Months](#)

The month value.

### Returns

[int](#)<sup>↗</sup>

The quarter corresponding to the given month.

## QuarterFromMonth(DateTime)

Calculates the quarter from the given month.

```
public static int QuarterFromMonth(this DateTime value)
```

### Parameters

value [DateTime](#)<sup>↗</sup>

The month value.

### Returns

[int](#)<sup>↗</sup>

The quarter corresponding to the given month.

## QuarterFromMonth(int)

Calculates the quarter from a given month value.

```
public static int QuarterFromMonth(this int value)
```

### Parameters

value [int](#)

The month value.

### Returns

[int](#)

The quarter corresponding to the month value.

## QuarterlyFromMonth(Months)

Calculates the quarterly value from a given month.

```
public static int QuarterlyFromMonth(Months value)
```

### Parameters

value [Months](#)

The month value.

### Returns

[int](#)

The quarterly value.

## QuarterlyFromMonth(DateTime)

Calculates the quarterly value from the given month.



```
public static int QuarterlyFromMonth(this DateTime value)
```

## Parameters

value [DateTime](#)

The month value.

## Returns

[int](#)

The quarterly value.

## QuarterlyFromMonth(int)

Calculates the quarterly value from a given month.

```
public static int QuarterlyFromMonth(this int value)
```

## Parameters

value [int](#)

The month value.

## Returns

[int](#)

The quarterly value.

## SerializeDateTime(DateTime)

Serializes a DateTime object into a JSON string representation.

```
public static string SerializeDateTime(this DateTime dateTime)
```

## Parameters

**dateTime** [DateTime](#)

The DateTime object to be serialized.

## Returns

[string](#)

A JSON string representation of the DateTime object.

## Subtract(DateTime, int, int, int, int, int, int)

Subtracts a specified number of years, months, days, hours, minutes, and seconds from the given DateTime value.

```
public static DateTime Subtract(this DateTime value, int years = 0, int months = 0,
int days = 0, int hours = 0, int minutes = 0, int seconds = 0)
```

## Parameters

**value** [DateTime](#)

The DateTime value to subtract from.

**years** [int](#)

The number of years to subtract. Default is 0.

**months** [int](#)

The number of months to subtract. Default is 0.

**days** [int](#)

The number of days to subtract. Default is 0.

**hours** [int](#)

The number of hours to subtract. Default is 0.

**minutes** [int](#)

The number of minutes to subtract. Default is 0.

**seconds** [int](#)

The number of seconds to subtract. Default is 0.

## Returns

[DateTime](#)

A new DateTime value that is the result of subtracting the specified years, months, days, hours, minutes, and seconds from the given DateTime value.

## ToDateTime(string)

Converts a string value to a nullable DateTime object.

```
public static DateTime? ToDateTime(this string value)
```

## Parameters

**value** [string](#)

The string value to convert.

## Returns

[DateTime](#)?

A nullable DateTime object representing the converted value, or null if the conversion fails.

# Namespace Refinity.Finance

## Classes

[FinanceUtility](#)

# Class FinanceUtility

Namespace: [Refinity.Finance](#)








Assembly: Refinity.dll

```
public static class FinanceUtility
```

## Inheritance

[object](#)  ← FinanceUtility

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) ,  
[object.GetType\(\)](#) , [object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) ,  
[object.ToString\(\)](#) 

## Methods

### ApplyTax(double, double)

Applies tax to the given amount.

```
public static double ApplyTax(double amount, double taxRate)
```

## Parameters

**amount** [double](#) 

The amount to apply tax to.

**taxRate** [double](#) 

The tax rate in percentage.

## Returns

[double](#) 

The amount with tax applied.

## BreakEvenPoint(double, double, double)

Calculates the break-even point based on fixed costs, variable costs, and selling price per unit.

```
public static double BreakEvenPoint(double fixedCosts, double variableCosts,  
double sellingPrice)
```

### Parameters

**fixedCosts** [double](#)

The total fixed costs.

**variableCosts** [double](#)

The variable costs per unit.

**sellingPrice** [double](#)

The selling price per unit.

### Returns

[double](#)

The break-even point in units.

## CalculateCompoundInterest(double, double, double)

Calculates the compound interest based on the principal amount, interest rate, and time period.

```
public static double CalculateCompoundInterest(double principal, double  
interestRate, double timePeriod)
```

### Parameters

**principal** [double](#)

The principal amount.

**interestRate** [double](#)

The interest rate.

`timePeriod` [double](#)

The time period in years.

## Returns

[double](#)

The calculated compound interest.

## CalculateInterest(double, double, double)

Calculates the interest based on the principal amount, interest rate, and time period.

```
public static double CalculateInterest(double principal, double interestRate,  
double timePeriod)
```

## Parameters

`principal` [double](#)

The principal amount.

`interestRate` [double](#)

The interest rate.

`timePeriod` [double](#)

The time period in years.

## Returns

[double](#)

The calculated interest.

## CalculateInternalRateOfReturn(List<double>)

Calculates the internal rate of return (IRR) of a series of cash flows.

```
public static double CalculateInternalRateOfReturn(List<double> cashFlows)
```

## Parameters

cashFlows [List](#) <[double](#)>

The cash flows.

