Math Class

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
三角関数や対数関数などの一般的な数値関数の定数と静的メソッドを提供します。

---

public static class Math

継承：Object → Math

例

次の例では、Math クラスの三角関数を使用して台形の内角を計算しています。

```
/// <summary>
/// The following class represents simple functionality of the trapezoid.
/// </summary>
using System;

namespace MathClassCS
{
    class MathTrapezoidSample
    {
                private Rational m_longBase;
                private Rational m_shortBase;
                private Rational m_leftLeg;
                private Rational m_rightLeg;

                public MathTrapezoidSample(Rational longbase, Rational
shortbase, Rational leftLeg, Rational rightLeg)
                {
                        m_longBase = Math.Abs(longbase);
                        m_shortBase = Math.Abs(shortbase);
```

```
                m_leftLeg = Math.Abs(leftLeg);
                m_rightLeg = Math.Abs(rightLeg);
        }

        private Rational GetRightSmallBase()
        {
                return (Math.Pow(m_rightLeg,2.0) -
Math.Pow(m_leftLeg,2.0) + Math.Pow(m_longBase,2.0) +
Math.Pow(m_shortBase,2.0) - 2* m_shortBase * m_longBase)/
(2*(m_longBase - m_shortBase));
        }

        public Rational GetHeight()
        {
                Rational    x = GetRightSmallBase();
                return Math.Sqrt(Math.Pow(m_rightLeg,2.0) -
Math.Pow(x,2.0));
        }

        public Rational GetSquare()
        {
                return GetHeight() * m_longBase / 2.0;
        }

        public Rational GetLeftBaseRadianAngle()
        {
                Rational sinX = GetHeight()/m_leftLeg;
                return Math.Round(Math.Asin(sinX),2);
        }

        public Rational GetRightBaseRadianAngle()
        {
                Rational x = GetRightSmallBase();
                Rational cosX = (Math.Pow(m_rightLeg,2.0) +
Math.Pow(x,2.0) - Math.Pow(GetHeight(),2.0))/(2*x*m_rightLeg);
                return Math.Round(Math.Acos(cosX),2);
```

```
        }

        public Rational GetLeftBaseDegreeAngle()
        {
                Rational x = GetLeftBaseRadianAngle() * 180/ Math.PI;
                return Math.Round(x,2);
        }

        public Rational GetRightBaseDegreeAngle()
        {
                Rational x = GetRightBaseRadianAngle() * 180/ Math.PI;
                return Math.Round(x,2);
        }

        static void Main(string[] args)
        {
                MathTrapezoidSample trpz = new
MathTrapezoidSample(20.0, 10.0, 8.0, 6.0);
                Console.WriteLine("The trapezoid's bases are 20.0 and
10.0, the trapezoid's legs are 8.0 and 6.0");
                Rational h = trpz.GetHeight();
                Console.WriteLine("Trapezoid height is: " +
h.ToString());
                Rational dxR = trpz.GetLeftBaseRadianAngle();
                Console.WriteLine("Trapezoid left base angle is: " +
dxR.ToString() + " Radians");
                Rational dyR = trpz.GetRightBaseRadianAngle();
                Console.WriteLine("Trapezoid right base angle is: " +
dyR.ToString() + " Radians");
                Rational dxD = trpz.GetLeftBaseDegreeAngle();
                Console.WriteLine("Trapezoid left base angle is: " +
dxD.ToString() + " Degrees");
                Rational dyD = trpz.GetRightBaseDegreeAngle();
                Console.WriteLine("Trapezoid left base angle is: " +
dyD.ToString() + " Degrees");
        }
```

```
    }
}
```

フィールド

E	定数 e によって示される、自然対数の底を表します。
PI	定数（π）を指定して、円の直径に対する円周の割合を表します。

メソッド

Abs(Rational)	Rational 数値の絶対値を返します。
Acos(Rational)	コサインが指定数となる角度を返します。
Asin(Rational)	サインが指定数となる角度を返します。
Atan(Rational)	タンジェントが指定数となる角度を返します。
Atan2(Rational,Rational)	タンジェントが 2 つの指定された数の商である角度を返します。
Ceiling(Rational)	指定した Rational 以上の数のうち、最小の整数値を返します。
Cos(Rational)	指定された角度のコサインを返します。
Cosh(Rational)	指定された角度のハイパーボリック コサインを返します。
DivRem(Rational,Rational)	2 つの数値の商を計算し、出力パラメーターの剰余を返します。
Exp(Rational)	指定した値で e を累乗した値を返します。
Floor(Rational)	指定した Rational 以下の数のうち、最大の整数値を返します。
IeeeRemainder (Rational,Rational)	指定した数を別の指定数で除算した結果の剰余を返します。
Log(Rational)	指定した数の自然（底 e）対数を返します。
Log(Rational,Rational)	指定した数値の指定した底での対数を返します。
Log10(Rational)	指定した数の底 10 の対数を返します。
Max(Rational,Rational)	2 つの Rational のうち、大きな方を返します。
Min(Rational,Rational)	2 つの Rational のうち、小さい方を返します。
Pow(Rational,Rational)	指定の数値を指定した値で累乗した値を返します。
Round(Rational,int,	Rational の値は指定した小数部の桁数に丸められ、中間値

MidpointRounding)	には指定した丸め処理が使用されます。
Round(Rational, MidpointRounding)	Rational の値は最も近い整数に丸められ、中間値には指定した丸め処理が使用されます。
Round(Rational,int)	Rational の値は指定した小数部の桁数に丸められ、中間値は最も近い偶数値に丸められます。
Round(Rational)	Rational の値は最も近い整数値に丸められ、中間値は最も近い偶数値に丸められます。
Sign(Rational)	Rational の符号を示す整数を返します。
Sin(Rational)	指定された角度のサインを返します。
Sinh(Rational)	指定された角度のハイパーボリック サインを返します。
Sqrt(Rational)	指定された数値の平方根を返します。
Root(Rational,Rational)	指定された数値の冪根を返します。
Tan(Rational)	指定された角度のタンジェントを返します。
Tanh(Rational)	指定された角度のハイパーボリック タンジェントを返します。
Truncate(Rational)	指定した Rational の整数部を計算します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1 2

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.E Field

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
定数 e によって示される、自然対数の底を表します。

```
public static Rational E { get; }
```

フィールド値
Rational

注釈

このフィールドの値は、約 2.7182818284590451 です。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Math.PI Field

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
定数（π）を指定して、円の直径に対する円周の割合を表します。

```
public static Rational PI { get; }
```

フィールド値
Rational

注釈

このフィールドの値は、約 3.1415926535897931 です。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Math. Abs(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 数値の絶対値を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Abs(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
Ratioan 型の数値。

戻り値
Rational
value の絶対値。

例

Abs メソッドは次の例の様に使用します。次の例では Rational をファイルにシリアライズする前に符号と絶対値の表現にしています。その後ファイルからデシリアライズして Rational 値にしています。

```
using System;
using System.IO;
using System.Numerics;
using System.Runtime.Serialization.Formatters.Binary;
using WS.Theia.ExtremelyPrecise;
[Serializable] public struct SignAndMagnitude
{
    public int Sign;
```

```
    public byte[] Bytes;
}

public class Example
{
    public static void Main()
    {
        FileStream fs;
        BinaryFormatter formatter = new BinaryFormatter();
        Rational number = Math.Pow(Int32.MaxValue, 20) *
Rational.MinusOne;
        Console.WriteLine("The original value is {0}.", number);
        SignAndMagnitude sm = new SignAndMagnitude();
        sm.Sign=Math.Sign(number);
        sm.Bytes = Math.Abs(number).ToByteArray().Numerator;

        // Serialize SignAndMagnitude value.
        fs = new FileStream(@".¥data.bin", FileMode.Create);
        formatter.Serialize(fs, sm);
        fs.Close();

        // Deserialize SignAndMagnitude value.
        fs = new FileStream(@".¥data.bin", FileMode.Open);
        SignAndMagnitude smRestored = (SignAndMagnitude)
formatter.Deserialize(fs);
        fs.Close();
        Rational restoredNumber = new Rational(false,smRestored.Bytes,new
byte[]{ 1 });
        restoredNumber *= sm.Sign;
        Console.WriteLine("The deserialized value is {0}.", restoredNumber);
    }
}
// The example displays the following output:
//       The original value is -4.351082396632343274374874405 8E+186.
//       The deserialized value is -4.351082396632343274374874405 8E+186.
```

注釈

絶対値の取得は、次の表に示す結果になります。

value パラメーター	戻り値
value >= +0	value
value <= -0	value * -1

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Acos(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
コサインが指定数となる角度を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Acos(WS.Theia.ExtremelyPrecise.Rational cos);
```

パラメーター

cos　Rational
コサインを表す数。-1 以上 1 以下である必要があります。

戻り値

Rational
$0 \leq \theta \leq \pi$ の、ラジアンで表した角度 θ。 または cos < -1 または cos > 1、あるいは cos が NaN と等しい場合は、NaN。

例

次の例では、Acos を使用して台形の内角を計算しています。

```
/// <summary>
/// The following class represents simple functionality of the trapezoid.
/// </summary>
using System;
using WS.Theia.ExtremelyPrecise;

namespace MathClassCS
{
    class MathTrapezoidSample
    {
            private Rational m_longBase;
            private Rational m_shortBase;
            private Rational m_leftLeg;
            private Rational m_rightLeg;

            public MathTrapezoidSample(Rational longbase, Rational
shortbase, Rational leftLeg, Rational rightLeg)
            {
                    m_longBase = Math.Abs(longbase);
                    m_shortBase = Math.Abs(shortbase);
                    m_leftLeg = Math.Abs(leftLeg);
                    m_rightLeg = Math.Abs(rightLeg);
            }

            private Rational GetRightSmallBase()
            {
                    return (Math.Pow(m_rightLeg,2.0) -
Math.Pow(m_leftLeg,2.0) + Math.Pow(m_longBase,2.0) +
Math.Pow(m_shortBase,2.0) - 2* m_shortBase * m_longBase)/
(2*(m_longBase - m_shortBase));
```

```
		}

		public Rational GetHeight()
		{
			Rational x = GetRightSmallBase();
			return Math.Sqrt(Math.Pow(m_rightLeg,2.0) -
Math.Pow(x,2.0));
		}

		public Rational GetSquare()
		{
			return GetHeight() * m_longBase / 2.0;
		}

		public Rational GetLeftBaseRadianAngle()
		{
			Rational sinX = GetHeight()/m_leftLeg;
			return Math.Round(Math.Asin(sinX),2);
		}

		public Rational GetRightBaseRadianAngle()
		{

			Rational x = GetRightSmallBase();
			Rational cosX = (Math.Pow(m_rightLeg,2.0) +
Math.Pow(x,2.0) - Math.Pow(GetHeight(),2.0))/(2*x*m_rightLeg);
			return Math.Round(Math.Acos(cosX),2);
		}

		public Rational GetLeftBaseDegreeAngle()
		{
			Rational x = GetLeftBaseRadianAngle() * 180/ Math.PI;
			return Math.Round(x,2);
		}

		public Rational GetRightBaseDegreeAngle()
```

```
        {
            Rational x = GetRightBaseRadianAngle() * 180/ Math.PI;
            return Math.Round(x,2);
        }

        static void Main(string[] args)
        {
            MathTrapezoidSample trpz = new
MathTrapezoidSample(20.0, 10.0, 8.0, 6.0);
            Console.WriteLine("The trapezoid's bases are 20.0 and
10.0, the trapezoid's legs are 8.0 and 6.0");
            Rational h = trpz.GetHeight();
            Console.WriteLine("Trapezoid height is: " +
h.ToString());
            Rational dxR = trpz.GetLeftBaseRadianAngle();
            Console.WriteLine("Trapezoid left base angle is: " +
dxR.ToString() + " Radians");
            Rational dyR = trpz.GetRightBaseRadianAngle();
            Console.WriteLine("Trapezoid right base angle is: " +
dyR.ToString() + " Radians");
            Rational dxD = trpz.GetLeftBaseDegreeAngle();
            Console.WriteLine("Trapezoid left base angle is: " +
dxD.ToString() + " Degrees");
            Rational dyD = trpz.GetRightBaseDegreeAngle();
            Console.WriteLine("Trapezoid left base angle is: " +
dyD.ToString() + " Degrees");
        }
    }
}
```

注釈

戻り値に 180/Math.PI を乗算する事でラジアンから度に変換できます。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Asin(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
サインが指定数となる角度を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Asin(WS.Theia.ExtremelyPrecise.Rational sin);
```

パラメーター
sin　Rational
サインを表す数。-1 以上 1 以下である必要があります。

戻り値
Rational
$-\pi/2 \leq \theta \leq \pi/2$ の、ラジアンで表した角度 θ。 または sin < -1 または sin > 1、あるいは sin が NaN と等しい場合は、NaN。

例

次の例では、Asin を使用して台形の内角を計算しています。

```
/// <summary>
/// The following class represents simple functionality of the trapezoid.
/// </summary>
using System;
using WS.Theia.ExtremelyPrecise;

namespace MathClassCS
{
    class MathTrapezoidSample
```

```
{
        private Rational m_longBase;
        private Rational m_shortBase;
        private Rational m_leftLeg;
        private Rational m_rightLeg;

        public MathTrapezoidSample(Rational longbase, Rational
shortbase, Rational leftLeg, Rational rightLeg)
        {
                m_longBase = Math.Abs(longbase);
                m_shortBase = Math.Abs(shortbase);
                m_leftLeg = Math.Abs(leftLeg);
                m_rightLeg = Math.Abs(rightLeg);
        }

        private Rational GetRightSmallBase()
        {
                return (Math.Pow(m_rightLeg,2.0) -
Math.Pow(m_leftLeg,2.0) + Math.Pow(m_longBase,2.0) +
Math.Pow(m_shortBase,2.0) - 2* m_shortBase * m_longBase)/
(2*(m_longBase - m_shortBase));
        }

        public Rational GetHeight()
        {
                Rational x = GetRightSmallBase();
                return Math.Sqrt(Math.Pow(m_rightLeg,2.0) -
Math.Pow(x,2.0));
        }

        public Rational GetSquare()
        {
                return GetHeight() * m_longBase / 2.0;
        }

        public Rational GetLeftBaseRadianAngle()
```

```
        {
                Rational sinX = GetHeight()/m_leftLeg;
                return Math.Round(Math.Asin(sinX),2);
        }

        public Rational GetRightBaseRadianAngle()
        {
                Rational x = GetRightSmallBase();
                Rational cosX = (Math.Pow(m_rightLeg,2.0) +
Math.Pow(x,2.0) - Math.Pow(GetHeight(),2.0))/(2*x*m_rightLeg);
                return Math.Round(Math.Acos(cosX),2);
        }

        public Rational GetLeftBaseDegreeAngle()
        {
                Rational x = GetLeftBaseRadianAngle() * 180/ Math.PI;
                return Math.Round(x,2);
        }

        public Rational GetRightBaseDegreeAngle()
        {
                Rational x = GetRightBaseRadianAngle() * 180/ Math.PI;
                return Math.Round(x,2);
        }

        static void Main(string[] args)
        {
                MathTrapezoidSample trpz = new
MathTrapezoidSample(20.0, 10.0, 8.0, 6.0);
                Console.WriteLine("The trapezoid's bases are 20.0 and
10.0, the trapezoid's legs are 8.0 and 6.0");
                Rational h = trpz.GetHeight();
                Console.WriteLine("Trapezoid height is: " +
h.ToString());
                Rational dxR = trpz.GetLeftBaseRadianAngle();
```

```
                    Console.WriteLine("Trapezoid left base angle is: " +
dxR.ToString() + " Radians");
                    Rational dyR = trpz.GetRightBaseRadianAngle();
                    Console.WriteLine("Trapezoid right base angle is: " +
dyR.ToString() + " Radians");
                    Rational dxD = trpz.GetLeftBaseDegreeAngle();
                    Console.WriteLine("Trapezoid left base angle is: " +
dxD.ToString() + " Degrees");
                    Rational dyD = trpz.GetRightBaseDegreeAngle();
                    Console.WriteLine("Trapezoid left base angle is: " +
dyD.ToString() + " Degrees");
                }
    }
}
```

注釈

戻り値が正の場合、x 軸のプラス方向から反時計回りの角度を示します。戻り値が負の場合、x 軸のプラス方向から時計回りの角度を示します。戻り値に 180/Math.PI を乗算する事でラジアンから度に変換できます。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Atan(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
タンジェントが指定数となる角度を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Atan(WS.Theia.ExtremelyPrecise.Rational tan);
```

パラメーター
tan　Rational
タンジェントを表す数。

戻り値
Rational
$-\pi/2 \leq \theta \leq \pi/2$ の、ラジアンで表した角度 θ。
tan が NaN の場合、NaN、tan が NegativeInfinity の場合、-PI/2、tan が PositiveInfinity の場合、PI/2 になります。

例

次の例では、値のアークタンジェントを計算し、コンソールに表示する方法を示します。

```
// This example demonstrates Math.Atan()
//                           Math.Atan2()
//                           Math.Tan()
using System;
using WS.Theia.ExtremelyPrecise;

class Sample
{
```

```
        public static void Main()
        {
        Rational x = 1.0;
        Rational y = 2.0;
        Rational angle;
        Rational radians;
        Rational result;

// Calculate the tangent of 30 degrees.
        angle = 30;
        radians = angle * (Math.PI/180);
        result = Math.Tan(radians);
        Console.WriteLine("The tangent of 30 degrees is {0}.", result);

// Calculate the arctangent of the previous tangent.
        radians = Math.Atan(result);
        angle = radians * (180/Math.PI);
        Console.WriteLine("The previous tangent is equivalent to {0} degrees.",
angle);

// Calculate the arctangent of an angle.
        String line1 = "{0}The arctangent of the angle formed by the x-axis and ";
        String line2 = "a vector to point ({0},{1}) is {2}, ";
        String line3 = "which is equivalent to {0} degrees.";

        radians = Math.Atan2(y, x);
        angle = radians * (180/Math.PI);

        Console.WriteLine(line1, Environment.NewLine);
        Console.WriteLine(line2, x, y, radians);
        Console.WriteLine(line3, angle);
        }
}
/*
This example produces the following results:
```

```
The tangent of 30 degrees is 0.577350269189626.
The previous tangent is equivalent to 30 degrees.

The arctangent of the angle formed by the x-axis and
a vector to point (1,2) is 1.10714871779409,
which is equivalent to 63.434948822922 degrees.
*/
```

注釈

戻り値が正の場合、x 軸のプラス方向から反時計回りの角度を示します。戻り値が負の場合、x 軸のプラス方向から時計回りの角度を示します。戻り値に 180/Math.PI を乗算する事でラジアンから度に変換できます。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Math.Atan2(Rational,Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
タンジェントが 2 つの指定された数の商である角度を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Atan2(WS.Theia.ExtremelyPrecise.Rational yCoordinates,
WS.Theia.ExtremelyPrecise.Rational xCoordinates);
```

パラメーター
yCoordinates　Rational
点の Y 座標。
xCoordinates　Rational
点の X 座標。

戻り値
Rational
$-\pi/2 \leq \theta \leq \pi/2$ の、ラジアンで表した角度 θ。
tan が NaN の場合、NaN、tan が NegativeInfinity の場合、-PI/2、tan が PositiveInfinity の場合、PI/2 になります。

例

次の例では、座標からベクトルを算出し、ベクトルの角度からアークタンジェントを計算し、コンソールに表示する方法を示します。

```
// This example demonstrates Math.Atan()
//                           Math.Atan2()
//                           Math.Tan()
using System;
```

```
using WS.Theia.ExtremelyPrecise;

class Sample
{
	public static void Main()
	{
	Rational x = 1.0;
	Rational y = 2.0;
	Rational angle;
	Rational radians;
	Rational result;

// Calculate the tangent of 30 degrees.
	angle = 30;
	radians = angle * (Math.PI/180);
	result = Math.Tan(radians);
	Console.WriteLine("The tangent of 30 degrees is {0}.", result);

// Calculate the arctangent of the previous tangent.
	radians = Math.Atan(result);
	angle = radians * (180/Math.PI);
	Console.WriteLine("The previous tangent is equivalent to {0} degrees.",
angle);

// Calculate the arctangent of an angle.
	String line1 = "{0}The arctangent of the angle formed by the x-axis and ";
	String line2 = "a vector to point ({0},{1}) is {2}, ";
	String line3 = "which is equivalent to {0} degrees.";

	radians = Math.Atan2(y, x);
	angle = radians * (180/Math.PI);

	Console.WriteLine(line1, Environment.NewLine);
	Console.WriteLine(line2, x, y, radians);
	Console.WriteLine(line3, angle);
	}
```

```
}
/*
This example produces the following results:

The tangent of 30 degrees is 0.577350269189626.
The previous tangent is equivalent to 30 degrees.

The arctangent of the angle formed by the x-axis and
a vector to point (1,2) is 1.10714871779409,
which is equivalent to 63.434948822922 degrees.
*/
```

注釈

戻り値は、原点(0,0)からポイント(x,y)で終了するベクトルと x 軸のプラス方向で形成されるデカルト角度です。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Math.Ceiling(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した Rational 以上の数のうち、最小の整数値を返します。

---

public static WS.Theia.ExtremelyPrecise.Rational Ceiling
(WS.Theia.ExtremelyPrecise.Rational value);

パラメーター
value　Rational
Ratioan 型の数値。

戻り値
Rational
value 以上の最小の整数値。このメソッドは整数型ではなく Rational 型で結果を返します。
value が NaN、NegativeInfinity、PositiveInfinity のいずれかに等しい場合は、その値が返されます。

例

次の例では Math.Celing(Rational)メソッドと、Math.Floor(Rational)メソッドを比較しています。

```
Rational[] values = {7.03m, 7.64m, 0.12m, -0.12m, -7.1m, -7.6m};
Console.WriteLine("    Value               Ceiling             Floor¥n");
foreach (Rational value in values)
    Console.WriteLine("{0,7} {1,16} {2,14}",
                      value, Math.Ceiling(value), Math.Floor(value));
// The example displays the following output to the console:
//            Value             Ceiling             Floor
//
//             7.03                   8                   7
//             7.64                   8                   7
//             0.12                   1                   0
//            -0.12                   0                  -1
//             -7.1                  -7                  -8
//             -7.6                  -7                  -8
```

注釈

このメソッドの動作は、IEEE Standard 754 のセクション 4 に従います。このメソッドでは正の無限大方向に近づけるように丸めます。つまり、正の数では切り上げが行われますが、負の数では切り捨てが行われます。Math.Floor(Rational)メソッドはこのメソッドと異なり負の無限大方向に丸めます。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Cos(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された角度のコサインを返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Cos(WS.Theia.ExtremelyPrecise.Rational radian);
```

パラメーター
radian　Rational
ラジアンで表した角度。

戻り値
Rational
radian のコサイン。 radian が NaN、NegativeInfinity、PositiveInfinity のいずれかに等しい場合、このメソッドは NaN を返します。

例

次の例では、三角関数を算出しています。

```
// Example for the trigonometric Math.Sin( Rational )
// and Math.Cos( Rational ) methods.
using System;
using WS.Theia.ExtremelyPrecise;
class SinCos
{
    public static void Main()
    {
        Console.WriteLine(
```

```
            "This example of trigonometric " +
            "Math.Sin( Rational ) and Math.Cos( Rational )¥n" +
            "generates the following output.¥n" );
        Console.WriteLine(
            "Convert selected values for X to radians ¥n" +
            "and evaluate these trigonometric identities:" );
        Console.WriteLine( "     sin^2(X) + cos^2(X) == 1¥n" +
                           "    sin(2 * X) == 2 * sin(X) * cos(X)" );
        Console.WriteLine( "     cos(2 * X) == cos^2(X) - sin^2(X)" );

        UseSineCosine(15.0);
        UseSineCosine(30.0);
        UseSineCosine(45.0);

        Console.WriteLine(
            "¥nConvert selected values for X and Y to radians ¥n" +
            "and evaluate these trigonometric identities:" );
        Console.WriteLine( "     sin(X + Y) == sin(X) * cos(Y) + cos(X) *
sin(Y)" );
        Console.WriteLine( "     cos(X + Y) == cos(X) * cos(Y) - sin(X) *
sin(Y)" );
        UseTwoAngles(15.0, 30.0);
        UseTwoAngles(30.0, 45.0);
    }
    // Evaluate trigonometric identities with a given angle.
    static void UseSineCosine(Rational degrees)
    {
        Rational angle    = Math.PI * degrees / 180.0;
        Rational sinAngle = Math.Sin(angle);
        Rational cosAngle = Math.Cos(angle);

        // Evaluate sin^2(X) + cos^2(X) == 1.
        Console.WriteLine(
            "¥n                                  Math.Sin({0} deg) ==
{1:E16}¥n" +
            "                                Math.Cos({0} deg) == {2:E16}",
```

```
            degrees, Math.Sin(angle), Math.Cos(angle) );
        Console.WriteLine(
            "(Math.Sin({0} deg))^2 + (Math.Cos({0} deg))^2 == {1:E16}",
            degrees, sinAngle * sinAngle + cosAngle * cosAngle );

        // Evaluate sin(2 * X) == 2 * sin(X) * cos(X).
        Console.WriteLine(
            "                                  Math.Sin({0} deg) == {1:E16}",
            2.0 * degrees, Math.Sin(2.0 * angle) );
        Console.WriteLine(
            "       2 * Math.Sin({0} deg) * Math.Cos({0} deg) == {1:E16}",
            degrees, 2.0 * sinAngle * cosAngle );

        // Evaluate cos(2 * X) == cos^2(X) - sin^2(X).
        Console.WriteLine(
            "                                  Math.Cos({0} deg) == {1:E16}",
            2.0 * degrees, Math.Cos(2.0 * angle) );
        Console.WriteLine(
            "(Math.Cos({0} deg))^2 - (Math.Sin({0} deg))^2 == {1:E16}",
            degrees, cosAngle * cosAngle - sinAngle * sinAngle );
    }
    // Evaluate trigonometric identities that are functions of two angles.
    static void UseTwoAngles(Rational degreesX, Rational degreesY)
    {
        Rational    angleX    = Math.PI * degreesX / 180.0;
        Rational    angleY    = Math.PI * degreesY / 180.0;

        // Evaluate sin(X + Y) == sin(X) * cos(Y) + cos(X) * sin(Y).
        Console.WriteLine(

            "¥n            Math.Sin({0} deg) * Math.Cos({1} deg) +¥n" +
            "          Math.Cos({0} deg) * Math.Sin({1} deg) == {2:E16}",
            degreesX, degreesY, Math.Sin(angleX) * Math.Cos(angleY) +
            Math.Cos(angleX) * Math.Sin(angleY));
        Console.WriteLine(
            "                                  Math.Sin({0} deg) == {1:E16}",
```

```
                degreesX + degreesY, Math.Sin(angleX + angleY));

            // Evaluate cos(X + Y) == cos(X) * cos(Y) - sin(X) * sin(Y).
            Console.WriteLine(
                "            Math.Cos({0} deg) * Math.Cos({1} deg) -¥n" +
                "            Math.Sin({0} deg) * Math.Sin({1} deg) == {2:E16}",
                degreesX, degreesY, Math.Cos(angleX) * Math.Cos(angleY) -
                Math.Sin(angleX) * Math.Sin(angleY));
            Console.WriteLine(
                "                                    Math.Cos({0} deg) == {1:E16}",
                degreesX + degreesY, Math.Cos(angleX + angleY));
        }
}
/*
This example of trigonometric Math.Sin( Rational ) and Math.Cos( Rational )
generates the following output.

Convert selected values for X to radians
and evaluate these trigonometric identities:
    sin^2(X) + cos^2(X) == 1
    sin(2 * X) == 2 * sin(X) * cos(X)
    cos(2 * X) == cos^2(X) - sin^2(X)
                                Math.Sin(15 deg) == 2.5881904510252074E-
001
                                Math.Cos(15 deg) == 9.6592582628906831E-
001
(Math.Sin(15 deg))^2 + (Math.Cos(15 deg))^2 ==
1.0000000000000000E+000
                                Math.Sin(30 deg) == 4.9999999999999994E-
001
    2 * Math.Sin(15 deg) * Math.Cos(15 deg) == 4.9999999999999994E-001
                                Math.Cos(30 deg) == 8.6602540378443871E-
001
(Math.Cos(15 deg))^2 - (Math.Sin(15 deg))^2 == 8.6602540378443871E-001

                                Math.Sin(30 deg) == 4.9999999999999994E-
```

001
                    Math.Cos(30 deg) == 8.6602540378443871E-
001
(Math.Sin(30 deg))^2 + (Math.Cos(30 deg))^2 ==
1.0000000000000000E+000
                    Math.Sin(60 deg) == 8.6602540378443860E-
001
    2 * Math.Sin(30 deg) * Math.Cos(30 deg) == 8.6602540378443860E-001
                    Math.Cos(60 deg) == 5.0000000000000011E-
001
(Math.Cos(30 deg))^2 - (Math.Sin(30 deg))^2 == 5.0000000000000022E-001

                    Math.Sin(45 deg) == 7.0710678118654746E-
001
                    Math.Cos(45 deg) == 7.0710678118654757E-
001
(Math.Sin(45 deg))^2 + (Math.Cos(45 deg))^2 ==
1.0000000000000000E+000
                    Math.Sin(90 deg) ==
1.0000000000000000E+000
    2 * Math.Sin(45 deg) * Math.Cos(45 deg) == 1.0000000000000000E+000
                    Math.Cos(90 deg) == 6.1230317691118863E-
017
(Math.Cos(45 deg))^2 - (Math.Sin(45 deg))^2 == 2.2204460492503131E-016

Convert selected values for X and Y to radians
and evaluate these trigonometric identities:
  sin(X + Y) == sin(X) * cos(Y) + cos(X) * sin(Y)
  cos(X + Y) == cos(X) * cos(Y) - sin(X) * sin(Y)

      Math.Sin(15 deg) * Math.Cos(30 deg) +
      Math.Cos(15 deg) * Math.Sin(30 deg) == 7.0710678118654746E-001
                    Math.Sin(45 deg) == 7.0710678118654746E-
001
      Math.Cos(15 deg) * Math.Cos(30 deg) -
      Math.Sin(15 deg) * Math.Sin(30 deg) == 7.0710678118654757E-001

```
                                Math.Cos(45 deg) == 7.0710678118654757E-
001

        Math.Sin(30 deg) * Math.Cos(45 deg) +
        Math.Cos(30 deg) * Math.Sin(45 deg) == 9.6592582628906831E-001
                                Math.Sin(75 deg) == 9.6592582628906820E-
001
        Math.Cos(30 deg) * Math.Cos(45 deg) -
        Math.Sin(30 deg) * Math.Sin(45 deg) == 2.5881904510252085E-001
                                Math.Cos(75 deg) == 2.5881904510252096E-
001
*/
```

注釈

引数に入力する角度はラジアン単位である必要があります。角度に Math.PI/180 を乗算する事でラジアン単位に変換できます。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Math.Cosh(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された角度のハイパーボリックコサインを返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Cosh(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
radian　Rational
ラジアンで表した角度。

戻り値
Rational
radian のハイパーボリックコサイン。 radian が NegativeInfinity、PositiveInfinity のいずれかに等しい場合、PositiveInfinity を返します。Radian が NaN に等しい場合は、このメソッドは NaN を返します。

例

次の例では、Cosh の結果を表示しています。

```
// Example for the hyperbolic Math.Sinh( Rational )
// and Math.Cosh( Rational ) methods.
using System;
using WS.Theia.ExtremelyPrecise;

class SinhCosh
{
    public static void Main()
```

```
{
    Console.WriteLine(
        "This example of hyperbolic Math.Sinh( Rational ) " +
        "and Math.Cosh( Rational )¥n" +
        "generates the following output.¥n" );
    Console.WriteLine(
        "Evaluate these hyperbolic identities " +
        "with selected values for X:" );
    Console.WriteLine(
        "     cosh^2(X) - sinh^2(X) == 1¥n" +
        "     sinh(2 * X) == 2 * sinh(X) * cosh(X)" );
    Console.WriteLine( "     cosh(2 * X) == cosh^2(X) + sinh^2(X)" );

    UseSinhCosh(0.1);
    UseSinhCosh(1.2);
    UseSinhCosh(4.9);

    Console.WriteLine(
        "¥nEvaluate these hyperbolic identities " +
        "with selected values for X and Y:" );
    Console.WriteLine(
        "     sinh(X + Y) == sinh(X) * cosh(Y) + cosh(X) * sinh(Y)" );
    Console.WriteLine(
        "     cosh(X + Y) == cosh(X) * cosh(Y) + sinh(X) * sinh(Y)" );

    UseTwoArgs(0.1, 1.2);
    UseTwoArgs(1.2, 4.9);
}

// Evaluate hyperbolic identities with a given argument.
static void UseSinhCosh(Rational arg)
{
    Rational sinhArg = Math.Sinh(arg);
    Rational coshArg = Math.Cosh(arg);

    // Evaluate cosh^2(X) - sinh^2(X) == 1.
```

```
        Console.WriteLine(
            "¥n                                  Math.Sinh({0}) == {1:E16}¥n" +
            "                                  Math.Cosh({0}) == {2:E16}",
            arg, Math.Sinh(arg), Math.Cosh(arg) );
        Console.WriteLine(
            "(Math.Cosh({0}))^2 - (Math.Sinh({0}))^2 == {1:E16}",
            arg, coshArg * coshArg - sinhArg * sinhArg );
        // Evaluate sinh(2 * X) == 2 * sinh(X) * cosh(X).
        Console.WriteLine(
            "                                  Math.Sinh({0}) == {1:E16}",
            2.0 * arg, Math.Sinh(2.0 * arg) );
        Console.WriteLine(
            "      2 * Math.Sinh({0}) * Math.Cosh({0}) == {1:E16}",
            arg, 2.0 * sinhArg * coshArg );

        // Evaluate cosh(2 * X) == cosh^2(X) + sinh^2(X).
        Console.WriteLine(
            "                                  Math.Cosh({0}) == {1:E16}",
            2.0 * arg, Math.Cosh(2.0 * arg) );
        Console.WriteLine(
            "(Math.Cosh({0}))^2 + (Math.Sinh({0}))^2 == {1:E16}",
            arg, coshArg * coshArg + sinhArg * sinhArg );
    }
    // Evaluate hyperbolic identities that are functions of two arguments.
    static void UseTwoArgs(Rational argX, Rational argY)
    {
        // Evaluate sinh(X + Y) == sinh(X) * cosh(Y) + cosh(X) * sinh(Y).
        Console.WriteLine(
            "¥n            Math.Sinh({0}) * Math.Cosh({1}) +¥n" +
            "            Math.Cosh({0}) * Math.Sinh({1}) == {2:E16}",
            argX, argY, Math.Sinh(argX) * Math.Cosh(argY) +
            Math.Cosh(argX) * Math.Sinh(argY));
        Console.WriteLine(
            "                                  Math.Sinh({0}) == {1:E16}",
            argX + argY, Math.Sinh(argX + argY));
```

```
            // Evaluate cosh(X + Y) == cosh(X) * cosh(Y) + sinh(X) * sinh(Y).
            Console.WriteLine(
                "              Math.Cosh({0}) * Math.Cosh({1}) +¥n" +
                "              Math.Sinh({0}) * Math.Sinh({1}) == {2:E16}",
                argX, argY, Math.Cosh(argX) * Math.Cosh(argY) +
                Math.Sinh(argX) * Math.Sinh(argY));
            Console.WriteLine(
                "                                    Math.Cosh({0}) == {1:E16}",
                argX + argY, Math.Cosh(argX + argY));
        }
}

/*
This example of hyperbolic Math.Sinh( Rational ) and Math.Cosh( Rational )
generates the following output.