## Returns

[double](#)

The calculated internal rate of return.

## CalculateNetPresentValue(double, List<double>)

Calculates the net present value (NPV) of a series of cash flows based on a discount rate.

```
public static double CalculateNetPresentValue(double discountRate,  
List<double> cashFlows)
```

## Parameters

discountRate [double](#)

The discount rate.

cashFlows [List](#) <[double](#)>

The cash flows.

## Returns

[double](#)

The calculated net present value.



## CalculateSimpleInterest(double, double, double)

Calculates the simple interest based on the principal amount, interest rate, and time period.

```
public static double CalculateSimpleInterest(double principal, double interestRate, double timePeriod)
```

### Parameters

**principal** [double](#)

The principal amount.

**interestRate** [double](#)

The interest rate.

**timePeriod** [double](#)

The time period in years.

### Returns

[double](#)

The calculated simple interest.

## DaysUntilDue(DateTime)

Calculates the number of days until the specified due date.

```
public static int DaysUntilDue(DateTime dueDate)
```

### Parameters

**dueDate** [DateTime](#)

The due date to calculate the days until.

### Returns

[int](#)

The number of days until the due date.

## DifferencePercentage(double, double)

Calculates the difference between two values as a percentage.

```
public static double DifferencePercentage(this double value, double otherValue)
```

### Parameters

value [double](#)

The first value.

otherValue [double](#)

The second value.

### Returns

[double](#)

The difference between the two values as a percentage.

## DifferencePercentage(int, int)

Calculates the difference between two integers as a percentage.

```
public static double DifferencePercentage(this int value, int otherValue)
```

### Parameters

value [int](#)

The first integer value.

otherValue [int](#)

The second integer value.

## Returns

[double](#)

The difference between the two integers as a percentage.

## PaybackPeriod(double, List<double>)

Calculates the time required to recover the cost of an investment.

```
public static double PaybackPeriod(double initialInvestment, List<double> cashFlows)
```

## Parameters

**initialInvestment** [double](#)

The initial investment cost.

**cashFlows** [List](#)<[double](#)>

The cash flows generated by the investment.

## Returns

[double](#)

The payback period in years.

## PredictFutureValue(double, double, DateTime, DateTime)

Predicts the future value based on the present value, interest rate, start date, and end date.

```
public static double PredictFutureValue(double presentValue, double interestRate,  
DateTime startDate, DateTime endDate)
```

## Parameters

**presentValue** [double](#)

The present value.

**interestRate** [double](#)

The interest rate.

**startDate** [DateTime](#)

The start date.

**endDate** [DateTime](#)

The end date.

## Returns

[double](#)

The predicted future value.

## PredictFutureValue(double, double, double)

Calculates the future value of a present value based on the interest rate and time period.

```
public static double PredictFutureValue(double presentValue, double interestRate,  
double timePeriod)
```

## Parameters

**presentValue** [double](#)

The present value.

**interestRate** [double](#)

The interest rate.

**timePeriod** [double](#)

The time period in years.

## Returns

[double](#)

The future value of the present value.

## ToCurrency(double)

Converts a double value to a currency string representation.

```
public static string ToCurrency(this double value)
```

### Parameters

value [double](#)

The double value to convert.

### Returns

[string](#)

A string representation of the double value formatted as currency.

## ToCurrency(double, CultureInfo)

Converts the specified integer value to a currency string representation using the specified culture information.

```
public static string ToCurrency(this double value, CultureInfo cultureInfo)
```

### Parameters

value [double](#)

The integer value to convert.

cultureInfo [CultureInfo](#)

The culture information used for formatting the currency string.

## Returns

[string](#) 

A string representation of the specified integer value formatted as a currency.

# Namespace Refinity.Geometry

## Classes

### [CoordinateModel](#)

Represents a coordinate model with X, Y, and optional Z values.

### [GeometryUtility](#)

# Class CoordinateModel

Namespace: [Refinity.Geometry](#)

Assembly: Refinity.dll








Represents a coordinate model with X, Y, and optional Z values.

```
public class CoordinateModel
```

## Inheritance

[object](#)  ← CoordinateModel

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) ,  
[object.GetType\(\)](#) , [object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) ,  
[object.ToString\(\)](#) 

## Properties

### X

Gets or sets the X value of the coordinate.

```
public double X { get; set; }
```

### Property Value

[double](#) 

### Y

Gets or sets the Y value of the coordinate.

```
public double Y { get; set; }
```

### Property Value



[double](#)<sup>↗</sup>

## Z

Gets or sets the optional Z value of the coordinate.

```
public double? Z { get; set; }
```

## Property Value

[double](#)<sup>↗</sup>?

# Class GeometryUtility

Namespace: [Refinity.Geometry](#)








Assembly: Refinity.dll

```
public static class GeometryUtility
```

## Inheritance

[object](#)  ← GeometryUtility

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) ,  
[object.GetType\(\)](#) , [object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) ,  
[object.ToString\(\)](#) 

## Methods

### AreaOfCircle(double)

Calculates the area of a circle.

```
public static double AreaOfCircle(double radius)
```

## Parameters

**radius** [double](#) 

The radius of the circle.