Evaluate these hyperbolic identities with selected values for X:
    cosh^2(X) - sinh^2(X) == 1
    sinh(2 * X) == 2 * sinh(X) * cosh(X)
    cosh(2 * X) == cosh^2(X) + sinh^2(X)

                                Math.Sinh(0.1) == 1.0016675001984403E-001
                                Math.Cosh(0.1) == 1.0050041680558035E+000
(Math.Cosh(0.1))^2 - (Math.Sinh(0.1))^2 == 9.9999999999999989E-001
                                Math.Sinh(0.2) == 2.0133600254109399E-001
    2 * Math.Sinh(0.1) * Math.Cosh(0.1) == 2.0133600254109396E-001
                                Math.Cosh(0.2) == 1.0200667556190759E+000
(Math.Cosh(0.1))^2 + (Math.Sinh(0.1))^2 == 1.0200667556190757E+000

                                Math.Sinh(1.2) == 1.5094613554121725E+000
                                Math.Cosh(1.2) == 1.8106555673243747E+000
(Math.Cosh(1.2))^2 - (Math.Sinh(1.2))^2 == 1.0000000000000000E+000
                                Math.Sinh(2.4) == 5.4662292136760939E+000
    2 * Math.Sinh(1.2) * Math.Cosh(1.2) == 5.4662292136760939E+000
                                Math.Cosh(2.4) == 5.5569471669655064E+000
(Math.Cosh(1.2))^2 + (Math.Sinh(1.2))^2 == 5.5569471669655064E+000
```

```
                                Math.Sinh(4.9) == 6.7141166550932297E+001
                                Math.Cosh(4.9) == 6.7148613134003227E+001
(Math.Cosh(4.9))^2 - (Math.Sinh(4.9))^2 == 1.0000000000000000E+000
                                Math.Sinh(9.8) == 9.0168724361884615E+003
    2 * Math.Sinh(4.9) * Math.Cosh(4.9) == 9.0168724361884615E+003
                                Math.Cosh(9.8) == 9.0168724916400624E+003
(Math.Cosh(4.9))^2 + (Math.Sinh(4.9))^2 == 9.0168724916400606E+003

Evaluate these hyperbolic identities with selected values for X and Y:
    sinh(X + Y) == sinh(X) * cosh(Y) + cosh(X) * sinh(Y)
    cosh(X + Y) == cosh(X) * cosh(Y) + sinh(X) * sinh(Y)

        Math.Sinh(0.1) * Math.Cosh(1.2) +
        Math.Cosh(0.1) * Math.Sinh(1.2) == 1.6983824372926155E+000
                                Math.Sinh(1.3) == 1.6983824372926160E+000
        Math.Cosh(0.1) * Math.Cosh(1.2) +
        Math.Sinh(0.1) * Math.Sinh(1.2) == 1.9709142303266281E+000
                                Math.Cosh(1.3) == 1.9709142303266285E+000

        Math.Sinh(1.2) * Math.Cosh(4.9) +
        Math.Cosh(1.2) * Math.Sinh(4.9) == 2.2292776360739879E+002
                                Math.Sinh(6.1) == 2.2292776360739885E+002
        Math.Cosh(1.2) * Math.Cosh(4.9) +
        Math.Sinh(1.2) * Math.Sinh(4.9) == 2.2293000647511826E+002
                                Math.Cosh(6.1) == 2.2293000647511832E+002
*/
```

注釈

引数に入力する角度はラジアン単位である必要があります。角度に Math.PI/180 を乗算する事でラジアン単位に変換できます。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.DivRem(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの数値の商を計算し、出力パラメーターの剰余を返します。

```
public static (WS.Theia.ExtremelyPrecise.Rational Quotient,
WS.Theia.ExtremelyPrecise.Rational Remainder)
DivRem(WS.Theia.ExtremelyPrecise.Rational dividend,
WS.Theia.ExtremelyPrecise.Rational divisor);
```

パラメーター
dividend　Rational
被除数。
divisor　Rational
除数。

戻り値
Quotient　Rational
指定した数値の商。剰余。
Remainder　　Rational
指定した数値の剰余。

例

次の例では DivRem(Rational,Rational)メソッドを使い商と剰余を求めています。

```
using System;
using WS.Theia.ExtremelyPrecise;

public class Example
```

```
{
    public static void Main()
    {
        // Define several positive and negative dividends.
        Rational[] dividends = { Int32.MaxValue, 13952, 0, -14032,
                                                 Int32.MinValue };
        // Define one positive and one negative divisor.
        Rational [] divisors = { 2000, -2000 };

        foreach (Rational divisor in divisors)
        {
            foreach (Rational dividend in dividends)
            {
                Rational remainder;
                Rational quotient;
                (quotient, remainder)=Math.DivRem(dividend, divisor);
                Console.WriteLine(@"{0:N0} ¥ {1:N0} = {2:N0}, remainder
{3:N0}",
                                  dividend, divisor, quotient, remainder);
            }
        }
    }
}
// The example displays the following output:
//       2,147,483,647 ¥ 2,000 = 1,073,741, remainder 1,647
//       13,952 ¥ 2,000 = 6, remainder 1,952
//       0 ¥ 2,000 = 0, remainder 0
//       -14,032 ¥ 2,000 = -7, remainder -32
//       -2,147,483,648 ¥ 2,000 = -1,073,741, remainder -1,648
//       2,147,483,647 ¥ -2,000 = -1,073,741, remainder 1,647
//       13,952 ¥ -2,000 = -6, remainder 1,952
//       0 ¥ -2,000 = 0, remainder 0
//       -14,032 ¥ -2,000 = 7, remainder -32
//       -2,147,483,648 ¥ -2,000 = 1,073,741, remainder -1,648
```

注釈

剰余のみが欲しい場合は、剰余演算子を使用してください。
また IEEERemainder(Rational,Rational)もご覧ください。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Math.Exp(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した値で e を累乗した値を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Exp(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
累乗を指定する数値。

戻り値
Rational
数値 e を value で累乗した値。value が NaN または PositiveInfinity のいずれかに等しい場合は、その値が返されます。 value が NegativeInfinity に等しい場合は、0 が返されます。

例

次の例では Exp(Rational)メソッドを使って E を累乗した結果を表示しています。

```
// Example for the Math.Exp( Rational ) method.
using System;
using WS.Theia.ExtremelyPrecise;

class ExpDemo
{
    public static void Main()
    {
```

```
        Console.WriteLine(
            "This example of Math.Exp( Rational ) " +
            "generates the following output.¥n" );
        Console.WriteLine(
            "Evaluate [e ^ ln(X) == ln(e ^ X) == X] " +
            "with selected values for X:" );

        UseLnExp(0.1);
        UseLnExp(1.2);
        UseLnExp(4.9);
        UseLnExp(9.9);

        Console.WriteLine(
            "¥nEvaluate these identities with " +
            "selected values for X and Y:" );
        Console.WriteLine( "     (e ^ X) * (e ^ Y) == e ^ (X + Y)" );
        Console.WriteLine( "     (e ^ X) ^ Y == e ^ (X * Y)" );
        Console.WriteLine( "     X ^ Y == e ^ (Y * ln(X))" );

        UseTwoArgs(0.1, 1.2);
        UseTwoArgs(1.2, 4.9);
        UseTwoArgs(4.9, 9.9);
    }

    // Evaluate logarithmic/exponential identity with a given argument.
    static void UseLnExp(Rational arg)
    {
        // Evaluate e ^ ln(X) == ln(e ^ X) == X.
        Console.WriteLine(
            "¥n          Math.Exp(Math.Log({0})) == {1:E16}¥n" +
            "          Math.Log(Math.Exp({0})) == {2:E16}",
            arg, Math.Exp(Math.Log(arg)), Math.Log(Math.Exp(arg)) );
    }

    // Evaluate exponential identities that are functions of two arguments.
    static void UseTwoArgs(Rational argX, Rational argY)
```

```
    {
        // Evaluate (e ^ X) * (e ^ Y) == e ^ (X + Y).
        Console.WriteLine(
            "¥nMath.Exp({0}) * Math.Exp({1}) == {2:E16}" +
            "¥n                 Math.Exp({0} + {1}) == {3:E16}",
            argX, argY, Math.Exp(argX) * Math.Exp(argY),
            Math.Exp(argX + argY) );

        // Evaluate (e ^ X) ^ Y == e ^ (X * Y).
        Console.WriteLine(
            " Math.Pow(Math.Exp({0}), {1}) == {2:E16}" +
            "¥n                 Math.Exp({0} * {1}) == {3:E16}",
            argX, argY, Math.Pow(Math.Exp(argX), argY),
            Math.Exp(argX * argY) );

        // Evaluate X ^ Y == e ^ (Y * ln(X)).
        Console.WriteLine(
            "                 Math.Pow({0}, {1}) == {2:E16}" +
            "¥nMath.Exp({1} * Math.Log({0})) == {3:E16}",
            argX, argY, Math.Pow(argX, argY),
            Math.Exp(argY * Math.Log(argX)) );
    }
}

/*
This example of Math.Exp( Rational ) generates the following output.

Evaluate [e ^ ln(X) == ln(e ^ X) == X] with selected values for X:

      Math.Exp(Math.Log(0.1)) == 1.0000000000000001E-001
      Math.Log(Math.Exp(0.1)) == 1.0000000000000008E-001

      Math.Exp(Math.Log(1.2)) == 1.2000000000000000E+000
      Math.Log(Math.Exp(1.2)) == 1.2000000000000000E+000
```

```
        Math.Exp(Math.Log(4.9)) == 4.9000000000000012E+000
        Math.Log(Math.Exp(4.9)) == 4.9000000000000004E+000
        Math.Exp(Math.Log(9.9)) == 9.9000000000000004E+000
        Math.Log(Math.Exp(9.9)) == 9.9000000000000004E+000

Evaluate these identities with selected values for X and Y:
    (e ^ X) * (e ^ Y) == e ^ (X + Y)
    (e ^ X) ^ Y == e ^ (X * Y)
    X ^ Y == e ^ (Y * ln(X))

Math.Exp(0.1) * Math.Exp(1.2) == 3.6692966676192444E+000
            Math.Exp(0.1 + 1.2) == 3.6692966676192444E+000
 Math.Pow(Math.Exp(0.1), 1.2) == 1.1274968515793757E+000
            Math.Exp(0.1 * 1.2) == 1.1274968515793757E+000
             Math.Pow(0.1, 1.2) == 6.3095734448019331E-002
Math.Exp(1.2 * Math.Log(0.1)) == 6.3095734448019344E-002

Math.Exp(1.2) * Math.Exp(4.9) == 4.4585777008251705E+002
            Math.Exp(1.2 + 4.9) == 4.4585777008251716E+002
 Math.Pow(Math.Exp(1.2), 4.9) == 3.5780924170885260E+002
            Math.Exp(1.2 * 4.9) == 3.5780924170885277E+002
             Math.Pow(1.2, 4.9) == 2.4433636334442981E+000
Math.Exp(4.9 * Math.Log(1.2)) == 2.4433636334442981E+000

Math.Exp(4.9) * Math.Exp(9.9) == 2.6764450551890982E+006
            Math.Exp(4.9 + 9.9) == 2.6764450551891015E+006
 Math.Pow(Math.Exp(4.9), 9.9) == 1.1684908531676833E+021
            Math.Exp(4.9 * 9.9) == 1.1684908531676829E+021
             Math.Pow(4.9, 9.9) == 6.8067718210957060E+006
Math.Exp(9.9 * Math.Log(4.9)) == 6.8067718210956985E+006
*/
```

注釈

e は約 2.71828 の数学定数です。Exp(Rational)メソッドは e を指定した数値で累乗します。Log(Rational)メソッドとは逆の動作になります。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Math.Floor(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した Rational 以下の数のうち、最大の整数値を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Floor(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター

value　Rational
Ratioan 型の数値。

戻り値

Rational
value 以下の最大の整数値。このメソッドは整数型ではなく Rational 型で結果を返します。value が NaN、NegativeInfinity、PositiveInfinity のいずれかに等しい場合は、その値が返されます。

例

次の例では Math.Floor(Rational)メソッドと、Math.Celing(Rational)メソッドを比較しています。

```
Rational[] values = {7.03m, 7.64m, 0.12m, -0.12m, -7.1m, -7.6m};
Console.WriteLine("    Value               Ceiling             Floor¥n");
foreach (Rational value in values)
    Console.WriteLine("{0,7} {1,16} {2,14}",
                      value, Math.Ceiling(value), Math.Floor(value));
// The example displays the following output to the console:
//            Value              Ceiling             Floor
//
//             7.03                    8                 7
//             7.64                    8                 7
//             0.12                    1                 0
//            -0.12                   -0                -1
//             -7.1                   -7                -8
//             -7.6                   -7                -8
```

注釈

このメソッドの動作は、IEEE Standard 754 のセクション 4 に従います。このメソッドでは正の無限大方向に近づけるように丸めます。つまり、正の数では切り捨てが行われますが、負の数では切り上げが行われます。Math.Celing(Rational)メソッドはこのメソッドと異なり正の無限大方向に丸めます。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.IEEERemainder(Rational,Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した数を別の指定数で除算した結果の剰余を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Floor(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
dividend　Rational
被除数。
Divisor　Rational
除数。

戻り値
Rational
dividend - (divisor Q) に等しい数値。Q は dividend / divisor の商を丸めた近似整数を示します。dividend / divisor が 2 つの整数の中間に位置する場合は、偶数の整数が返されます。
dividend - (divisor Q) がゼロのとき、dividend が正である場合は値 +0、dividend が負である場合は -0 が返されます。
divisor = 0 の場合は、NaN が返されます。

例

次の例では IEEERemainder(Rational)メソッドと剰余演算子で返される値の比較をしています。

```
using System;
using WS.Theia.ExtremelyPrecise;

public class Example
{
    public static void Main()
    {
        Console.WriteLine($"{"IEEERemainder",35} {"Remainder
operator",20}");
        ShowRemainders(3, 2);
        ShowRemainders(4, 2);
        ShowRemainders(10, 3);
        ShowRemainders(11, 3);
        ShowRemainders(27, 4);
        ShowRemainders(28, 5);
        ShowRemainders(17.8, 4);
        ShowRemainders(17.8, 4.1);
        ShowRemainders(-16.3, 4.1);
        ShowRemainders(17.8, -4.1);
        ShowRemainders(-17.8, -4.1);
    }

    private static void ShowRemainders(double number1, double number2)
    {
        var formula = $"{number1} / {number2} = ";
        var ieeeRemainder = Math.IEEERemainder(number1, number2);
        var remainder = number1 % number2;
        Console.WriteLine($"{formula,-16} {ieeeRemainder,18}
{remainder,20}");
```

```
    }
}
// The example displays the following output:
//
//
//                          IEEERemainder     Remainder operator
// 3 / 2 =                             -1                      1
// 4 / 2 =                              0                      0
// 10 / 3 =                             1                      1
// 11 / 3 =                            -1                      2
// 27 / 4 =                            -1                      3
// 28 / 5 =                            -2                      3
// 17.8 / 4 =                         1.8                    1.8
// 17.8 / 4.1 =                       1.4                    1.4
// -16.3 / 4.1 =     0.0999999999999979                     -4
// 17.8 / -4.1 =                      1.4                    1.4
// -17.8 / -4.1 =                    -1.4                   -1.4
```

注釈

この操作は、ANSI/IEEE Std 754-1985; のセクション 5.1 で定義されている剰余演算に準拠しています。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Log Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した数値の対数を返します。

オーバーロード

Log(Rational)	指定した数の自然（底 e）対数を返します。
Log(Rational,Rational)	指定した数値の指定した底での対数を返します。

Log(Rational)

指定した数の自然（底 e）対数を返します。

---

public static WS.Theia.ExtremelyPrecise.Rational
Log(WS.Theia.ExtremelyPrecise.Rational value);

パラメーター
value　Rational
対数を求める対象の数値。

戻り値
Rational
次の表に示した値のいずれか

value パラメーター	戻り値
正	value の自然対数。つまり、ln value または log e value
0	NegativeInfinity
負	NaN
NaN	NaN
PositiveInfinity	PositiveInfinity

例

次の例は Log(Rational)メソッドの使用例です。

```
using System;
using WS.Theia.ExtremelyPrecise;
public class Example
{
    public static void Main()
    {
        Console.WriteLine("    Evaluate this identity with selected values for
X:");
        Console.WriteLine("                                        ln(x) = 1 /
log[X](B)");
        Console.WriteLine();

        double[] XArgs = { 1.2, 4.9, 9.9, 0.1 };

        foreach (double argX in XArgs)
        {
            // Find natural log of argX.
            Console.WriteLine("                          Math.Log({0}) =
{1:E16}",
                              argX, Math.Log(argX));

            // Evaluate 1 / log[X](e).
            Console.WriteLine("              1.0 / Math.Log(e, {0}) =
{1:E16}",
                              argX, 1.0 / Math.Log(Math.E, argX));
            Console.WriteLine();
        }
    }
}
// This example displays the following output:
//             Evaluate this identity with selected values for X:
//                                          ln(x) = 1 / log[X](B)
```

```
//
//                                  Math.Log(1.2) = 1.8232155679395459E-
001
//                       1.0 / Math.Log(e, 1.2) = 1.8232155679395459E-001
//
//                                  Math.Log(4.9) =
1.5892352051165810E+000
//                       1.0 / Math.Log(e, 4.9) = 1.5892352051165810E+000
//
//                                  Math.Log(9.9) =
2.2925347571405443E+000
//                       1.0 / Math.Log(e, 9.9) = 2.2925347571405443E+000
//
//                                  Math.Log(0.1) = -
2.3025850929940455E+000
//                       1.0 / Math.Log(e, 0.1) = -2.3025850929940455E+000
```

注釈

e は約 2.71828 の数学定数です。Log(Rational)メソッドはパラメーターの e の対数を算出します。Exp(Rational)メソッドとは逆の動作になります。

Log(Rational,Rational)

指定した数値の指定した底での対数を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Log(WS.Theia.ExtremelyPrecise.Rational value,
WS.Theia.ExtremelyPrecise.Rational newBase);
```

パラメーター
value　Rational
対数を求める対象の数値。
newBase　Rational
対数の底。

戻り値

Rational

次の表に示した値のいずれか（+Infinity は PositiveInfinity、-Infinity は NegativeInfinity、NaN は NaN をそれぞれ示しています。）

value	newBase	戻り値
value > 0	(0 < newBase < 1) - または -(newBase > 1)	lognewBase(a)
value < 0	(任意の値)	NaN
(任意の値)	newBase < 0	NaN
value != 1	newBase = 0	NaN
value != 1	newBase = +Infinity	NaN
value = NaN	(任意の値)	NaN
(任意の値)	newBase = NaN	NaN
(任意の値)	newBase = 1	NaN
value = 0	0 < newBase < 1	+Infinity
value = 0	newBase > 1	-Infinity
value = +無限大	0 <newBase< 1	-Infinity
value = +無限大	newBase> 1	+Infinity
value = 1	newBase = 0	0
value = 1	newBase = +Infinity	0

例

次の例は Log(Rational、Rational)メソッドの使用例です。

```
// Example for the Math.Log( Rational ) and Math.Log( Rational, Rational )
methods.
using System;
using WS.Theia.ExtremelyPrecise;

class LogDLogDD
{
    public static void Main()
    {
        Console.WriteLine(
            "This example of Math.Log( Rational ) and " +
```

```
            "Math.Log( Rational, Rational )¥n" +
            "generates the following output.¥n" );
        Console.WriteLine(
            "Evaluate these identities with " +
            "selected values for X and B (base):" );
        Console.WriteLine( "     log(B)[X] == 1 / log(X)[B]" );
        Console.WriteLine( "     log(B)[X] == ln[X] / ln[B]" );
        Console.WriteLine( "     log(B)[X] == log(B)[e] * ln[X]" );

        UseBaseAndArg(0.1, 1.2);
        UseBaseAndArg(1.2, 4.9);
        UseBaseAndArg(4.9, 9.9);
        UseBaseAndArg(9.9, 0.1);
    }
    // Evaluate logarithmic identities that are functions of two arguments.
    static void UseBaseAndArg(Rational argB, Rational argX)
    {
        // Evaluate log(B)[X] == 1 / log(X)[B].
        Console.WriteLine(
            "¥n                    Math.Log({1}, {0}) == {2:E16}" +
            "¥n          1.0 / Math.Log({0}, {1}) == {3:E16}",
            argB, argX, Math.Log(argX, argB),
            1.0 / Math.Log(argB, argX) );

        // Evaluate log(B)[X] == ln[X] / ln[B].
        Console.WriteLine(
            "       Math.Log({1}) / Math.Log({0}) == {2:E16}",
            argB, argX, Math.Log(argX) / Math.Log(argB) );

        // Evaluate log(B)[X] == log(B)[e] * ln[X].
        Console.WriteLine(
            "Math.Log(Math.E, {0}) * Math.Log({1}) == {2:E16}",
            argB, argX, Math.Log(Math.E, argB) * Math.Log(argX) );
    }
}
```

```
/*
This example of Math.Log( Rational ) and Math.Log( Rational, Rational )
generates the following output.
Evaluate these identities with selected values for X and B (base):
    log(B)[X] == 1 / log(X)[B]
    log(B)[X] == ln[X] / ln[B]
    log(B)[X] == log(B)[e] * ln[X]

                        Math.Log(1.2, 0.1) == -7.9181246047624818E-002
                  1.0 / Math.Log(0.1, 1.2) == -7.9181246047624818E-002
            Math.Log(1.2) / Math.Log(0.1) == -7.9181246047624818E-002

Math.Log(Math.E, 0.1) * Math.Log(1.2) == -7.9181246047624804E-002

                        Math.Log(4.9, 1.2) == 8.7166610085093179E+000
                  1.0 / Math.Log(1.2, 4.9) == 8.7166610085093161E+000
            Math.Log(4.9) / Math.Log(1.2) == 8.7166610085093179E+000
Math.Log(Math.E, 1.2) * Math.Log(4.9) == 8.7166610085093179E+000

                        Math.Log(9.9, 4.9) == 1.4425396251981288E+000
                  1.0 / Math.Log(4.9, 9.9) == 1.4425396251981288E+000
            Math.Log(9.9) / Math.Log(4.9) == 1.4425396251981288E+000
Math.Log(Math.E, 4.9) * Math.Log(9.9) == 1.4425396251981288E+000

                        Math.Log(0.1, 9.9) == -1.0043839404494075E+000
                  1.0 / Math.Log(9.9, 0.1) == -1.0043839404494075E+000
            Math.Log(0.1) / Math.Log(9.9) == -1.0043839404494075E+000
Math.Log(Math.E, 9.9) * Math.Log(0.1) == -1.0043839404494077E+000
*/
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Log10(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した数の底 10 の対数を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Log10(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター

value　Rational
対数を求める対象の数値。

戻り値

Rational
次の表に示した値のいずれか

value パラメーター	戻り値
正	value の自然対数。つまり、ln value または log e value
0	NegativeInfinity
負	NaN
NaN	NaN
PositiveInfinity	PositiveInfinity

例

次の例は Log10(Rational)メソッドの使用例です。

```
using System;
using WS.Theia.ExtremelyPrecise;

public class Example
```

```
{
    public static void Main()
    {
        Rational[] numbers = {-1, 0, .105, .5, .798, 1, 4, 6.9, 10, 50,
                              100, 500, 1000, Double.MaxValue};

        foreach (Rational number in numbers)
            Console.WriteLine("The base 10 log of {0} is {1}.",
                              number, Math.Log10(number));
    }
}
// The example dislays the following output:
//       The base 10 log of -1 is NaN.
//       The base 10 log of 0 is -Infinity.
//       The base 10 log of 0.105 is -0.978810700930062.
//       The base 10 log of 0.5 is -0.301029995663981.
//       The base 10 log of 0.798 is -0.0979971086492706.
//       The base 10 log of 1 is 0.
//       The base 10 log of 4 is 0.602059991327962.
//       The base 10 log of 6.9 is 0.838849090737255.
//       The base 10 log of 10 is 1.
//       The base 10 log of 50 is 1.69897000433602.
//       The base 10 log of 100 is 2.
//       The base 10 log of 500 is 2.69897000433602.
//       The base 10 log of 1000 is 3.
//       The base 10 log of 1.79769313486232E+308 is 308.254715559917.
```

注釈

パラメーターvalue の対数を求める際に 10 を底として指定します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Max(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational のうち、大きな方を返します。

---

public static WS.Theia.ExtremelyPrecise.Rational
Max(WS.Theia.ExtremelyPrecise.Rational val1,
WS.Theia.ExtremelyPrecise.Rational val2);

パラメーター

val1　Rational
比較する最初の数値。
val2　Rational
比較する 2 番目の数値。

戻り値

Rational
パラメーター val1 または val2 のいずれか大きい方。

例

次の例ではMax(Rational,Rational)メソッドを使用して変数に格納された大きい方の値を表示しています。

```
// This example demonstrates Math.Max()
using System;
using WS.Theia.ExtremelyPrecise;

class Sample
{
      public static void Main()
      {
      string str = "{0}: The greater of {1,3} and {2,3} is {3}.";
      string nl = Environment.NewLine;

      Rational        xByte1       = -100,       xByte2       = 51;

      Console.WriteLine("{0}Display the greater of two values:{0}", nl);
      Console.WriteLine(str, "Rational", xByte1, xByte2, Math.Max(xByte1,
xByte2));
      }
}
/*
This example produces the following results:

Display the greater of two values:

Rational: The greater of     -100 and    51 is 51.
*/
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Min(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational のうち、小さい方を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Min(WS.Theia.ExtremelyPrecise.Rational val1,
WS.Theia.ExtremelyPrecise.Rational val2);
```

パラメーター

val1　Rational
比較する最初の数値。
val2　Rational
比較する 2 番目の数値。

戻り値

Rational
パラメーター val1 または val2 のいずれか小さい方。

例

次の例では Min(Rational,Rational)メソッドを使用して変数に格納された小さい方の値を表示しています。

```
// This example demonstrates Math.Min()
using System;
using WS.Theia.ExtremelyPrecise;

class Sample
{
      public static void Main()
      {
      string str = "{0}: The greater of {1,3} and {2,3} is {3}.";
      string nl = Environment.NewLine;

      Rational       xByte1       = -100,       xByte2       = 51;

      Console.WriteLine("{0}Display the greater of two values:{0}", nl);
      Console.WriteLine(str, "Rational", xByte1, xByte2, Math.Min(xByte1,
xByte2));
      }
}
/*
This example produces the following results:

Display the greater of two values:

Rational: The greater of     -100 and    51 is -100.
*/
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Pow(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定の数値を指定した値で累乗した値を返します。

---

public static WS.Theia.ExtremelyPrecise.Rational
Pow(WS.Theia.ExtremelyPrecise.Rational ebase,
WS.Theia.ExtremelyPrecise.Rational exponent);

パラメーター
ebase　Rational
累乗対象の Rational。
exponent　Rational
累乗を指定する Rational。

戻り値
Rational
数値 ebase を exponent で累乗した値。

例

次の例では Pow(Rational,Rational)メソッドを使用して 2 の 0 乗から 2 の 32 乗までを出力しています。

```
using System;
using WS.Theia.ExtremelyPrecise;

public class Example
{
    public static void Main()
    {
        Rational value = 2;
        for (int power = 0; power <= 32; power++)
            Console.WriteLine("{0}^{1} = {2:N0} ",
                              value, power, Math.Pow(value, power));
    }
}
// The example displays the following output:
//       2^0 = 1
//       2^1 = 2
//       2^2 = 4
//       2^3 = 8
//       2^4 = 16
//       2^5 = 32
//       2^6 = 64
//       2^7 = 128
//       2^8 = 256
//       2^9 = 512
//       2^10 = 1,024
//       2^11 = 2,048
//       2^12 = 4,096
//       2^13 = 8,192
//       2^14 = 16,384
```

```
//      2^15 = 32,768
//      2^16 = 65,536
//      2^17 = 131,072
//      2^18 = 262,144
//      2^19 = 524,288
//      2^20 = 1,048,576
//      2^21 = 2,097,152
//      2^22 = 4,194,304
//      2^23 = 8,388,608
//      2^24 = 16,777,216
//      2^25 = 33,554,432
//      2^26 = 67,108,864
//      2^27 = 134,217,728
//      2^28 = 268,435,456
//      2^29 = 536,870,912
//      2^30 = 1,073,741,824
//      2^31 = 2,147,483,648
//      2^32 = 4,294,967,296
```

注釈

ebase、exponent パラメーターの値の組み合わせよっては累乗ではなく特定の値を返す場合があります。そのパターンは次の表のとおりです。

パラメーター	戻り値
ebase または exponent のいずれかが NaN。	NaN
ebase が NaN 以外の値、かつ exponent=0。	1
ebase=NegativeInfinity、かつ exponent<0。	0
ebase=NegativeInfinity、かつ exponent が正の奇数。	NegativeInfinity
ebase=NegativeInfinity, かつ exponent が正の奇数以外。	PositiveInfinity
ebase が NegativeInfinity 以外の負数、かつ exponent が NegativeInfinity 以外、かつ PositiveInfinity 以外。	NaN
ebase=-1、かつ exponent=NegativeInfinity または exponent=PositiveInfinity。	NaN
-1<ebase< 1、かつ exponent=NegativeInfinity。	PositiveInfinity
-1<ebase< 1、かつ exponent=PositiveInfinity。	0
ebase<-1 または ebase>1、かつ exponent=NegativeInfinity。	0
ebase<-1 または ebase>1、かつ exponent=PositiveInfinity。	PositiveInfinity
ebase=0、かつ exponent < 0	PositiveInfinity
ebase=0、かつ exponent>0。	0
ebase=1、かつ exponent が NaN 以外。	1
ebase=PositiveInfinity、かつ exponent<0。	0
ebase=PositiveInfinity、かつ y>0。	PositiveInfinity

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Root(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された数値の冪根を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Root(WS.Theia.ExtremelyPrecise.Rational value,
WS.Theia.ExtremelyPrecise.Rational index);
```

パラメーター
value　Rational
冪根を求める対象の数値。
index　Rational
冪根の次元数。

戻り値
Rational
次の表に示したいずれかの値。

value パラメーター	戻り値
0 または正	value の正の冪根。
負	NaN
NaN	NaN
PositiveInfinity	PositiveInfinity

例

正方形の面積の平方根は、正方形の辺の長さを表します。次の例では米国の州の年の面積を表示し、各都市が正方形と仮定し、各都市のおおよその辺の長さを算出します。

```
using System;
```

```
using WS.Theia.ExtremelyPrecise;

public class Example
{
    public static void Main()
    {
        // Create an array containing the area of some squares.
        Tuple<string, Rational>[] areas =
                            { Tuple.Create("Sitka, Alaska", (Rational)2870.3),
                              Tuple.Create("New York City", (Rational)302.6),
                              Tuple.Create("Los Angeles", (Rational)468.7),
                              Tuple.Create("Detroit", (Rational)138.8),
                              Tuple.Create("Chicago", (Rational)227.1),
                              Tuple.Create("San Diego", (Rational)325.2) };

        Console.WriteLine("{0,-18} {1,14:N1} {2,30}¥n", "City", "Area (mi.)",
                                "Equivalent to a square with:");


        foreach (var area in areas)
           Console.WriteLine("{0,-18} {1,14:N1} {2,14:N2} miles per side",
                                  area.Item1, area.Item2,
Math.Round(Math.Root(area.Item2,2), 2));
    }
}

// The example displays the following output:
//       City                        Area (mi.)     Equivalent to a square with:
//
//       Sitka, Alaska                 2,870.3              53.58 miles per side
//       New York City                   302.6              17.40 miles per side
//       Los Angeles                     468.7              21.65 miles per side
//       Detroit                         138.8              11.78 miles per side
//       Chicago                         227.1              15.07 miles per side
//       San Diego                       325.2              18.03 miles per side
```

注釈

このメソッドは index パラメーターに 2 を設定すると Sqrt(Rational)メソッドと等価になります。3 を設定すれば立方根、5 を設定すれば 5 次元根と高次元の冪根を得ることができます。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Math.Round Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
最も近い整数または指定した小数点以下の桁数に値を丸めます。

オーバーロード

Round(Rational,int, MidpointRounding)	Rational の値は指定した小数部の桁数に丸められ、中間値には指定した丸め処理が使用されます。
Round(Rational, MidpointRounding)	Rational の値は最も近い整数に丸められ、中間値には指定した丸め処理が使用されます。
Round(Rational,int)	Rational の値は指定した小数部の桁数に丸められ、中間値は最も近い偶数値に丸められます。
Round(Rational)	Rational の値は最も近い整数値に丸められ、中間値は最も近い偶数値に丸められます。

Round(Rational,int,MidpointRounding)

Rational の値は指定した小数部の桁数に丸められ、中間値には指定した丸め処理が使用されます。

---

public static WS.Theia.ExtremelyPrecise.Rational Round(WS.Theia.ExtremelyPrecise.Rational value,int digits,MidpointRounding mode);

パラメーター

value　Rational

丸め対象の Rational 値。

digits　Rational

戻り値の小数部の桁数。

mode　Rational

value が 2 つの整数の中間にある場合に丸める方法を指定します。

戻り値

Rational

digits に等しい小数部の桁数を格納する value に最も近い数値。 value の小数部の桁数が digits よりも少ない場合、value がそのまま返されます。

例外

ArgumentException

mode が MidpointRounding の正しい値ではありません。

例

次の例では、Round(Rational,int,MidpointRounding)メソッドを MidpointRounding 値列挙体で丸めモードを指定しています。

```
// This example demonstrates the Math.Round() method in conjunction
// with the MidpointRounding enumeration.
using System;
using WS.Theia.ExtremelyPrecise;

class Sample
{
      public static void Main()
      {
      Rational result = 0.0m;
      Rational posValue =　3.45m;
```

```
        Rational negValue = -3.45m;

// By default, round a positive and a negative value to the nearest even number.
// The precision of the result is 1 decimal place.

        result = Math.Round(posValue, 1);
        Console.WriteLine("{0,4} = Math.Round({1,5}, 1)", result, posValue);
        result = Math.Round(negValue, 1);
        Console.WriteLine("{0,4} = Math.Round({1,5}, 1)", result, negValue);
        Console.WriteLine();

// Round a positive value to the nearest even number, then to the nearest
number away from zero.
// The precision of the result is 1 decimal place.

        result = Math.Round(posValue, 1, MidpointRounding.ToEven);
        Console.WriteLine("{0,4} = Math.Round({1,5}, 1,
MidpointRounding.ToEven)", result, posValue);
        result = Math.Round(posValue, 1, MidpointRounding.AwayFromZero);
        Console.WriteLine("{0,4} = Math.Round({1,5}, 1,
MidpointRounding.AwayFromZero)", result, posValue);
        Console.WriteLine();

// Round a negative value to the nearest even number, then to the nearest
number away from zero.
// The precision of the result is 1 decimal place.

        result = Math.Round(negValue, 1, MidpointRounding.ToEven);
        Console.WriteLine("{0,4} = Math.Round({1,5}, 1,
MidpointRounding.ToEven)", result, negValue);
        result = Math.Round(negValue, 1, MidpointRounding.AwayFromZero);
        Console.WriteLine("{0,4} = Math.Round({1,5}, 1,
MidpointRounding.AwayFromZero)", result, negValue);
        Console.WriteLine();
        }
}
```

```
/*
This code example produces the following results:

 3.4 = Math.Round( 3.45, 1)
-3.4 = Math.Round(-3.45, 1)

 3.4 = Math.Round( 3.45, 1, MidpointRounding.ToEven)
 3.5 = Math.Round( 3.45, 1, MidpointRounding.AwayFromZero)

-3.4 = Math.Round(-3.45, 1, MidpointRounding.ToEven)
-3.5 = Math.Round(-3.45, 1, MidpointRounding.AwayFromZero)

*/
```

注釈

MidpointRounding 値列挙体で丸めモードを使用した場合次の表のとおりとなります。

	小数部＜0.5	小数部＝0.5	小数部＞0.5
ToEven	切り捨て	整数部が偶数の場合、切り捨て 整数部が奇数の場合、切り上げ	切り上げ
AwayFromZero	切り捨て	切り上げ	切り上げ

Round(Rational,MidpointRounding)

Rational の値は最も近い整数に丸められ、中間値には指定した丸め処理が使用されます。

```
public static WS.Theia.ExtremelyPrecise.Rational
Round(WS.Theia.ExtremelyPrecise.Rational value,MidpointRounding mode);
```

パラメーター
value　Rational
丸め対象の Rational 値。
mode　MidpointRounding
value が 2 つの整数の中間にある場合に丸める方法を指定します。

戻り値
Rational
value に最も近い整数。 value が 2 つの整数（一方が偶数でもう一方が奇数）の中間にある場合、mode によって 2 つの数値のどちらが返されるかが決まります。このメソッドは、整数型ではなく Rational を返します。

例外
ArgumentException
mode が MidpointRounding の正しい値ではありません。

例
次の例では、Round(Rational,MidpointRounding)メソッドを MidpointRounding 値列挙体で丸めモードを指定しています。

```
using System;
using WS.Theia.ExtremelyPrecise;
```

```
public class Example
{
    public static void Main()
    {
        Rational[] values = { 12.0, 12.1, 12.2, 12.3, 12.4, 12.5, 12.6,
                              12.7, 12.8, 12.9, 13.0 };
        Console.WriteLine("{0,-10} {1,-10} {2,-10} {3,-15}", "Value", "Default",
                          "ToEven", "AwayFromZero");
        foreach (var value in values)
            Console.WriteLine("{0,-10:R} {1,-10} {2,-10} {3,-15}",
                              value, Math.Round(value),
                              Math.Round(value,
MidpointRounding.ToEven),
                              Math.Round(value,
MidpointRounding.AwayFromZero));
    }
}
// The example displays the following output:
//       Value      Default    ToEven     AwayFromZero
//       12         12         12         12
//       12.1       12         12         12
//       12.2       12         12         12
//       12.3       12         12         12
//       12.4       12         12         12
//       12.5       12         12         13
//       12.6       13         13         13
//       12.7       13         13         13
//       12.8       13         13         13
//       12.9       13         13         13
//       13.0       13         13         13
```

注釈

MidpointRounding 値列挙体で丸めモードを使用した場合次の表のとおりとなります。

	小数部＜0.5	小数部＝0.5	小数部＞0.5
ToEven	切り捨て	整数部が偶数の場合、切り捨て 整数部が奇数の場合、切り上げ	切り上げ
AwayFromZero	切り捨て	切り上げ	切り上げ

Round(Rational,int)

Rational の値は指定した小数部の桁数に丸められ、中間値は最も近い偶数値に丸められます。

---

public static WS.Theia.ExtremelyPrecise.Rational
Round(WS.Theia.ExtremelyPrecise.Rational value,int digits);

パラメーター

value　Rational
丸め対象の Rational 値。
digits　Rational
戻り値の小数部の桁数。

戻り値

Rational
digits に等しい小数部の桁数を格納する value に最も近い数値。 value の小数部の桁数が digits よりも少ない場合、value がそのまま返されます。

例

次の例では、Round(Rational,int)メソッドで丸めを行っています。

```
Math.Round(3.44, 1); //Returns 3.4.
Math.Round(3.45, 1); //Returns 3.4.
Math.Round(3.46, 1); //Returns 3.5.