## Returns

[double](#) 

The area of the circle.

### AreaOfParallelogram(double, double)

Calculates the area of a parallelogram.

```
public static double AreaOfParallelogram(double @base, double height)
```

## Parameters

**base** [double](#)

The base of the parallelogram.

**height** [double](#)

The height of the parallelogram.

## Returns

[double](#)

The area of the parallelogram.

## AreaOfRectangle(double, double)

Calculates the area of a rectangle.

```
public static double AreaOfRectangle(double length, double width)
```

## Parameters

**length** [double](#)

The length of the rectangle.

**width** [double](#)

The width of the rectangle.

## Returns

[double](#)

The area of the rectangle.

## AreaOfRhombus(double, double)

Calculates the area of a rhombus.

```
public static double AreaOfRhombus(double diagonal1, double diagonal2)
```

### Parameters

**diagonal1** [double](#)

The first diagonal of the rhombus.

**diagonal2** [double](#)

The second diagonal of the rhombus.

### Returns

[double](#)

The area of the rhombus.

## AreaOfSector(double, double)

Calculates the area of a sector.

```
public static double AreaOfSector(double radius, double angle)
```

### Parameters

**radius** [double](#)

The radius of the sector.

**angle** [double](#)

The angle of the sector.

### Returns

[double](#)

The area of the sector.

## AreaOfSquare(double)

Calculates the area of a square.

```
public static double AreaOfSquare(double side)
```

### Parameters

**side** [double](#)

The side of the square.

### Returns

[double](#)

The area of the square.

## AreaOfTrapezoid(double, double, double)

Calculates the area of a trapezoid.

```
public static double AreaOfTrapezoid(double base1, double base2, double height)
```

### Parameters

**base1** [double](#)

The first base of the trapezoid.

**base2** [double](#)

The second base of the trapezoid.

**height** [double](#)

The height of the trapezoid.

## Returns

[double](#) 

The area of the trapezoid.

## AreaOfTriangle(double, double)

Calculates the area of a triangle.

```
public static double AreaOfTriangle(double @base, double height)
```

## Parameters

**base** [double](#) 

The base of the triangle.

**height** [double](#) 

The height of the triangle.

## Returns

[double](#) 

The area of the triangle.

## CalculateDistance(double, double, double, double)

Calculates the distance between two points in a two-dimensional space.

```
public static double CalculateDistance(double x1, double y1, double x2, double y2)
```

## Parameters

**x1** [double](#) 

The x-coordinate of the first point.

y1 [double](#)

The y-coordinate of the first point.

x2 [double](#)

The x-coordinate of the second point.

y2 [double](#)

The y-coordinate of the second point.

Returns

[double](#)

The distance between the two points.

## CalculateIsoscelesTrapezoidPerimeter(double, double, double)

Calculates the perimeter of an isosceles trapezoid.

```
public static double CalculateIsoscelesTrapezoidPerimeter(double base1, double  
base2, double side)
```

Parameters

base1 [double](#)

The length of the first base.

base2 [double](#)

The length of the second base.

side [double](#)

The length of the side.

Returns

[double](#)

The perimeter of the isosceles trapezoid.

## CalculateIsoscelesTrapezoidSide(double, double, double)

Calculates the length of a side of an isosceles trapezoid given the lengths of its bases and height.

```
public static double CalculateIsoscelesTrapezoidSide(double base1, double base2,  
double height)
```

### Parameters

**base1** [double](#)

The length of the first base of the trapezoid.

**base2** [double](#)

The length of the second base of the trapezoid.

**height** [double](#)

The height of the trapezoid.

### Returns

[double](#)

The length of the side of the trapezoid.

## CalculateParallelogramPerimeter(double, double)

Calculates the perimeter of a parallelogram.

```
public static double CalculateParallelogramPerimeter(double baseLength,  
double sideLength)
```

### Parameters

**baseLength** [double](#)



The length of the base of the parallelogram.

**sideLength** [double](#)

The length of the side of the parallelogram.

Returns

[double](#)

The perimeter of the parallelogram.

## CalculateRectangleDiagonal(double, double)

Calculates the diagonal length of a rectangle using its length and width.

```
public static double CalculateRectangleDiagonal(double length, double width)
```

Parameters

**length** [double](#)

The length of the rectangle.

**width** [double](#)

The width of the rectangle.

Returns

[double](#)

The diagonal length of the rectangle.

## CalculateRectanglePerimeter(double, double)

Calculates the perimeter of a rectangle.

```
public static double CalculateRectanglePerimeter(double length, double width)
```

## Parameters

**length** [double](#)

The length of the rectangle.

**width** [double](#)

The width of the rectangle.

## Returns

[double](#)

The perimeter of the rectangle.

## CalculateRhombusPerimeter(double)

Calculates the perimeter of a rhombus given the length of its side.

```
public static double CalculateRhombusPerimeter(double side)
```

## Parameters

**side** [double](#)

The length of the side of the rhombus.