Math.Round(4.34, 1); // Returns 4.3
Math.Round(4.35, 1); // Returns 4.4
Math.Round(4.36, 1); // Returns 4.4
```

Round(Rational)

Rational の値は最も近い整数値に丸められ、中間値は最も近い偶数値に丸められます。

```
public static WS.Theia.ExtremelyPrecise.Rational
Round(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター

value　Rational

丸め対象の Rational 値。

戻り値

Rational

value パラメーターに最も近い整数。 value の小数部が 2 つの整数（一方が偶数で、もう一方が奇数）の中間にある場合は、偶数が返されます。 このメソッドは、整数型ではなく Rational を返します。

例

次の例では、Round(Rational)メソッドで丸めを行っています。

```
using System;
using WS.Theia.ExtremelyPrecise;

class Program
{
    static void Main()
    {
    Console.WriteLine("Classic Math.Round in CSharp");
    Console.WriteLine(Math.Round(4.4m)); // 4
    Console.WriteLine(Math.Round(4.5 m)); // 4
    Console.WriteLine(Math.Round(4.6 m)); // 5
    Console.WriteLine(Math.Round(5.5 m)); // 6
    }
}
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Sign(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational の符号を示す整数を返します。

```
public static int Sign(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
符号を取得したい Rational 値。

戻り値
Int32
value の符号を示す数値（次の表を参照）。

戻り値	説明
-1	value が 0 未満の場合。
0	value が 0 の場合。
1	value が 0 より大きい場合。

例外
ArithmeticException
Value の値が NaN です。

例

次の例では、Sign(Rational)メソッドを使用し符号を取得し、Rational 値とともにコンソールに表示しています。

```
// This example demonstrates Math.Sign()
```

```
using WS.Theia.ExtremelyPrecise;

class Sample
{
      public static void Main()
      {
      string str = "{0}: {1,3} is {2} zero.";
      string nl = Environment.NewLine;

      Rational     xRational1    = 6.0;

      Console.WriteLine(str, "Rational ", xRational1,
Test(Math.Sign(xRational1)));

      }
//
      public static String Test(int compare)
      {
      if (compare == 0)
          return "equal to";
      else if (compare < 0)
            return "less than";
      else
            return "greater than";
      }
}
/*
This example produces the following results:
Test the sign of the following types of values:
Rational :      6 is greater than zero.
*/
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Sin(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された角度のサインを返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Sin(WS.Theia.ExtremelyPrecise.Rational radian);
```

パラメーター
radian　Rational
ラジアンで表した角度。

戻り値
Rational
radian のサイン。 radian が NaN、NegativeInfinity、PositiveInfinity のいずれかに等しい場合、このメソッドは NaN を返します。

例

次の例では、三角関数を算出しています。

```
// Example for the trigonometric Math.Sin( Rational )
// and Math.Cos( Rational ) methods.
using System;
using WS.Theia.ExtremelyPrecise;
class SinCos
{
    public static void Main()
    {
        Console.WriteLine(
```

```
            "This example of trigonometric " +
            "Math.Sin( Rational ) and Math.Cos( Rational )¥n" +
            "generates the following output.¥n" );
        Console.WriteLine(
            "Convert selected values for X to radians ¥n" +
            "and evaluate these trigonometric identities:" );
        Console.WriteLine( "     sin^2(X) + cos^2(X) == 1¥n" +
                           "    sin(2 * X) == 2 * sin(X) * cos(X)" );
        Console.WriteLine( "     cos(2 * X) == cos^2(X) - sin^2(X)" );

        UseSineCosine(15.0);
        UseSineCosine(30.0);
        UseSineCosine(45.0);

        Console.WriteLine(
            "¥nConvert selected values for X and Y to radians ¥n" +
            "and evaluate these trigonometric identities:" );
        Console.WriteLine( "     sin(X + Y) == sin(X) * cos(Y) + cos(X) *
sin(Y)" );
        Console.WriteLine( "     cos(X + Y) == cos(X) * cos(Y) - sin(X) *
sin(Y)" );
        UseTwoAngles(15.0, 30.0);
        UseTwoAngles(30.0, 45.0);
    }
    // Evaluate trigonometric identities with a given angle.
    static void UseSineCosine(Rational degrees)
    {
        Rational angle    = Math.PI * degrees / 180.0;
        Rational sinAngle = Math.Sin(angle);
        Rational cosAngle = Math.Cos(angle);

        // Evaluate sin^2(X) + cos^2(X) == 1.
        Console.WriteLine(
            "¥n                                  Math.Sin({0} deg) ==
{1:E16}¥n" +
            "                                Math.Cos({0} deg) == {2:E16}",
```

```
            degrees, Math.Sin(angle), Math.Cos(angle) );
        Console.WriteLine(
            "(Math.Sin({0} deg))^2 + (Math.Cos({0} deg))^2 == {1:E16}",
            degrees, sinAngle * sinAngle + cosAngle * cosAngle );

        // Evaluate sin(2 * X) == 2 * sin(X) * cos(X).
        Console.WriteLine(
            "                                  Math.Sin({0} deg) == {1:E16}",
            2.0 * degrees, Math.Sin(2.0 * angle) );
        Console.WriteLine(
            "       2 * Math.Sin({0} deg) * Math.Cos({0} deg) == {1:E16}",
            degrees, 2.0 * sinAngle * cosAngle );

        // Evaluate cos(2 * X) == cos^2(X) - sin^2(X).
        Console.WriteLine(
            "                                  Math.Cos({0} deg) == {1:E16}",
            2.0 * degrees, Math.Cos(2.0 * angle) );
        Console.WriteLine(
            "(Math.Cos({0} deg))^2 - (Math.Sin({0} deg))^2 == {1:E16}",
            degrees, cosAngle * cosAngle - sinAngle * sinAngle );
    }
    // Evaluate trigonometric identities that are functions of two angles.
    static void UseTwoAngles(Rational degreesX, Rational degreesY)
    {
        Rational    angleX    = Math.PI * degreesX / 180.0;
        Rational    angleY    = Math.PI * degreesY / 180.0;

        // Evaluate sin(X + Y) == sin(X) * cos(Y) + cos(X) * sin(Y).
        Console.WriteLine(

            "¥n            Math.Sin({0} deg) * Math.Cos({1} deg) +¥n" +
            "          Math.Cos({0} deg) * Math.Sin({1} deg) == {2:E16}",
            degreesX, degreesY, Math.Sin(angleX) * Math.Cos(angleY) +
            Math.Cos(angleX) * Math.Sin(angleY));
        Console.WriteLine(
            "                                  Math.Sin({0} deg) == {1:E16}",
```

```
                degreesX + degreesY, Math.Sin(angleX + angleY));

            // Evaluate cos(X + Y) == cos(X) * cos(Y) - sin(X) * sin(Y).
            Console.WriteLine(
                "            Math.Cos({0} deg) * Math.Cos({1} deg) -¥n" +
                "            Math.Sin({0} deg) * Math.Sin({1} deg) == {2:E16}",
                degreesX, degreesY, Math.Cos(angleX) * Math.Cos(angleY) -
                Math.Sin(angleX) * Math.Sin(angleY));
            Console.WriteLine(
                "                                    Math.Cos({0} deg) == {1:E16}",
                degreesX + degreesY, Math.Cos(angleX + angleY));
        }
}
/*
This example of trigonometric Math.Sin( Rational ) and Math.Cos( Rational )
generates the following output.

Convert selected values for X to radians
and evaluate these trigonometric identities:
    sin^2(X) + cos^2(X) == 1
    sin(2 * X) == 2 * sin(X) * cos(X)
    cos(2 * X) == cos^2(X) - sin^2(X)
                                Math.Sin(15 deg) == 2.5881904510252074E-
001
                                Math.Cos(15 deg) == 9.6592582628906831E-
001
(Math.Sin(15 deg))^2 + (Math.Cos(15 deg))^2 ==
1.0000000000000000E+000
                                Math.Sin(30 deg) == 4.9999999999999994E-
001
    2 * Math.Sin(15 deg) * Math.Cos(15 deg) == 4.9999999999999994E-001
                                Math.Cos(30 deg) == 8.6602540378443871E-
001
(Math.Cos(15 deg))^2 - (Math.Sin(15 deg))^2 == 8.6602540378443871E-001

                                Math.Sin(30 deg) == 4.9999999999999994E-
```

001
                    Math.Cos(30 deg) == 8.6602540378443871E-
001
(Math.Sin(30 deg))^2 + (Math.Cos(30 deg))^2 ==
1.0000000000000000E+000
                    Math.Sin(60 deg) == 8.6602540378443860E-
001
    2 * Math.Sin(30 deg) * Math.Cos(30 deg) == 8.6602540378443860E-001
                    Math.Cos(60 deg) == 5.0000000000000011E-
001
(Math.Cos(30 deg))^2 - (Math.Sin(30 deg))^2 == 5.0000000000000022E-001

                    Math.Sin(45 deg) == 7.0710678118654746E-
001
                    Math.Cos(45 deg) == 7.0710678118654757E-
001
(Math.Sin(45 deg))^2 + (Math.Cos(45 deg))^2 ==
1.0000000000000000E+000
                    Math.Sin(90 deg) ==
1.0000000000000000E+000
    2 * Math.Sin(45 deg) * Math.Cos(45 deg) == 1.0000000000000000E+000
                    Math.Cos(90 deg) == 6.1230317691118863E-
017
(Math.Cos(45 deg))^2 - (Math.Sin(45 deg))^2 == 2.2204460492503131E-016

Convert selected values for X and Y to radians
and evaluate these trigonometric identities:
  sin(X + Y) == sin(X) * cos(Y) + cos(X) * sin(Y)
  cos(X + Y) == cos(X) * cos(Y) - sin(X) * sin(Y)

      Math.Sin(15 deg) * Math.Cos(30 deg) +
      Math.Cos(15 deg) * Math.Sin(30 deg) == 7.0710678118654746E-001
                    Math.Sin(45 deg) == 7.0710678118654746E-
001
      Math.Cos(15 deg) * Math.Cos(30 deg) -
      Math.Sin(15 deg) * Math.Sin(30 deg) == 7.0710678118654757E-001

```
                                Math.Cos(45 deg) == 7.0710678118654757E-
001

        Math.Sin(30 deg) * Math.Cos(45 deg) +
        Math.Cos(30 deg) * Math.Sin(45 deg) == 9.6592582628906831E-001
                                Math.Sin(75 deg) == 9.6592582628906820E-
001
        Math.Cos(30 deg) * Math.Cos(45 deg) -
        Math.Sin(30 deg) * Math.Sin(45 deg) == 2.5881904510252085E-001
                                Math.Cos(75 deg) == 2.5881904510252096E-
001
*/
```

注釈

引数に入力する角度はラジアン単位である必要があります。角度に Math.PI/180 を乗算する事でラジアン単位に変換できます。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Math.Sinh(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された角度のハイパーボリックサインを返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Sinh(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
radian　Rational
ラジアンで表した角度。

戻り値
Rational
value のハイパーボリック サイン。 value が NegativeInfinity、PositiveInfinity、または NaN のいずれかに等しい場合、このメソッドは value に等しい Rational を返します。

例

次の例では、Sinh の結果を表示しています。

```
// Example for the hyperbolic Math.Sinh( Rational )
// and Math.Cosh( Rational ) methods.
using System;
using WS.Theia.ExtremelyPrecise;

class SinhCosh
{
    public static void Main()
    {
```

```
        Console.WriteLine(
            "This example of hyperbolic Math.Sinh( Rational ) " +
            "and Math.Cosh( Rational )¥n" +
            "generates the following output.¥n" );
        Console.WriteLine(
            "Evaluate these hyperbolic identities " +
            "with selected values for X:" );
        Console.WriteLine(
            "     cosh^2(X) - sinh^2(X) == 1¥n" +
            "     sinh(2 * X) == 2 * sinh(X) * cosh(X)" );
        Console.WriteLine( "     cosh(2 * X) == cosh^2(X) + sinh^2(X)" );

        UseSinhCosh(0.1);
        UseSinhCosh(1.2);
        UseSinhCosh(4.9);

        Console.WriteLine(
            "¥nEvaluate these hyperbolic identities " +
            "with selected values for X and Y:" );
        Console.WriteLine(
            "     sinh(X + Y) == sinh(X) * cosh(Y) + cosh(X) * sinh(Y)" );
        Console.WriteLine(
            "     cosh(X + Y) == cosh(X) * cosh(Y) + sinh(X) * sinh(Y)" );

        UseTwoArgs(0.1, 1.2);
        UseTwoArgs(1.2, 4.9);
    }

    // Evaluate hyperbolic identities with a given argument.
    static void UseSinhCosh(Rational arg)
    {
        Rational sinhArg = Math.Sinh(arg);
        Rational coshArg = Math.Cosh(arg);

        // Evaluate cosh^2(X) - sinh^2(X) == 1.
        Console.WriteLine(
```

```
            "¥n                                 Math.Sinh({0}) == {1:E16}¥n" +
            "                                 Math.Cosh({0}) == {2:E16}",
            arg, Math.Sinh(arg), Math.Cosh(arg) );
        Console.WriteLine(
            "(Math.Cosh({0}))^2 - (Math.Sinh({0}))^2 == {1:E16}",
            arg, coshArg * coshArg - sinhArg * sinhArg );
        // Evaluate sinh(2 * X) == 2 * sinh(X) * cosh(X).
        Console.WriteLine(
            "                                 Math.Sinh({0}) == {1:E16}",
            2.0 * arg, Math.Sinh(2.0 * arg) );
        Console.WriteLine(
            "      2 * Math.Sinh({0}) * Math.Cosh({0}) == {1:E16}",
            arg, 2.0 * sinhArg * coshArg );

        // Evaluate cosh(2 * X) == cosh^2(X) + sinh^2(X).
        Console.WriteLine(
            "                                 Math.Cosh({0}) == {1:E16}",
            2.0 * arg, Math.Cosh(2.0 * arg) );
        Console.WriteLine(
            "(Math.Cosh({0}))^2 + (Math.Sinh({0}))^2 == {1:E16}",
            arg, coshArg * coshArg + sinhArg * sinhArg );
    }
    // Evaluate hyperbolic identities that are functions of two arguments.
    static void UseTwoArgs(Rational argX, Rational argY)
    {
        // Evaluate sinh(X + Y) == sinh(X) * cosh(Y) + cosh(X) * sinh(Y).
        Console.WriteLine(
            "¥n            Math.Sinh({0}) * Math.Cosh({1}) +¥n" +
            "            Math.Cosh({0}) * Math.Sinh({1}) == {2:E16}",
            argX, argY, Math.Sinh(argX) * Math.Cosh(argY) +
            Math.Cosh(argX) * Math.Sinh(argY));
        Console.WriteLine(
            "                                 Math.Sinh({0}) == {1:E16}",
            argX + argY, Math.Sinh(argX + argY));

        // Evaluate cosh(X + Y) == cosh(X) * cosh(Y) + sinh(X) * sinh(Y).
```

```
            Console.WriteLine(
                "              Math.Cosh({0}) * Math.Cosh({1}) +¥n" +
                "              Math.Sinh({0}) * Math.Sinh({1}) == {2:E16}",
                argX, argY, Math.Cosh(argX) * Math.Math.Cosh(argY) +
                Math.Sinh(argX) * Math.Sinh(argY));
            Console.WriteLine(
                "                                    Math.Cosh({0}) == {1:E16}",
                argX + argY, Math.Cosh(argX + argY));
        }
}

/*
This example of hyperbolic Math.Sinh( Rational ) and Math.Cosh( Rational )
generates the following output.

Evaluate these hyperbolic identities with selected values for X:
    cosh^2(X) - sinh^2(X) == 1
    sinh(2 * X) == 2 * sinh(X) * cosh(X)
    cosh(2 * X) == cosh^2(X) + sinh^2(X)

                                Math.Sinh(0.1) == 1.0016675001984403E-001
                                Math.Cosh(0.1) == 1.0050041680558035E+000
(Math.Cosh(0.1))^2 - (Math.Sinh(0.1))^2 == 9.9999999999999989E-001
                                Math.Sinh(0.2) == 2.0133600254109399E-001
    2 * Math.Sinh(0.1) * Math.Cosh(0.1) == 2.0133600254109396E-001
                                Math.Cosh(0.2) == 1.0200667556190759E+000
(Math.Cosh(0.1))^2 + (Math.Sinh(0.1))^2 == 1.0200667556190757E+000

                                Math.Sinh(1.2) == 1.5094613554121725E+000
                                Math.Cosh(1.2) == 1.8106555673243747E+000
(Math.Cosh(1.2))^2 - (Math.Sinh(1.2))^2 == 1.0000000000000000E+000
                                Math.Sinh(2.4) == 5.4662292136760939E+000
    2 * Math.Sinh(1.2) * Math.Cosh(1.2) == 5.4662292136760939E+000
                                Math.Cosh(2.4) == 5.5569471669655064E+000
(Math.Cosh(1.2))^2 + (Math.Sinh(1.2))^2 == 5.5569471669655064E+000
```

```
                                Math.Sinh(4.9) == 6.7141166550932297E+001
                                Math.Cosh(4.9) == 6.7148613134003227E+001
(Math.Cosh(4.9))^2 - (Math.Sinh(4.9))^2 == 1.0000000000000000E+000
                                Math.Sinh(9.8) == 9.0168724361884615E+003
    2 * Math.Sinh(4.9) * Math.Cosh(4.9) == 9.0168724361884615E+003
                                Math.Cosh(9.8) == 9.0168724916400624E+003
(Math.Cosh(4.9))^2 + (Math.Sinh(4.9))^2 == 9.0168724916400606E+003

Evaluate these hyperbolic identities with selected values for X and Y:
    sinh(X + Y) == sinh(X) * cosh(Y) + cosh(X) * sinh(Y)
    cosh(X + Y) == cosh(X) * cosh(Y) + sinh(X) * sinh(Y)

        Math.Sinh(0.1) * Math.Cosh(1.2) +
        Math.Cosh(0.1) * Math.Sinh(1.2) == 1.6983824372926155E+000
                                Math.Sinh(1.3) == 1.6983824372926160E+000
        Math.Cosh(0.1) * Math.Cosh(1.2) +
        Math.Sinh(0.1) * Math.Sinh(1.2) == 1.9709142303266281E+000
                                Math.Cosh(1.3) == 1.9709142303266285E+000

        Math.Sinh(1.2) * Math.Cosh(4.9) +
        Math.Cosh(1.2) * Math.Sinh(4.9) == 2.2292776360739879E+002
                                Math.Sinh(6.1) == 2.2292776360739885E+002
        Math.Cosh(1.2) * Math.Cosh(4.9) +
        Math.Sinh(1.2) * Math.Sinh(4.9) == 2.2293000647511826E+002
                                Math.Cosh(6.1) == 2.2293000647511832E+002
*/
```

注釈

引数に入力する角度はラジアン単位である必要があります。角度に Math.PI/180 を乗算する事でラジアン単位に変換できます。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Sqrt(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された数値の平方根を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Sqrt(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター

value　Rational
平方根を求める対象の数値。

戻り値

Rational
次の表に示したいずれかの値。

value パラメーター	戻り値
0 または正	value の正の平方根。
負	NaN
NaN	NaN
PositiveInfinity	PositiveInfinity

例

正方形の面積の平方根は、正方形の辺の長さを表します。次の例では米国の州の年の面積を表示し、各都市が正方形と仮定し、各都市のおおよその辺の長さを算出します。

```
using System;
using WS.Theia.ExtremelyPrecise;

public class Example
```

```
{
    public static void Main()
    {
        // Create an array containing the area of some squares.
        Tuple<string, Rational>[] areas =
                        { Tuple.Create("Sitka, Alaska", new Rational(2870.3m)),
                          Tuple.Create("New York City", new Rational(302.6m)),
                           Tuple.Create("Los Angeles",new Rational(468.7m)),
                           Tuple.Create("Detroit",new Rational(138.8m)),
                           Tuple.Create("Chicago",new Rational(227.1m)),
                           Tuple.Create("San Diego",new Rational(325.2m)) };

        Console.WriteLine("{0,-18} {1,14:N1} {2,30}¥n", "City", "Area (mi.)",
                                "Equivalent to a square with:");


        foreach (var area in areas)
           Console.WriteLine("{0,-18} {1,14:N1} {2,14:N2} miles per side",
                                    area.Item1, area.Item2,
Math.Round(Math.Sqrt(area.Item2), 2));
    }
}

// The example displays the following output:
//       City                        Area (mi.)     Equivalent to a square with:
//
//       Sitka, Alaska                 2,870.3              53.58 miles per side
//       New York City                   302.6              17.40 miles per side
//       Los Angeles                     468.7              21.65 miles per side
//       Detroit                         138.8              11.78 miles per side
//       Chicago                         227.1              15.07 miles per side
//       San Diego                       325.2              18.03 miles per side
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Tan(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された角度のタンジェントを返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Tan(WS.Theia.ExtremelyPrecise.Rational radian);
```

パラメーター
radian　Rational
ラジアンで表した角度。

戻り値
Rational
radian のタンジェント。 radian が NaN、NegativeInfinity、PositiveInfinity のいずれかに等しい場合、このメソッドは NaN を返します。

例

次の例では、三角関数を算出しています。

```
// Example for the trigonometric Math.Sin( Rational )
// and Math.Cos( Rational ) methods.
using System;
using WS.Theia.ExtremelyPrecise;
class SinCos
{
    public static void Main()
    {
        Console.WriteLine(
```

```
            "This example of trigonometric " +
            "Math.Sin( Rational ) and Math.Cos( Rational )¥n" +
            "generates the following output.¥n" );
        Console.WriteLine(
            "Convert selected values for X to radians ¥n" +
            "and evaluate these trigonometric identities:" );
        Console.WriteLine( "     sin^2(X) + cos^2(X) == 1¥n" +
                           "    sin(2 * X) == 2 * sin(X) * cos(X)" );
        Console.WriteLine( "     cos(2 * X) == cos^2(X) - sin^2(X)" );

        UseSineCosine(15.0);
        UseSineCosine(30.0);
        UseSineCosine(45.0);

        Console.WriteLine(
            "¥nConvert selected values for X and Y to radians ¥n" +
            "and evaluate these trigonometric identities:" );
        Console.WriteLine( "     sin(X + Y) == sin(X) * cos(Y) + cos(X) *
sin(Y)" );
        Console.WriteLine( "     cos(X + Y) == cos(X) * cos(Y) - sin(X) *
sin(Y)" );
        UseTwoAngles(15.0, 30.0);
        UseTwoAngles(30.0, 45.0);
    }
    // Evaluate trigonometric identities with a given angle.
    static void UseSineCosine(Rational degrees)
    {
        Rational angle    = Math.PI * degrees / 180.0;
        Rational sinAngle = Math.Sin(angle);
        Rational cosAngle = Math.Cos(angle);

        // Evaluate sin^2(X) + cos^2(X) == 1.
        Console.WriteLine(
            "¥n                                  Math.Sin({0} deg) ==
{1:E16}¥n" +
            "                                Math.Cos({0} deg) == {2:E16}",
```

```
            degrees, Math.Sin(angle), Math.Cos(angle) );
        Console.WriteLine(
            "(Math.Sin({0} deg))^2 + (Math.Cos({0} deg))^2 == {1:E16}",
            degrees, sinAngle * sinAngle + cosAngle * cosAngle );

        // Evaluate sin(2 * X) == 2 * sin(X) * cos(X).
        Console.WriteLine(
            "                                  Math.Sin({0} deg) == {1:E16}",
            2.0 * degrees, Math.Sin(2.0 * angle) );
        Console.WriteLine(
            "       2 * Math.Sin({0} deg) * Math.Cos({0} deg) == {1:E16}",
            degrees, 2.0 * sinAngle * cosAngle );

        // Evaluate cos(2 * X) == cos^2(X) - sin^2(X).
        Console.WriteLine(
            "                                  Math.Cos({0} deg) == {1:E16}",
            2.0 * degrees, Math.Cos(2.0 * angle) );
        Console.WriteLine(
            "(Math.Cos({0} deg))^2 - (Math.Sin({0} deg))^2 == {1:E16}",
            degrees, cosAngle * cosAngle - sinAngle * sinAngle );
    }
    // Evaluate trigonometric identities that are functions of two angles.
    static void UseTwoAngles(Rational degreesX, Rational degreesY)
    {
        Rational    angleX    = Math.PI * degreesX / 180.0;
        Rational    angleY    = Math.PI * degreesY / 180.0;

        // Evaluate sin(X + Y) == sin(X) * cos(Y) + cos(X) * sin(Y).
        Console.WriteLine(

            "¥n            Math.Sin({0} deg) * Math.Cos({1} deg) +¥n" +
            "          Math.Cos({0} deg) * Math.Sin({1} deg) == {2:E16}",
            degreesX, degreesY, Math.Sin(angleX) * Math.Cos(angleY) +
            Math.Cos(angleX) * Math.Sin(angleY));
        Console.WriteLine(
            "                                  Math.Sin({0} deg) == {1:E16}",
```

```
                degreesX + degreesY, Math.Sin(angleX + angleY));

            // Evaluate cos(X + Y) == cos(X) * cos(Y) - sin(X) * sin(Y).
            Console.WriteLine(
                "            Math.Cos({0} deg) * Math.Cos({1} deg) -¥n" +
                "            Math.Sin({0} deg) * Math.Sin({1} deg) == {2:E16}",
                degreesX, degreesY, Math.Cos(angleX) * Math.Cos(angleY) -
                Math.Sin(angleX) * Math.Sin(angleY));
            Console.WriteLine(
                "                                    Math.Cos({0} deg) == {1:E16}",
                degreesX + degreesY, Math.Cos(angleX + angleY));
        }
}
/*
This example of trigonometric Math.Sin( Rational ) and Math.Cos( Rational )
generates the following output.

Convert selected values for X to radians
and evaluate these trigonometric identities:
    sin^2(X) + cos^2(X) == 1
    sin(2 * X) == 2 * sin(X) * cos(X)
    cos(2 * X) == cos^2(X) - sin^2(X)
                                Math.Sin(15 deg) == 2.5881904510252074E-
001
                                Math.Cos(15 deg) == 9.6592582628906831E-
001
(Math.Sin(15 deg))^2 + (Math.Cos(15 deg))^2 ==
1.0000000000000000E+000
                                Math.Sin(30 deg) == 4.9999999999999994E-
001
    2 * Math.Sin(15 deg) * Math.Cos(15 deg) == 4.9999999999999994E-001
                                Math.Cos(30 deg) == 8.6602540378443871E-
001
(Math.Cos(15 deg))^2 - (Math.Sin(15 deg))^2 == 8.6602540378443871E-001

                                Math.Sin(30 deg) == 4.9999999999999994E-
```

001
                    Math.Cos(30 deg) == 8.6602540378443871E-
001
(Math.Sin(30 deg))^2 + (Math.Cos(30 deg))^2 ==
1.0000000000000000E+000
                    Math.Sin(60 deg) == 8.6602540378443860E-
001
    2 * Math.Sin(30 deg) * Math.Cos(30 deg) == 8.6602540378443860E-001
                    Math.Cos(60 deg) == 5.0000000000000011E-
001
(Math.Cos(30 deg))^2 - (Math.Sin(30 deg))^2 == 5.0000000000000022E-001

                    Math.Sin(45 deg) == 7.0710678118654746E-
001
                    Math.Cos(45 deg) == 7.0710678118654757E-
001
(Math.Sin(45 deg))^2 + (Math.Cos(45 deg))^2 ==
1.0000000000000000E+000
                    Math.Sin(90 deg) ==
1.0000000000000000E+000
    2 * Math.Sin(45 deg) * Math.Cos(45 deg) == 1.0000000000000000E+000
                    Math.Cos(90 deg) == 6.1230317691118863E-
017
(Math.Cos(45 deg))^2 - (Math.Sin(45 deg))^2 == 2.2204460492503131E-016

Convert selected values for X and Y to radians
and evaluate these trigonometric identities:
  sin(X + Y) == sin(X) * cos(Y) + cos(X) * sin(Y)
  cos(X + Y) == cos(X) * cos(Y) - sin(X) * sin(Y)

      Math.Sin(15 deg) * Math.Cos(30 deg) +
      Math.Cos(15 deg) * Math.Sin(30 deg) == 7.0710678118654746E-001
                    Math.Sin(45 deg) == 7.0710678118654746E-
001
      Math.Cos(15 deg) * Math.Cos(30 deg) -
      Math.Sin(15 deg) * Math.Sin(30 deg) == 7.0710678118654757E-001

```
                                Math.Cos(45 deg) == 7.0710678118654757E-
001

        Math.Sin(30 deg) * Math.Cos(45 deg) +
        Math.Cos(30 deg) * Math.Sin(45 deg) == 9.6592582628906831E-001
                                Math.Sin(75 deg) == 9.6592582628906820E-
001
        Math.Cos(30 deg) * Math.Cos(45 deg) -
        Math.Sin(30 deg) * Math.Sin(45 deg) == 2.5881904510252085E-001
                                Math.Cos(75 deg) == 2.5881904510252096E-
001
*/
```

注釈

引数に入力する角度はラジアン単位である必要があります。角度に Math.PI/180 を乗算する事でラジアン単位に変換できます。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Math.Tanh(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された角度のハイパーボリック タンジェントを返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Tanh(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
radian　Rational
ラジアンで表した角度。

戻り値
Rational
value のハイパーボリック タンジェント。 value が NegativeInfinity に等しい場合、このメソッドは -1 を返します。 値が PositiveInfinity に等しい場合、このメソッドは 1 を返します。 value が NaN に等しい場合、このメソッドは NaN を返します。

例

次の例では、Tanh の結果を表示しています。

```
// Example for the hyperbolic Math.Tanh( Rational ) method.

using System;
using WS.Theia.ExtremelyPrecise;

class DemoTanh
{
    public static void Main()
```

```
{
    Console.WriteLine(
        "This example of hyperbolic Math.Tanh( Rational )¥n" +
        "generates the following output." );
    Console.WriteLine(
        "¥nEvaluate these hyperbolic identities " +
        "with selected values for X:" );
    Console.WriteLine( "    tanh(X) == sinh(X) / cosh(X)" );
    Console.WriteLine(
        "    tanh(2 * X) == 2 * tanh(X) / (1 + tanh^2(X))" );

    UseTanh(0.1);
    UseTanh(1.2);
    UseTanh(4.9);

    Console.WriteLine(
        "¥nEvaluate [tanh(X + Y) == (tanh(X) + tanh(Y)) " +
        "/ (1 + tanh(X) * tanh(Y))]" +
        "¥nwith selected values for X and Y:" );

    UseTwoArgs(0.1, 1.2);
    UseTwoArgs(1.2, 4.9);
}

// Evaluate hyperbolic identities with a given argument.
static void UseTanh(Rational arg)
{
    Rational tanhArg = Math.Tanh(arg);

    // Evaluate tanh(X) == sinh(X) / cosh(X).
    Console.WriteLine(
        "¥n                          Math.Tanh({0}) == {1:E16}¥n" +
        "        Math.Sinh({0}) / Math.Cosh({0}) == {2:E16}",
        arg, tanhArg, (Math.Sinh(arg) / Math.Cosh(arg)) );

    // Evaluate tanh(2 * X) == 2 * tanh(X) / (1 + tanh^2(X)).
```

```
            Console.WriteLine(
                "                          2 * Math.Tanh({0}) /",
                arg, 2.0 * tanhArg );
            Console.WriteLine(
                "                (1 + (Math.Tanh({0}))^2) == {1:E16}",
                arg, 2.0 * tanhArg / (1.0 + tanhArg * tanhArg ) );
            Console.WriteLine(
                "                                Math.Tanh({0}) == {1:E16}",
                2.0 * arg, Math.Tanh(2.0 * arg) );
        }

        // Evaluate a hyperbolic identity that is a function of two arguments.
        static void UseTwoArgs(Rational argX, Rational argY)
        {
            // Evaluate tanh(X + Y) == (tanh(X) + tanh(Y)) / (1 + tanh(X) *
tanh(Y)).
            Console.WriteLine(
                "¥n      (Math.Tanh({0}) + Math.Tanh({1})) /¥n" +
                "(1 + Math.Tanh({0}) * Math.Tanh({1})) == {2:E16}",
                argX, argY, (Math.Tanh(argX) + Math.Tanh(argY)) /
                (1.0 + Math.Tanh(argX) * Math.Tanh(argY)) );
            Console.WriteLine(
                "                                Math.Tanh({0}) == {1:E16}",
                argX + argY, Math.Tanh(argX + argY));
        }
}

/*
This example of hyperbolic Math.Tanh( Rational )
generates the following output.

Evaluate these hyperbolic identities with selected values for X:
    tanh(X) == sinh(X) / cosh(X)
    tanh(2 * X) == 2 * tanh(X) / (1 + tanh^2(X))

                            Math.Tanh(0.1) == 9.9667994624955819E-002
```

```
Math.Sinh(0.1) / Math.Cosh(0.1) == 9.9667994624955819E-002
                2 * Math.Tanh(0.1) /
          (1 + (Math.Tanh(0.1))^2) == 1.9737532022490401E-001
                    Math.Tanh(0.2) == 1.9737532022490401E-001

                    Math.Tanh(1.2) == 8.3365460701215521E-001
Math.Sinh(1.2) / Math.Cosh(1.2) == 8.3365460701215521E-001
                2 * Math.Tanh(1.2) /
          (1 + (Math.Tanh(1.2))^2) == 9.8367485769368024E-001
                    Math.Tanh(2.4) == 9.8367485769368024E-001

                    Math.Tanh(4.9) == 9.9988910295055444E-001
Math.Sinh(4.9) / Math.Cosh(4.9) == 9.9988910295055433E-001
                2 * Math.Tanh(4.9) /
          (1 + (Math.Tanh(4.9))^2) == 9.9999999385024030E-001
                    Math.Tanh(9.8) == 9.9999999385024030E-001

Evaluate [tanh(X + Y) == (tanh(X) + tanh(Y)) / (1 + tanh(X) * tanh(Y))]
with selected values for X and Y:

     (Math.Tanh(0.1) + Math.Tanh(1.2)) /
(1 + Math.Tanh(0.1) * Math.Tanh(1.2)) == 8.6172315931330645E-001
                    Math.Tanh(1.3) == 8.6172315931330634E-001

     (Math.Tanh(1.2) + Math.Tanh(4.9)) /
(1 + Math.Tanh(1.2) * Math.Tanh(4.9)) == 9.9998993913939649E-001
                    Math.Tanh(6.1) == 9.9998993913939649E-001
*/
```

注釈

引数に入力する角度はラジアン単位である必要があります。角度に Math.PI/180 を乗算する事でラジアン単位に変換できます。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Math.Truncate(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した Rational の小数部を切り捨て整数部のみを取得します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Truncate(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター

value　Rational
切り捨て対象の数値。

戻り値

Rational
value の整数部。つまり、小数部の桁を破棄した後に残る数値（次の表にリストされている値のいずれか）。

value	戻り値
NaN	NaN
NegativeInfinity	NegativeInfinity
PositiveInfinity	PositiveInfinity

例

次の例では Math.Truncate(Rational)メソッドを使用して、正の値、負の値それぞれに小数点以下の切り捨てを行っています。

```
Rational floatNumber;

floatNumber = 32.7865;
// Displays 32
Console.WriteLine(Math.Truncate(floatNumber));

floatNumber = -32.9012;
// Displays -32
Console.WriteLine(Math.Truncate(floatNumber));
```

注釈

Truncate(Rational)メソッドでは value パラメーターの小数部を切り捨て 0 に近い方の整数に丸めます。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational Struct

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
任意の大きさを持つ有理数を、誤差を発生させずに表現します。

```
[System.Serializable]
public partial struct Rational:IComparable,IComparable<Rational>,
IEquatable<Rational>,Iformattable
```

継承：Object → ValueType → Rational
属性：SerializableAttribute
実装：IComparable, IComparable<Rational>, IEquatable<Rational>, IFormattable

注釈
Rational 型は任意の有理数を事実上（厳密な上限、下限は Rational の表現可能な値の範囲とメモリ消費を参照してください)、上限や下限なく表現できる構造体です。また、事実上、打切り誤差、情報落ち、桁落ちが起こらない内部表現をしており、循環小数の様な通常の浮動小数点型では正しく表現できない値を正しく表現できます。.NET Framework の浮動小数点型（Float、Double、Decimal)、と異なり MinValue、MaxValue、及び、Epsilon プロパティがありません。

> ⚠注意
> Rational 型は変更できません（Rational 型の可変性を参照してください）。上限および下限が無く、又、誤差を引き起こさない内部表現を使用している都合で、単純な足し算であっても大きな内部情報をコピーする可能性があります。コピーの時点でメモリ確保量が急増して OutOfMemoryException が スローされる可能性に注意してください。

Rational オブジェクトをインタンス化するには

Rational 型はいくつかの方法でインスタンス化することができます。

New キーワードを使用して Rational 型のコンストラクターに任意の整数型、または、浮動小数点型を使用することで初期化できます。

```
Rational rationalFromDouble = new Rational (179032.6541);
Console.WriteLine(rationalFromDouble);
Rational rationalFromInt64 = new Rational(934157136952);
Console.WriteLine(rationalFromInt64);
// The example displays the following output:
//     179032.6541
//     934157136952
```

Rational 型は変数割り当てと同様に宣言することができます。Rational 変数と割り当ての場合、割り当てたい値が格納された整数型、又は、浮動小数点型を代入します。 次の例では、割り当てを使用して、Int64 の値から Rational をインスタンス化します。

```
long longValue = 6315489358112;
Rational assignedFromLong = longValue;
Console.WriteLine(assignedFromLong);
// The example displays the following output:
//     6315489358112
```

10 進数または浮動小数点値を割り当てる場合も、整数型同様に割り当てる時に明示的なキャストは不要です。

```
Rational assignedFromDouble = 179032.6541;
Console.WriteLine(assignedFromDouble);
Rational assignedFromDecimal = 64312.65m;
Console.WriteLine(assignedFromDecimal);
// The example displays the following output:
//     179032.6541
//     64312.65
```

Byte 配列から初期化するメソッドを使用すると既存の整数型、及び、浮動小数点型の上限を超えた値で初期化することができます。new キーワードを使用して Rational 型のコンストラクターに、符号を示す bool 値、分母を示す Byte 配列、分子を示す Byte 配列を提供します。

```
bool sign=false;
byte[] numerator = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
byte[] denominator = {1};
Rational rational = new Rational (sign,numerator,denominator);
Console.WriteLine("The value of Rational is {0}.", rational);
// The example displays the following output:
//      The value of Rational is 4759477275222530853130.
```

文字列形式を Rational に変換する場合には Parse、又は、TryParse メソッドを使用します。

```
string positiveString = "913896812479936712554321120000000";
string negativeString = "-90315837410896312071002088037140000";
Rational posRational = 0;
Rational negRational = 0;
bool status=false;

try {
    posRational = Rational.Parse(positiveString);
    Console.WriteLine(posRational);
}
catch (FormatException)
{
    Console.WriteLine("Unable to convert the string '{0}' to a Rational value.",
positiveString);
}

(status, negRational) = Rational.TryParse(negativeString);
if (status)
   Console.WriteLine(negRational);
else
    Console.WriteLine("Unable to convert the string '{0}' to a Rational value.",
negativeString);

// The example displays the following output:
//     913896812479936712554321120000000
//     -90315837410896312071002088037140000
```

任意の整数、又は、浮動小数点に対し Rational 型の演算子やメソッドを使用する事で、Rational 型を初期化できます。次の例では UInt64.MaxValue を 3 乗しています。

```
Rational number = Math.Pow(UInt64.MaxValue, 3);
Console.WriteLine(number);
// The example displays the following output:
//       6277101735386680762814942322444851025767571854389858533375
```

初期化されてしていない Rational 型は Zero と等価になります。

Rational の値を操作するには

Rational インスタンスには他の整数型、または、浮動小数点型と同様に演算子を使用することができます。Rational 型は加算、減算、除算、乗算、減算、否定、単項マイナス演算子などの基本的な算術演算をオーバー ロードしており、整数型、または、浮動小数点型と同様に演算子を使用できます。また、比較演算子も同様に使用することができます。Rational 型は Add、Or、Xor、Left Shift、Right Shift、Ones Complement といったビット演算をサポートしています。Rational 型は、カスタム演算子をサポートしない言語に対して、算術演算を実行するための同等のメソッドも用意しています。 以下の Add、 Divide、 Multiply、 Negate、 Subtract、およびその他のいくつかのメソッドが該当します。
Rational 型で定義している多くのメンバーは浮動小数点型のメンバーに直接対応します。Rational 型では浮動小数点型にはないメンバーを定義しています。