## Returns

[double](#)

The perimeter of the rhombus.

## CalculateRightTrapezoidDiagonal(double, double, double, double)

Calculates the diagonal of a right trapezoid.

```
public static double CalculateRightTrapezoidDiagonal(double base1, double base2,  
double height, double side)
```

## Parameters

**base1** [double](#)

The length of the first base of the trapezoid.

**base2** [double](#)

The length of the second base of the trapezoid.

**height** [double](#)

The height of the trapezoid.

**side** [double](#)

The length of the side of the trapezoid.

## Returns

[double](#)

The diagonal of the right trapezoid.

## CalculateRightTrapezoidPerimeter(double, double, double, double)

Calculates the perimeter of a right trapezoid.

```
public static double CalculateRightTrapezoidPerimeter(double base1, double base2,  
double height, double side)
```

## Parameters

**base1** [double](#)

The length of the first base.

**base2** [double](#)

The length of the second base.

**height** [double](#)

The height of the trapezoid.

side [double](#)

The length of the side.

Returns

[double](#)

The perimeter of the right trapezoid.

## CalculateSlope(double, double, double, double)

Calculates the slope between two points on a Cartesian plane.

```
public static double CalculateSlope(double x1, double y1, double x2, double y2)
```

Parameters

x1 [double](#)

The x-coordinate of the first point.

y1 [double](#)

The y-coordinate of the first point.

x2 [double](#)

The x-coordinate of the second point.

y2 [double](#)

The y-coordinate of the second point.

Returns

[double](#)

The slope between the two points.

## CalculateSquarePerimeter(double)

Calculates the perimeter of a square.

```
public static double CalculateSquarePerimeter(double side)
```

### Parameters

side [double](#)

The length of a side of the square.

### Returns

[double](#)

The perimeter of the square.

## CalculateTrapezoidPerimeter(double, double, double, double)

Calculates the perimeter of a trapezoid.

```
public static double CalculateTrapezoidPerimeter(double base1, double base2, double side1, double side2)
```

### Parameters

base1 [double](#)

The length of the first base of the trapezoid.

base2 [double](#)

The length of the second base of the trapezoid.

side1 [double](#)

The length of the first side of the trapezoid.

side2 [double](#)

The length of the second side of the trapezoid.

## Returns

[double](#) 

The perimeter of the trapezoid.

## ToCartesianCoordinates(double, double)

Converts Cartesian coordinates to cartesian coordinates.

```
public static CoordinateModel ToCartesianCoordinates(double radius, double angle)
```

## Parameters

radius [double](#) 

The radius.

angle [double](#) 

The angle.

## Returns

[CoordinateModel](#)

## ToPolarCoordinates(double, double)

Converts Cartesian coordinates to polar coordinates.

```
public static CoordinateModel ToPolarCoordinates(double x, double y)
```

## Parameters

x [double](#) 

The x-coordinate.

y [double](#) 

The y-coordinate.

Returns

[CoordinateModel](#)

# Namespace Refinity.Logging

## Classes

[LoggingUtility](#)



# Class LoggingUtility

Namespace: [Refinity.Logging](#)








Assembly: Refinity.dll

```
public class LoggingUtility
```

## Inheritance

[object](#)  ← LoggingUtility

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) ,  
[object.GetType\(\)](#) , [object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) ,  
[object.ToString\(\)](#) 

## Constructors

### LoggingUtility(string, LogFileType)

Represents a utility class for logging. logFileName should not include the file extension.

```
public LoggingUtility(string logFileName, LogFileType logFileType = LogFileType.TXT)
```

## Parameters

logFileName [string](#) 

logFileType [LogFileType](#)

## Methods

### ConvertLogToCSV(char)

```
public byte[] ConvertLogToCSV(char csvDelimiter = ';')
```

## Parameters

csvDelimiter [char](#)

Returns

[byte](#)[]

## Debug(string, int)

Writes a debug message to the log.

```
public void Debug(string message, int severity = 0)
```

Parameters

message [string](#)

The message to be logged.

severity [int](#)

The severity of the error.

## Error(string, int)

Logs an error message.

```
public void Error(string message, int severity = 0)
```

Parameters

message [string](#)

The message to be logged.

severity [int](#)

The severity of the error.

## Fatal(string, int)

Logs a fatal error message.

```
public void Fatal(string message, int severity = 0)
```

### Parameters

message [string](#)

The message to be logged.

severity [int](#)

The severity of the error.

## Info(string, int)

Logs an informational message.

```
public void Info(string message, int severity = 0)
```

### Parameters

message [string](#)

The message to be logged.

severity [int](#)

The severity of the error.

## Warn(string, int)

Logs a warning message.

```
public void Warn(string message, int severity = 0)
```

### Parameters

**message** [string](#)

The message to be logged.

**severity** [int](#)

The severity of the error.