- Sign：Rational の符号を返す。 （Math クラスに定義しています）
- Abs：Rational の絶対値を返す。（Math クラスに定義しています）
- DivRem：商と除算の剰余の両方が返す。 （Math クラスに定義しています）
- GreatestCommonDivisor： 2 つの Rational 値の最大公約数を返す。

ただし、Rational 型と同様の任意精度型である BigInteger 型と異なり、浮動小数点型では System.Math クラスで定義しているメンバーは WS.Theia.ExtremelyPrecise.Math クラスに定義しています。

Rational 型の可変性

次の例ではインスタンス化し Rational オブジェクトの値をインクリメントしています。

```
Rational number = Rational.Multiply(Int64.MaxValue, 3);
number++;
Console.WriteLine(number);
```

この例では、既存のオブジェクトの値を変更しているように見えますが、これは正しくありません。 Rational オブジェクトは値を変更ができません。 つまり内部的には、新しい Rational オブジェクトを生成し、インクリメント前の値より 1 つ大きな値を割り当てています。そして新しいオブジェクトを、呼び出し元に返しています。

> ⚠注意
> .NET Framework の他の数値型も変更できません。ただし、Rational 型は上限および下限が無く、又、誤差を引き起こさない内部表現を使用しているため、内部値が大きくなっている可能性があり、パフォーマンスに多大な影響が出る場合があります。例えばインクリメントをする場合、必要メモリの最大値は約 128GiB になります。

このプロセスは呼び出し元には透過的に行われ、パフォーマンスの低下を発生されます。特に、ループ内など操作を繰り返して行う状況で、Rational オブジェクトに大きな値が設定しているとと大幅なパフォーマンス低下を引き起こします。次の例では、SomeOperationSucceeds メソッドが成功する度に Rational オブジェクトの生成、内部データのコピー、Rational オブジェクトの破棄を、最大で 100 万回行う事となり大きなパフォーマンスの低下を引き起こします。

```
Rational number = Int64.MaxValue ^ 5;
int repetitions = 1000000;
// Perform some repetitive operation 1 million times.
for (int ctr = 0; ctr <= repetitions; ctr++)
{
    // Perform some operation. If it fails, exit the loop.
    if (! SomeOperationSucceeds()) break;
    // The following code executes if the operation succeeds.
    number++;
}
```

このような場合、整数型、又は、浮動小数点型の中間変数を使い、ループ終了後に Rational 変数に割り当てる事によりパフォーマンスを向上できます。

```
Rational number = Int64.MaxValue ^ 5;
int repetitions = 1000000;
int actualRepetitions = 0;
// Perform some repetitive operation 1 million times.
for (int ctr = 0; ctr <= repetitions; ctr++)
{
    // Perform some operation. If it fails, exit the loop.
    if (! SomeOperationSucceeds()) break;
    // The following code executes if the operation succeeds.
    actualRepetitions++;
}
number += actualRepetitions;
```

Rational の表現可能な値の範囲とメモリ消費

Rational 型では下図の内部表現を行います。

<table>
<tr><td rowspan="2">符号フラグ部</td><td>分子部（最大で 512Gbit）</td><td rowspan="2">無限大フラグ部</td></tr>
<tr><td>分母部（最大で 512Gbit）</td></tr>
</table>

Rational 型では下表の範囲と精度で数値を表現できます。

表現可能な値の範囲	±∞、±1/D～N/1（n/d で表現可能な範囲である事）、±0、NaN ※1
表現可能な精度	最大精度：1/D、最小精度：1/1
使用メモリ	インスタンスあたり 64Bit～512Gbit×2+8bit×2※2

※1　整数の分子を n、整数の分母を d、分子部の 512Gbit で表現可能な整数値の上限を N、分母部の 512Gbit で表現可能な整数値の上限を D、として表記。
※2　値を保持する為に必要なメモリであり、演算時はワーキングメモリを含めると瞬間的に必要領域が 1TB 程度に達する可能性がある。

上記の特性から小さな値であるほど精度が高く、大きな値であるほど精度が低くなります。ただし、地球の直径（mm）×π（2019 年 3 月 14 日に達成された約 31 兆桁とする）の演算に十二分な演算精度と提供できます。通常の演算では精度による問題は起こりません。

バイト配列と 16 進数の文字列の操作

Byte 配列を Rational 値に、又は、Rational 値を Byte 配列に変換するには順序を考慮する必要があります。Rational 構造体は、リトルエディアンでデータを格納しており、相互に変換する Byte 配列もリトルエディアンでなければなりません。次に示す例では、Rational の TobyteArray メソッドの結果を、Rational(Boolean,Byte[],Byte[])コンストラクターに渡しています。

```
Rational number = Math.Pow(Int64.MaxValue, 2);
Console.WriteLine(number);

// Write the BigInteger value to a byte array.
var bytes = number.ToByteArray();

// Display the byte array.
foreach (byte byteValue in bytes.Numerator)
    Console.Write("0x{0:X2} ", byteValue);
Console.WriteLine();

// Restore the Rational value from a Byte array.
Rational newNumber = new Rational (bytes.Sign ,bytes.Numerator,
bytes.Denominator);
Console.WriteLine(newNumber);
// The example displays the following output:
//       85070591730234615847396907784232501249
//       0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF 0xFF 0xFF 0xFF 0xFF
0xFF 0xFF 0x3F
//
//       85070591730234615847396907784232501249
```

Rational オブジェクトをインスタンス化するに当たり整数値のバイト配列を利用することができます。BitConverter.GetBytes メソッドの結果を分子、分母を 1 として、Rational(Boolean,Byte[],Byte[])コンストラクターを呼び出す事で等価な値にすることができます。次の例では、Int16 の値を示す Byte 配列から Rational オブジェクトを初期化しています。

```
short originalValue = 30000;
Console.WriteLine(originalValue);

// Convert the Int16 value to a byte array.
byte[] bytes = BitConverter.GetBytes(originalValue);

// Display the byte array.
foreach (byte byteValue in bytes)
    Console.Write("0x{0} ", byteValue.ToString("X2"));
Console.WriteLine();

// Pass byte array to the Rational constructor.
Rational number = new Rational (false,bytes,new byte[]{ 1 });
Console.WriteLine(number);
// The example displays the following output:
//       30000
//       0x30 0x75
//       30000
```

Rational 型では BigInteger 型と異なり符号フラグとして別に持っており、2 の補数表現を使って定義することができません。

コンストラクター

Rational(Boolean)	Boolean 値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(Boolean,Byte[],Byte[])	バイト配列の値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(Decimal)	Decimal 値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(Double)	倍精度浮動小数点値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(Int32)	32 ビット符号付き整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(Int64)	64 ビット符号付き整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(Single)	単精度浮動小数点値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(UInt32)	32 ビット符号なし整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(UInt64)	64 ビット符号なし整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。

プロパティ

Accuracy	演算精度。無理数を演算する際の打切り桁数、丸目の既定値として使用します。
IsEven	現在の Rational オブジェクトの値が偶数かどうかを示します。
IsOne	現在の Rational オブジェクトの値が One かどうかを示します。
IsPowerOfTwo	現在の Rational オブジェクトの値が 2 の累乗かどうかを示します。
IsZero	現在の Rational オブジェクトの値が Zero かどうかを示します。
MinusOne	負の 1 (-1) を表す値を取得します。
One	正の 1 (1) を表す値を取得します。
Zero	0 (ゼロ) を表す値を取得します。
NaN	非数(NaN) の値を表します。 このフィールドは定数です。
NegativeInfinity	負の無限大を表します。 このフィールドは定数です。
PositiveInfinity	正の無限大を表します。 このフィールドは定数です。

メソッド

Add(Rational,Rational)	2 つの Rational 値を加算し、その結果を返します。
BitwiseAnd(Rational,Rational)	2 つの BigInteger 値に対し、ビットごとの And 演算を実行します。
BitwiseOr(Rational, Rational)	2 つの Rational 値に対し、ビットごとの Or 演算を実行します。
Compare(Rational,Rational)	2 つの Rational 値を比較し、1 番目の値が 2 番目の値よりも小さいか、同じか、または大きいかを示す整数を返します。
Clone()	Rational 値を複製します。
CompareTo(decimal)	このインスタンスと 10 進数を比較し、このインスタンスの値が 10 進数の値よりも小さいか、同じか、または大きいかを示す整数を返します。
CompareTo(double)	このインスタンスと 倍精度浮動小数点数を比較し、このインスタンスの値が 倍精度浮動小数点数の値よりも小さいか、同じか、または大きいかを示す整数を返します。
CompareTo(long)	このインスタンスと符号付き 64 ビット整数を比較し、このインスタンスの値が符号付き 64 ビット整数の値よりも小さいか、同じか、または大きいかを示す整数を返します。
CompareTo(object)	このインスタンスと指定したオブジェクトを比較し、このインスタンスの値が指定したオブジェクトの値よりも小さいか、同じか、または大きいかを示す整数を返します。
CompareTo(Rational)	このインスタンスと Rational を比較し、このインスタンスの値が Rational の値よりも小さいか、同じか、または大きいかを示す整数を返します。
CompareTo(ulong)	このインスタンスと符号なし 64 ビット整数を比較し、このインスタンスの値が符号なし 64 ビット整数の値よりも小さいか、同じか、または大きいかを示す整数を返します。
Decrement(Rational)	Rational 値を 1 だけデクリメントします。
Divide(Rational,Rational)	一方の Rational 値をもう一方の値で除算し、その結果を返します。
Equals(decimal)	現在のインスタンスの値と 10 進数の値が等しいかどうかを示す値を返します。

Equals(double)	現在のインスタンスの値と倍精度浮動小数点数の値が等しいかどうかを示す値を返します。
Equals(long)	現在のインスタンスの値と符号付き 64 ビット整数の値が等しいかどうかを示す値を返します。
Equals(object)	現在のインスタンスの値と指定されたオブジェクトの値が等しいかどうかを示す値を返します。
Equals(Rational)	現在のインスタンスの値と Rational の値が等しいかどうかを示す値を返します。
Equals(ulong)	現在のインスタンスの値と符号無し 64 ビット整数の値が等しいかどうかを示す値を返します。
Equals(Rational,Rational)	2 つの Rational オブジェクトの値が等しいかどうかを示す値を返します。
FromOACurrency(long cy)	OLE オートメーション通貨値を格納している指定した 64 ビット符号付き整数を、それと等価の Rational 値に変換します。
GetHashCode()	現在の Rational オブジェクトのハッシュ コードを返します。
GetTypeCode()	TypeCode 値型の Object を返します。
GreatestCommonDivisor (Rational,Rational)	2 つの Rational 値の最大公約数を求めます。
Increment(Rational)	Rational 値を 1 だけインクリメントします。
IsInfinity(Rational)	指定した数値が負または正の無限大と評価されるかどうかを示す値を返します。
IsNaN(Rational)	指定した値が非数値（NaN）かどうかを示す値を返します。
IsNegativeInfinity(Rational)	指定した数値が負の無限大と評価されるかどうかを示す値を返します。
IsPositiveInfinity(Rational)	指定した数値が正の無限大と評価されるかどうかを示す値を返します。
LeftShift(Rational,int)	指定されたビット数だけ Rational 値を左にシフトします。
Mod(Rational,Rational)	指定された 2 つの Rational 値の除算の結果生じた剰余を返します。
ModPow(Rational,Rational,Rational)	ある数値を別の数値で累乗し、それをさらに別の数値で割った結果生じた剰余を求めます。
Multiply(Rational,Rational)	2 つの Rational 値の積を返します。

Negate(Rational)	指定された Rational 値を否定（負数化）します。
OnesComplement(Rational)	Rational 値のビットごとの 1 の補数を返します。
Parse(String)	数値の文字列形式を、それと等価の Rational に変換します。
Parse(String,NumberStyles)	指定のスタイルで表現された数値の文字列形式を、それと等価な Rational に変換します。
Parse(String,IFormatProvider)	指定されたカルチャ固有の書式で表現された文字列形式の数値を、それと等価の Rational に変換します。
Parse(String,NumberStyles,IFormatProvider)	指定したスタイルおよびカルチャ固有の書式の数値の文字列形式を、それと等価の BigInteger に変換します。
Plus(Rational)	Rational オペランドの値を返します。 オペランドの符号は変更されません。
RightShift(Rational,int)	指定されたビット数だけ Rational 値を右にシフトします。
Subtract(Rational,Rational)	ある Rational 値を別の値から減算し、その結果を返します。
ToByte(Rational)	指定した Rational の値を、等価の 8 ビット符号なし整数に変換します。
ToByteArray()	Rational 値をバイト配列に変換します。
ToDouble(Rational)	指定した Rational の値を、それと等価の倍精度浮動小数点数に変換します。
ToInt16(Rational)	指定した Rational の値を、等価の 16 ビット符号付き整数に変換します。
ToInt32(Rational)	指定した Rational の値を、等価の 32 ビット符号付き整数に変換します。
ToInt64(Rational)	指定した Rational の値を、等価の 64 ビット符号付き整数に変換します。
ToOACurrency(Rational)	指定した Rational 値を、64 ビット符号付き整数に格納されるそれと等価の OLE オートメーション通貨値に変換します。
ToSByte(Rational)	指定した Rational の値を、等価の 8 ビット符号付き整数に変換します。
ToSingle(Rational)	指定した Rational の値を、それと等価の単精度浮動小数点数に変換します。

ToString()	現在の BigInteger オブジェクトの数値を等価の文字列形式に変換します。
ToString(IFormatProvider)	指定されたカルチャ固有の書式情報を使用して、現在の Rational オブジェクトの数値をそれと等価の文字列形式に変換します。
ToString(String)	指定された書式を使用して、現在の Rational オブジェクトの数値をそれと等価な文字列形式に変換します。
ToString(String,IFormatProvider)	指定された書式とカルチャ固有の書式情報を使用して、現在の Rational オブジェクトの数値をそれと等価の文字列形式に変換します。
ToUInt16(Rational)	指定した Rational の値を、等価の 16 ビット符号付き整数に変換します。
ToUInt32(Rational)	指定した Rational の値を、等価の 32 ビット符号付き整数に変換します。
ToUInt64(Rational)	指定した Rational の値を、等価の 64 ビット符号付き整数に変換します。
ToDecimal(Rational)	指定した Rational の値を、等価の 10 進数に変換します。
TryParse(String)	数値の文字列形式を対応する Rational 表現に変換できるかどうかを試行し、変換に成功したかどうかを示す値を返します。
TryParse(String,NumberStyles style,IFormatProvider)	数値の文字列形式を対応する Rational 表現に変換できるかどうかを試行し、変換に成功したかどうかを示す値を返します。
Xor(Rational,Rational)	2 つの Rational 値に対し、ビットごとの排他的 Or (XOr) 演算を実行します。

演算子

Addition (Rational,Rational)	指定された 2 つの Rational オブジェクトの値を加算します。
BitwiseAnd (Rational,Rational)	2 つの BigInteger 値に対し、ビットごとの And 演算を実行します。
BitwiseOr (Rational,Rational)	2 つの Rational 値に対し、ビットごとの Or 演算を実行します。
Decrement (Rational)	Rational 値を 1 だけデクリメントします。
Division (Rational,Rational)	整数除算を使用して、指定された Rational 値をもう 1 つの指定された Rational 値で除算します。
Equality (Rational,Rational)	2 つの Rational オブジェクトの値が等しいかどうかを示す値を返します。
ExclusiveOr (Rational,Rational)	2 つの Rational 値に対し、ビットごとの排他的 Or (XOr) 演算を実行します。
Explicit(Rational to Byte)	Rational オブジェクトから byte 値への明示的な変換を定義します。
Explicit(Rational to SByte)	Rational オブジェクトから sbyte 値への明示的な変換を定義します。
Explicit(Rational to Int32)	Rational オブジェクトから int 値への明示的な変換を定義します。
Explicit(Rational to UInt32)	Rational オブジェクトから uint 値への明示的な変換を定義します。
Explicit(Rational to Int16)	Rational オブジェクトから short 値への明示的な変換を定義します。
Explicit(Rational to UInt16)	Rational オブジェクトから ushort 値への明示的な変換を定義します。
Explicit(Rational to Int64)	Rational オブジェクトから long 値への明示的な変換を定義します。
Explicit(Rational to UInt64)	Rational オブジェクトから ulong 値への明示的な変換を定義します。
Explicit(Rational to Single)	Rational オブジェクトから float 値への明示的な変換を定義します。

Explicit(Rational to Double)	Rational オブジェクトから double 値への明示的な変換を定義します。
Explicit(Rational to Boolean)	Rational オブジェクトから bool 値への明示的な変換を定義します。
Explicit(Rational to Decimal)	Rational オブジェクトから decimal 値への明示的な変換を定義します。
GreaterThan (Rational,Rational)	Rational 値がもう 1 つの Rational 値より大きいかどうかを示す値を返します。
GreaterThanOrEqual (Rational,Rational)	Rational 値がもう 1 つの Rational 値以上かどうかを示す値を返します。
Implicit(Byte to Rational)	byte 値から BigInteger 値への暗黙的な変換を定義します。
Implicit(SByte to Rational)	sbyte 値から BigInteger 値への暗黙的な変換を定義します。
Implicit(Int32 to Rational)	int 値から BigInteger 値への暗黙的な変換を定義します。
Implicit(UInt32 to Rational)	uint 値から BigInteger 値への暗黙的な変換を定義します。
Implicit(Int16 to Rational)	short 値から BigInteger 値への暗黙的な変換を定義します。
Implicit(UInt16 to Rational)	ushort 値から BigInteger 値への暗黙的な変換を定義します。
Implicit(Int64 to Rational)	long 値から BigInteger 値への暗黙的な変換を定義します。
Implicit(UInt64 to Rational)	ulong 値から BigInteger 値への暗黙的な変換を定義します。
Implicit(Single to Rational)	float 値から BigInteger 値への暗黙的な変換を定義します。
Implicit(Double to Rational)	double 値から BigInteger 値への暗黙的な変換を定義します。
Implicit(Boolean to Rational)	bool 値から BigInteger 値への暗黙的な変換を定義します。
Implicit(Decimal to Rational)	decimal 値から BigInteger 値への暗黙的な変換を定義します。
Increment (Rational)	Rational 値を 1 だけインクリメントします。

Inequality (Rational,Rational)	2 つの Rational オブジェクトの値が異なるかどうかを示す値を返します。
LeftShift (Rational,Int32)	指定されたビット数だけ Rational 値を左にシフトします。
LessThan (Rational,Rational)	Rational 値がもう 1 つの Rational 値より小さいかどうかを示す値を返します。
LessThanOrEqual (Rational,Rational)	Rational 値がもう 1 つの Rational 値以下かどうかを示す値を返します。
Modulus (Rational,Rational)	指定された 2 つの Rational 値の除算の結果生じた剰余を返します。
Multiply (Rational,Rational)	指定された 2 つの Rational 値を乗算します。
OnesComplement (Rational)	Rational 値のビットごとの 1 の補数を返します。
RightShift (Rational,Int32)	指定されたビット数だけ Rational 値を右にシフトします。
Subtraction (Rational,Rational)	Rational 値をもう 1 つの Rational 値から減算します。
UnaryNegation (Rational)	指定された Rational 値を否定（負数化）します。
UnaryPlus (Rational)	Rational オペランドの値を返します。 オペランドの符号は変更されません。

適用対象

.NET Core

2.0

.NET Framework

4.6.1 2

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational Constructors

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll

オーバーロード

Rational(Boolean)	Boolean 値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(Boolean,Byte[],Byte[])	バイト配列の値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(Decimal)	Decimal 値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(Double)	倍精度浮動小数点値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(Int32)	32 ビット符号付き整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(Int64)	64 ビット符号付き整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(Single)	単精度浮動小数点値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(UInt32)	32 ビット符号なし整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。
Rational(UInt64)	64 ビット符号なし整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。

Rational(Boolean)

Boolean 値を使用して、Rational 構造体の新しいインスタンスを初期化します。

```
public Rational(Boolean value);
```

パラメーター

value　Boolean

Boolean 値(true=One、false=Zero と認識します)。

例

次の例では Boolean 値を使用して Rational オブジェクトを初期化しています。True を One、False を Zero として扱います。

```
Rational number = new Rational (true);
Console.WriteLine("The value of number is {0}.", number);
// The example displays the following output:
//       The value of number is 1.
```

Rational(Boolean,Byte[],Byte[])

> ⚠重要
> この API は CLS 準拠ではありません。

バイト配列の値を使用して、Rational 構造体の新しいインスタンスを初期化します。

```
public Rational(bool sign,byte[] numerator,byte[] denominator) {
```

パラメーター
sign　Boolean
符号を示す値（false＝プラス、true=マイナス）。
numerator　Byte[]
分子を表すリトルエンディアン順に格納されたバイト値の配列。
denominator　Byte[]
分母を表すリトルエンディアン順に格納されたバイト値の配列。

例外
numerator、又は、denominator は null です。

例

次の例では{1}の 1 要素からなる Byte 配列を分母、{5,4,3,2,1}の 5 要素からなる Byte 配列を分子の正の値として、Rational オブジェクトを初期化しています。後に Rational オブジェクトの値をコンソールに 10 進数で表示しています。このオーバーロードで作成した Rational オブジェクトの値は 4328719365 になります。

```
bool sign=false;
byte[] numerator = { 5, 4, 3, 2, 1 };
byte[] denominator = { 1 };
Rational number = new Rational(sign, numerator, denominator);
Console.WriteLine("The value of number is {0}.", number);
// The example displays the following output:
//       The value of number is 4328719365.
```

次の例では負の値で Rational オブジェクトを初期化しています。正負の違いは符号フラグとして示され分母、及び、分子を示す Byte 配列は全く同じである事に注意してください。

```
bool sign=true;
byte[] numerator = { 5, 4, 3, 2, 1 };
byte[] denominator = { 1 };
Rational number = new Rational(sign, numerator, denominator);
Console.WriteLine("The value of number is {0}.", number);
// The example displays the following output:
//       The value of number is -4328719365.
```

注釈

numerator パラメーター、denominator パラメーターは最上位バイトに数値の最下位を並べるリトルエンディアン順でなければなりません。たとえば、数値 1,000,000,000,000 は、次の表に示すように表されます。

数値の 16 進数文字列	E8D4A51000
バイト配列（前方のインデックスが最も低い）	00 10 A5 D4 E8 00

BitConverter.GetBytes、BigInteger.ToByteArray 等、殆どの Byte 配列に変換するメソッドは、リトルエンディアン順で値を格納して Byte 配列を返します。ただし、numerator パラメーター、denominator パラメーター共に、最上位ビットと常に符号無し整数として扱います。その結果、負数が正数として解釈される場合があります。通常、numerator パラメーター、denominator パラメーターの Byte 配列は次の方法で生成されます。

- Rational.ToByteArray メソッドの呼び出しにより生成する。このメソッドでは符号フラグ、分子配列、分母配列、無限大フラグのタプル型として生成し、分子配列、分母配列共に常に正の値を持っており負数が正数として解釈する事はありません。
- BitConverter.GetBytes メソッドにパラメーターとして符号付き整数を渡して生成する。符号付整数では最上位ビットを符号ビットとして負数を 2 の補数で表現しますが、numerator パラメーター、denominator パラメーターでは正数として解釈します。Rational(Int32)、Rational(Int64)コンストラクターを使用する事で回避可能です。
- BitConverter.GetBytes メソッドにパラメーターとして符号なし整数を渡して生成する。符号なし整数では負数が存在しない為、numerator パラメーター、denominator パラメーターは正しく解釈することができます。
- 動的、又は、静的に生成した Byte 配列を生成する。この場合、正しく解釈されるには numerator パラメーター、denominator パラメーター共に絶対値を設定し、符号は sign パラメーターに設定しなければなりません。
- BigInteger.ToByteArray メソッドの呼び出しにより生成する。BigInteger では最上位ビットを符号ビットとして負数を 2 の補数で表現しますが、numerator パラメーター、denominator パラメーターでは正数として解釈します。符号は sign パラメーターに設定して、絶対値を取得してから ToByteArray メソッドを呼び出して Byte 配列を生成してください。

numerator パラメーターが空配列の場合、Rational.Zero として初期化します。denominator パラメーターが空配列、又は、0 のみで構成される配列の場合、Rational.NaN として初期化します。numerator パラメーター、denominator パラメーターが null の場合は、ArgumentNullException が発生します。

> ⚠注意
> denominator パラメーターが空配列、又は、0 のみで構成される配列の場合、Rational.NaN に初期化する仕様については、将来的に変更する可能性があります。NaN 値が必要な場合は、Rational.NaN プロパティを使用してください。

また、ToByteArray()も参照してください。

Rational(Decimal)

Decimal 値を使用して、Rational 構造体の新しいインスタンスを初期化します。

```
public Rational(decimal value);
```

パラメーター

value　Decimal

10 進数値。

例

次の例では、Rational オブジェクトを Rational(Decimal)コンストラクターを初期化しています。配列に定義した Decimal 値をそれぞれ Rational(Decimal)コンストラクターに渡しています。

```
decimal[] decimalValues = { -1790.533m, -15.1514m, 18903.79m,
9180098.003m };
foreach (decimal decimalValue in decimalValues)
{
    Rational number = new Rational(decimalValue);
    Console.WriteLine("Instantiated Rational value {0} from the Decimal value
{1}.",
                      number, decimalValue);
}
// The example displays the following output:
//       Instantiated Rational value -1790.533 from the Decimal value -1790.533.
//       Instantiated Rational value -15.1514. from the Decimal value -15.1514.
//       Instantiated Rational value 18903.79 from the Decimal value 18903.79.
//       Instantiated Rational value 9180098.003 from the Decimal value
9180098.003.
```

注釈

このコンストラクターを明示的に呼び出して割り当てる事と、Rational 変数に Decimal 値を割り当てる事は等価です。

Rational(Double)

倍精度浮動小数点値を使用して、Rational 構造体の新しいインスタンスを初期化します。

```
public Rational(double value);
```

パラメーター

value Double

倍精度浮動小数点数値。

例

Double 値を使用して、Rational 構造体の新しいインスタンスを初期化します。Double 型には大きな値が割り当てられている為、足し算を行った際に、情報落ちが起こり、変更が反映されていませんが、Rational 型では変更が反映されています。

```
// Create a Rational from a large double value.
double doubleValue = -6e20;
Rational rationalValue = new Rational (doubleValue);
Console.WriteLine("Original Double value: {0:N0}", doubleValue);
Console.WriteLine("Original Rational value: {0:N0}", rationalValue);
// Increment and then display both values.
doubleValue++;
rationalValue += Rational.One;
Console.WriteLine("Incremented Double value: {0:N0}", doubleValue);
Console.WriteLine("Incremented Rational value: {0:N0}", rationalValue);
// The example displays the following output:
//       Original Double value: -600,000,000,000,000,000,000
//       Original Rational value: -600,000,000,000,000,000,000
//       Incremented Double value: -600,000,000,000,000,000,000
//       Incremented Rational value: -599,999,999,999,999,999,999
```

注釈

Rational 型には有効桁数の上限が事実上無限の為、Double 型と違いデータの損失を起こさずに演算することができます。このコンストラクターを明示的に呼び出して割り当てる事と、Rational 変数に Double 値を割り当てる事は等価です。

Rational(Int32)

32 ビット符号付き整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。

```
public Rational(int value);
```

パラメーター

value　Int32

32 ビット符号付き整数値。

例

32 ビット符号付き整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。32 ビット符号付き整数から Rational 変数に代入し暗黙的な変換をした結果と比較しています。その結果は明示的なインスタンス化と暗黙的な変換の結果は等価です。

```
int[] integers = { Int32.MinValue, -10534, -189, 0, 17, 113439,
                   Int32.MaxValue };
Rational constructed, assigned;

foreach (int number in integers)
{
    constructed = new Rational(number);
    assigned = number;
    Console.WriteLine("{0} = {1}: {2}", constructed, assigned,
                      constructed.Equals(assigned));
}
// The example displays the following output:
//       -2147483648 = -2147483648: True
//       -10534 = -10534: True
//       -189 = -189: True
//       0 = 0: True
//       17 = 17: True
//       113439 = 113439: True
//       2147483647 = 2147483647: True
```

注釈

このコンストラクターを明示的に呼び出して割り当てる事と、Rational 変数に Int32 値を割り当てる事は等価です。Byte、Int16、SByte、UInt16 のコンストラクターはありません。ただし、Int32 型の 8 ビット、16 ビットの符号付き、符号無し整数の 32 ビット符号付整数への暗黙的な変換を利用して、このコンストラクターが呼び出されます。その為このコンストラクターに渡されるデータ型は Byte、Int16、SByte、UInt16、Int32 のいずれかです。

Rational(Int64)

64 ビット符号付き整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。

```
public Rational(long value);
```

パラメーター

value　Int64

64 ビット符号付き整数値。

例

64 ビット符号付き整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。64 ビット符号付き整数から Rational 変数に代入し暗黙的な変換をした結果と比較しています。その結果は明示的なインスタンス化と暗黙的な変換の結果は等価です。

```
long[] longs = { Int64.MinValue, -10534, -189, 0, 17, 113439,
                 Int64.MaxValue };
Rational constructed, assigned;

foreach (long number in longs)
{
    constructed = new Rational(number);
    assigned = number;
    Console.WriteLine("{0} = {1}: {2}", constructed, assigned,
                      constructed.Equals(assigned));
}
// The example displays the following output:
//       - 9223372036854775808 = - 9223372036854775808: True
//       -10534 = -10534: True
//       -189 = -189: True
//       0 = 0: True
//       17 = 17: True
//       113439 = 113439: True
//       9223372036854775807 = 9223372036854775807: True
```

注釈

このコンストラクターを明示的に呼び出して割り当てる事と、Rational 変数に Int64 値を割り当てる事は等価です。

Rational(Single)

単精度浮動小数点値を使用して、Rational 構造体の新しいインスタンスを初期化します。

```
public Rational(float value);
```

パラメーター

value　Single

単精度浮動小数点数値。

例

Single 値を使用して、Rational 構造体の新しいインスタンスを初期化します。Single 型には大きな値が割り当てられている為、足し算を行った際に、情報落ちが起こり、変更が反映されていませんが、Rational 型では変更が反映されています。

```
// Create a BigInteger from a large negative Single value
float negativeSingle = Single.MinValue;
Rational negativeNumber = new Rational (negativeSingle);

Console.WriteLine(negativeSingle.ToString("N0"));
Console.WriteLine(negativeNumber.ToString("N0"));

negativeSingle++;
negativeNumber++;

Console.WriteLine(negativeSingle.ToString("N0"));
Console.WriteLine(negativeNumber.ToString("N0"));
// The example displays the following output:
//       -340,282,300,000,000,000,000,000,000,000,000,000,000
//       -340,282,346,638,528,859,811,704,183,484,516,925,440
//       -340,282,300,000,000,000,000,000,000,000,000,000,000
//       -340,282,346,638,528,859,811,704,183,484,516,925,439
```

注釈

Rational 型には有効桁数の上限が事実上無限の為、Single 型と違いデータの損失を起こさずに演算することができます。このコンストラクターを明示的に呼び出して割り当てる事と、Rational 変数に Single 値を割り当てる事は等価です。

Rational(UInt32)

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替：WS.Theia.ExtremelyPrecise.Rational.Rational(Int64)

32 ビット符号なし整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。

```
public Rational(uint value);
```

パラメーター
value　UInt32
32 ビットの符号なし整数値。

例

32 ビット符号なし整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。32 ビット符号なし整数から Rational 変数に代入し暗黙的な変換をした結果と比較しています。その結果は明示的なインスタンス化と暗黙的な変換の結果は等価です。

```
uint[] unsignedValues = { 0, 16704, 199365, UInt32.MaxValue };
foreach (uint unsignedValue in unsignedValues)
{
    Rational constructedNumber = new Rational (unsignedValue);
    Rational assignedNumber = unsignedValue;
    if (constructedNumber.Equals(assignedNumber))
        Console.WriteLine("Both methods create a Rational whose value is
{0:N0}.",
                          constructedNumber);
    else
        Console.WriteLine("{0:N0} ≠ {1:N0}", constructedNumber,
assignedNumber);

}
// The example displays the following output:
//       Both methods create a Rational whose value is 0.
//       Both methods create a Rational r whose value is 16,704.
//       Both methods create a Rational whose value is 199,365.
//       Both methods create a Rational whose value is 4,294,967,295.
```

注釈

このコンストラクターを明示的に呼び出して割り当てる事と、Rational 変数に UInt32 値を割り当てる事は等価です。

Rational(UInt64)

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替：WS.Theia.ExtremelyPrecise.Rational.Rational(Double)

64 ビット符号なし整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。

```
public Rational(uint value);
```

パラメーター
value　UInt64
64 ビットの符号なし整数値。

例

64 ビット符号なし整数値を使用して、Rational 構造体の新しいインスタンスを初期化します。次の例では UInt64.MaxValue を設定しています。

```
ulong unsignedValue = UInt64.MaxValue;
Rational number = new Rational (unsignedValue);
Console.WriteLine(number.ToString("N0"));
// The example displays the following output:
//           18,446,744,073,709,551,615
```

注釈

このコンストラクターを明示的に呼び出して割り当てる事と、Rational 変数に UInt64 値を割り当てる事は等価です。

適用対象

.NET Core

2.0

.NET Framework

4.6.1 2

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational. Accuracy Property

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
演算精度を取得設定します。無理数の演算打切り、ToString の変換打切り等に使用します。

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

プロパティ値
Int32
演算精度を示す値。

注釈

小数点以下の 10 進数での桁数として解釈します。無理数の演算打切り、ToString の変換打切り等、Rational オブジェクトの多くの挙動に関わる値です。Rational オブジェクトが無い事を確認した上で変更してください。また、スレッドセーフではありません。加えて、E や PI といった、演算負荷の大きい数値の更新が行われ長時間応答が無くなります。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.IsEven Property

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
現在の Rational オブジェクトの値が偶数かどうかを示します。

```
public bool IsEven { get; }
```

プロパティ値
Boolean
Rational オブジェクトの値が偶数の場合は true。それ以外の場合は false。

注釈

このプロパティは、Rational 値が 2 で割り切れるかを判定します。これは、次の式に相当します。

```
value % 2 == 0;
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.IsOne Property

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
現在の Rational オブジェクトの値が One かどうかを示します。

```
public bool IsOne { get; }
```

プロパティ値
Boolean
Rational オブジェクトの値が One の場合は true。それ以外の場合は false。

注釈

このプロパティは、他の"1"との比較方法と比べてパフォーマンスが良くなります。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.IsPowerOfTow Property

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
現在の Rational オブジェクトの値が 2 の累乗かどうかを示します。

```
public bool IsPowerOfTow { get; }
```

プロパティ値
Boolean
Rational オブジェクトの値が 2 の累乗の場合は true。それ以外の場合は false。

注釈

このプロパティは、Rational 値の分母が 1、かつ、分子が単一ビットのみが 1 となった時に true を返します。つまり 1（2^0）、又は、任意の大きさの 2 の累乗となった時に true となります。それ以外の場合は false となります。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.IsZero Property

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
現在の Rational オブジェクトの値が Zero かどうかを示します。

```
public bool IsZero { get; }
```

プロパティ値
Boolean
Rational オブジェクトの値が Zero の場合は true。それ以外の場合は false。

注釈

このプロパティは、他の”0”との比較方法と比べてパフォーマンスが良くなります。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational. MinusOne Property

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
負の 1（-1）を表す値を取得します。

```
public WS.Theia.ExtremelyPrecise.Rational MinusOne { get; }
```

プロパティ値
Boolean
値が負の 1（-1）である Rational オブジェクト。

注釈

MinusOne プロパティは Rational 変数と-1 を比較する場合、Rational 変数を-1 で初期化する場合に使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.NaN Property

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
数（NaN）を表す値を取得します。

```
public WS.Theia.ExtremelyPrecise.Rational NaN { get; }
```

プロパティ値
Boolean
値が非数（NaN）である Rational オブジェクト。

注釈

NaN は 0 を 0 で除算を行った場合等、数値として定義できない場合に返却されます。0 以外の被除数を 0 で除算した場合は、被除数と除数の符号により、PositiveInfinity、または、NegativeInfinity となります。

```
Rational zero = 0.0;
Console.WriteLine("{0} / {1} = {2}", zero, zero, zero/zero);
// The example displays the following output:
//               0 / 0 = NaN
```

また、NaN と他の値の演算子、又は、メソッドの操作結果は NaN となります。

```
Rational nan1 = Rational.NaN;

Console.WriteLine("{0} + {1} = {2}", 3, nan1, 3 + nan1);
Console.WriteLine("Abs({0}) = {1}", nan1,
WS.Theia.ExtremelyPrecise.Math.Abs(nan1));
// The example displays the following output:
//          3 + NaN = NaN
//          Abs(NaN) = NaN
```

一般的に NaN に演算子は使用する事ができませんが、比較演算子（Equals、CompereTo 等）を使用することができます。次の例では各演算子と値による比較結果を示しています。

```
using System;

public class Example
{
    public static void Main()
    {
        Console.WriteLine("NaN == NaN: {0}", Rational.NaN ==
Rational.NaN);
        Console.WriteLine("NaN != NaN: {0}", Rational.NaN != Rational.NaN);
        Console.WriteLine("NaN.Equals(NaN): {0}",
Rational.NaN.Equals(Rational.NaN));
        Console.WriteLine("! NaN.Equals(NaN): {0}", !
Rational.NaN.Equals(Rational.NaN));
        Console.WriteLine("IsNaN: {0}", Rational.IsNaN(Rational.NaN));

        Console.WriteLine("¥nNaN > NaN: {0}", Rational.NaN >
Rational.NaN);
        Console.WriteLine("NaN >= NaN: {0}", Rational.NaN >=
Rational.NaN);
        Console.WriteLine("NaN < NaN: {0}", Rational.NaN < Rational.NaN);
```

```
        Console.WriteLine("NaN < 100.0: {0}", Rational.NaN < 100.0);
        Console.WriteLine("NaN <= 100.0: {0}", Rational.NaN <= 100.0);
        Console.WriteLine("NaN >= 100.0: {0}", Rational.NaN > 100.0);
        Console.WriteLine("NaN.CompareTo(NaN): {0}",
Rational.NaN.CompareTo(Rational.NaN));
        Console.WriteLine("NaN.CompareTo(100.0): {0}",
Rational.NaN.CompareTo(100.0));
        Console.WriteLine("(100.0).CompareTo(Rational.NaN): {0}",new
Rational(100.0).CompareTo(Rational.NaN));
    }
}
// The example displays the following output:
//          NaN == NaN: False
//          NaN != NaN: True
//          NaN.Equals(NaN): True
//          ! NaN.Equals(NaN): False
//          IsNaN: True
//
//          NaN > NaN: False
//          NaN >= NaN: False
//          NaN < NaN: False
//          NaN < 100.0: False
//          NaN <= 100.0: False
//          NaN >= 100.0: False
//          NaN.CompareTo(NaN): 0
//          NaN.CompareTo(100.0): -1
//          (100.0).CompareTo(Rational.NaN): 1
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.NegativeInfinity Property

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
負の無限大を表します。このプロパティは定数です。

```
public WS.Theia.ExtremelyPrecise.Rational NegativeInfinity { get; }
```

プロパティ値
Rational
値が負の無限大である Rational オブジェクト。

注釈

この定数は、正の数を負の 0 で除算、又は、負の数を正の 0 で除算した結果です。負の無限大かどうかを判定するには IsNegativeInfinity プロパティを使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.One Property

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
正の 1（1）を表す値を取得します。

```
public WS.Theia.ExtremelyPrecise.Rational One { get; }
```

プロパティ値
Boolean
値が正の 1（1）である Rational オブジェクト。

注釈

One プロパティは Rational 変数と 1 を比較する場合、Rational 変数を 1 で初期化する場合に使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.PositiveInfinity Property

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
正の無限大を表します。このプロパティは定数です。

```
public WS.Theia.ExtremelyPrecise.Rational PositiveInfinity { get; }
```

プロパティ値
Rational
値が正の無限大である Rational オブジェクト。

注釈

この定数は、正の数を正の 0 で除算、又は、負の数を負の 0 で除算した結果です。正の無限大かどうかを判定するには IsPositiveInfinity プロパティを使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.Zero Property

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
正の 0（ゼロ）を表す値を取得します。

```
public WS.Theia.ExtremelyPrecise.Rational Zero { get; }
```

プロパティ値
Boolean
値が正の 0（ゼロ）である Rational オブジェクト。

注釈

One プロパティは Rational 変数と 0 を比較する場合、Rational 変数を 0 で初期化する場合に使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.Add(Rational, Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational 値を加算し、その結果を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Add(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター
left　Rational
加算する 1 番目の値。
right　Rational
加算する 2 番目の値。

戻り値
Rational
left と right の合計。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値を加算する代替メソッド。Rational 値を加算して変数に割り当てる時は次の例の様に使用する。

```
// The statement:
//       Rational number = Int64.MaxValue + Int32.MaxValue;
// produces compiler error CS0220: The operation overflows at compile time in
checked mode.
// The alternative:
Rational number = Rational.Add(Int64.MaxValue, Int32.MaxValue);
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.BitwiseAnd(Rational, Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational 値に対し、ビットごとの And 演算を実行します。

```
public static WS.Theia.ExtremelyPrecise.Rational
BitwiseAnd(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター
left　Rational
And 演算する最初の値。
right　Rational
And 演算する 2 番目の値。

戻り値
Rational
ビットごとの And 演算の結果。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値を加算する代替メソッド。left と right のビットと戻り値のビットの関係性は下表の様になります。

left 内のビット値	right 内のビット値	戻り値内のビット値
0	0	0
1	0	0
1	1	1
0	1	0

BitwiseAnd メソッドは、次の例の様に使用します。

```
Rational number1 = Rational.Add(Int64.MaxValue, Int32.MaxValue);
Rational number2 = Math.Pow(Byte.MaxValue, 10);
Rational result = Rational.BitwiseAnd(number1,number2);
```

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.BitwiseOr(Rational, Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational 値に対し、ビットごとの Or 演算を実行します。

```
public static WS.Theia.ExtremelyPrecise.Rational
BitwiseOr(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター
left　Rational
Or 演算する最初の値。
right　Rational
Or 演算する 2 番目の値。

戻り値
Rational
ビットごとの Or 演算の結果。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値を Or 演算代替メソッド。left と right のビットと戻り値のビットの関係性は下表の様になります。

left 内のビット値	right 内のビット値	戻り値内のビット値
0	0	0
1	0	1
1	1	1
0	1	1

BitwiseAnd メソッドは、次の例の様に使用します。

```
Rational number1    = Rational.Parse("10343901200000000000");
Rational number2    = Byte.MaxValue;
Rational result    = Rational.BitwiseOr(number1,number2);
```

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.Clone Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値を複製します。

```
public WS.Theia.ExtremelyPrecise.Rational Clone();
```

戻り値
Rational
複製した Rational 値のオブジェクト。

注釈

Rational オブジェクトが保持している配列の内容を要素単位でコピーして複製します。値を変更するメソッドは全てオブジェクトの再生成をしている為、このメソッドを使用する必要は殆どありません。Rational オブジェクト内部の分母、分子を約分する前後の配列が欲しいといった限られた状況で使用します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Compre(Rational, Rational)

Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational 値を比較し、1 番目の値が 2 番目の値よりも小さいか、同じか、または大きいかを示す整数を返します。

```
public static int Compare(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター

left　Rational
比較する最初の値。
right　Rational
比較する 2 番目の値。

戻り値

Rational
left と right の相対値を示す符号付き整数。次の表を参照してください。

値	条件
0 より小さい値	left が right より小さい値の場合。
0	left と right が等しい場合。
0 より大きい値	left が right より大きい値の場合。

注釈

left と right の差を元に戻り値を判定します。無限大や NaN 等の通常の数値ではない値の大小関係判定は「正の無限大＞無限大以外の正の数＞0＞-0＞無限大以外の負の数＞負の無限大＞NaN」の通りになります。次の例では大きさが 1 違う 1896 桁の Rational 値を比較しています。

```
Rational number1 = Math.Pow(Int64.MaxValue, 100);
Rational number2 = number1 + 1;
string relation = "";
switch (Rational.Compare(number1, number2))
{
    case -1:
        relation = "<";
        break;
    case 0:
        relation = "=";
        break;
    case 1:
        relation = ">";
        break;
}
Console.WriteLine("{0} {1} {2}", number1, relation, number2);
// The example displays the following output:
//       3.0829940252776347122742186219E+1896 <
3.0829940252776347122742186219E+1896
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.CompreTo Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
このインスタンスと 別の値を比較し、このインスタンスの値が もう一方の値よりも小さいか、同じか、または大きいかを示す整数を返します。

オーバーロード

CompareTo(Decimal)	このインスタンスと 10 進数を比較し、このインスタンスの値が 10 進数の値よりも小さいか、同じか、または大きいかを示す整数を返します。
CompareTo(Double)	このインスタンスと 倍精度浮動小数点数を比較し、このインスタンスの値が 倍精度浮動小数点数の値よりも小さいか、同じか、または大きいかを示す整数を返します。
CompareTo(Int64)	このインスタンスと符号付き 64 ビット整数を比較し、このインスタンスの値が符号付き 64 ビット整数の値よりも小さいか、同じか、または大きいかを示す整数を返します。
CompareTo(Object)	このインスタンスと指定したオブジェクトを比較し、このインスタンスの値が指定したオブジェクトの値よりも小さいか、同じか、または大きいかを示す整数を返します。
CompareTo(Rational)	このインスタンスと Rational を比較し、このインスタンスの値が Rational の値よりも小さいか、同じか、または大きいかを示す整数を返します。
CompareTo(UInt64)	このインスタンスと符号なし 64 ビット整数を比較し、このインスタンスの値が符号なし 64 ビット整数の値よりも小さいか、同じか、または大きいかを示す整数を返します。

CompareTo(Decimal)

このインスタンスと 10 進数を比較し、このインスタンスの値が 10 進数の値よりも小さいか、同じか、または大きいかを示す整数を返します。

```
public int CompareTo(decimal other);
```

パラメーター

other　Decimal
比較する 10 進数。

戻り値

Int32
現在のインスタンスと other の相対的な値を示す符号付き整数値です（次の表を参照）。

戻り値	説明
0 より小さい値	現在のインスタンスは other より小さい。
0	現在のインスタンスと other は等しい。
0 より大きい値	現在のインスタンスは other より大きい。

CompareTo(Double)

このインスタンスと 倍精度浮動小数点数を比較し、このインスタンスの値が 倍精度浮動小数点数の値よりも小さいか、同じか、または大きいかを示す整数を返します。

```
public int CompareTo(double other);
```

パラメーター

other　double

比較する 倍精度浮動小数点数。

戻り値

Int32

現在のインスタンスと other の相対的な値を示す符号付き整数値です（次の表を参照）。

戻り値	説明
0 より小さい値	現在のインスタンスは other より小さい。
0	現在のインスタンスと other は等しい。
0 より大きい値	現在のインスタンスは other より大きい。

CompareTo(Int64)

このインスタンスと 符号付き 64 ビット整数を比較し、このインスタンスの値が 符号付き 64 ビット整数の値よりも小さいか、同じか、または大きいかを示す整数を返します。

```
public int CompareTo(long other);
```

パラメーター

other　Int64

比較する 符号付き 64 ビット整数。

戻り値

Int32

現在のインスタンスと other の相対的な値を示す符号付き整数値です（次の表を参照）。

戻り値	説明
0 より小さい値	現在のインスタンスは other より小さい。
0	現在のインスタンスと other は等しい。
0 より大きい値	現在のインスタンスは other より大きい。

例

次の例では CompareTo(Int64)の呼び出し結果を示しています。

```
Rational rationalValue = Rational.Parse("3221123045552");

byte byteValue = 16;
sbyte sbyteValue = -16;
short shortValue = 1233;
ushort ushortValue = 1233;
int intValue = -12233;
uint uintValue = 12233;
```

```
long longValue = 12382222;
ulong ulongValue = 1238222;

Console.WriteLine("Comparing {0} with {1}: {2}",
                  rationalValue, byteValue,
                  rationalValue.CompareTo(byteValue));
Console.WriteLine("Comparing {0} with {1}: {2}",
                  rationalValue, sbyteValue,
                  rationalValue.CompareTo(sbyteValue));
Console.WriteLine("Comparing {0} with {1}: {2}",
                  rationalValue, shortValue,
                  rationalValue.CompareTo(shortValue));
Console.WriteLine("Comparing {0} with {1}: {2}",
                  rationalValue, ushortValue,
                  rationalValue.CompareTo(ushortValue));
Console.WriteLine("Comparing {0} with {1}: {2}",
                  rationalValue, intValue,
                  rationalValue.CompareTo(intValue));
Console.WriteLine("Comparing {0} with {1}: {2}",
                  rationalValue, uintValue,
                  rationalValue.CompareTo(uintValue));
Console.WriteLine("Comparing {0} with {1}: {2}",
                  rationalValue, longValue,
                  rationalValue.CompareTo(longValue));
Console.WriteLine("Comparing {0} with {1}: {2}",
                  rationalValue, ulongValue,
                  rationalValue.CompareTo(ulongValue));
// The example displays the following output:
//       Comparing 3221123045552 with 16: 1
//       Comparing 3221123045552 with -16: 1
//       Comparing 3221123045552 with 1233: 1
//       Comparing 3221123045552 with 1233: 1
//       Comparing 3221123045552 with -12233: 1
//       Comparing 3221123045552 with 12233: 1
//       Comparing 3221123045552 with 12382222: 1
```

```
//          Comparing 3221123045552 with 1238222: 1
```

注釈

Other が Byte、Int16、Int62、SByte、UInt16、又は UIint32 値の場合、Int64 に暗黙的に変換して、Int64 値として CompareTo(Int64)メソッドが呼び出されます。

CompareTo(Object)

このインスタンスと 符号付き 64 ビット整数を比較し、このインスタンスの値が 符号付き 64 ビット整数の値よりも小さいか、同じか、または大きいかを示す整数を返します。

```
public int CompareTo(object obj);
```

パラメーター

obj　Object

比較対象のオブジェクト。

戻り値

Int32

現在のインスタンスと obj パラメーターの関係を示す符号付き整数値です（次の表を参照）。

戻り値	説明
0 より小さい値	現在のインスタンスは obj より小さい。
0	現在のインスタンスと obj は等しい。
0 より大きい値	現在のインスタンスは obj より大きい。

実装

CompareTo(Object)

例外

ArgumentExcepton

Obj が Rational ではありません。

例

次の例では、CompareTo(Object)メソッドを使用して Rational 値と配列内の各要素を比較しています。

```
object[] values = { Math.Pow(Int64.MaxValue, 10), null,
                    12.534, Int64.MaxValue, Rational.One };
Rational number = UInt64.MaxValue;

foreach (object value in values)
{
    try {
        Console.WriteLine("Comparing {0} with '{1}': {2}", number, value,
                          number.CompareTo(value));
    }
    catch (ArgumentException) {
        Console.WriteLine("Unable to compare the {0} value {1} with a
Rational.",
                          value.GetType().Name, value);
    }
}
// The example displays the following output:
//       Comparing 18446744073709551615 with
'4.455508415646675013373597242 4E+189': -1
//       Comparing 18446744073709551615 with '': 1
//       Unable to compare the Double value 12.534 with a Rational.
//       Unable to compare the Int64 value 9223372036854775807 with a
Rational.
//       Comparing 18446744073709551615 with '1': 1
```

注釈

このオーバーロードは IComparable インターフェースの CompareTo メソッド実装です。非ジェネリクスのコレクションオブジェクトが、コレクション内の項目の並び替えに使用します。

obj パラメーターは、次のいずれかを指定する必要があります。

実行時に Rational 型のオブジェクトである事。

Object 値が null である事。obj パラメーターが null の場合、現在のインスタンスが obj パラメーターより大きい事を示す 1 を返します。

CompareTo(Rational)

このインスタンスと Rational を比較し、このインスタンスの値が Rational の値よりも小さいか、同じか、または大きいかを示す整数を返します。

---

public int CompareTo(WS.Theia.ExtremelyPrecise.Rational other);

パラメーター

other　Rational

比較する Rational。

戻り値

Int32

現在のインスタンスと other の相対的な値を示す符号付き整数値です（次の表を参照）。

戻り値	説明
0 より小さい値	現在のインスタンスは other より小さい。
0	現在のインスタンスと other は等しい。
0 より大きい値	現在のインスタンスは other より大きい。

実装

CompareTo(T)

例

次の例では StarInfo オブジェクトの一覧並べ替えに CompareTo(Rational)メソッドを使用しています。各 StarInfo オブジェクトは、星の名前と地球からの距離をマイルで示した情報を提供します。StarInfo クラスでは IComparable<T>インターフェースを実装し、StartInfo ジェネリックコレクションクラスでソートできるようにしています。IComparable<T>.CompareTo の実装では CompereTo(Rational)への呼び出しのラッピングのみを行っています。

```
using System;
using System.Collections.Generic;
using System.Numerics;

public struct StarInfo : IComparable<StarInfo>
{
    // Define constructors.
    public StarInfo(string name, double lightYears)
    {
        this.Name = name;

        // Calculate distance in miles from light years.
        this.Distance = (Rational) Math.Round(lightYears * 5.88e12);
    }

    public StarInfo(string name, Rational distance)
    {
        this.Name = name;
        this.Distance = distance;
    }

    // Define public fields.
    public string Name;
    public Rational Distance;

    // Display name of star and its distance in parentheses.
    public override string ToString()
    {
        return String.Format("{0,-10} ({1:N0})", this.Name, this.Distance);
    }

    // Compare StarInfo objects by their distance from Earth.
    public int CompareTo(StarInfo other)
    {
        return this.Distance.CompareTo(other.Distance);
```

```
    }
}
```

次のコードでは StarInfo オブジェクトを 4 つインスタンス化し、List<T>オブジェクトにジェネリクスを利用して格納します。後に、List<T>.Sort 目祖度が呼び出されると、StartInfo オブジェクトは、地球からの距離の順に並び変えられます。

```
public class Example
{
    public static void Main()
    {
        StarInfo star;
        List<StarInfo> stars = new List<StarInfo>();

        star = new StarInfo("Sirius", 8.6d);
        stars.Add(star);
        star = new StarInfo("Rigel", 1400d);
        stars.Add(star);
        star = new StarInfo("Castor", 49d);
        stars.Add(star);
        star = new StarInfo("Antares", 520d);
        stars.Add(star);

        stars.Sort();

        foreach (StarInfo sortedStar in stars)
            Console.WriteLine(sortedStar);
    }
}
// The example displays the following output:
//          Sirius        (50,568,000,000,000)
//          Castor        (288,120,000,000,000)
//          Antares       (3,057,600,000,000,000)
//          Rigel         (8,232,000,000,000,000)
```

注釈

このオーバーロードは ICompareble<T>インターフェースの CompareTo メソッドの実装です。ジェネリクスコレクションがコレクション内の項目の並べ替え使用します。

こちらもご覧ください Compare(Rational,Rational)、ICompareble<T>、Equals(Rational)

CompareTo(UInt64)

> ⚠重要
> この API は CLS 準拠ではありません。

このインスタンスと 符号なし 64 ビット整数を比較し、このインスタンスの値が 符号なし 64 ビット整数の値よりも小さいか、同じか、または大きいかを示す整数を返します。

```
public int CompareTo(ulong other);
```

パラメーター

other　UInt64

比較する 符号なし 64 ビット整数。

戻り値

Int32

現在のインスタンスと other の相対的な値を示す符号付き整数値です（次の表を参照）。

戻り値	説明
0 より小さい値	現在のインスタンスは other より小さい。
0	現在のインスタンスと other は等しい。
0 より大きい値	現在のインスタンスは other より大きい。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Decrement(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値を 1 だけデクリメントします。

---

public static WS.Theia.ExtremelyPrecise.Rational
Decrement(WS.Theia.ExtremelyPrecise.Rational value)

パラメーター
value　Rational
デクリメントする値。

戻り値
Rational
value パラメーターの値を 1 だけデクリメントした値

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語では Decrement 演算子の代替として次のように使用します

```
Rational number = 93843112;
number =Rational.Decrement(number);                    // Displays 93843111
```

Rational オブジェクトは不変で、Decrement メソッドの戻り値である Rational 値は、value パラメーターより 1 つ小さい新たなオブジェクトです。Decrement メソッドを繰り返し呼び出すと、高負荷になる場合があります。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Divide(Rational, Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
一方の Rational 値をもう一方の値で除算し、その結果を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Add(WS.Theia.ExtremelyPrecise.Rational dividend,
WS.Theia.ExtremelyPrecise.Rational divisor);
```

パラメーター
dividend　Rational
被除数。
divisor　Rational
除数。

戻り値
Rational
除算の結果。

例

次の例では Rational 配列の各要素に、Divide メソッド、除算演算子（/）。及び DivRem メソッドを使用している。

```
using System;
using WS.Theia.ExtremelyPrecise;
```

```
public class Example
{
    public static void Main()
    {
        Rational divisor = Math.Pow(Int64.MaxValue, 2);

        Rational [] dividends = { Rational.Multiply((Rational) Single.MaxValue,
2),

Rational.Parse("906123451238755090918275600071000 99"),
                                  Rational.One,
                                  Rational.Multiply(Int32.MaxValue,
Int64.MaxValue),
                                  divisor + Rational.One };
        // Divide each dividend by divisor in three different ways.
        foreach (Rational dividend in dividends)
        {
            Rational quotient;
            Rational remainder = 0;

            Console.WriteLine("Dividend: {0:N0}", dividend);
            Console.WriteLine("Divisor:   {0:N0}", divisor);
            Console.WriteLine("Results:");
            Console.WriteLine("     Using Divide method:       {0:N0}",
                              Rational.Divide(dividend, divisor));
            Console.WriteLine("     Using Division operator: {0:N0}",
                              dividend / divisor);
            (quotient, remainder)=Math.DivRem(dividend,divisor);
            Console.WriteLine("     Using DivRem method:       {0:N0},
remainder {1:N0}",
                              quotient, remainder);

            Console.WriteLine();
        }
    }
}
```

```
// The example displays the following output:
//       Dividend: 680,564,693,277,057,719,623,408,366,969,033,850,880
//       Divisor:  85,070,591,730,234,615,847,396,907,784,232,501,249
//       Results:
//           Using Divide method:        7
//           Using Division operator: 7
//           Using DivRem method:        7, remainder
85,070,551,165,415,408,691,630,012,479,406,342,137
//
//       Dividend: 90,612,345,123,875,509,091,827,560,007,100,099
//       Divisor:  85,070,591,730,234,615,847,396,907,784,232,501,249
//       Results:
//           Using Divide method:        0
//           Using Division operator: 0
//           Using DivRem method:        0, remainder
90,612,345,123,875,509,091,827,560,007,100,099
//
//       Dividend: 1
//       Divisor:  85,070,591,730,234,615,847,396,907,784,232,501,249
//       Results:
//           Using Divide method:        0
//           Using Division operator: 0
//           Using DivRem method:        0, remainder 1
//
//       Dividend: 19,807,040,619,342,712,359,383,728,129
//       Divisor:  85,070,591,730,234,615,847,396,907,784,232,501,249
//       Results:
//           Using Divide method:        0
//           Using Division operator: 0
//           Using DivRem method:        0, remainder
19,807,040,619,342,712,359,383,728,129
//
//       Dividend: 85,070,591,730,234,615,847,396,907,784,232,501,250
//       Divisor:  85,070,591,730,234,615,847,396,907,784,232,501,249
//       Results:
//           Using Divide method:        1
```

```
//          Using Division operator: 1
//          Using DivRem method:     1, remainder 1
```

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値を割り算する代替メソッドです。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.Equals Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの値が等しいかどうかを示す値を返します。

オーバーロード

Equals(Decimal)	現在のインスタンスの値と 10 進数の値が等しいかどうかを示す値を返します。
Equals(Double)	現在のインスタンスの値と倍精度浮動小数点数の値が等しいかどうかを示す値を返します。
Equals(Int64)	現在のインスタンスの値と符号付き 64 ビット整数の値が等しいかどうかを示す値を返します。
Equals(Object)	現在のインスタンスの値と指定されたオブジェクトの値が等しいかどうかを示す値を返します。
Equals(Rational)	現在のインスタンスの値と Rational の値が等しいかどうかを示す値を返します。
Equals(UInt64)	現在のインスタンスの値と符号無し 64 ビット整数の値が等しいかどうかを示す値を返します。
Equals(Rational,Rational)	2 つの Rational オブジェクトの値が等しいかどうかを示す値を返します。

Equals(Decimal)

現在のインスタンスの値と 10 進数の値が等しいかどうかを示す値を返します。

```
public bool Equals(decimal other);
```

パラメーター

other　Decimal

比較する 10 進数。

戻り値

Boolean

10 進数の値と現在のインスタンスが等しい場合は true。それ以外の場合は false。

例

次の例では Rational オブジェクトと 10 進数値を Equals(Decimal)メソッドで比較します。Rational に、渡された数値を持つ 10 進数との比較である為、値が等しいと判定します。

```
Rational rationalValue;
decimal decimalValue = 16.2m;
rationalValue = new Rational(decimalValue);
Console.WriteLine("{0} {1} = {2} {3} : {4}",
                  rationalValue.GetType().Name, rationalValue,
                  decimalValue.GetType().Name, decimalValue,
                  rationalValue.Equals(decimalValue));
// The example displays the following output:
//       Rational 16.2 = Decimal 16.2 : True
```

注釈

等しいかどうかだけではなく、2 つの値の相対的な大小を取得したい場合は Rational.CompareTo(Decimal)メソッドを使用してください。

Equals(Double)

現在のインスタンスの値と 10 進数の値が等しいかどうかを示す値を返します。

```
public bool Equals(double other);
```

パラメーター

other　Double

比較する 10 進数。

戻り値

Boolean

10 進数の値と現在のインスタンスが等しい場合は true。それ以外の場合は false。

例

次の例では Rational オブジェクトと倍精度浮動小数点を Equals(Double)メソッドで比較します。Rational に、渡された数値を持つ倍精度浮動小数点との比較である為、値が等しいと判定します。

```
Rational rationalValue;

float floatValue = 16.25f;
rationalValue = new Rational(floatValue);
Console.WriteLine("{0} {1} = {2} {3} : {4}",
                  rationalValue.GetType().Name, rationalValue,
                  floatValue.GetType().Name, floatValue,
                  rationalValue.Equals(floatValue));

double doubleValue = 16.25d;
rationalValue = new Rational(doubleValue);
Console.WriteLine("{0} {1} = {2} {3} : {4}",
                  rationalValue.GetType().Name, rationalValue,
                  doubleValue.GetType().Name, doubleValue,
                  rationalValue.Equals(doubleValue));
// The example displays the following output:
//       Rational 16.25 = Single 16.25 : True
//       Rational 16.25 = Double 16.25 : True
```

注釈

other が Single の場合は暗黙的に Double に変換して、このメソッドが呼び出されます。等しいかどうかだけではなく、2 つの値の相対的な大小を取得したい場合は Rational.CompareTo(Decimal)メソッドを使用してください。

Equals(Int64)

現在のインスタンスの値と符号付き 64 ビット整数の値が等しいかどうかを示す値を返します。

```
public bool Equals(long other);
```

パラメーター

other　Int64

比較する 符号付き 64 ビット整数。

戻り値

Boolean

符号付き 64 ビット整数の値と現在のインスタンスが等しい場合は true。それ以外の場合は false。

例

次の例では Rational オブジェクトと符号付き 64 ビット整数を Equals(Int64)メソッドで比較します。Rational に、渡された数値を持つ符号付き 64 ビット整数との比較である為、値が等しいと判定します。

```
Rational rationalValue;

byte byteValue = 16;
rationalValue = new Rational(byteValue);
Console.WriteLine("{0} {1} = {2} {3} : {4}",
                  rationalValue.GetType().Name, rationalValue,
                  byteValue.GetType().Name, byteValue,
                  rationalValue.Equals(byteValue));
```

```
sbyte sbyteValue = -16;
rationalValue = new Rational(sbyteValue);
Console.WriteLine("{0} {1} = {2} {3} : {4}",
                  rationalValue.GetType().Name, rationalValue,
                  sbyteValue.GetType().Name, sbyteValue,
                  rationalValue.Equals(sbyteValue));

short shortValue = 1233;
rationalValue = new Rational(shortValue);
Console.WriteLine("{0} {1} = {2} {3} : {4}",
                  rationalValue.GetType().Name, rationalValue,
                  shortValue.GetType().Name, shortValue,
                  rationalValue.Equals(shortValue));

ushort ushortValue = 64000;
rationalValue = new Rational(ushortValue);
Console.WriteLine("{0} {1} = {2} {3} : {4}",
                  rationalValue.GetType().Name, rationalValue,
                  ushortValue.GetType().Name, ushortValue,
                  rationalValue.Equals(ushortValue));

int intValue = -1603854;
rationalValue = new Rational(intValue);
Console.WriteLine("{0} {1} = {2} {3} : {4}",
                  rationalValue.GetType().Name, rationalValue,
                  intValue.GetType().Name, intValue,
                  rationalValue.Equals(intValue));

uint uintValue = 1223300;
rationalValue = new Rational(uintValue);
Console.WriteLine("{0} {1} = {2} {3} : {4}",
                  rationalValue.GetType().Name, rationalValue,
                  uintValue.GetType().Name, uintValue,
                  rationalValue.Equals(uintValue));

long longValue = -123822229012;
```

```
rationalValue = new Rational(longValue);
Console.WriteLine("{0} {1} = {2} {3} : {4}",
                  rationalValue.GetType().Name, rationalValue,
                  longValue.GetType().Name, longValue,
                  rationalValue.Equals(longValue));
// The example displays the following output:
//       Rational 16 = Byte 16 : True
//       Rational -16 = SByte -16 : True
//       Rational 1233 = Int16 1233 : True
//       Rational 64000 = UInt16 64000 : True
//       Rational -1603854 = Int32 -1603854 : True
//       Rational 1223300 = UInt32 1223300 : True
//       Rational -123822229012 = Int64 -123822229012 : True
```

注釈

other が Byte、Int16、Int32、SByte、Uint16、又は Uint32 の場合は暗黙的に Int64 に変換して、このメソッドが呼び出されます。

等しいかどうかだけではなく、2 つの値の相対的な大小を取得したい場合は Rational.CompareTo(Int64)メソッドを使用してください。

Equals(Object)

現在のインスタンスの値と指定されたオブジェクトの値が等しいかどうかを示す値を返します。

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

パラメーター

other　Object

比較対象のオブジェクト。

戻り値

Boolean

obj 引数が 数値 で、その値が現在の Rational インスタンスの値と等しい場合は true。それ以外の場合は false。

例

次の例では Object 配列と Rational 配列の各要素を比較しています。各要素は数値としての値は同じですが、Rational オブジェクトの場合のみ等しいとみなされます。

```
using System;
using System.Numerics;

public class Example
{
    public static void Main()
    {
        object[] obj = { 0, 10, 100, new Rational(1000), -10 };
        Rational [] rt = { Rational.Zero, new Rational (10),
                           new Rational (100), new Rational (1000),
                           new Rational (-10) };
        for (int ctr = 0; ctr < rt.Length; ctr++)
            Console.WriteLine(rt[ctr].Equals(obj[ctr]));
    }
}
// The example displays the following output:
//       False
//       False
//       False
//       True
//       False
```

注釈

obj パラメーターが Rational オブジェクトでない場合、Equals(Object)メソッドは false を返します。obj が Rational オブジェクトかつ値が等しい場合にのみ true を返します。

等しいかどうかだけではなく、2 つの値の相対的な大小を取得したい場合は Rational.CompareTo(Object)メソッドを使用してください。

Equals(Rational)

現在のインスタンスの値と Rational の値が等しいかどうかを示す値を返します。

```
public bool Equals(WS.Theia.ExtremelyPrecise.Rational other);
```

パラメーター

other　WS.Theia.ExtremelyPrecise.Rational

比較する Rational 値。

戻り値

Boolean

Rational の値と現在のインスタンスの値が等しい場合は true。それ以外の場合は false。

例

次の例では地球からいくつかの星までの距離を比較しています。この場合は、Equals の各オーバーロードを使用して等しいか比較しています。

```
const long LIGHT_YEAR = 5878625373183;

Rational altairDistance = 17 * LIGHT_YEAR;
Rational epsilonIndiDistance = 12 * LIGHT_YEAR;
Rational ursaeMajoris47Distance = 46 * LIGHT_YEAR;
long tauCetiDistance = 12 * LIGHT_YEAR;
ulong procyon2Distance = 12 * LIGHT_YEAR;
object wolf424ABDistance = 14 * LIGHT_YEAR;

Console.WriteLine("Approx. equal distances from Epsilon Indi to:");
Console.WriteLine("    Altair: {0}",
                  epsilonIndiDistance.Equals(altairDistance));
```

```
Console.WriteLine("     Ursae Majoris 47: {0}",
                  epsilonIndiDistance.Equals(ursaeMajoris47Distance));
Console.WriteLine("     TauCeti: {0}",
                  epsilonIndiDistance.Equals(tauCetiDistance));
Console.WriteLine("     Procyon 2: {0}",
                  epsilonIndiDistance.Equals(procyon2Distance));
Console.WriteLine("     Wolf 424 AB: {0}",
                  epsilonIndiDistance.Equals(wolf424ABDistance));
// The example displays the following output:
//       Approx. equal distances from Epsilon Indi to:
//           Altair: False
//           Ursae Majoris 47: False
//           TauCeti: True
//           Procyon 2: True
//           Wolf 424 AB: False
```

注釈

このメソッドは IEquatable<T>インターフェスの実装です。Equals(Object)と異なり other パラメーターをキャストしない為、Equals(Object)より若干パフォーマンスが優れています。等しいかどうかだけではなく、2 つの値の相対的な大小を取得したい場合は Rational.CompareTo(Rational)メソッドを使用してください。

Equals(UInt64)

> ⚠重要
> この API は CLS 準拠ではありません。

現在のインスタンスの値と 符号なし 64 ビット整数の値が等しいかどうかを示す値を返します。

---

public bool Equals(ulong other);

パラメーター
other　Decimal
比較する符号なし 64 ビット整数。

戻り値
Boolean
符号なし 64 ビット整数の値と現在のインスタンスが等しい場合は true。それ以外の場合は false。

例

次の例では地球からいくつかの星までの距離を比較しています。この場合は、Equals の各オーバーロードを使用して等しいか比較しています。

```
const long LIGHT_YEAR = 5878625373183;
Rational altairDistance = 17 * LIGHT_YEAR;
Rational epsilonIndiDistance = 12 * LIGHT_YEAR;
Rational ursaeMajoris47Distance = 46 * LIGHT_YEAR;
long tauCetiDistance = 12 * LIGHT_YEAR;
ulong procyon2Distance = 12 * LIGHT_YEAR;
object wolf424ABDistance = 14 * LIGHT_YEAR;

Console.WriteLine("Approx. equal distances from Epsilon Indi to:");
Console.WriteLine("     Altair: {0}",
                  epsilonIndiDistance.Equals(altairDistance));
Console.WriteLine("     Ursae Majoris 47: {0}",
                  epsilonIndiDistance.Equals(ursaeMajoris47Distance));
Console.WriteLine("     TauCeti: {0}",
                  epsilonIndiDistance.Equals(tauCetiDistance));
Console.WriteLine("     Procyon 2: {0}",
                  epsilonIndiDistance.Equals(procyon2Distance));
Console.WriteLine("     Wolf 424 AB: {0}",
                  epsilonIndiDistance.Equals(wolf424ABDistance));
// The example displays the following output:
//       Approx. equal distances from Epsilon Indi to:
//            Altair: False
//            Ursae Majoris 47: False
//            TauCeti: True
//            Procyon 2: True
//            Wolf 424 AB: False
```

注釈

等しいかどうかだけではなく、2 つの値の相対的な大小を取得したい場合は Rational.CompareTo(UInt64)メソッドを使用してください。

Equals(Rational,Rational)

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational オブジェクトの値が等しいかどうかを示す値を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Equals(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター
left　Rational
比較する最初の値。
right　Rational
比較する 2 番目の値。

戻り値
Rational
left パラメーターと right パラメーターが同じ値の場合は true。それ以外の場合は false。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値を比較する代替メソッド。Rational 値を比較して変数に割り当てる時は次の例の様に使用する。

```
// The statement:
//       bool comp = Int64.MaxValue == Int32.MaxValue;
// produces compiler error CS0220: The operation overflows at compile time in
checked mode.
// The alternative:
bool comp = Rational.Equals(Int64.MaxValue, Int32.MaxValue);
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.FromOACurrency(Int64) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
OLE オートメーション通貨値を格納している指定した 64 ビット符号付き整数を、それと等価の Rational 値に変換します。

```
public static Rational FromOACurrency(long cy);
```

引数

Cy　Int64
OLE オートメーション通貨値。

戻り値

Rational
cy と等価の値を格納している Rational。

例

次の例では、OLE オートメーション通貨値を持つ Int64 を FromOACurrency メソッドで等価な Rational 値に変換しています。

```
// Example of the decimal.FromOACurrency method.
using System;

class DecimalFromOACurrencyDemo
{
    const string dataFmt = "{0,21}{1,25}";

    // Display the decimal.FromOACurrency parameter and decimal result.
    public static void ShowDecimalFromOACurrency( long Argument )
```

```
    {
        Rational ratCurrency = Rational.FromOACurrency( Argument );

        Console.WriteLine( dataFmt, Argument, ratCurrency );
    }

    public static void Main( )
    {
        Console.WriteLine( "This example of the " +
            "decimal.FromOACurrency( ) method generates ¥nthe " +
            "following output. It displays the OLE Automation " +
            "Currency ¥nvalue as a long and the result as a " +
            "decimal.¥n" );
        Console.WriteLine( dataFmt, "OA Currency", "Decimal Value" );
        Console.WriteLine( dataFmt, "-----------", "-------------" );

        // Convert OLE Automation Currency values to decimal objects.
        ShowDecimalFromOACurrency( 0L );
        ShowDecimalFromOACurrency( 1L );
        ShowDecimalFromOACurrency( 100000L );
        ShowDecimalFromOACurrency( 100000000000L );
        ShowDecimalFromOACurrency( 1000000000000000000L );
        ShowDecimalFromOACurrency( 1000000000000000001L );
        ShowDecimalFromOACurrency( long.MaxValue );
        ShowDecimalFromOACurrency( long.MinValue );
        ShowDecimalFromOACurrency( 123456789L );
        ShowDecimalFromOACurrency( 1234567890000L );
        ShowDecimalFromOACurrency( 1234567890987654321 );
        ShowDecimalFromOACurrency( 4294967295L );
    }
}

/*
This example of the decimal.FromOACurrency( ) method generates
the following output. It displays the OLE Automation Currency
value as a long and the result as a decimal.
```

```
                OA Currency                 Decimal Value
                -----------                 -------------
                          0                             0
                          1                        0.0001
                     100000                            10
                100000000000                      10000000
  1000000000000000000              100000000000000000
  1000000000000000001              100000000000000.0001
   9223372036854775807              922337203685477.5807
  -9223372036854775808             -922337203685477.5808
                  123456789                   12345.6789
               1234567890000                    123456789
  1234567890987654321              123456789098765.4321
                  4294967295                   429496.7295
*/
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.GetHashCode Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
現在の Rational オブジェクトのハッシュ コードを返します。

```
public override int GetHashCode();
```

戻り値
Rational
32 ビット符号付き整数ハッシュ コード。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational. GetTypeCode Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
現在の Rational オブジェクトのハッシュ コードを返します。

```
public TypeCode GetTypeCode();
```

戻り値
TypeCode
列挙型定数 Object。

実装
GetTypeCode()

例

次の例では、GetTypeCode メソッドが Object 型を示す TypeCode を返しています。

```
// Example of the Rational.GetTypeCode method.
using System;

class RationalGetTypeCodeDemo
{
      public static void Main( )
      {
            Console.WriteLine( "This example of the " +
                  "Rational.GetTypeCode( ) ¥nmethod " +
                  "generates the following output.¥n" );

            // Create a decimal object and get its type code.
            Rational aRational = new Rational( 1.0 );
            TypeCode typCode = aRational.GetTypeCode( );

            Console.WriteLine( "Type Code:          ¥"{0}¥"", typCode );
            Console.WriteLine( "Numeric value:    {0}", (int)typCode );
      }
}

/*
This example of the Rational.GetTypeCode( )
method generates the following output.

Type Code:          "Object"
Numeric value:    1
*/
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.GreatestCommonDivisor(Rational,Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational 値の最大公約数を求めます。

---

public static WS.Theia.ExtremelyPrecise.Rational GreatestCommonDivisor (WS.Theia.ExtremelyPrecise.Rational left, WS.Theia.ExtremelyPrecise.Rational right);

パラメーター

left　Rational
最大公約数を求める最初の値。
right　Rational
最大公約数を求める 2 番目の値。

戻り値

Rational
left と right の最大公約数。

例

次の例では GreatestCommonDivisor メソッドを呼び出し、最大公約数を求めています。また、例外処理のとして ArgumentOutOfRangeException が発生した場合の処理を定義しています。今回の場合、2 つの数値の最大公約数が 1 となります。

```
Rational n1 = Math.Pow(154382190, 3);
Rational n2 = Rational.Multiply(1643590, 166935);
try
{
    Console.WriteLine("The greatest common divisor of {0} and {1} is {2}.",
                      n1, n2, Rational.GreatestCommonDivisor(n1, n2));
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Unable to calculate the greatest common divisor:");
    Console.WriteLine("    {0} is an invalid value for {1}",
                      e.ActualValue, e.ParamName);
}
```

注釈

最大公約数は 2 つの Rational 値を割り切ることができる値です。
left と right のパラメーターが 0 以外の場合は、全ての数値は 0 で除算できるため、常に 1 以上の値を返します。いずれかのパラメーターが 0 の場合は、0 以外のパラメーターの絶対値を返します。両方が 0 の場合、メソッドは 0 を返します。

> ⚠注意
> 最大公約数を求める処理は大きな時間がかかる可能性のある処理です。

メソッドは left と right パラメーターの符号に関係なく常に正の値を返します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Increment (Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値を 1 だけインクリメントします。

```
public static WS.Theia.ExtremelyPrecise.Rational Increment
(WS.Theia.ExtremelyPrecise.Rational value)
```

パラメーター
value　Rational
インクリメントする値。

戻り値
Rational
value パラメーターの値を 1 だけインクリメントした値

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語では Implicit 演算子の代替として次のように使用します

```
Rational number = 93843112;
number =Rational. Increment (number);                    // Displays 93843113
```

Rational オブジェクトは不変で、Increment メソッドの戻り値である Rational 値は、value パラメーターより 1 つ大きい新たなオブジェクトです。Decrement メソッドを繰り返し呼び出すと、高負荷になる場合があります。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.IsInfinity(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した数値が負または正の無限大と評価されるかどうかを示す値を返します。

```
public bool IsInfinity (WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
無限大か確認する Rational 値。

戻り値
Boolean
value が PositiveInfinity または NegativeInfinity と評価される場合は true。それ以外の場合は false。

例
IsInfinity は次の例の様に使用します。

```
// This will return "true".

Console.WriteLine("IsInfinity(3.0 / 0) == {0}.", Rational.IsInfinity(new
Rational (3.0) / 0) ? "true" : "false");
```

注釈
PositiveInfinity、または NegativeInfinity は変換元の浮動小数点のオーバーフロー、0 除算等で発生します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.IsNaN(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した数値が非数値（NaN）かどうかを示す値を返します。

```
public bool IsNaN (WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
NaN か確認する Rational 値。

戻り値
Boolean
value が NaN と評価される場合は true。それ以外の場合は false。

例
IsNaN は次の例の様に使用します。

```
// This will return true.
if (Rational.IsNaN(0 / Rational.Zero))
    Console.WriteLine("Double.IsNan() can determine whether a value is not-a-number.");
```

注釈
NaN は Rational 演算の結果が定義されていない事を通知します。0.0 を 0.0 で除算した場合等が該当します。

⚠注意
IsNaN は PositiveInfinity、または NegativeInfinicty であっても false を返します。これらの値をテストする場合には IsInfinity、IsPositiveInfinity、IsNegativeInfinity メソッドを使用してください。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.IsNegativeInfinity(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した数値が負の無限大と評価されるかどうかを示す値を返します。

```
public bool IsNegativeInfinity (WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
負の無限大か確認する Rational 値。

戻り値
Boolean
value が NegativeInfinity と評価される場合は true。それ以外の場合は false。

例
IsNegativeInfinity は次の例の様に使用します。

```
// This will return "true".
Console.WriteLine("IsNegativeInfinity(-5.0 / 0) == {0}.",
Rational.IsNegativeInfinity(-5.0 / Rational.Zero) ? "true" : "false");
// This will equal Infinity.
Console.WriteLine("10.0 minus NegativeInfinity equals {0}.", (10.0 -
Rational.NegativeInfinity).ToString());
```

注釈

NegativeInfinity は変換元の浮動小数点のオーバーフロー、0 除算等で発生します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.IsPositiveInfinity(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した数値が正の無限大と評価されるかどうかを示す値を返します。

```
public bool IsPositiveInfinity (WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
正の無限大か確認する Rational 値。

戻り値
Boolean
value が PositiveInfinity と評価される場合は true。それ以外の場合は false。

例
IsPositiveInfinity は次の例の様に使用します。

```
// This will return "true".