# Namespace Refinity.Logging.Enums

## Enums

[LogFileType](#)

[LogLevel](#)

# Enum LogFileType

Namespace: [Refinity.Logging.Enums](#)

Assembly: Refinity.dll

```
public enum LogFileType
```

## Fields

CONSOLE = 2

LOG = 3

RTF = 0

TXT = 1

# Enum LogLevel

Namespace: [Refinity.Logging.Enums](#)

Assembly: Refinity.dll

```
public enum LogLevel
```

## Fields

DEBUG = 1

ERROR = 4

FATAL = 5

INFO = 2

TRACE = 0

WARNING = 3

# Namespace Refinity.Logging.Models

## Classes

[LogColorHelper](#)



# Class LogColorHelper

Namespace: [Refinity.Logging.Models](#)








Assembly: Refinity.dll

```
public class LogColorHelper
```

## Inheritance

[object](#)  ← LogColorHelper

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) ,  
[object.GetType\(\)](#) , [object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) ,  
[object.ToString\(\)](#) 

## Methods

### GetLogLevelColor(LogLevel)

```
public static ConsoleColor GetLogLevelColor(LogLevel logLevel)
```

## Parameters

**logLevel** [LogLevel](#)

## Returns

[ConsoleColor](#) 

# Namespace Refinity.Math

## Classes

[LinearRegressionModel](#)

[MathUtility](#)

# Class LinearRegressionModel

Namespace: [Refinity.Math](#)








Assembly: Refinity.dll

```
public class LinearRegressionModel
```

## Inheritance

[object](#)  ← LinearRegressionModel

## Inherited Members

[object.Equals\(object\)](#)  , [object.Equals\(object, object\)](#)  , [object.GetHashCode\(\)](#)  ,  
[object.GetType\(\)](#)  , [object.MemberwiseClone\(\)](#)  , [object.ReferenceEquals\(object, object\)](#)  ,  
[object.ToString\(\)](#) 

## Constructors

LinearRegressionModel(double, double, double)

```
public LinearRegressionModel(double m, double b, double r)
```

## Parameters

**m** [double](#) 

**b** [double](#) 

**r** [double](#) 

## Properties

### Correlation

```
public double Correlation { get; set; }
```

Property Value

[double](#)

Intercept

```
public double Intercept { get; set; }
```

Property Value

[double](#)

Slope

```
public double Slope { get; set; }
```

Property Value

[double](#)

# Class MathUtility

Namespace: [Refinity.Math](#)








Assembly: Refinity.dll

```
public static class MathUtility
```

## Inheritance

[object](#)  ← MathUtility

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) ,  
[object.GetType\(\)](#) , [object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) ,  
[object.ToString\(\)](#) 

## Methods

### Clamp(double, double, double)

Clamps a value between a minimum and maximum value.

```
public static double Clamp(this double value, double min, double max)
```

## Parameters

value [double](#) 

The value to clamp.

min [double](#) 

The minimum value.

max [double](#) 

The maximum value.

## Returns

[double](#)

The clamped value.

## ConvertToBaseBinary(int)

Converts an integer to its binary representation in base 2.

```
public static int ConvertToBaseBinary(this int n)
```

### Parameters

**n** [int](#)

The integer to convert.

### Returns

[int](#)

The binary representation of the input integer.

## ConvertToBaseHexadecimal(int)

Converts an integer to its hexadecimal representation.

```
public static int ConvertToBaseHexadecimal(this int n)
```

### Parameters

**n** [int](#)

The integer to convert.

### Returns

[int](#)

The hexadecimal representation of the input integer.

## ConvertToBaseOctal(int)

Converts an integer to its octal representation.

```
public static int ConvertToBaseOctal(this int n)
```

### Parameters

**n** [int](#)

The integer to be converted.

### Returns

[int](#)

The octal representation of the input integer.

## DegreesToHMS(double)

Converts degrees to hours, minutes, and seconds.

```
public static (int hours, int minutes, double seconds) DegreesToHMS(double degrees)
```

### Parameters

**degrees** [double](#)

The degrees to convert.

### Returns

([int](#) [hours](#), [int](#) [minutes](#), [double](#) [seconds](#))

A tuple containing the hours, minutes, and seconds.

## DegreesToHMSString(double)

Converts the given degrees to a string representation in hours, minutes, and seconds format.

```
public static string DegreesToHMSSString(this double degrees)
```

## Parameters

**degrees** [double](#)

The degrees to convert.

## Returns

[string](#)

A string representation of the degrees in hours, minutes, and seconds format.

## DegreesToRadians(double)

Converts degrees to radians.

```
public static double DegreesToRadians(this double degrees)
```

## Parameters

**degrees** [double](#)

The angle in degrees.

## Returns

[double](#)

The angle in radians.

## Divisors(int)

Returns an array of divisors for the given integer.

```
public static int[] Divisors(this int n)
```



## Parameters

**n** [int](#)

The integer for which to find divisors.

## Returns

[int](#) []

An array of divisors.

## Factorial(int)

Calculates the factorial of a non-negative integer.

```
public static int Factorial(this int n)
```

## Parameters

**n** [int](#)

The non-negative integer.

## Returns

[int](#)

The factorial of the input integer.

## Fibonacci(int)

Calculates the Fibonacci number for a given integer.

```
public static int Fibonacci(this int n)
```

## Parameters

**n** [int](#)

The input integer.

## Returns

[int](#)

The Fibonacci number.

## GetStandardDeviation(double[])

Calculates the mean and standard deviation of an array of doubles.

```
public static (double mean, double standardDeviation) GetStandardDeviation(this double[] values)
```

## Parameters

values [double](#) []

The array of doubles.

## Returns

([double](#) mean, [double](#) standardDeviation)

A tuple containing the mean and standard deviation.

## GetStandardDeviation(int[])

Calculates the mean and standard deviation of an array of integers.

```
public static (double mean, double standardDeviation) GetStandardDeviation(this int[] values)
```

## Parameters

values [int](#) []

The array of integers.

## Returns

([double](#) [mean](#), [double](#) [standardDeviation](#))

A tuple containing the mean and standard deviation.

## GreatestCommonDivisor(int, int)

Calculates the greatest common divisor (GCD) of two integers.

```
public static int GreatestCommonDivisor(this int a, int b)
```

## Parameters

**a** [int](#)

The first integer.

**b** [int](#)

The second integer.

## Returns

[int](#)

The greatest common divisor of the two integers.

## Invert(double)

Inverts the specified value.

```
public static double Invert(this double value)
```

## Parameters

**value** [double](#)

The value to invert.

## Returns

[double](#)

The inverted value.

## Invert(int)

Inverts the specified integer value.

```
public static double Invert(this int value)
```

## Parameters

value [int](#)

The value to invert.

## Returns

[double](#)

The inverted value.

## IsApproximatelyEqualTo(double, double, double)

Determines whether the specified value is approximately equal to the other value within the given tolerance.

```
public static bool IsApproximatelyEqualTo(this double value, double otherValue,  
double tolerance)
```

## Parameters

value [double](#)

The value to compare.

otherValue [double](#)

The other value to compare.

**tolerance** [double](#)

The tolerance within which the values are considered equal.

## Returns

[bool](#)

**true** if the values are approximately equal; otherwise, **false**.

## IsDivisibleBy(int, int)

Determines whether an integer is divisible by a given divisor.

```
public static bool IsDivisibleBy(this int n, int divisor)
```

## Parameters

**n** [int](#)

The integer to check for divisibility.

**divisor** [int](#)

The divisor to check against.

## Returns

[bool](#)

True if the integer is divisible by the divisor, otherwise false.

## IsDivisibleBy(int, int, out int)

Determines whether an integer is divisible by a given divisor and calculates the quotient.

```
public static bool IsDivisibleBy(this int n, int divisor, out int quotient)
```

## Parameters

**n** [int](#)

The integer to check for divisibility.

**divisor** [int](#)

The divisor to check against.

**quotient** [int](#)

The calculated quotient if the integer is divisible by the divisor.

## Returns

[bool](#)

True if the integer is divisible by the divisor, false otherwise.

## IsEven(int)

Determines whether the specified integer is even.

```
public static bool IsEven(this int n)
```

## Parameters

**n** [int](#)

The integer to check.

## Returns

[bool](#)

true if the integer is even; otherwise, false.

## IsInRange(double, double, double)

Determines whether the specified value is within the specified range.

```
public static bool IsInRange(this double value, double min, double max)
```

## Parameters

value [double](#)

The value to check.

min [double](#)

The minimum value of the range.

max [double](#)

The maximum value of the range.

## Returns

[bool](#)

true if the value is within the range; otherwise, false.

## IsOdd(int)

Determines whether the specified integer is odd.

```
public static bool IsOdd(this int n)
```

## Parameters

n [int](#)

The integer to check.

## Returns

[bool](#)

true if the specified integer is odd; otherwise, false.

## IsPrime(double)

Determines whether the specified number is a prime number.

```
public static bool IsPrime(this double n)
```

### Parameters

**n** [double](#)

The number to check.

### Returns

[bool](#)

**true** if the number is prime; otherwise, **false**.

## IsPrime(int)

Determines whether the specified number is a prime number.

```
public static bool IsPrime(this int n)
```

### Parameters

**n** [int](#)

The number to check.

### Returns

[bool](#)

**true** if the number is prime; otherwise, **false**.

## LeastCommonMultiple(int, int)

Calculates the least common multiple (LCM) of two integers.



```
public static int LeastCommonMultiple(this int a, int b)
```

## Parameters

**a** [int](#)

The first integer.

**b** [int](#)

The second integer.

## Returns

[int](#)

The least common multiple of the two integers.

## LogBaseN(double, double)

Calculates the logarithm of a specified value in a specified base.

```
public static double LogBaseN(this double value, double n)
```

## Parameters

**value** [double](#)

The value for which to calculate the logarithm.

**n** [double](#)

The base of the logarithm.

## Returns

[double](#)

The logarithm of the specified value in the specified base.

## MatrixAddition(dynamic, dynamic)

Adds two matrices together.

```
public static dynamic MatrixAddition(dynamic matrix1, dynamic matrix2)
```

### Parameters

**matrix1** dynamic

The first matrix.

**matrix2** dynamic

The second matrix.