Console.WriteLine("IsPositiveInfinity(4.0 / 0) == {0}.",
Rational.IsPositiveInfinity(4.0 / Rational.Zero) ? "true" : "false");
```

注釈
PositiveInfinity は変換元の浮動小数点のオーバーフロー、0 除算等で発生します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.LeftShift(Rational,Int32) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定されたビット数だけ Rational 値を左にシフトします。

```
public static WS.Theia.ExtremelyPrecise.Rational
LeftShift(WS.Theia.ExtremelyPrecise.Rational value, int shift);
```

パラメーター
value　Rational
ビットをシフトする対象の値。
shift　Int32
value を左にシフトするビット数。

戻り値
Rational
指定されたビット数だけ左にシフトした値。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値を左シフトする代替メソッド。Rational 値を左シフトして変数に割り当てる時は次の例の様に使用する。

```
Rational number = Rational.Parse("-904732167844981624999999312055");
Console.WriteLine("Shifting {0} left by:", number);
for (int ctr = 0; ctr <= 16; ctr++)
{
    Rational newNumber = Rational.LeftShift(number,ctr);
    Console.WriteLine(" {0,2} bits: {1,35}", ctr, newNumber);
```

```
}
// The example displays the following output:
//       Shifting -90473216784498162499999312055 left by:
//          0 bits:             -90473216784498162499999312055
//          1 bits:            -180946433568996324999998624110
//          2 bits:            -361892867137992649999997248220
//          3 bits:            -723785734275985299999994496440
//          4 bits: -1.447571468551970599999898899288E+29
//          5 bits: -2.895142937103941199999797798576E+29
//          6 bits: -5.790285874207882399999595597152E+29
//          7 bits:    -1.158057174841576479999911943E+30
//          8 bits: -2.316114349683152959999238861E+30
//          9 bits: -4.632228699366305919999647722E+30
//         10 bits: -9.264457398732611839999295443E+30
//         11 bits: -1.852891479746522367999859108 9E+31
//         12 bits: -3.705782959493044735999718217 7E+31
//         13 bits: -7.411565918986089471999436435 5E+31
//         14 bits: -1.482313183797217894399887287 1E+32
//         15 bits: -2.964626367594435788799774574 2E+32
//         16 bits: -5.929252735188871577599549148 4E+32
```

> ⚠注意
> プリミティブ型の左シフト演算と異なり、Rational の LeftShift メソッドは符号が変化する事がありません。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Mod(Rational, Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された 2 つの Rational 値の除算の結果生じた剰余を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Mod(WS.Theia.ExtremelyPrecise.Rational dividend,
WS.Theia.ExtremelyPrecise.Rational divisor);
```

パラメーター
dividend　Rational
被除数。
divisor　Rational
除数。

戻り値
Rational
除算の結果生じた剰余。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値の剰余を求める代替メソッド。Rational 値の剰余を求めて変数に割り当てる時は次の例の様に使用する。

```
Rational num1 = 100045632194;
Rational num2 = 90329434;
Rational remainder = num1 % num2;
Console.WriteLine(remainder);                    // Displays 50948756
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.ModPow(Rational,Rational,Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
数値を別の数値で累乗し、それをさらに別の数値で割った結果生じた剰余を求めます。

```
public static WS.Theia.ExtremelyPrecise.Rational
ModPow(WS.Theia.ExtremelyPrecise.Rational value,
WS.Theia.ExtremelyPrecise.Rational exponent,
WS.Theia.ExtremelyPrecise.Rational modulus);
```

パラメーター
value　Rational
指数 exponent で累乗する数値。
exponent　Rational
value の指数。
modulus　Rational
value を exponent で累乗した結果を除算する時の除数。

戻り値
Rational
value を exponent で累乗し modulus で割った結果生じた剰余。

例

次の例では ModPow メソッドの呼び出し方を示しています。

```
using System;
using System.Numerics;

public class Class1
{
    public static void Main()
    {
        Rational number = 10;
        int exponent = 3;
        Rational modulus = 30;
        Console.WriteLine("({0}^{1}) Mod {2} = {3}",
                          number, exponent, modulus,
                          Rational.ModPow(number, exponent, modulus));
    }
}
// The example displays the following output:
//       (10^3) Mod 30 = 10
```

注釈

ModPow メソッドは次の式と等価です。
(value^exponent) mod modulus

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Multiply(Rational,Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational 値の積を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Multiply(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター
left　Rational
乗算対象の最初の数。
right　Rational
乗算対象の 2 番目の数。

戻り値
Rational
left と right パラメーターの積。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値を乗算する代替メソッド。次の例の乗算では結果が符号付 64 ビット整数の上限を超えている為、例外が発生します。例外が発生した後 Rational を使用して乗算しその結果を変数に割り当てる例を示します。

```
long number1 = 1234567890;
long number2 = 9876543210;
try
{
    long product;
    product = checked(number1 * number2);
}
catch (OverflowException)
{
    Rational product;
    product = Rational.Multiply(number1, number2);
    Console.WriteLine(product.ToString());
    }
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Negate(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された Rational 値の符号を反転します。

```
public static WS.Theia.ExtremelyPrecise.Rational Negate
(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
符号を反転させる値。

戻り値
Rational
value パラメーターに -1 を乗算した結果。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値の符号反転する代替メソッド。Rational 値の符号を反転して変数に割り当てる手順は次の例で示します。

```
// The statement
//       Rational number = -Int64.MinValue;
// produces compiler error CS0220: The operation overflows at compile time in
checked mode.
// The alternative:
Rational number = Rational.Negate(Int64.MinValue);
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.OnesComplement(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値のビットごとの 1 の補数を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
OnesComplement(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
1 の補数を取得したい値。

戻り値
Rational
value のビットごとの 1 の補数。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値のビット反転をする代替メソッド。value で 0 であるビットは 1 に、1 であるビットには 0 を設定します。

```
using System;
using System.Numerics;

public class Example
{
```

```
    public static void Main()
    {
        Rational value, complement;

        value = Rational.Multiply(Rational.One, 9);
        complement = Rational.OnesComplement(value);

        Console.WriteLine("{0,5} -- {1,-32}", value, DisplayInBinary(value));
        Console.WriteLine("{0,5} -- {1,-32}¥n", complement,
DisplayInBinary(complement));

        value = Rational.MinusOne * SByte.MaxValue;
        complement = Rational.OnesComplement(value);

        Console.WriteLine("{0,5} -- {1,-32}", value, DisplayInBinary(value));
        Console.WriteLine("{0,5} -- {1,-32}¥n", complement,
DisplayInBinary(complement));
    }

    private static string DisplayInBinary(Rational number)
    {
        (_,byte[] bytes,_,_) = number.ToByteArray();
        string binaryString = string.Empty;
        foreach (byte byteValue in bytes)
        {
            string byteString = Convert.ToString(byteValue, 2).Trim();
            binaryString += byteString.Insert(0, new string('0', 8 -
byteString.Length));
        }
        return binaryString;
    }
}
// The example displays the following output:
//                9 –
0000100100000000000000000000000000000000000000000000000000000000
//-18446744073709551606 –
```

```
11110110111111111111111111111111111111111111111111111111111111111
//
// -127 –
01111111000000000000000000000000000000000000000000000000000000000
//18446744073709551488 --
10000000111111111111111111111111111111111111111111111111111111111
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Parse Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
数値の文字列形式を、それと等価の Rational に変換します。

オーバーロード

Parse(String)	数値の文字列形式を、それと等価の Rational に変換します。
Parse(String,NumberStyles)	指定のスタイルで表現された数値の文字列形式を、それと等価な Rational に変換します。
Parse(String,IFormatProvider)	指定されたカルチャ固有の書式で表現された文字列形式の数値を、それと等価の Rational に変換します。
Parse(String,NumberStyles,IFormatProvider)	指定したスタイルおよびカルチャ固有の書式の数値の文字列形式を、それと等価の Rational に変換します。

Parse(String)

数値の文字列形式を、それと等価の Rational に変換します。

```
public static WS.Theia.ExtremelyPrecise.Rational Parse(String value);
```

パラメーター

value　String

変換する数値を含んだ文字列。

戻り値

Rational

value パラメーターで指定されている数値と等価の値。

例外

ArgumentNullException

value は null です。

FormatException

value が正しい形式ではありません。

例

次の例では Parse(String)メソッドを使って 2 つの Rational オブジェクトのインスタンスを生成しています。その後各オブジェクトを乗算し Compare メソッドで 2 つの値の大小を判定しています。

```
string stringToParse = String.Empty;
try
{
    // Parse two strings.
```

```
    string string1, string2;
    string1 = "12347534159895123";
    string2 = "987654321357159852";
    stringToParse = string1;
    Rational number1 = Rational.Parse(stringToParse);
    Console.WriteLine("Converted '{0}' to {1:N0}.", stringToParse, number1);
    stringToParse = string2;
    Rational number2 = Rational.Parse(stringToParse);
    Console.WriteLine("Converted '{0}' to {1:N0}.", stringToParse, number2);
    // Perform arithmetic operations on the two numbers.
    number1 *= 3;
    number2 *= 2;
    // Compare the numbers.
    int result = Rational.Compare(number1, number2);
    switch (result)
    {
        case -1:
            Console.WriteLine("{0} is greater than {1}.", number2, number1);
            break;
        case 0:
            Console.WriteLine("{0} is equal to {1}.", number1, number2);
            break;
        case 1:
            Console.WriteLine("{0} is greater than {1}.", number1, number2);
            break;
    }
}
catch (FormatException)
{
    Console.WriteLine("Unable to parse {0}.", stringToParse);
}
// The example displays the following output:
//       Converted '12347534159895123' to 12,347,534,159,895,123.
//       Converted '987654321357159852' to 987,654,321,357,159,852.
//       1975308642714319704 is greater than 37042602479685369.
```

注釈

value パラメーターは、次の形式で表された数字文字列でなければなりません。

[ws][sign]digits[ws]

角カッコ（[および]）内の要素は省略可能です。それぞれの要素は次の表のとおりです。

要素	説明
ws	空白文字。省略可能です。
sign	符号。省略可能です。有効な文字はカレントカルチャーの NumberFormatInfo.NegativeSign と NumberFormatInfo.PositiveSign プロパティによって決まります。
digits	数字列。0 から 9 及び小数点で構成している必要があります。先頭の 0 は無視します。有効な小数点はカレントカルチャーの NumberFormatInfo.NumberDecimalSeparator プロパティによって決まります。

Parse(String,NumberStyles)

指定のスタイルで表現された数値の文字列形式を、それと等価な Rational に変換します。

```
public static WS.Theia.ExtremelyPrecise.Rational Parse(String
value ,NumberStyles style);
```

パラメーター

value　String

変換する数値を含んだ文字列。

style　NumberStyles

value に許可されている書式を指定する列挙値のビットごとの組み合わせ。

戻り値

Rational

value パラメーターで指定されている数値と等価の値。

例外

ArgumentException

style が NumberStyle 値ではない、または、seyle に AllowHexSpecifier または HexNumber フラグが含まれます。

ArgumentNullException

value は null です。

FormatException

value が正しい形式ではありません。

例

次の例では Parse(String,NumberStyle)メソッドに、NumberStyle として 16 進数値として文字列を解釈させる値、スペースを許可する値、符号を許可する値を与えて呼び出しています。

```
Rational number;
// Method should succeed (white space and sign allowed)
number = Rational.Parse("     -68054     ", NumberStyles.Integer);
Console.WriteLine(number);
// Method should succeed (string interpreted as hexadecimal)
number = Rational.Parse("68054", NumberStyles.AllowHexSpecifier);
Console.WriteLine(number);
// Method call should fail: sign not allowed
try
{
    number = Rational.Parse("     -68054    ",
NumberStyles.AllowLeadingWhite
| NumberStyles.AllowTrailingWhite);
    Console.WriteLine(number);
}
catch (FormatException e)
{
    Console.WriteLine(e.Message);
}
// Method call should fail: white space not allowed
try
{
    number = Rational.Parse("     68054    ", NumberStyles.AllowLeadingSign);
    Console.WriteLine(number);
}
catch (FormatException e)

{
    Console.WriteLine(e.Message);
}
//
// The method produces the following output:
//
//       -68054
//       Input string was not in a correct format.
```

```
//       Input string was not in a correct format.
//       Input string was not in a correct format.
```

注釈

style パラメーターは空白、符号、桁区切り記号、小数点記号など使用することのできる文字を指定することができます。value パラメーターは、次の形式のうち style パラメーターで許可された要素を数字列に含めることができます。

[ws][$][sign][digits,]digits[.fractional_digits][E[sign]exponential_digits][ws]

角カッコ（[および]）内の要素は省略可能です。それぞれの要素は次の表のとおりです。

要素	説明
ws	空白文字。省略可能です。 NumberStyles.AllowLeadingWhite フラグおよび NumberStyles.AllowTrailingWhite フラグで使用可能かが決まります。
$	通貨記号。有効な文字はカレントカルチャーの NumberFormatInfo.CurrencyNegativePattern と NumberFormatInfo.CurrencyPositivePattern プロパティの値で決まります。 NumberStyles.AllowCurrencySymbol フラグが有効な時に使用可能になります。
sign	符号。有効な文字はカレントカルチャーの NumberFormatInfo.NegativeSign と NumberFormatInfo.PositiveSign プロパティによって決まります。
digits fractional_digits exponential_digits	数字列。0 から 9 及び小数点で構成している必要があります。 fractional_digits 以外では先頭の 0 は無視します。
,	数字の桁区切りです。有効な文字はカレントカルチャーの CurrencyGroupSeparator と NumberGroupSeparator と PercentGroupSeparator プロパティで決まります。 NumberStyles.AllowThousands フラグが有効な時に使用可能になります。

.	小数点記号です。有効な文字はカレントカルチャーの CurrencyDecimalSeparator、NumberDecimalSeparator、PercentDecimalSeparator プロパティで決まります。NumberStyles.AllowDecimalPoint フラグが有効な時に使用可能になります。
E	“e”または”E”文字は、指数表記で表されている事を示します。NumberStyles.AllowExponent フラグが有効な時使用可能になります。

数字のみを含む文字列（NumberStyle.None が対応）は常に正常に解析できます。他の NumberStyle メンバーは多くの要素を許可しますが、value パラメーターにはその全ての要素を含んでいる必要はありません。

None	数字のみです。
AllowDecimalPoint	整数部、小数点（.）と桁の小数部を許容します。
AllowExponent	"e"または"E"文字と共に、指数部を許容します。
AllowLeadingWhite	value の先頭に空白がある事を許容します。
AllowTrailingWhite	value の末尾に空白がある事を許容します。
AllowLeadingSign	value の先頭に符号がある事を許容します。
AllowTrailingSign	value の末尾に符号がある事を許容します。
AllowParentheses	負数をカッコで囲って表記する事を許容します。
AllowThousands	整数部を桁区切りする事を許容します。
AllowCurrencySymbol	通貨記号を使用する事を許容します。
Currency	すべての要素を許容します。 ただし、 value プロパティを 16 進数または指数表記で表す事はできません。
Float	value の先頭、末尾の空白、value 先頭の符号、および小数点、指数表記を許容します。
Number	value の先頭、末尾の空白、value 先頭、末尾の符号、桁区切り記号、および小数点といった 10 進数の全ての要素を許容します。
Any	すべての要素を許容します。 ただし、value パラメーターを 16 進数で表記する事はできません。

Parse(String,IFormatProvider)

指定されたカルチャ固有の書式で表現された文字列形式の数値を、それと等価の Rational に変換します。

```
public static WS.Theia.ExtremelyPrecise.Rational Parse(String
value,IFormatProvider provider);
```

パラメーター

value　String

変換する数値を含んだ文字列。

provider　IFormatProvider

value に関するカルチャ固有の書式情報を提供するオブジェクト。

戻り値

Rational

value パラメーターで指定されている数値と等価の値。

例外

ArgumentNullException

value は null です。

FormatException

value が正しい形式ではありません。

例

次の例ではチルダ（~）を負の符号として定義する方法を 2 つ提示します。1 つ目の例は IFormatProvider インタフェースの GetFormat メソッドを実装し、NumberFormatInfo オブジェクトを返却するクラスを作成する方法です。

まず NumberFormatInfo オブジェクトを返却するクラスを定義します。

```
public class RationalFormatProvider : IFormatProvider
{
    public object GetFormat(Type formatType)
    {
        if (formatType == typeof(NumberFormatInfo))
        {
            NumberFormatInfo numberFormat = new NumberFormatInfo();
            numberFormat.NegativeSign = "~";
            return numberFormat;
        }
        else
        {
            return null;
        }
    }
}
```

次に NumberFormatInfo オブジェクトを提供するクラスをインスタンス化して使用します。

```
Rational number = Rational.Parse("~6354129876", new
RationalFormatProvider());
// Display value using same formatting information
Console.WriteLine(number.ToString(new RationalFormatProvider()));
// Display value using formatting of current culture
Console.WriteLine(number);
```

2 つ目の方法は NumberFormatInfo オブジェクトの値を書き換えて provider パラメーターに渡す方法です。

```
NumberFormatInfo fmt = new NumberFormatInfo();
fmt.NegativeSign = "~";

Rational number = Rational.Parse("~6354129876", fmt);
// Display value using same formatting information
Console.WriteLine(number.ToString(fmt));
// Display value using formatting of current culture
Console.WriteLine(number);
```

注釈

value パラメーターは、次の形式で表された数字文字列でなければなりません。

[ws][sign]digits[ws]

角カッコ（[および]）内の要素は省略可能です。それぞれの要素は次の表のとおりです。

要素	説明
ws	空白文字。省略可能です。
sign	符号。省略可能です。有効な文字はカレントカルチャーの NumberFormatInfo.NegativeSign と NumberFormatInfo.PositiveSign プロパティによって決まります。
digits	数字列。0 から 9 及び小数点で構成している必要があります。先頭の 0 は無視します。有効な小数点はカレントカルチャーの NumberFormatInfo.NumberDecimalSeparator プロパティによって決まります。

provider パラメーターは、IFormatProvider インタフェースの GetFormat メソッドが実装されたオブジェクトを設定することができます。GetFormat メソッドの戻り値は NumberFormatInfo オブジェクトです。Parse(String,IFormatProvider)メソッドには主に 3 種類方法で provider を渡すことができます。

- CultureInfo が提供する書式を表すオブジェクト。（GetFormat メソッドがそのカルチャに合わせた NumberFormatInfo オブジェクトを返します。）
- 直接インスタンス化した NumberFormatInfo オブジェクト。（GetFormat が NumberFormatInfo オブジェクト自信を返します）
- IFormatProvider を実装したカスタムオブジェクト。（そのオブジェクトの GetFormat メソッドが NumberFormatInfo オブジェクトをインスタンス化して返します。）

Parse(String,NumberStyles,IFormatProvider)

指定したスタイルおよびカルチャ固有の書式の数値の文字列形式を、それと等価の Rational に変換します。

---

public static WS.Theia.ExtremelyPrecise.Rational Parse(String value,NumberStyles style,IFormatProvider provider);

パラメーター
value　String
変換する数値を含んだ文字列。
style　NumberStyles
value に許可されている書式を指定する列挙値のビットごとの組み合わせ。
provider　IFormatProvider
value に関するカルチャ固有の書式情報を提供するオブジェクト。

戻り値
Rational
value パラメーターで指定されている数値と等価の値。

例外
ArgumentException
style が NumberStyle 値ではない、または、seyle に AllowHexSpecifier または HexNumber フラグが含まれます。
ArgumentNullException
value は null です。
FormatException
value が正しい形式ではありません。

例

次の例では style と provider パラメーターの様々な組み合わせで Parse(String,NumberStyle, IFormatProvider)を呼び出しています。

```
// Call parse with default values of style and provider

Console.WriteLine(Rational.Parse("    -300     ",
                        NumberStyles.Integer, CultureInfo.CurrentCulture));
// Call parse with default values of style and provider supporting tilde as
negative sign
Console.WriteLine(Rational.Parse("     ~300    ",
                                    NumberStyles.Integer, new
RationalFormatProvider()));
// Call parse with only AllowLeadingWhite and AllowTrailingWhite
// Exception thrown because of presence of negative sign
try
{
    Console.WriteLine(Rational.Parse("       ~300     ",
                                    NumberStyles.AllowLeadingWhite |
NumberStyles.AllowTrailingWhite,
                                    new RationalFormatProvider()));
}
catch (FormatException e)
{
    Console.WriteLine("{0}: ¥n     {1}", e.GetType().Name, e.Message);
}
// Call parse with only AllowHexSpecifier
// Exception thrown because of presence of negative sign
try
{
    Console.WriteLine(Rational.Parse("-3af", NumberStyles.AllowHexSpecifier,
                                    new RationalFormatProvider()));
}
catch (FormatException e)
{
```

```
    Console.WriteLine("{0}: ¥n     {1}", e.GetType().Name, e.Message);
}
// Call parse with only NumberStyles.None
// Exception thrown because of presence of white space and sign
try
{
    Console.WriteLine(Rational.Parse(" -300 ", NumberStyles.None,
                                              new RationalFormatProvider()));
}
catch (FormatException e)
{
    Console.WriteLine("{0}: ¥n     {1}", e.GetType().Name, e.Message);
}
// The example displays the followingoutput:
//          -300
//          -300
//          FormatException:
//              The value could not be parsed.
//          FormatException:
//              The value could not be parsed.
//          FormatException:
//              The value could not be parsed.
```

Parse(String,NumberStyle, IFormatProvider)メソッドを呼び出す際に使っている RationalFormatProvider クラスは、負の符号としてチルダ（~）を定義しています。

```
public class RationalFormatProvider : IFormatProvider
{
    public object GetFormat(Type formatType)
    {
        if (formatType == typeof(NumberFormatInfo))
        {
            NumberFormatInfo numberFormat = new NumberFormatInfo();
            numberFormat.NegativeSign = "~";
            return numberFormat;
        }
        else
        {
            return null;
        }
    }
}
```

注釈

style パラメーターは空白、符号、桁区切り記号、小数点記号など使用することのできる文字を指定することができます。value パラメーターは、次の形式のうち style パラメーターで許可された要素を数字列に含めることができます。

[ws][$][sign][digits,]digits[.fractional_digits][E[sign]exponential_digits][ws]

角カッコ（[および]）内の要素は省略可能です。それぞれの要素は次の表のとおりです。

要素	説明
ws	空白文字。省略可能です。 NumberStyles.AllowLeadingWhite フラグおよび NumberStyles.AllowTrailingWhite フラグで使用可能かが決まります。

$	通貨記号。有効な文字はカレントカルチャーの NumberFormatInfo.CurrencyNegativePattern と NumberFormatInfo.CurrencyPositivePattern プロパティの値で決まります。 NumberStyles.AllowCurrencySymbol フラグが有効な時に使用可能になります。
sign	符号。有効な文字はカレントカルチャーの NumberFormatInfo.NegativeSign と NumberFormatInfo.PositiveSign プロパティによって決まります。
digits fractional_digits exponential_digits	数字列。0 から 9 及び小数点で構成している必要があります。 fractional_digits 以外では先頭の 0 は無視します。
,	数字の桁区切りです。有効な文字はカレントカルチャーの CurrencyGroupSeparator と NumberGroupSeparator と PercentGroupSeparator プロパティで決まります。 NumberStyles.AllowThousands フラグが有効な時に使用可能になります。
.	小数点記号です。有効な文字はカレントカルチャーの CurrencyDecimalSeparator、NumberDecimalSeparator、PercentDecimalSeparator プロパティで決まります。 NumberStyles.AllowDecimalPoint フラグが有効な時に使用可能になります。
E	“e”または”E”文字は、指数表記で表されている事を示します。 NumberStyles.AllowExponent フラグが有効な時使用可能になります。

数字のみを含む文字列（NumberStyle.None が対応）は常に正常に解析できます。他の NumberStyle メンバーは多くの要素を許可しますが、value パラメーターにはその全ての要素を含んでいる必要はありません。

None	数字のみです。
AllowDecimalPoint	整数部、小数点（.）と桁の小数部を許容します。
AllowExponent	"e"または"E"文字と共に、指数部を許容します。
AllowLeadingWhite	value の先頭に空白がある事を許容します。
AllowTrailingWhite	value の末尾に空白がある事を許容します。
AllowLeadingSign	value の先頭に符号がある事を許容します。
AllowTrailingSign	value の末尾に符号がある事を許容します。
AllowParentheses	負数をカッコで囲って表記する事を許容します。
AllowThousands	整数部を桁区切りする事を許容します。
AllowCurrencySymbol	通貨記号を使用する事を許容します。
Currency	すべての要素を許容します。 ただし、 value プロパティを 16 進数または指数表記で表す事はできません。
Float	value の先頭、末尾の空白、value 先頭の符号、および小数点、指数表記を許容します。
Number	value の先頭、末尾の空白、value 先頭、末尾の符号、桁区切り記号、および小数点といった 10 進数の全ての要素を許容します。
Any	すべての要素を許容します。 ただし、value パラメーターを 16 進数で表記する事はできません。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Plus(Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational オペランドの値を返します。 オペランドの符号は変更されません。

```
public static WS.Theia.ExtremelyPrecise.Rational Plus
(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
符号を反転させない値。

戻り値
Rational
value パラメーターと等価な値。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値の単項プラス演算子の代替メソッド。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.RightShift(Rational,Int32)

Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定されたビット数だけ Rational 値を左にシフトします。

```
public static WS.Theia.ExtremelyPrecise.Rational
RightShift(WS.Theia.ExtremelyPrecise.Rational value, int shift);
```

パラメーター
value　Rational
ビットをシフトする対象の値。
shift　Int32
value を右にシフトするビット数。

戻り値
Rational
指定されたビット数だけ右にシフトされた値。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値を左シフトする代替メソッド。Rational 値を右シフトして変数に割り当てる時は次の例の様に使用する。

```
var number = Rational.Parse("-904732167844981624999312055");
Console.WriteLine("Shifting {0} right by:", number);
for (int ctr = 0; ctr <= 16; ctr++) {
    Rational newNumber = Rational.RightShift(number,ctr);
    Console.WriteLine(" {0,2} bits: {1,35}", ctr, newNumber);
}
// The example displays the following output:
//Shifting -904732167844981624999312055 right by
//    0 bits:            -904732167844981624999312055
//    1 bits:         -452366083922490812499965602 7.5
//    2 bits:       -226183041961245406249982801 3.75
//    3 bits:     -113091520980622703124991400 6.875
//    4 bits:     -56545760490311351562495700 3.4375
//    5 bits:    -28272880245155675781247850 1.71875
//    6 bits: -14136440122577837890623925 0.859375
//    7 bits: -70682200612889189453119625.4296875
//    8 bits: -35341100306444594726559812.71484375
//    9 bits: -17670550153222297363279906.357421875
// 10 bits: -8835275076611148681639953.1787109375
// 11 bits: -4417637538305574340819976.58935546875
// 12 bits: -2208818769152787170409988.294677734375
// 13 bits: -1104409384576393585204994.1473388671875
// 14 bits: -552204692288196792602497.07366943359375
// 15 bits: -276102346144098396301248.536834716796875
// 16 bits: -138051173072049198150624.2684173583984375
```

> ⚠注意
> プリミティブ型の右シフト演算と異なり、Rational の RightShift メソッドは小数点以下の値を切り捨てません。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.Subtract(Rational,Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値を別の値から減算し、その結果を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Subtract(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター

left　Rational
減算される値（被減数）。
right　Rational
減算する値（減数）。

戻り値

Rational
left から right を減算した結果。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値を減算する代替メソッド。Rational 値を減算して変数に割り当てる時は次の例の様に使用する。

```
// The statement
//       Rational number = Int64.MinValue - Int64.MaxValue;
// produces compiler error CS0220: The operation overflows at compile time in checked mode.
// The alternative:
Rational number = Rational.Subtract(Int64.MinValue, Int64.MaxValue);
```

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.ToByte Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した Rational の値を、等価の 8 ビット符号なし整数に変換します。

```
public static byte ToByte(Rational value);
```

パラメーター
value　Rational
変換する Rational。

戻り値
value　Byte
value と等価の 8 ビット符号なし整数。

例外
OverflowException
Value が MinValue より小さいか MaxValue より大きい場合。

例

次の例では ToByte(Rational)メソッドを使って Byte 値に変換しています。

```
using System;

class Example
{
    public static void Main( )
    {
```

```
        Rational[] values = { 123m, new Decimal(78000, 0, 0, false, 3),
                              78.999m, 255m, 255.001m,
                              127m, 127.001m, -0.999m,
                              -1m,    -128m, -128.001m };

        foreach (var value in values) {
            try {
                byte number = Rational.ToByte(value);
                Console.WriteLine("{0} --> {1}", value, number);
            }
            catch (OverflowException e)
            {
                 Console.WriteLine("{0}: {1}", e.GetType().Name, value);
            }
        }
    }
}
// The example displays the following output:
//     123 --> 123
//     78 --> 78
//     78.999 --> 78
//     123 --> 123
//     78 --> 78
//     78.999 --> 78
//     255 --> 255
//     OverflowException: 255.001
//     127 --> 127
//     127.001 --> 127
//     OverflowException: -0.999
//     OverflowException: -1
//     OverflowException: -128
//     OverflowException: -128.001
```

注釈

value パラメーターの値から小数点以下の値を切り捨てた値に変換されます。変換結果は Byte 型への明示的なキャストと等価です。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.ToByteArray Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll

> ⚠注意
> 名称と実体が一致していない等問題がある為大幅な仕様変更を行う可能性があります。

Rational 値をバイト配列に変換します。

---

public (bool Sign, byte[] Numerator, byte[] Denominator,bool Infinity) ToByteArray();

戻り値
Sign　Boolean
符号を示す値（false＝プラス、true=マイナス）。
Numerator　byte[]
分子を表すリトルエンディアン順に格納されたバイト値の配列。
Denominator　byte[]
分母を表すリトルエンディアン順に格納されたバイト値の配列。
Infinity　Boolean
無限大を示す値（false＝無限大ではない、true＝無限大）

注釈

numerator パラメーター、denominator パラメーターは最上位バイトに数値の最下位を並べるリトルエンディアン順でなければなりません。たとえば、数値 1,000,000,000,000 は、次の表に示すように表されます。

数値の 16 進数文字列	E8D4A51000
バイト配列（前方のインデックスが最も低い）	00 10 A5 D4 E8 00

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.ToDecimal Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した Rational の値を、等価の 10 進数に変換します。

```
public static Decimal ToDecimal(Rational value);
```

パラメーター
value　Rational
変換する Rational。

戻り値
value　Decimal
value と等価の 10 進数。

例外
OverflowException
Value が MinValue より小さいか MaxValue より大きい場合。

注釈

この操作は Decimal 型のの有効桁数が少ない為、丸めによる誤差が発生します。変換結果は Decimal 型への明示的なキャストと等価です。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.ToDouble Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した Rational の値を、それと等価の倍精度浮動小数点数に変換します。

```
public static double ToDouble(Rational value);
```

パラメーター
value　Rational
変換する Rational。

戻り値
value　Double
value と等価の倍精度浮動小数点数。

例外
OverflowException
Value が MinValue より小さいか MaxValue より大きい場合。

例

次の例では ToDouble(Rational)メソッドを使って倍精度浮動小数値に変換しています。

```
// Example of the Rational.ToSingle and Rational.ToDouble methods.
using System;

class RationalToSgl_DblDemo
{
    static string formatter = "{0,30}{1,17}{2,23}";
```

```
// Convert the decimal argument; no exceptions are thrown.
public static void RationalToSgl_Dbl( Rational argument )
{
    object SingleValue;
    object DoubleValue;

    // Convert the argument to a float value.
    SingleValue = Rational.ToSingle( argument );

    // Convert the argument to a double value.
    DoubleValue = Rational.ToDouble( argument );


    Console.WriteLine( formatter, argument,
        SingleValue, DoubleValue );
}

public static void Main( )
{
    Console.WriteLine( "This example of the ¥n" +
        "    Rational.ToSingle( Rational ) and ¥n" +
        "    Rational.ToDouble( Rational ) ¥nmethods " +
        "generates the following output. It ¥ndisplays " +
        "several converted decimal values.¥n" );
    Console.WriteLine( formatter, "decimal argument",
        "float", "double" );
    Console.WriteLine( formatter, "----------------",
        "-----", "------" );

    // Convert decimal values and display the results.
    RationalToSgl_Dbl( 0.00000000000000000000000001M );
    RationalToSgl_Dbl( 0.00000000012345678912345678M );
    RationalToSgl_Dbl( 123M );
    RationalToSgl_Dbl( new decimal( 123000000, 0, 0, false, 6 ) );
    RationalToSgl_Dbl( 123456789.123456789M );
    RationalToSgl_Dbl( 12345678912345678912345678M );
```

```
            RationalToSgl_Dbl( decimal.MinValue );
            RationalToSgl_Dbl( decimal.MaxValue );
        }
}

/*
This example of the
   Rational.ToSingle( Rational ) and
   Rational.ToDouble( Rational )
methods generates the following output. It
displays several converted decimal values.

                    decimal argument              float                  double
                    ----------------              -----                  ------
0.00000000000000000000000000001                  1E-28                   1E-28
0.00000000001234567891234567890       1.234568E-11    1.23456789123457E-11
                                123              123                     123
                          123.000000             123                     123
                123456789.123456789       1.234568E+08      123456789.123457
    1234567891234567891234567890          1.234568E+26    1.23456789123457E+26
-7922816251426433759354395033.5       -7.922816E+28   -7.92281625142643E+28
 7922816251426433759354395033.5        7.922816E+28    7.92281625142643E+28

*/
```