### Returns

dynamic

The result of the matrix addition.

## MatrixInverse(dynamic)

Represents a type that can hold values of any type.

```
public static dynamic MatrixInverse(dynamic matrix)
```

### Parameters

**matrix** dynamic

### Returns

dynamic

## MatrixMultiplication(dynamic, dynamic)

Performs matrix multiplication on two dynamic matrices.

```
public static dynamic MatrixMultiplication(dynamic matrix1, dynamic matrix2)
```

## Parameters

**matrix1** dynamic

The first matrix.

**matrix2** dynamic

The second matrix.

## Returns

dynamic

The result of the matrix multiplication.

## MatrixScalarMultiplication(dynamic, double)

Performs scalar multiplication on a matrix.

```
public static dynamic MatrixScalarMultiplication(dynamic matrix, double scalar)
```

## Parameters

**matrix** dynamic

The matrix to be multiplied.

**scalar** [double](#)

The scalar value to multiply the matrix by.

## Returns

dynamic

The result of the matrix scalar multiplication.

## MatrixSubtraction(dynamic, dynamic)

Performs subtraction of two matrices.

```
public static dynamic MatrixSubtraction(dynamic matrix1, dynamic matrix2)
```

### Parameters

**matrix1** dynamic

The first matrix.

**matrix2** dynamic

The second matrix.

### Returns

dynamic

The result of the matrix subtraction.

## MatrixTranspose(dynamic)

Transposes a matrix.

```
public static dynamic MatrixTranspose(dynamic matrix)
```

### Parameters

**matrix** dynamic

The matrix to transpose.

### Returns

dynamic

The transposed matrix.

## Median(params double[])

Calculates the median value of an array of numbers.

```
public static double Median(params double[] numbers)
```

### Parameters

**numbers** [double](#) []

The array of numbers.

### Returns

[double](#)

The median value.

## Mode(params double[])

Calculates the mode value of an array of numbers.

```
public static double Mode(params double[] numbers)
```

### Parameters

**numbers** [double](#) []

The array of numbers.

### Returns

[double](#)

The mode value.

## NextPrime(double)

Finds the next prime number greater than the specified number.

```
public static double NextPrime(this double n)
```

## Parameters

**n** [double](#)

The number to find the next prime number after.

## Returns

[double](#)

The next prime number greater than **n**.

## NextPrime(int)

Finds the next prime number greater than the specified number.

```
public static int NextPrime(this int n)
```

## Parameters

**n** [int](#)

The number to find the next prime number after.

## Returns

[int](#)

The next prime number greater than **n**.

## Normalize(double, double, double)

Normalizes a value within a specified range.

```
public static double Normalize(this double value, double min, double max)
```

## Parameters

**value** [double](#)

The value to be normalized.

**min** [double](#)

The minimum value of the range.

**max** [double](#)

The maximum value of the range.

## Returns

[double](#)

The normalized value.

## PercentageOf(double, double)

Calculates the percentage of a value relative to another value.

```
public static double PercentageOf(this double value, double otherValue)
```

## Parameters

**value** [double](#)

The value to calculate the percentage of.

**otherValue** [double](#)

The value to calculate the percentage relative to.

## Returns

[double](#)

The percentage of the value relative to the other value.

## PercentageOf(int, int)

Calculates the percentage of a value relative to another value.

```
public static double PercentageOf(this int value, int otherValue)
```

### Parameters

value [int](#)

The value to calculate the percentage of.

otherValue [int](#)

The value to calculate the percentage relative to.

### Returns

[double](#)

The percentage of the value relative to the other value.

## PerformLinearRegression(double[], double[])

Performs linear regression on the given arrays of x and y values.

```
public static LinearRegressionModel PerformLinearRegression(this double[] x,  
double[] y)
```

### Parameters

x [double](#) []

The array of x values.

y [double](#) []

The array of y values.

### Returns



## [LinearRegressionModel](#)

An object containing the slope (m), y-intercept (b), and correlation coefficient (r).

## PerformLinearRegression(int[], int[])

Performs linear regression on the given arrays of x and y values.

```
public static LinearRegressionModel PerformLinearRegression(this int[] x, int[] y)
```

### Parameters

x [int\[\]](#)

The array of x values.

y [int\[\]](#)

The array of y values.

### Returns

## [LinearRegressionModel](#)

An object containing the slope (m), y-intercept (b), and correlation coefficient (r).

## RadiansToDegrees(double)

Converts an angle from radians to degrees.

```
public static double RadiansToDegrees(this double radians)
```

### Parameters

radians [double](#)

The angle in radians.

### Returns

[double](#)

The angle in degrees.

## SimpsonRuleIntegration(Func<double, double>, double, double, int)

Performs numerical integration using Simpson's rule.

```
public static double SimpsonRuleIntegration(Func<double, double> function, double a, double b, int n)
```

### Parameters

**function** [Func](#)<[double](#), [double](#)>

The function to integrate.

**a** [double](#)

The lower limit of integration.

**b** [double](#)

The upper limit of integration.

**n** [int](#)

The number of intervals.

### Returns

[double](#)

The approximate value of the integral.