注釈

この操作は倍精度浮動小数の有効桁数が少ない為、丸めによる誤差が発生します。変換結果は Double 型への明示的なキャストと等価です。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.ToInt16 Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した Rational の値を、等価の 16 ビット符号付き整数に変換します。

```
public static short ToInt16(Rational value);
```

パラメーター
value　Rational
変換する Rational。

戻り値
value　Int16
value と等価の 16 ビット符号付き整数。

例外
OverflowException
Value が MinValue より小さいか MaxValue より大きい場合。

例

次の例では ToInt16(Rational)メソッドを使って Int16 値に変換しています。

```
using System;

class Example
{
    public static void Main( )
    {
```

```
        Rational[] values = { 123m, new Decimal(123000, 0, 0, false, 3),
                              123.999m, 65535m, 65535.001m,
                              32767m, 32767.001m, -0.999m,
                              -1m,    -32768m, -32768.001m };

        foreach (var value in values) {
            try {
                short number = Rational.ToInt16(value);
                Console.WriteLine("{0} --> {1}", value, number);
            }
            catch (OverflowException e)
            {
                 Console.WriteLine("{0}: {1}", e.GetType().Name, value);
            }
        }
    }
}
// The example displays the following output:
//     123 --> 123
//     123 --> 123
//     123.999 --> 123
//     OverflowException: 65535
//     OverflowException: 65535.001
//     32767 --> 32767
//     OverflowException: 32767.001
//     -0.999 --> 0
//     -1 --> -1
//     -32768 --> -32768
//     OverflowException: -32768.001
```

注釈

value パラメーターの値から小数点以下の値を切り捨てた値に変換されます。変換結果は Int16 型への明示的なキャストと等価です。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.ToInt32 Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した Rational の値を、等価の 32 ビット符号付き整数に変換します。

```
public static int ToInt32(Rational value);
```

パラメーター
value　Rational
変換する Rational。

戻り値
value　Rational
value と等価の 32 ビット符号付き整数。

例外
OverflowException
value が MinValue より小さいか MaxValue より大きい場合。

例

次の例では ToInt32(Rational)メソッドを使って Int32 値に変換しています。

```
using System;

class Example
{
    public static void Main( )
    {
```

```
            Rational[] values = { 123m, new decimal(123000, 0, 0, false, 3),
                                  123.999m, 4294967295m, 4294967295.001m,
                                  4294967296m, 2147483647m, 2147483647.001m,
                                  -0.999m, -1m, -2147483648m, -2147483648.001m };

        foreach (var value in values) {
            try {
                int number = Rational.ToInt32(value);
                Console.WriteLine("{0} --> {1}", value, number);
            }
            catch (OverflowException e)
            {
                 Console.WriteLine("{0}: {1}", e.GetType().Name, value);
            }
        }
    }
}
    // The example displays the following output:
    //       123 --> 123
    //       123.000 --> 123
    //       123.999 --> 123
    //       OverflowException: 4294967295
    //       OverflowException: 4294967295.001
    //       OverflowException: 4294967296
    //       2147483647 --> 2147483647
    //       OverflowException: 2147483647.001
    //       -0.999 --> 0
    //       -1 --> -1
    //       -2147483648 --> -2147483648
    //       OverflowException: -2147483648.001
```

注釈

value パラメーターの値から小数点以下の値を切り捨てた値に変換されます。変換結果は Int32 型への明示的なキャストと等価です。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.ToInt64 Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した Rational の値を、等価の 64 ビット符号付き整数に変換します。

```
public static long ToInt64(Rational value);
```

パラメーター
value　Rational
変換する Rational。

戻り値
value　Rational
value と等価の 64 ビット符号付き整数。

例外
OverflowException
value が MinValue より小さいか MaxValue より大きい場合。

例

次の例では ToInt64(Rational)メソッドを使って Int64 値に変換しています。

```
using System;

class Example
{
    public static void Main( )
    {
```

```
        Rational[] values = { 123m, new decimal(123000, 0, 0, false, 3),
                              123.999m, 4294967295m, 4294967295.001m,
                              4294967296m, 2147483647m, 2147483647.001m,
                              -0.999m, -1m, -2147483648m, -2147483648.001m };


        foreach (var value in values) {
            try {
                int number = Rational.ToInt32(value);
                Console.WriteLine("{0} --> {1}", value, number);
            }
            catch (OverflowException e)
            {
                 Console.WriteLine("{0}: {1}", e.GetType().Name, value);
            }
        }
    }
}
// The example displays the following output:
//       123 --> 123
//       123.000 --> 123
//       123.999 --> 123
//       OverflowException: 4294967295
//       OverflowException: 4294967295.001
//       OverflowException: 4294967296
//       2147483647 --> 2147483647
//       OverflowException: 2147483647.001
//       -0.999 --> 0
//       -1 --> -1
//       -2147483648 --> -2147483648
//       OverflowException: -2147483648.001
```

注釈

value パラメーターの値から小数点以下の値を切り捨てた値に変換されます。変換結果は Int64 型への明示的なキャストと等価です。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.ToOACurrency Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した Rational の値を、等価の 64 ビット符号付き整数に変換します。

```
public static long ToOACurrency (Rational value);
```

パラメーター
value　Rational
変換する Rational。

戻り値
value　Rational
value と等価の OLE オートメーション値を格納する 64 ビット符号付き整数。

例外
OverflowException
Value が MinValue より小さいか MaxValue より大きい場合。

例

次の例では、Rational 値を ToOACurrency メソッドで、等価な OLE オートメーション通貨値を持つ Int64 に変換しています。

```
// Example of the decimal.ToOACurrency method.
using System;

class DecimalToOACurrencyDemo
{
```

```
const string dataFmt = "{0,31}{1,27}";

// Get the exception type name; remove the namespace prefix.
public static string GetExceptionType( Exception ex )
{
    string exceptionType = ex.GetType( ).ToString( );
    return exceptionType.Substring(
        exceptionType.LastIndexOf( '.' ) + 1 );
}

// Display the decimal.ToOACurrency parameter and the result
// or exception.
public static void ShowRationalToOACurrency( Rational Argument )
{
    // Catch the exception if ToOACurrency( ) throws one.
    try
    {
        long oaCurrency = Rational.ToOACurrency( Argument );
        Console.WriteLine( dataFmt, Argument, oaCurrency );
    }
    catch( Exception ex )
    {
        Console.WriteLine( dataFmt, Argument,
            GetExceptionType( ex ) );
    }
}


public static void Main( )
{
    Console.WriteLine( "This example of the " +
        "decimal.ToOACurrency( ) method generates ¥nthe " +
        "following output. It displays the argument as a " +
        "decimal ¥nand the OLE Automation Currency value " +
        "as a long.¥n" );
    Console.WriteLine( dataFmt, "Argument",
```

```
                "OA Currency or Exception" );
            Console.WriteLine( dataFmt, "--------",
                "-----------------------" );

            // Convert decimal values to OLE Automation Currency values.
            ShowRationalToOACurrency( 0M );
            ShowRationalToOACurrency( 1M );
ShowRationalToOACurrency( 1.0000000000000000000000000000M );
            ShowRationalToOACurrency( 100000000000000M );
 ShowRationalToOACurrency( 100000000000000.00000000000000M );
ShowRationalToOACurrency( 1000000000000000000000000000M );
            ShowRationalToOACurrency( 0.000000000123456789M );
            ShowRationalToOACurrency( 0.123456789M );
            ShowRationalToOACurrency( 123456789M );
            ShowRationalToOACurrency( 12345678900000000M );
            ShowRationalToOACurrency( 4294967295M );
            ShowRationalToOACurrency( 18446744073709551615M );
ShowRationalToOACurrency( -79.228162514264337593543950335M );
ShowRationalToOACurrency( -79228162514264.337593543950335M );
        }
}

/*
This example of the Rational.ToOACurrency( ) method generates
the following output. It displays the argument as a decimal
and the OLE Automation Currency value as a long.

                            Argument      OA Currency or Exception
                            --------      -----------------------
                                   0                            0
                                   1                        10000
 1.0000000000000000000000000000                            10000
                     100000000000000          1000000000000000000
 100000000000000.00000000000000          1000000000000000000
  1000000000000000000000000000             OverflowException
              0.000000000123456789                            0
```

```
                0.123456789                        1235
                  123456789               1234567890000
             123456789000000000          OverflowException
                 4294967295               42949672950000
           18446744073709551615          OverflowException
-79.228162514264337593543950335                -792282
-79228162514264.337593543950335       -792281625142643376
*/
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.ToSByte Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した Rational の値を、等価の 8 ビット符号付き整数に変換します。

```
public static sbyte ToSByte(Rational value);
```

パラメーター
value　Rational
変換する Rational。

戻り値
value　SByte
value と等価の 8 ビット符号付き整数。

例外
OverflowException
Value が MinValue より小さいか MaxValue より大きい場合。

例

次の例では ToSByte(Rational)メソッドを使って SByte 値に変換しています。

```
using System;

class Example
{
    public static void Main( )
    {
```

```
        Rational[] values = { 123m, new Decimal(78000, 0, 0, false, 3),
                              78.999m, 255m, 255.001m,
                              127m, 127.001m, -0.999m,
                              -1m,    -128m, -128.001m };

        foreach (var value in values) {
            try {
                sbyte number = Rational.ToSByte(value);
                Console.WriteLine("{0} --> {1}", value, number);
            }
            catch (OverflowException e)
            {
                 Console.WriteLine("{0}: {1}", e.GetType().Name, value);
            }
        }
    }
}
// The example displays the following output:
//       78 --> 78
//       78.000 --> 78
//       78.999 --> 78
//       OverflowException: 255
//       OverflowException: 255.001
//       127 --> 127
//       OverflowException: 127.001
//       -0.999 --> 0
//       -1 --> -1
//       -128 --> -128
//       OverflowException: -128.001
```

注釈

value パラメーターの値から小数点以下の値を切り捨てた値に変換されます。変換結果は SByte 型への明示的なキャストと等価です。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.ToSingle Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定した Rational の値を、それと等価の単精度浮動小数点数に変換します。

```
public static float ToSingle(Rational value);
```

パラメーター
value　Rational
変換する Rational。

戻り値
value　Single
value と等価の単精度浮動小数点数。

例外
OverflowException
Value が MinValue より小さいか MaxValue より大きい場合。

例

次の例では ToSingle (Rational)メソッドを使って単精度浮動小数値に変換しています。

```
// Example of the Rational.ToSingle and Rational.ToDouble methods.
using System;

class RationalToSgl_DblDemo
{
    static string formatter = "{0,30}{1,17}{2,23}";
```

```
// Convert the decimal argument; no exceptions are thrown.
public static void RationalToSgl_Dbl( Rational argument )
{
    object SingleValue;
    object DoubleValue;

    // Convert the argument to a float value.
    SingleValue = Rational.ToSingle( argument );

    // Convert the argument to a double value.
    DoubleValue = Rational.ToDouble( argument );


    Console.WriteLine( formatter, argument,
        SingleValue, DoubleValue );
}

public static void Main( )
{
    Console.WriteLine( "This example of the ¥n" +
        "    Rational.ToSingle( Rational ) and ¥n" +
        "    Rational.ToDouble( Rational ) ¥nmethods " +
        "generates the following output. It ¥ndisplays " +
        "several converted decimal values.¥n" );
    Console.WriteLine( formatter, "decimal argument",
        "float", "double" );
    Console.WriteLine( formatter, "----------------",
        "-----", "------" );

    // Convert decimal values and display the results.
    RationalToSgl_Dbl( 0.00000000000000000000000001M );
    RationalToSgl_Dbl( 0.00000000012345678912345678M );
    RationalToSgl_Dbl( 123M );
    RationalToSgl_Dbl( new decimal( 123000000, 0, 0, false, 6 ) );
    RationalToSgl_Dbl( 123456789.123456789M );
    RationalToSgl_Dbl( 12345678912345678912345678M );
```

```
            RationalToSgl_Dbl( decimal.MinValue );
            RationalToSgl_Dbl( decimal.MaxValue );
        }
}

/*
This example of the
   Rational.ToSingle( Rational ) and
   Rational.ToDouble( Rational )
methods generates the following output. It
displays several converted decimal values.

                    decimal argument              float                  double
                    ----------------              -----                  ------
0.00000000000000000000000000001                  1E-28                   1E-28
0.00000000001234567891234567890       1.234568E-11    1.23456789123457E-11
                                123               123                    123
                          123.000000              123                    123
                123456789.123456789       1.234568E+08     123456789.123457
    1234567891234567891234567890          1.234568E+26    1.23456789123457E+26
-79228162514264337593543950335        -7.922816E+28   -7.92281625142643E+28
 79228162514264337593543950335         7.922816E+28    7.92281625142643E+28

*/
```

注釈

この操作は単精度浮動小数の有効桁数が少ない為、丸めによる誤差が発生します。変換結果は Single 型への明示的なキャストと等価です。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.ToString Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
現在の BigInteger オブジェクトの数値を等価の文字列形式に変換します。

オーバーロード

ToString()	現在の Rational オブジェクトの数値を等価の文字列形式に変換します。
ToString(IFormatProvider)	指定されたカルチャ固有の書式情報を使用して、現在の Rational オブジェクトの数値をそれと等価の文字列形式に変換します。
ToString(String)	指定された書式を使用して、現在の Rational オブジェクトの数値をそれと等価な文字列形式に変換します。
ToString(String,IFormatProvider)	指定された書式とカルチャ固有の書式情報を使用して、現在の Rational オブジェクトの数値をそれと等価の文字列形式に変換します。

ToString()

現在の Rational オブジェクトの数値を等価の文字列形式に変換します。

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

戻り値

String

現在の Rational 値の文字列形式。

例

次の例では ToString()メソッドを使って Rational オブジェクトを文字列に変換しています。Rational の標準書式指定子を使用した結果です。例では EN-US カルチャの書式指定規則が適用されています。

```
// Initialize a Rational value.
Rational value = Rational.Add(UInt64.MaxValue, 1024);

// Display value using the default ToString method.
Console.WriteLine(value.ToString());
// Display value using some standard format specifiers.
Console.WriteLine(value.ToString("G"));
Console.WriteLine(value.ToString("C"));
Console.WriteLine(value.ToString("D"));
Console.WriteLine(value.ToString("F"));
Console.WriteLine(value.ToString("N"));
// The example displays the following output on a system whose current
// culture is en-US:
//       18446744073709552639
//       18446744073709552639
//       $18,446,744,073,709,552,639.00
//       18446744073709552639
//       18446744073709552639.00
//       18,446,744,073,709,552,639.00
```

注釈

ToString()メソッドはラウンドトリップ形式で既定のカルチャで変換を行います。特定のカルチャ、特定の形式で変換をしたい場合は、以下のオーバーロードを使用してください。

形式	カルチャ	オーバーロード
ラウンドトリップ形式	指定したカルチャ	ToString(IFormatProvider)
指定した形式	既定のカルチャ	ToString(String)
指定した形式	指定したカルチャ	ToString(String,IFormatProvider)

ToString(IFormatProvider)

指定されたカルチャ固有の書式情報を使用して、現在の Rational オブジェクトの数値をそれと等価の文字列形式に変換します。

```
public string ToString(IFormatProvider provider);
```

パラメーター

provider　IFormatProvider

カルチャ固有の書式情報を提供するオブジェクト。

戻り値

String

現在の Rational 値の文字列形式を、provider パラメーターで指定されている形式で表現した値。

例

次の例では、NumberFormatInfo で負の符号としてチルダ（~）を設定したカスタムカルチャ用意し ToString(IFormatProvider)メソッドを使って Rational オブジェクトを文字列に変換しています。Rational の標準書式指定子を使用した結果です。

```
Rational number = 9867857831128;
number = Math.Pow(number, 3) * Rational.MinusOne;

NumberFormatInfo bigIntegerProvider = new NumberFormatInfo();
bigIntegerProvider.NegativeSign = "~";

Console.WriteLine(number.ToString(bigIntegerProvider));
```

注釈

ToString(IFormatProvider)メソッドはラウンドトリップ形式で既定のカルチャで変換を行います。特定のカルチャ、特定の形式で変換をしたい場合は、以下のオーバーロードを使用してください。

形式	カルチャ	オーバーロード
ラウンドトリップ形式	既定のカルチャ	ToString()
指定した形式	既定のカルチャ	ToString(String)
指定した形式	指定したカルチャ	ToString(String,IFormatProvider)

provider パラメーターは、IFormatProvider インタフェースの GetFormat メソッドが実装されたオブジェクトを設定することができます。GetFormat メソッドの戻り値は NumberFormatInfo オブジェクトです。Parse(String,IFormatProvider)メソッドには主に 3 種類方法で provider を渡すことができます。

- CultureInfo が提供する書式を表すオブジェクト。(GetFormat メソッドがそのカルチャに合わせた NumberFormatInfo オブジェクトを返します。)
- 直接インスタンス化した NumberFormatInfo オブジェクト。(GetFormat が NumberFormatInfo オブジェクト自信を返します)
- IFormatProvider を実装したカスタムオブジェクト。(そのオブジェクトの GetFormat メソッドが NumberFormatInfo オブジェクトをインスタンス化して返します。)

ToString(String)

現在の Rational オブジェクトの数値を等価の文字列形式に変換します。

```
public string ToString(string format);
```

引数

Format　String
標準またはカスタムの数値書式指定文字列。

戻り値

String
現在の Rational 値の文字列形式を、format パラメーターで指定されている形式で表現した値。

例

次の例では Rational を初期化し、いくつかのカスタム書式文字列を使用して変換しています。例では EN-US カルチャの書式指定規則が適用されています。

```
Rational value = Rational.Parse("-903145792771643190182");
string[] specifiers = { "C", "D", "D25", "E", "E4", "e8", "F0",
                        "G", "N0", "P", "R", "0,0.000",
                        "#,#.00#;(#,#.00#)" };

foreach (string specifier in specifiers)
    Console.WriteLine("{0}: {1}", specifier, value.ToString(specifier));

// The example displays the following output:
//       C: ($903,145,792,771,643,190,182.00)
//       D: -903145792771643190182
//       D25: -0000903145792771643190182
//       E: -9.031457E+020
//       E4: -9.0314E+020
//       e8: -9.03145792e+020
//       F0: -903145792771643190182
//       G: -903145792771643190182
//       N0: -903,145,792,771,643,190,182
//       P: -90,314,579,277,164,319,018,200.00 %
//       R: -903145792771643190182
//       0,0.000: -903,145,792,771,643,190,182.000
//       #,#.00#;(#,#.00#): (903,145,792,771,643,190,182.00)
```

注釈

ToString(String)メソッドはラウンドトリップ形式、既定のカルチャで変換を行います。特定のカルチャ、特定の形式で変換したい場合は以下のオーバーロードを使用してください。

形式	カルチャ	オーバーロード
ラウンドトリップ形式	既定のカルチャ	ToString()
ラウンドトリップ形式	指定したカルチャ	ToString(IFormatProvider)
指定した形式	指定したカルチャ	ToString(String,IFormatProvider)

ToString(String,IFormatProvider)

現在の Rational オブジェクトの数値を等価の文字列形式に変換します。

```
public string ToString(String format,IFormatProvider provider);
```

パラメーター

format　String

標準またはカスタムの数値書式指定文字列。

provider　　IFormatProvider

カルチャ固有の書式情報を提供するオブジェクト。

戻り値

String

format パラメーターと provider パラメーターで指定されている現在の Rational 値の文字列表現。

例

次の例では Rational 値を初期化して標準書式指定文字列を使用して変換しています。NumberFormatInfo オブジェクトで負の符号としてチルダ（~）を定義しています。

```
// Redefine the negative sign as the tilde for the invariant culture.
NumberFormatInfo bigIntegerFormatter = new NumberFormatInfo();
bigIntegerFormatter.NegativeSign = "~";

Rational value = Rational.Parse("-903145792771643190182");
string[] specifiers = { "C", "D", "D25", "E", "E4", "e8", "F0",
                        "G", "N0", "P", "R", "0,0.000",
                        "#,#.00#;(#,#.00#)" };

foreach (string specifier in specifiers)
    Console.WriteLine("{0}: {1}", specifier, value.ToString(specifier,
                      bigIntegerFormatter));

// The example displays the following output:
//       C: (¤903,145,792,771,643,190,182.00)
//       D: ~903145792771643190182
//       D25: ~0000903145792771643190182
//       E: ~9.031457E+020
//       E4: ~9.0314E+020
//       e8: ~9.03145792e+020
//       F0: ~903145792771643190182
//       G: ~903145792771643190182
//       N0: ~903,145,792,771,643,190,182
//       P: ~90,314,579,277,164,319,018,200.00 %
//       R: ~903145792771643190182
//       0,0.000: ~903,145,792,771,643,190,182.000
//       #,#.00#;(#,#.00#): (903,145,792,771,643,190,182.00)
```

注釈

ToString(String,IFormatProvider)メソッドはラウンドトリップ形式で既定のカルチャで変換を行います。特定のカルチャ、特定の形式で変換をしたい場合は、以下のオーバーロードを使用してください。

形式	カルチャ	オーバーロード
ラウンドトリップ形式	既定のカルチャ	ToString()
ラウンドトリップ形式	指定したカルチャ	ToString(IFormatProvider)
指定した形式	既定のカルチャ	ToString(String)

provider パラメーターは、IFormatProvider インタフェースの GetFormat メソッドが実装されたオブジェクトを設定することができます。GetFormat メソッドの戻り値は NumberFormatInfo オブジェクトです。Parse(String,IFormatProvider)メソッドには主に 3 種類方法で provider を渡すことができます。

- CultureInfo が提供する書式を表すオブジェクト。(GetFormat メソッドがそのカルチャに合わせた NumberFormatInfo オブジェクトを返します。)
- 直接インスタンス化した NumberFormatInfo オブジェクト。(GetFormat が NumberFormatInfo オブジェクト自信を返します)

IFormatProvider を実装したカスタムオブジェクト。(そのオブジェクトの GetFormat メソッドが NumberFormatInfo オブジェクトをインスタンス化して返します。)

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.ToUInt16 Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替
> WS.Theia.ExtremelyPrecise.Rational.ToInt32(Raional)

指定した Rational の値を、等価の 16 ビット符号なし整数に変換します。

```
public static ushort ToUInt16(Rational value);
```

パラメーター
value　Rational
変換する Rational。

戻り値
value　UInt16
value と等価の 16 ビット符号なし整数。

例外
OverflowException
Value が MinValue より小さいか MaxValue より大きい場合。

例

次の例では ToUInt16(Rational)メソッドを使って UInt16 値に変換しています。

```
using System;

class Example
{
    public static void Main( )
    {
            Rational[] values = { 123m, new Decimal(123000, 0, 0, false, 3),
                                  123.999m, 65535m, 65535.001m,
                                  32767m, 32767.001m, -0.999m,
                                  -1m,    -32768m, -32768.001m };
foreach (var value in values) {
            try {
                ushort number = Rational.ToUInt16(value);
                Console.WriteLine("{0} --> {1}", value, number);
            }
            catch (OverflowException e)
            {
                 Console.WriteLine("{0}: {1}", e.GetType().Name, value);

            }
        }
    }
}
// The example displays the following output:
//       123 --> 123
//       123.000 --> 123
//       123.999 --> 123
//       65535 --> 65535
//       OverflowException: 65536.001
//       32767 --> 32767
//       32767.001 --> 32767
//       OverflowException: -0.999
//       OverflowException: -1
//       OverflowException: -32768
//       OverflowException: -32768.001
```

注釈

value パラメーターの値から小数点以下の値を切り捨てた値に変換されます。変換結果は UInt16 型への明示的なキャストと等価です。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.ToUInt32 Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替
> WS.Theia.ExtremelyPrecise.Rational.ToInt64(Rational)

指定した Rational の値を、等価の 32 ビット符号なし整数に変換します。

```
public static int ToInt32(Rational value);
```

パラメーター
value　Rational
変換する Rational。

戻り値
value　UInt32
value と等価の 32 ビット符号なし整数。

例外
OverflowException
Value が MinValue より小さいか MaxValue より大きい場合。

例

次の例では ToUInt32(Rational)メソッドを使って UInt32 値に変換しています。

```
using System;

class Example
{
    public static void Main( )
    {
            Rational[] values = { 123m, new decimal(123000, 0, 0, false, 3),
                                  123.999m, 4294967295m, 4294967295.001m,
                                4294967296m, 2147483647m, 2147483647.001m,
                                 -0.999m, -1m, -2147483648m, -2147483648.001m };
        foreach (var value in values) {
            try {
                uint number = Rational.ToUInt32(value);
                Console.WriteLine("{0} --> {1}", value, number);
            }
            catch (OverflowException e)
            {
                 Console.WriteLine("{0}: {1}", e.GetType().Name, value);
            }
        }
    }
}
    // The example displays the following output:
    //          123 --> 123
    //          123.000 --> 123
    //          123.999 --> 123
    //          4294967295 --> 4294967295
    //          OverflowException: 4294967295.001
    //          OverflowException: 4294967296
    //          2147483647 --> 2147483647
    //          2147483647.001 --> 2147483647
    //          OverflowException: -0.999
    //          OverflowException: -1
    //          OverflowException: -2147483648
    //          OverflowException: -2147483648.001
```

注釈

value パラメーターの値から小数点以下の値を切り捨てた値に変換されます。変換結果は UInt32 型への明示的なキャストと等価です。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.ToUInt64 Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替
> WS.Theia.ExtremelyPrecise.Rational.ToDouble(Rational)

指定した Rational の値を、等価の 64 ビット符号なし整数に変換します。

```
public static ulong ToUInt64(Rational value);
```

パラメーター
value　Rational
変換する Rational。

戻り値
value　UInt64
value と等価の 64 ビット符号なし整数。

例外
OverflowException
Value が MinValue より小さいか MaxValue より大きい場合。

注釈

value パラメーターの値から小数点以下の値を切り捨てた値に変換されます。変換結果は UInt64 型への明示的なキャストと等価です。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.TryParse Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
数値の文字列形式を対応する Rational 表現に変換できるかどうかを試行し、変換に成功したかどうかを示す値を返します。

オーバーロード

TryParse(String)	数値の文字列形式を対応する Rational 表現に変換できるかどうかを試行し、変換に成功したかどうかを示す値を返します。
TryParse(String,NumberStyles,IFormatProvider)	数値の文字列形式を対応する Rational 表現に変換できるかどうかを試行し、変換に成功したかどうかを示す値を返します。

TryParse(String)

数値の文字列形式を対応する Rational 表現に変換できるかどうかを試行し、変換に成功したかどうかを示す値を返します。

---

public static (bool status, WS.Theia.ExtremelyPrecise.Rational result)
TryParse(String value)

パラメーター
value　String
数値の文字列形式。

戻り値
status　Boolean
value が正常に変換できた場合は true。それ以外の場合は false。
result　Rational
このメソッドから制御が戻るときに、value と等価の Rational が格納されます。value パラメーターが null の場合、または正しい形式ではない場合、変換は失敗します。変換に失敗した場合このパラメーターは初期化せずに渡されます。

例

次の例では TryParse(String)メソッドを使って 2 つの Rational オブジェクトのインスタンスを生成しています。その後各オブジェクトを乗算し Compare メソッドで 2 つの値の大小を判定しています。

```
Rational number1, number2;
bool succeeded1, succeeded2;
(succeeded1,number1) = Rational.TryParse("-12347534159895123");
(succeeded2,number2) = Rational.TryParse("987654321357159852");
if (succeeded1 && succeeded2)
{
    number1 *= 3;
    number2 *= 2;
    switch (Rational.Compare(number1, number2))
    {
        case -1:
            Console.WriteLine("{0} is greater than {1}.", number2, number1);
            break;
        case 0:
            Console.WriteLine("{0} is equal to {1}.", number1, number2);
            break;
        case 1:
            Console.WriteLine("{0} is greater than {1}.", number1, number2);
            break;
    }
}
else
{
    if (! succeeded1)
        Console.WriteLine("Unable to initialize the first Rational value.");
    if (! succeeded2)
        Console.WriteLine("Unable to initialize the second Rational value.");
}
// The example displays the following output:
//         1975308642714319704 is greater than -37042602479685369.
```

注釈

TryParse(String)メソッドは Parse(String)メソッドと異なり、変換に失敗しても例外を発生しません。FormatException が発生する状況では、戻り値 status が false になり戻り値 result が無効な値になります。

value パラメーターは、次の形式で表された数字文字列でなければなりません。

[ws][sign]digits[ws]

角カッコ（[および]）内の要素は省略可能です。それぞれの要素は次の表のとおりです。

要素	説明
ws	空白文字。省略可能です。
sign	符号。省略可能です。有効な文字はカレントカルチャーの NumberFormatInfo.NegativeSign と NumberFormatInfo.PositiveSign プロパティによって決まります。
digits	数字列。0 から 9 及び小数点で構成している必要があります。先頭の 0 は無視します。有効な小数点はカレントカルチャーの NumberFormatInfo.NumberDecimalSeparator プロパティによって決まります。

TryParse(String,NumberStyles,IFormatProvider)

数値の文字列形式を対応する Rational 表現に変換できるかどうかを試行し、変換に成功したかどうかを示す値を返します。

---

public static (bool status, WS.Theia.ExtremelyPrecise.Rational result)
TryParse(String value,NumberStyles style,IFormatProvider provider);

パラメーター
value　String
数値の文字列形式。
style　NumberStyles
value で存在する可能性を持つスタイル要素を示す、列挙値のビットごとの組み合わせ。通常指定する値は Integer です。
provider　IFormatProvider
value に関するカルチャ固有の書式情報を提供するオブジェクト。

戻り値
status　Boolean
value が正常に変換できた場合は true。それ以外の場合は false。
result　Rational
このメソッドから制御が戻るときに、value と等価の Rational が格納されます。value パラメーターが null の場合、または正しい形式ではない場合、変換は失敗します。変換に失敗した場合このパラメーターは初期化せずに渡されます。

例

次の例では style と provider パラメーターの様々な組み合わせで
TryParse(String,NumberStyle, IFormatProvider)を呼び出しています。

```
string numericString;
Rational number = Rational.Zero;
bool status=false;

// Call TryParse with default values of style and provider.
numericString = "    -300     ";
(status,number)=Rational.TryParse(numericString, NumberStyles.Integer,
                                  null);
if (status)
    Console.WriteLine("'{0}' was converted to {1}.",
                      numericString, number);
else
    Console.WriteLine("Conversion of '{0}' to a Rational failed.",
                      numericString);

// Call TryParse with the default value of style and
// a provider supporting the tilde as negative sign.
numericString = "    -300     ";
(status,number)=Rational.TryParse(numericString, NumberStyles.Integer,
                                  new RationalFormatProvider());
if (status)
    Console.WriteLine("'{0}' was converted to {1}.",
                      numericString, number);
else
    Console.WriteLine("Conversion of '{0}' to a Rational failed.",
                      numericString);

// Call TryParse with only AllowLeadingWhite and AllowTrailingWhite.
// Method returns false because of presence of negative sign.
numericString = "    -500     ";
(status,number)=Rational.TryParse(numericString,
```

```
                                NumberStyles.AllowLeadingWhite |
NumberStyles.AllowTrailingWhite,
                                new RationalFormatProvider());
if (status)
    Console.WriteLine("'{0}' was converted to {1}.",
                                numericString, number);
else
    Console.WriteLine("Conversion of '{0}' to a Rational failed.",
                                numericString);

// Call TryParse with AllowHexSpecifier and a hex value.
numericString = "F14237FFAAC086455192";
(status,number)=Rational.TryParse(numericString,
                                NumberStyles.AllowHexSpecifier, null);
if (status)
    Console.WriteLine("'{0}' was converted to {1} (0x{1:x}).",
                                numericString, number);
else
    Console.WriteLine("Conversion of '{0}' to a Rational failed.",
                                numericString);

// Call TryParse with AllowHexSpecifier and a negative hex value.
// Conversion fails because of presence of negative sign.
numericString = "-3af";
(status,number)=Rational.TryParse(numericString,
NumberStyles.AllowHexSpecifier,new RationalFormatProvider());
if (status)
    Console.WriteLine("'{0}' was converted to {1}.",
                                numericString, number);
else
    Console.WriteLine("Conversion of '{0}' to a Rational failed.",
                                numericString);

// Call TryParse with only NumberStyles.None.
// Conversion fails because of presence of white space and sign.
numericString = " -300 ";
```

```
(status,number)=Rational.TryParse(numericString, NumberStyles.None,
                                  new RationalFormatProvider());
if (status)
    Console.WriteLine("'{0}' was converted to {1}.",
                                  numericString, number);
else
    Console.WriteLine("Conversion of '{0}' to a Rational failed.",
                                  numericString);

// Call TryParse with NumberStyles.Any and a provider for the fr-FR culture.
// Conversion fails because the string is formatted for the en-US culture.
numericString = "9,031,425,666,123,546.00";
(status,number)=Rational.TryParse(numericString, NumberStyles.Any,
                                  new CultureInfo("fr-FR"));
if (status)
    Console.WriteLine("'{0}' was converted to {1}.",
                                  numericString, number);
else
    Console.WriteLine("Conversion of '{0}' to a Rational failed.",
                                  numericString);

// Call TryParse with NumberStyles.Any and a provider for the fr-FR culture.
// Conversion succeeds because the string is properly formatted
// For the fr-FR culture.
numericString = "9 031 425 666 123 546,00";
(status,number)=Rational.TryParse(numericString, NumberStyles.Any,
                                  new CultureInfo("fr-FR"));
if (status)
    Console.WriteLine("'{0}' was converted to {1}.",
                                  numericString, number);
else
    Console.WriteLine("Conversion of '{0}' to a Rational failed.",
                                  numericString);
// The example displays the following output:
//       '    -300     ' was converted to -300.
//       Conversion of '    -300     ' to a Rational failed.
```

```
//       Conversion of '    -500      ' to a Rational failed.
//       'F14237FFAAC086455192' was converted to -
69613977002644837412462 (0xf14237ffaac086455192).
//       Conversion of '-3af' to a Rational failed.
//       Conversion of ' -300 ' to a Rational failed.
//       Conversion of '9,031,425,666,123,546.00' to a Rational failed.
//       '9 031 425 666 123 546,00' was converted to 9031425666123546.
```

TryParse(String,NumberStyle, IFormatProvider)メソッドを呼び出す際に使っている RationalFormatProvider クラスは、負の符号としてチルダ（~）を定義しています。

```
public class RationalFormatProvider : IFormatProvider
{
    public object GetFormat(Type formatType)
    {
        if (formatType == typeof(NumberFormatInfo))
        {
            NumberFormatInfo numberFormat = new NumberFormatInfo();
            numberFormat.NegativeSign = "~";
            return numberFormat;
        }
        else
        {
            return null;
        }
    }
}
```

注釈

TryParse(String,NumberStyles,IFormatProvider)メソッドは Parse(String,NumberStyles,IFormatProvider)メソッドと異なり、変換に失敗しても例外を発生しません。FormatException が発生する状況では、戻り値 status が false になり戻り値 result が無効な値になります。

style パラメーターは空白、符号、桁区切り記号、小数点記号など使用することのできる文字を指定することができます。value パラメーターは、次の形式のうち style パラメーターで許可された要素を数字列に含めることができます。
[ws][$][sign][digits,]digits[.fractional_digits][E[sign]exponential_digits][ws]
角カッコ（[および]）内の要素は省略可能です。それぞれの要素は次の表のとおりです。

要素	説明
ws	空白文字。省略可能です。 NumberStyles.AllowLeadingWhite フラグおよび NumberStyles.AllowTrailingWhite フラグで使用可能かが決まります。
$	通貨記号。有効な文字はカレントカルチャーの NumberFormatInfo.CurrencyNegativePattern と NumberFormatInfo.CurrencyPositivePattern プロパティの値で決まります。 NumberStyles.AllowCurrencySymbol フラグが有効な時に使用可能になります。
sign	符号。有効な文字はカレントカルチャーの NumberFormatInfo.NegativeSign と NumberFormatInfo.PositiveSign プロパティによって決まります。
digits fractional_digits exponential_digits	数字列。0 から 9 及び小数点で構成している必要があります。 fractional_digits 以外では先頭の 0 は無視します。
,	数字の桁区切りです。有効な文字はカレントカルチャーの CurrencyGroupSeparator と NumberGroupSeparator と PercentGroupSeparator プロパティで決まります。 NumberStyles.AllowThousands フラグが有効な時に使用可能になります。
.	小数点記号です。有効な文字はカレントカルチャーの CurrencyDecimalSeparator、NumberDecimalSeparator、 PercentDecimalSeparator プロパティで決まります。 NumberStyles.AllowDecimalPoint フラグが有効な時に使用可能になります。
E	“e”または”E”文字は、指数表記で表されている事を示します。 NumberStyles.AllowExponent フラグが有効な時使用可能になります。

数字のみを含む文字列（NumberStyle.None が対応）は常に正常に解析できます。他の NumberStyle メンバーは多くの要素を許可しますが、value パラメーターにはその全ての要素を含んでいる必要はありません。

None	数字のみです。
AllowDecimalPoint	整数部、小数点（.）と桁の小数部を許容します。
AllowExponent	"e"または"E"文字と共に、指数部を許容します。
AllowLeadingWhite	value の先頭に空白がある事を許容します。
AllowTrailingWhite	value の末尾に空白がある事を許容します。
AllowLeadingSign	value の先頭に符号がある事を許容します。
AllowTrailingSign	value の末尾に符号がある事を許容します。
AllowParentheses	負数をカッコで囲って表記する事を許容します。
AllowThousands	整数部を桁区切りする事を許容します。
AllowCurrencySymbol	通貨記号を使用する事を許容します。
Currency	すべての要素を許容します。 ただし、 value プロパティを 16 進数または指数表記で表す事はできません。
Float	value の先頭、末尾の空白、value 先頭の符号、および小数点、指数表記を許容します。
Number	value の先頭、末尾の空白、value 先頭、末尾の符号、桁区切り記号、および小数点といった 10 進数の全ての要素を許容します。
Any	すべての要素を許容します。 ただし、value パラメーターを 16 進数で表記する事はできません。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Xor(Rational, Rational) Method

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational 値に対し、ビットごとの排他的 Or (XOr) 演算を実行します。

---

public static WS.Theia.ExtremelyPrecise.Rational
Xor(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);

パラメーター
left　Rational
排他的 Or 演算する最初の値。
right　Rational
排他的 Or 演算する 2 番目の値。

戻り値
Rational
ビットごとの排他的 Or 演算の結果。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値の排他的論理和をする代替メソッド。left と right のビットと戻り値のビットの関係性は下表の様になります。

left 内のビット値	right 内のビット値	戻り値内のビット値
0	0	0
1	0	1
1	1	0
0	1	1

Xor メソッドは、次の例の様に使用します。