## SumTo(double, double)

Calculates the sum of an arithmetic series up to a given number of terms.

```
public static double SumTo(this double value, double n)
```

## Parameters

value [double](#)

The first term of the series.

n [double](#)

The number of terms in the series.

## Returns

[double](#)

The sum of the arithmetic series.

## SumTo(int, int)

Calculates the sum of an arithmetic series up to a given number of terms.

```
public static double SumTo(this int value, int n)
```

## Parameters

value [int](#)

The first term of the series.

n [int](#)

The number of terms in the series.

## Returns

[double](#)

The sum of the arithmetic series.

## ToScientificNotation(double)

Converts a number to scientific notation.

```
public static string ToScientificNotation(this double number)
```

### Parameters

**number** [double](#)

The number to convert.

### Returns

[string](#)

The number in scientific notation.

## ToStringPercentage(double, int)

Formats a double value as a percentage string.

```
public static string ToStringPercentage(this double value, int decimalPlaces = 2)
```

### Parameters

**value** [double](#)

The double value to format.

**decimalPlaces** [int](#)

The number of decimal places to include in the formatted string. Default is 2.

### Returns

[string](#)

A string representation of the double value formatted as a percentage.

# Namespace Refinity.Strings

## Classes

[StringsUtility](#)


# Class StringsUtility

Namespace: [Refinity.Strings](#)








Assembly: Refinity.dll

```
public static class StringsUtility
```

## Inheritance

[object](#)  ← StringsUtility

## Inherited Members

[object.Equals\(object\)](#) , [object.Equals\(object, object\)](#) , [object.GetHashCode\(\)](#) ,  
[object.GetType\(\)](#) , [object.MemberwiseClone\(\)](#) , [object.ReferenceEquals\(object, object\)](#) ,  
[object.ToString\(\)](#) 

## Methods

### FromBase64(string)

Converts a base64 encoded string to its original UTF-8 representation.

```
public static string FromBase64(this string base64)
```

## Parameters

**base64** [string](#) 

The base64 encoded string to convert.

## Returns

[string](#) 

The original UTF-8 representation of the base64 encoded string.

### FromBase64(string, Encoding)

Converts a Base64 encoded string to its original form using the specified encoding.

```
public static string FromBase64(this string base64, Encoding encoding)
```

## Parameters

**base64** [string](#)

The Base64 encoded string to convert.

**encoding** [Encoding](#)

The encoding to use for decoding the Base64 string.

## Returns

[string](#)

The original string represented by the Base64 encoded string.

## IsPalindrome(string)

Determines whether a string is a palindrome.

```
public static bool IsPalindrome(this string input)
```

## Parameters

**input** [string](#)

The string to check.

## Returns

[bool](#)

True if the string is a palindrome; otherwise, false.

## RemoveHTMLTags(string)

Removes HTML tags from a string.

```
public static string RemoveHTMLTags(this string input)
```

## Parameters

input [string](#)

The input string.

## Returns

[string](#)

The string with HTML tags removed.

## RemoveTabs(string, bool)

Removes tabs and optionally new lines from a string.

```
public static string RemoveTabs(this string input, bool removeNewLines)
```

## Parameters

input [string](#)

The input string.

removeNewLines [bool](#)

A boolean value indicating whether to remove new lines.

## Returns

[string](#)

The modified string with tabs and new lines removed.

## RemoveWhitespace(string)



Removes all whitespace characters from the input string.

```
public static string RemoveWhitespace(this string input)
```

## Parameters

input [string](#)

The string to remove whitespace from.

## Returns

[string](#)

A new string with all whitespace characters removed.

## Reverse(string)

Reverses the characters in a string.

```
public static string Reverse(this string input)
```

## Parameters

input [string](#)

The string to be reversed.

## Returns

[string](#)

The reversed string.

## SplitCamelCase(string)

Splits a camel case string into an array of strings.

```
public static string[] SplitCamelCase(this string input)
```

## Parameters

**input** [string](#)

The camel case string to split.

## Returns

[string](#) []

An array of strings representing the split camel case string.

## ToBase64(string)

Converts a string to its Base64 representation.

```
public static string ToBase64(this string text)
```

## Parameters

**text** [string](#)

The string to convert.

## Returns

[string](#)

The Base64 representation of the input string.

## ToBase64(string, Encoding)

Converts a string to its Base64 representation using the specified encoding.

```
public static string ToBase64(this string text, Encoding encoding)
```

## Parameters

**text** [string](#)

The string to convert.

**encoding** [Encoding](#)

The encoding to use.

## Returns

[string](#)

The Base64 representation of the input string.

## ToPascalCase(string)

Converts the specified string to title case.

```
public static string ToPascalCase(this string input)
```

## Parameters

**input** [string](#)

The string to convert.

## Returns

[string](#)

The specified string converted to title case.

## Truncate(string, int)

Truncates a string to the specified maximum length.

```
public static string Truncate(this string input, int maxLength)
```

## Parameters

**input** [string](#)

The input string.

**maxLength** [int](#)

The maximum length to truncate the string.

## Returns

[string](#)

The truncated string.