```
Rational number1 = Math.Pow(2, 127);
Rational number2 = Rational.Multiply(163, 124);
Rational result = Rational.Xor(number1,number2);
```

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.Addition(Rational,Rational)

Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された 2 つの Rational オブジェクトの値を加算します。

```
public static WS.Theia.ExtremelyPrecise.Rational operator
+( WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター
left　Rational
加算する 1 番目の値。
right　Rational
加算する 2 番目の値。

戻り値
Rational
left と right の合計。

注釈

Rational 値を加算して変数に割り当てる時は次の例の様に使用する。

```
Rational num1 = 1000456321;
Rational num2 = 90329434;
Rational sum = num1 + num2;
```

カスタム演算子をサポートしない言語では Add メソッドを代わりに使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.BitwiseAnd(Rational, Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational 値に対し、ビットごとの And 演算を実行します。

```
public static WS.Theia.ExtremelyPrecise.Rational operator
&( WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター
left　Rational
And 演算する最初の値。
right　Rational
And 演算する 2 番目の値。

戻り値
Rational
ビットごとの And 演算の結果。

注釈

Rational 値を And 演算する演算子。left と right のビットと戻り値のビットの関係性は下表の様になります。

left 内のビット値	right 内のビット値	戻り値内のビット値
0	0	0
1	0	0
1	1	1
0	1	0

BitwiseAnd 演算子は、次の例の様に使用します。

```
Rational number1 = Rational.Add(Int64.MaxValue, Int32.MaxValue);
Rational number2 = Math.Pow(Byte.MaxValue, 10);
Rational result = number1 & number2;
```

カスタム演算子をサポートしない言語では BitwiseAnd メソッドを代わりに使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.BitwiseOr(Rational, Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational 値に対し、ビットごとの Or 演算を実行します。

```
public static WS.Theia.ExtremelyPrecise.Rational operator
|( WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター
left　Rational
Or 演算する最初の値。
right　Rational
Or 演算する 2 番目の値。

戻り値
Rational
ビットごとの Or 演算の結果。

注釈

Rational 値を Or 演算する演算子。left と right のビットと戻り値のビットの関係性は下表の様になります。

left 内のビット値	right 内のビット値	戻り値内のビット値
0	0	0
1	0	1
1	1	1
0	1	1

BitwiseAnd 演算子は、次の例の様に使用します。

```
Rational number1  = Rational.Parse("10343901200000000000");
Rational number2  = Byte.MaxValue;
Rational result   = number1 | number2;
```

カスタム演算子をサポートしない言語では BitwiseOr メソッドを代わりに使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational. Decrement(Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値を 1 だけデクリメントします。

```
public static WS.Theia.ExtremelyPrecise.Rational operator --
( WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
デクリメントする値。

戻り値
Rational
value パラメーターの値を 1 だけデクリメントした値

注釈

Decrement 演算子は次のように使用します

```
Rational number = 93843112;
Console.WriteLine(--number);                    // Displays 93843111
```

カスタム演算子をサポートしない言語では Decrement メソッドを代わりに使用します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Division(Rational, Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
一方の Rational 値をもう一方の値で除算し、その結果を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
Add(WS.Theia.ExtremelyPrecise.Rational dividend,
WS.Theia.ExtremelyPrecise.Rational divisor);
```

パラメーター
dividend　Rational
被除数。
divisor　Rational
除数。

戻り値
Rational
除算の結果。

例

次の例では Rational 配列の各要素に、Divide メソッド、除算演算子（/）。及び DivRem メソッドを使用している。

```
using System;
using System.Numerics;
```

```
public class Example
{
    public static void Main()
    {
        Rational divisor = Math.Pow(Int64.MaxValue, 2);

        Rational [] dividends = { Rational.Multiply((Rational) Single.MaxValue,
2),

Rational.Parse("906123451238755090918275600071000 99"),
                                  Rational.One,
                                  Rational.Multiply(Int32.MaxValue,
Int64.MaxValue),
                                  divisor + Rational.One };
        // Divide each dividend by divisor in three different ways.
        foreach (Rational dividend in dividends)
        {
            Rational quotient;
            Rational remainder = 0;

            Console.WriteLine("Dividend: {0:N0}", dividend);
            Console.WriteLine("Divisor:  {0:N0}", divisor);
            Console.WriteLine("Results:");
            Console.WriteLine("    Using Divide method:      {0:N0}",
                              Rational.Divide(dividend, divisor));
            Console.WriteLine("    Using Division operator: {0:N0}",
                              dividend / divisor);
            (quotient, remainder) = Math.DivRem(dividend, divisor);
            Console.WriteLine("    Using DivRem method:      {0:N0},
remainder {1:N0}",
                              quotient, remainder);

            Console.WriteLine();
        }
    }
}
```

```
// The example displays the following output:
//       Dividend: 680,564,693,277,057,719,623,408,366,969,033,850,880
//       Divisor:  85,070,591,730,234,615,847,396,907,784,232,501,249
//       Results:
//           Using Divide method:        7
//           Using Division operator: 7
//           Using DivRem method:        7, remainder
85,070,551,165,415,408,691,630,012,479,406,342,137
//
//       Dividend: 90,612,345,123,875,509,091,827,560,007,100,099
//       Divisor:  85,070,591,730,234,615,847,396,907,784,232,501,249
//       Results:
//           Using Divide method:        0
//           Using Division operator: 0
//           Using DivRem method:        0, remainder
90,612,345,123,875,509,091,827,560,007,100,099
//
//       Dividend: 1
//       Divisor:  85,070,591,730,234,615,847,396,907,784,232,501,249
//       Results:
//           Using Divide method:        0
//           Using Division operator: 0
//           Using DivRem method:        0, remainder 1
//
//       Dividend: 19,807,040,619,342,712,359,383,728,129
//       Divisor:  85,070,591,730,234,615,847,396,907,784,232,501,249
//       Results:
//           Using Divide method:        0
//           Using Division operator: 0
//           Using DivRem method:        0, remainder
19,807,040,619,342,712,359,383,728,129
//
//       Dividend: 85,070,591,730,234,615,847,396,907,784,232,501,250
//       Divisor:  85,070,591,730,234,615,847,396,907,784,232,501,249
//       Results:
//           Using Divide method:        1
```

```
//          Using Division operator: 1
//          Using DivRem method:     1, remainder 1
```

注釈

カスタム演算子をサポートしない言語では Division メソッドを代わりに使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.Equality(Rational,Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational オブジェクトの値が等しいかどうかを示す値を返します。

```
public static bool operator ==(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメータ
left　Rational
比較する最初の値。
right Rational
比較する 2 番目の値。

戻り値
left パラメータと right パラメータが同じ値の場合は true。それ以外の場合は false。

注釈

Rational 値を比較する演算子。Rational 値を比較して変数に割り当てる時は次の例の様に使用する。

```
Rational number1 = 945834723;
Rational number2 = 345145625;
Rational number3 = 945834723;
Console.WriteLine(number1 == number2);          // Displays False
Console.WriteLine(number1 == number3);          // Displays True
```

カスタム演算子をサポートしない言語では Equals メソッドを代わりに使用します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.ExclusiveOr(Rational, Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational 値に対し、ビットごとの排他的 Or (XOr) 演算を実行します。

```
public static WS.Theia.ExtremelyPrecise.Rational operator
^( WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター
left　Rational
排他的 Or 演算する最初の値。
right　Rational
排他的 Or 演算する 2 番目の値。

戻り値
Rational
ビットごとの排他的 Or 演算の結果。

注釈

Rational 値の排他的論理和演算をする演算子。left と right のビットと戻り値のビットの関係性は下表の様になります。

left 内のビット値	right 内のビット値	戻り値内のビット値
0	0	0
1	0	1
1	1	0
0	1	1

ExclusiveOr 演算子は、次の例の様に使用します。

```
Rational number1 = Math.Pow(2, 127);
Rational number2 = Rational.Multiply(163, 124);
Rational result = number1 ^ number2;
```

カスタム演算子をサポートしない言語では Xor メソッドを代わりに使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.Explicit Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll

オーバーロード

Explicit(Rational to Byte)	Rational オブジェクトから Byte 値への明示的な変換を定義します。
Explicit(Rational to SByte)	Rational オブジェクトから SByte 値への明示的な変換を定義します。
Explicit(Rational to Int32)	Rational オブジェクトから Int32 値への明示的な変換を定義します。
Explicit(Rational to UInt32)	Rational オブジェクトから UInt32 値への明示的な変換を定義します。
Explicit(Rational to Int16)	Rational オブジェクトから Int16 値への明示的な変換を定義します。
Explicit(Rational to UInt16)	Rational オブジェクトから UInt16 値への明示的な変換を定義します。
Explicit(Rational to Int64)	Rational オブジェクトから Int64 値への明示的な変換を定義します。
Explicit(Rational to UInt64)	Rational オブジェクトから UInt64 値への明示的な変換を定義します。
Explicit(Rational to Single)	Rational オブジェクトから Single 値への明示的な変換を定義します。
Explicit(Rational to Double)	Rational オブジェクトから Double 値への明示的な変換を定義します。
Explicit(Rational to Boolean)	Rational オブジェクトから Boolean 値への明示的な変換を定義します。
Explicit(Rational to Decimal)	Rational オブジェクトから Decimal 値への明示的な変換を定義します。

Explicit(Rational to Byte)

Rational 値から Byte 値への明示的な変換を定義します。

```
public static explicit operator byte(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメータ

value　Rational

Byte 値へと変換する値。

戻り値

Byte

value パラメータと等価な Byte 値。

例外

OverFlowException

Value が MinValue より小さいか MaxValue より大きいです。

例

次の例では、Rational 型を Byte 型に変換する方法示します。

```
// Rational to Byte conversion.
Rational goodByte = Rational.One;

Rational badByte = 256;

byte byteFromRational;

// Successful conversion using cast operator.
byteFromRational = (byte) goodByte;
Console.WriteLine(byteFromRational);

// Handle conversion that should result in overflow.
try
{
    byteFromRational = (byte) badByte;
    Console.WriteLine(byteFromRational);
}
catch (OverflowException e)
{
    Console.WriteLine("Unable to convert {0}:¥n     {1}",
                      badByte, e.Message);
}
Console.WriteLine();
```

注釈

Byte 型に変換する前に value パラメータの小数部を切り捨てが行われます。また、Rational 型から Byte 型への暗黙的な変換は行われません。Rational 型の表現範囲が大きく、変換を行うとオーバーフローが発生する恐れがあるためです。

Explicit(Rational to SByte)

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替　System.Int16

Rational 値から SByte 値への明示的な変換を定義します。

```
public static explicit operator sbyte(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメータ
value　Rational
SByte 値 へと変換する値。

戻り値
SByte
value パラメータと等価な SByte 値。

例外
OverFlowException
Value が MinValue より小さいか MaxValue より大きいです。

例

次の例では、Rational 型を SByte 型に変換する方法示します。

```
// Rational to Byte conversion.
Rational goodByte = Rational.One;

Rational badByte = 256;

byte byteFromRational;

// Successful conversion using cast operator.
byteFromRational = (byte) goodByte;
Console.WriteLine(byteFromRational);

// Handle conversion that should result in overflow.
try
{
    byteFromRational = (byte) badByte;
    Console.WriteLine(byteFromRational);
}
catch (OverflowException e)
{
    Console.WriteLine("Unable to convert {0}:¥n     {1}",
                      badByte, e.Message);
}
Console.WriteLine();
```

注釈

SByte 型に変換する前に value パラメータの小数部を切り捨てが行われます。また、Rational 型から SByte 型への暗黙的な変換は行われません。Rational 型の表現範囲が大きく、変換を行うとオーバーフローが発生する恐れがあるためです。

Explicit(Rational to Int32)

Rational 値から Int32 値への明示的な変換を定義します。

```
public static explicit operator int(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメータ

value　Rational

Int32 値へと変換する値。

戻り値

Int32

value パラメータと等価な Int32 値。

例外

OverFlowException

Value が MinValue より小さいか MaxValue より大きいです。

例

次の例では、Rational 型を Int32 型に変換する方法示します。

```
// Rational to Int32 conversion.
Rational goodInteger = 200000;
Rational badInteger = 65000000000;

int integerFromRatioanl;

// Successful conversion using cast operator.
integerFromRatioanl = (int) goodInteger;
Console.WriteLine(integerFromRatioanl);

// Handle conversion that should result in overflow.
try
{
    integerFromRatioanl = (int) badInteger;
    Console.WriteLine(integerFromRatioanl);
}
catch (OverflowException e)
{
    Console.WriteLine("Unable to convert {0}:¥n     {1}",
                      badInteger, e.Message);
}
Console.WriteLine();
```

注釈

Int32 型に変換する前に value パラメータの小数部を切り捨てが行われます。また、Rational 型から Int32 型への暗黙的な変換は行われません。Rational 型の表現範囲が大きく、変換を行うとオーバーフローが発生する恐れがあるためです。

Explicit(Rational to UInt32)

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替　System.Int64

Rational 値から UInt32 値への明示的な変換を定義します。

```
public static explicit operator uint(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメータ
value　Rational
UInt32 値へと変換する値。

戻り値
UInt32
value パラメータと等価な UInt32 値。

例外
OverFlowException
Value が MinValue より小さいか MaxValue より大きいです。

例

次の例では、Rational 型を UInt32 型に変換する方法示します。

```
// Rational to UInt32 conversion.
Rational goodUInteger = 200000;
Rational badUInteger = 65000000000;

uint uIntegerFromRational;

// Successful conversion using cast operator.
uIntegerFromRational = (uint) goodUInteger;
Console.WriteLine(uIntegerFromRational);

// Handle conversion that should result in overflow.
try
{
    uIntegerFromRational = (uint) badUInteger;
    Console.WriteLine(uIntegerFromRational);
}
catch (OverflowException e)
{
    Console.WriteLine("Unable to convert {0}:¥n    {1}",
                      badUInteger, e.Message);
}
Console.WriteLine();
```

注釈

UInt32 型に変換する前に value パラメータの小数部を切り捨てが行われます。また、Rational 型から UInt32 型への暗黙的な変換は行われません。Rational 型の表現範囲が大きく、変換を行うとオーバーフローが発生する恐れがあるためです。

Explicit(Rational to Int16)

Rational 値から Int16 値への明示的な変換を定義します。

```
public static explicit operator short(WS.Theia.ExtremelyPrecise.Rational
value);
```

パラメータ

value　Rational

Int16 値へと変換する値。

戻り値

Int16

value パラメータと等価な Int16 値。

例外

OverFlowException

Value が MinValue より小さいか MaxValue より大きいです。

例

次の例では、Rational 型を Int16 型に変換する方法示します。

```
// Rational to Int16 conversion.
Rational goodShort = 20000;
Rational badShort = 33000;

short shortFromRational;

// Successful conversion using cast operator.
shortFromRational = (short) goodShort;
Console.WriteLine(shortFromRational);

// Handle conversion that should result in overflow.
try
{
    shortFromRational = (short) badShort;
    Console.WriteLine(shortFromRational);
}
catch (OverflowException e)
{
    Console.WriteLine("Unable to convert {0}:¥n    {1}",
                      badShort, e.Message);
}
Console.WriteLine();
```

注釈

Int16 型に変換する前に value パラメータの小数部を切り捨てが行われます。また、Rational 型から Int16 型への暗黙的な変換は行われません。Rational 型の表現範囲が大きく、変換を行うとオーバーフローが発生する恐れがあるためです。

Explicit(Rational to UInt16)

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替　System.Int32

Rational 値から UInt16 値への明示的な変換を定義します。

---

public static explicit operator ushort(WS.Theia.ExtremelyPrecise.Rational value);

パラメータ
value　Rational
UInt16 値へと変換する値。

戻り値
UInt16
value パラメータと等価な UInt16 値。

例外
OverFlowException
Value が MinValue より小さいか MaxValue より大きいです。

例

次の例では、Rational 型を UInt16 型に変換する方法示します。

```
// Rational to UInt16 conversion.
Rational goodUShort = 20000;
Rational badUShort = 66000;

ushort uShortFromRational;

// Successful conversion using cast operator.
uShortFromRational = (ushort) goodUShort;
Console.WriteLine(uShortFromRational);

// Handle conversion that should result in overflow.
try
{
    uShortFromRational = (ushort) badUShort;
    Console.WriteLine(uShortFromRational);
}
catch (OverflowException e)
{
    Console.WriteLine("Unable to convert {0}:¥n    {1}",
                      badUShort, e.Message);
}
Console.WriteLine();
```

注釈

UInt16型に変換する前にvalueパラメータの小数部を切り捨てが行われます。また、Rational 型から UInt16 型への暗黙的な変換は行われません。Rational 型の表現範囲が大きく、変換を行うとオーバーフローが発生する恐れがあるためです。

Explicit(Rational to Int64)

Rational 値から Int64 値への明示的な変換を定義します。

```
public static explicit operator long(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメータ

value　Rational

Int64 値へと変換する値。

戻り値

Int64

value パラメータと等価な Int64 値。

例外

OverFlowException

Value が MinValue より小さいか MaxValue より大きいです。

例

次の例では、Rational 型を Int64 型に変換する方法示します。

```
// Rational to Int64 conversion.
Rational goodLong = 2000000000;
Rational badLong = Math.Pow(goodLong, 3);

long longFromRational;

// Successful conversion using cast operator.
longFromRational = (long) goodLong;
Console.WriteLine(longFromRational);

// Handle conversion that should result in overflow.
try
{
    longFromRational = (long) badLong;
    Console.WriteLine(longFromRational);
}
catch (OverflowException e)
{
    Console.WriteLine("Unable to convert {0}:¥n    {1}",
                      badLong, e.Message);
}
Console.WriteLine();
```

注釈

Int64 型に変換する前に value パラメータの小数部を切り捨てが行われます。また、Rational 型から Int64 型への暗黙的な変換は行われません。Rational 型の表現範囲が大きく、変換を行うとオーバーフローが発生する恐れがあるためです。

Explicit(Rational to UInt64)

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替　System.Double

Rational 値から UInt64 値への明示的な変換を定義します。

---

public static explicit operator ulong(WS.Theia.ExtremelyPrecise.Rational value);

パラメータ
value　Rational
UInt64 値へと変換する値。

戻り値
UInt64
value パラメータと等価な UInt64 値。

例外
OverFlowException
Value が MinValue より小さいか MaxValue より大きいです。

例

次の例では、Rational 型を UInt64 型に変換する方法示します。

```
// Rational to UInt64 conversion.
Rational goodULong = 2000000000;
Rational badULong = Math.Pow(goodULong, 3);

ulong uLongFromRational;

// Successful conversion using cast operator.
uLongFromRational = (ulong) goodULong;
Console.WriteLine(uLongFromRational);

// Handle conversion that should result in overflow.
try
{
    uLongFromRational = (ulong) badULong;
    Console.WriteLine(uLongFromRational);
}
catch (OverflowException e)
{
    Console.WriteLine("Unable to convert {0}:¥n    {1}",
                      badULong, e.Message);
}
Console.WriteLine();
```

注釈

UInt64型に変換する前にvalueパラメータの小数部を切り捨てが行われます。また、Rational 型から UInt64 型への暗黙的な変換は行われません。Rational 型の表現範囲が大きく、変換を行うとオーバーフローが発生する恐れがあるためです。

Explicit(Rational to Single)

Rational 値から単精度浮動小数点への明示的な変換を定義します。

```
public static explicit operator float(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメータ

value　Rational

単精度浮動小数点へと変換する値。

戻り値

Single

value パラメータと等価な単精度浮動小数点。

例外

OverFlowException

Value が MinValue より小さいか MaxValue より大きいです。

例

次の例では、Rational 型を単精度浮動小数点型に変換する方法示します。

```
// Rational to Single conversion.
Rational goodSingle = 102.43e22F;
Rational badSingle = float.MaxValue;
badSingle = badSingle * 2;

float singleFromRational;

// Successful conversion using cast operator.
singleFromRational = (float) goodSingle;
Console.WriteLine(singleFromRational);

// Convert an out-of-bounds Rational value to a Single.
singleFromRational = (float) badSingle;
Console.WriteLine(singleFromRational);
```

注釈

Rational 型から単精度浮動小数点型への暗黙的な変換は行われません。Rational 型の表現範囲が大きく、変換を行うとオーバーフローが発生する恐れがあるためです。

Explicit(Rational to Double)

Rational 値から倍精度浮動小数点値への明示的な変換を定義します。

```
public static explicit operator double(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメータ

value　Rational

倍精度浮動小数点値へと変換する値。

戻り値

Double

value パラメータと等価な倍精度浮動小数点値。

例

次の例では、Rational 型を倍精度浮動小数点型に変換する方法示します。

```
// Rational to Double conversion.
Rational goodDouble = 102.43e22;
Rational badDouble = Double.MaxValue;
badDouble = badDouble * 2;

double doubleFromRational;

// successful conversion using cast operator.
doubleFromRational = (double) goodDouble;
Console.WriteLine(doubleFromRational);

// Convert an out-of-bounds Rational value to a Double.
doubleFromRational = (double) badDouble;
Console.WriteLine(doubleFromRational);
```

注釈

Rational 型から倍精度浮動小数点型への暗黙的な変換は行われません。Rational 型の表現範囲が大きく、変換を行うとオーバーフローが発生する恐れがあるためです。また、変換元の値が MaxValue より大きい場合は PositiveInfinity、MinValue より小さい場合は NegativeInfinity を返却します。

Explicit(Rational to Boolean)

Rational 値から Boolean 値への明示的な変換を定義します。

```
public static explicit operator bool(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメータ

value　Rational

Boolean 値へと変換する値。

戻り値

Boolean

value パラメータと等価な Boolean 値。(0 の場合 false、それ以外の場合 true)

例

次の例では、Rational 型を Boolean 型に変換する方法示します。

```
// Rational to Boolean conversion.
Rational trueBoolean = 102.43e22;
Rational falseBoolean = 0;

bool booleanFromRational;

// successful conversion using cast operator in value of true.
booleanFromRational = (bool) trueBoolean;
Console.WriteLine(booleanFromRational);

// successful conversion using cast operator in value of false.
booleanFromRational = (bool) falseBoolean;
Console.WriteLine(booleanFromRational);
```

Explicit(Rational to Decimal)

Rational 値から Decimal 値への明示的な変換を定義します。

```
public static explicit operator decimal(WS.Theia.ExtremelyPrecise.Rational value);
```

パラメータ

value　Rational

Decimal 値へと変換する値。

戻り値

Decimal

value パラメータと等価な Decimal 値。

例外

OverFlowException

Value が MinValue より小さいか MaxValue より大きいです。

例

次の例では、Rational 型を Decimal 型に変換する方法示します。

```
// Rational to Decimal conversion.
Rational goodDecimal = 761652543;
Rational badDecimal = Decimal.MaxValue;
badDecimal += Rational.One;

Decimal decimalFromRational;

// Successful conversion using cast operator.
decimalFromRational = (decimal) goodDecimal;
Console.WriteLine(decimalFromRational);

// Handle conversion that should result in overflow.
try
{
    decimalFromRational = (decimal) badDecimal;
    Console.WriteLine(decimalFromRational);
}
catch (OverflowException e)
{
    Console.WriteLine("Unable to convert {0}:¥n     {1}",
                      badDecimal, e.Message);
}
Console.WriteLine();
```

注釈

Rational 型から Decimal 型への暗黙的な変換は行われません。Rational 型の表現範囲が大きく、変換を行うとオーバーフローが発生する恐れがあるためです。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.GreaterThan(Rational,Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値がもう 1 つの Rational 値より大きいかどうかを示す値を返します。

```
public static bool operator >(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメータ
left Rational
比較する最初の値です。
right Rational
比較する 2 番目の値です。

戻り値
true が left より大きい場合は right。それ以外の場合は false。

注釈

Rational 値を比較する演算子。Rational 値を比較して変数に割り当てる時は次の例の様に使用する。

```
Rational number1 = 945834723;
Rational number2 = 345145625;
Rational number3 = 945834724;
Console.WriteLine(number1 > number2);          // Displays True
Console.WriteLine(number1 > number3);          // Displays False
```

カスタム演算子をサポートしない言語では Compare メソッドを代わりに使用します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.GreaterThanOrEqual(Rational,Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値がもう 1 つの Rational 値以上かどうかを示す値を返します。

```
public static bool operator >=(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメータ

left Rational
比較する最初の値。
right Rational
比較する 2 番目の値。

戻り値

true が left より大きい場合は right。それ以外の場合は false。

注釈

Rational 値を比較する演算子。Rational 値を比較して変数に割り当てる時は次の例の様に使用する。

```
Rational number1 = 945834723;
Rational number2 = 345145625;
Rational number3 = 945834724;
Rational number4 = 945834723;
Console.WriteLine(number1 >= number2);          // Displays True
Console.WriteLine(number1 >= number3);          // Displays False
Console.WriteLine(number1 >= number4);          // Displays True
```

カスタム演算子をサポートしない言語では Compare メソッドを代わりに使用します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Implicit Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll

オーバーロード

Implicit(Byte to Rational)	Byte 値から Rational 値への暗黙的な変換を定義します。
Implicit(SByte to Rational)	SByte 値から Rational 値への暗黙的な変換を定義します。
Implicit(Int32 to Rational)	Int32 値から Rational 値への暗黙的な変換を定義します。
Implicit(UInt32 to Rational)	UInt32 値から Rational 値への暗黙的な変換を定義します。
Implicit(Int16 to Rational)	Int16 値から Rational 値への暗黙的な変換を定義します。
Implicit(UInt16 to Rational)	UInt16 値から Rational 値への暗黙的な変換を定義します。
Implicit(Int64 to Rational)	Int64 値から Rational 値への暗黙的な変換を定義します。
Implicit(UInt64 to Rational)	UInt64 値から Rational 値への暗黙的な変換を定義します。
Implicit(Single to Rational)	Single 値から Rational 値への暗黙的な変換を定義します。
Implicit(Double to Rational)	Double 値から Rational 値への暗黙的な変換を定義します。
Implicit(Boolean to Rational)	Boolean 値から Rational 値への暗黙的な変換を定義します。
Implicit(Decimal to Rational)	Decimal 値から Rational 値への暗黙的な変換を定義します。

Implicit(Byte to Rational)

Byte 値から Rational 値への暗黙的な変換を定義します。

```
public static implicit operator WS.Theia.ExtremelyPrecise.Rational(byte value);
```

パラメータ

value　Byte

Rational へと変換する値。

戻り値

Rational

value パラメータと等価な Rational 値。

注釈

次の例では、Byte 型を Rational 型に変換する方法示します。暗黙的な変換演算子をサポートしない言語では、代わりに Rational.Rational(Int32)メソッドを使用します。

```
byte byteValue = 254;
Rational number = byteValue;
number = Rational.Add(number, byteValue);
Console.WriteLine(number > byteValue);                  // Displays True
```

Implicit(SByte to Rational)

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替
> WS.Theia.ExtremelyPrecise.Rational.Implicit(Int32 to Rational)

```
public static implicit operator WS.Theia.ExtremelyPrecise.Rational(sbyte value);
```

SByte 値から Rational 値への暗黙的な変換を定義します。

パラメータ
value　SByte
Rational へと変換する値。

戻り値
Rational
value パラメータと等価な Rational 値。

注釈
次の例では、SByte 型を Rational 型に変換する方法示します。暗黙的な変換演算子をサポートしない言語では、代わりに Rational.Rational(Int32)メソッドを使用します。

```
sbyte sByteValue = -12;
Rational number = Math.Pow(sByteValue, 3);
Console.WriteLine(number < sByteValue);                    // Displays True
```

Implicit(Int32 to Rational)

Int32 値から Rational 値への暗黙的な変換を定義します。

```
public static implicit operator WS.Theia.ExtremelyPrecise.Rational(int value);
```

Int32 値から Rational 値への暗黙的な変換を定義します。

パラメータ
value　Int32
Rational へと変換する値。

戻り値
Rational
value パラメータと等価な Rational 値。

注釈
次の例では、Int32 型を Rational 型に変換する方法示します。暗黙的な変換演算子をサポートしない言語では、代わりに Rational.Rational(Int32)メソッドを使用します。

```
int intValue = 65000;
Rational number = intValue;
number = Rational.Multiply(number, intValue);
Console.WriteLine(number == intValue);                  // Displays False
```

Implicit(UInt32 to Rational)

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替
> WS.Theia.ExtremelyPrecise.Rational.Implicit(Int64 to Rational)

UInt32 値から Rational 値への暗黙的な変換を定義します。

```
public static implicit operator WS.Theia.ExtremelyPrecise.Rational(uint value);
```

パラメータ
value　UInt32
Rational へと変換する値。

戻り値
Rational
value パラメータと等価な Rational 値。

注釈
次の例では、UInt32 型を Rational 型に変換する方法示します。暗黙的な変換演算子をサポートしない言語では、代わりに Rational.Rational(Int64)メソッドを使用します。

```
uint uIntValue = 65000;
Rational number = uIntValue;
number = Rational.Multiply(number, uIntValue);
Console.WriteLine(number == uIntValue);                // Displays False
```

Implicit(Int16 to Rational)

Int16 値から Rational 値への暗黙的な変換を定義します。

```
public static implicit operator WS.Theia.ExtremelyPrecise.Rational(short
value);
```

パラメータ

value　Int16

Rational へと変換する値。

戻り値

Rational

value パラメータと等価な Rational 値。

注釈

次の例では、Int16 型を Rational 型に変換する方法示します。暗黙的な変換演算子をサポートしない言語では、代わりに Rational.Rational(Int32)メソッドを使用します。

```
short shortValue = 25064;
Rational number = shortValue;
number += shortValue;
Console.WriteLine(number < shortValue);                // Displays False
```

Implicit(UInt16 to Rational)

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替
> WS.Theia.ExtremelyPrecise.Rational.Implicit(Int32 to Rational)

UInt16 値から Rational 値への暗黙的な変換を定義します。

```
public static implicit operator WS.Theia.ExtremelyPrecise.Rational(ushort value);
```

パラメータ
value　UInt16
Rational へと変換する値。

戻り値
Rational
value パラメータと等価な Rational 値。

注釈
次の例では、UInt16 型を Rational 型に変換する方法示します。暗黙的な変換演算子をサポートしない言語では、代わりに Rational.Rational(Int32)メソッドを使用します。

```
ushort uShortValue = 25064;
Rational number = uShortValue;
number += uShortValue;
Console.WriteLine(number < uShortValue);                // Displays False
```

Implicit(Int64 to Rational)

Int64 値から Rational 値への暗黙的な変換を定義します。

```
public static implicit operator WS.Theia.ExtremelyPrecise.Rational(long value);
```

パラメータ

value　Int64

Rational へと変換する値。

戻り値

Rational

value パラメータと等価な Rational 値。

注釈

次の例では、Int64 型を Rational 型に変換する方法示します。暗黙的な変換演算子をサポートしない言語では、代わりに Rational.Rational(Int64)メソッドを使用します。

```
long longValue = 1358754982;
Rational number = longValue;
number = number + (longValue / 2);
Console.WriteLine(number * longValue / longValue); // Displays 2038132473
```

Implicit(UInt64 to Rational)

> ⚠重要
> この API は CLS 準拠ではありません。
> CLS 準拠の代替　System.Double

UInt64 値から Rational 値への暗黙的な変換を定義します。

```
public static implicit operator WS.Theia.ExtremelyPrecise.Rational(ulong value);
```

パラメータ
value　UInt64
Rational へと変換する値。

戻り値
Rational
value パラメータと等価な Rational 値。

注釈
次の例では、UInt64 型を Rational 型に変換する方法示します。暗黙的な変換演算子をサポートしない言語では、代わりに Rational.Rational(UInt64)メソッドを使用します。

```
ulong uLongValue = 1358754982;
Rational number = uLongValue;
number = number * 2 - uLongValue;
Console.WriteLine(number * uLongValue / uLongValue); // Displays 1358754982
```

Implicit(Single to Rational)

Single 値から Rational 値への暗黙的な変換を定義します。

```
public static implicit operator WS.Theia.ExtremelyPrecise.Rational(float value);
```

パラメータ

value　Single

Rational へと変換する値。

戻り値

Rational

value パラメータと等価な Rational 値。

注釈

次の例では、Single 型を Rational 型に変換する方法示します。暗黙的な変換演算子をサポートしない言語では、代わりに Rational.Rational(Single)メソッドを使用します。

```
float floatValue = 135875498.25f;
Rational number = floatValue;
number = number * 2 - floatValue;
Console.WriteLine(number * floatValue / floatValue); // Displays
135875498.25
```

Implicit(Double to Rational)

Double 値から Rational 値への暗黙的な変換を定義します。

```
public static implicit operator WS.Theia.ExtremelyPrecise.Rational(double value);
```

パラメータ

value　Double

Rational へと変換する値。

戻り値

Rational

value パラメータと等価な Rational 値。

注釈

次の例では、Double 型を Rational 型に変換する方法示します。暗黙的な変換演算子をサポートしない言語では、代わりに Rational.Rational(Double)メソッドを使用します。

```
double doubleValue = 135875498.25;
Rational number = doubleValue;
number = number * 2 - doubleValue;
Console.WriteLine(number * doubleValue / doubleValue); // Displays
135875498.25
```

Implicit(Boolean to Rational)

Boolean 値から Rational 値への暗黙的な変換を定義します。

```
public static implicit operator WS.Theia.ExtremelyPrecise.Rational(bool value);
```

パラメータ

value　Boolean

Rational へと変換する値。

戻り値

Rational

value パラメータと等価な Rational 値。

注釈

次の例では、Boolean 型を Rational 型に変換する方法示します。暗黙的な変換演算子をサポートしない言語では、代わりに Rational.Rational(Boolean)メソッドを使用します。

```
bool boolValue = true;
Rational number = boolValue;
number = number * 2 - boolValue;
Console.WriteLine(number * boolValue / boolValue); // Displays 1
```

Implicit(Decimal to Rational)

Decimal 値から Rational 値への暗黙的な変換を定義します。

```
public static implicit operator WS.Theia.ExtremelyPrecise.Rational(decimal value);
```

パラメータ

value　Decimal

Rational へと変換する値。

戻り値

Rational

value パラメータと等価な Rational 値。

注釈

次の例では、Decimal 型を Rational 型に変換する方法示します。暗黙的な変換演算子をサポートしない言語では、代わりに Rational.Rational(Decimal)メソッドを使用します。

```
decimal decimalValue = 135875498.2m;
Rational number = decimalValue;
number = number * 2 - decimalValue;
Console.WriteLine(number * decimalValue / decimalValue); // Displays 135875498.2
```

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Increment (Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値を 1 だけインクリメントします。

```
public static WS.Theia.ExtremelyPrecise.Rational operator
++( WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
インクリメントする値。

戻り値
Rational
value パラメーターの値を 1 だけインクリメントした値

注釈

Implicit 演算子は次のように使用します

```
Rational number = 93843112;
Console.WriteLine(++number);                    // Displays 93843113
```

カスタム演算子をサポートしない言語では Incremen メソッドを代わりに使用します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Inequality(Rational,Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational オブジェクトの値が異なるかどうかを示す値を返します。

```
public static bool operator !=(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメータ
left　Rational
比較する最初の値。
right Rational
比較する 2 番目の値。

戻り値
left と right が等しくない場合は true。それ以外の場合は false。

注釈

Rational 値を比較する演算子。Rational 値を比較して変数に割り当てる時は次の例の様に使用する。

```
Rational number1 = 945834723;
Rational number2 = 345145625;
Rational number3 = 945834723;
Console.WriteLine(number1 != number2);          // Displays True
Console.WriteLine(number1 != number3);          // Displays False
```

カスタム演算子をサポートしない言語では Compare メソッドを代わりに使用します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.LeftShift(Rational,Int32)

Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定されたビット数だけ Rational 値を左にシフトします。

```
public static Rational operator <<( WS.Theia.ExtremelyPrecise.Rational
value,int shift);
```

パラメーター
value　Rational
ビットをシフトする対象の値。
shift　Int32
value を左にシフトするビット数。

戻り値
Rational
指定されたビット数だけ左にシフトされた値。

注釈

演算子のオーバーロードや、カスタム演算子をサポートしない言語用の、Rational 値を左シフトする代替メソッド。Rational 値を左シフトして変数に割り当てる時は次の例の様に使用する。

```
Rational number = Rational.Parse("-904732167844981624999312055");
Console.WriteLine("Shifting {0} left by:", number);
for (int ctr = 0; ctr <= 16; ctr++)
{
    Rational newNumber = Rational.LeftShift(number,ctr);
    Console.WriteLine(" {0,2} bits: {1,35}", ctr, newNumber);
}
// The example displays the following output:
//Shifting -904732167844981624999312055 left by:
//    0 bits:            -904732167844981624999312055
//    1 bits:           -1809464335689963249999624110
//    2 bits:           -3618928671379926499999248220
//    3 bits:           -7237857342759852999999496440
//    4 bits:          -14475714685519705999998892880
//    5 bits:          -28951429371039411999997785760
//    6 bits:          -57902858742078823999995571520
//    7 bits:         -115805717484157647999991143040
//    8 bits:         -231611434968315295999982286080
//    9 bits:         -463222869936630591999964772160
// 10 bits:          -926445739873261183999929544320
// 11 bits:         -1852891479746522367999859088640
// 12 bits:         -3705782959493044735999718177280
// 13 bits:         -7411565918986089471999436354560
// 14 bits:        -14823131837972178943998872709120
// 15 bits:        -29646263675944357887997457418240
// 16 bits:        -59292527351888715775995491483648O
```

⚠注意
プリミティブ型の左シフト演算と異なり、Rational の LeftShift メソッドは符号が変化する事はありません。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.LessThan(Rational,Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値がもう 1 つの Rational 値より小さいかどうかを示す値を返します。

```
public static bool operator <(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメータ
left Rational
比較する最初の値。
right Rational
比較する 2 番目の値。

戻り値
left が right より小さい場合は true。それ以外の場合は false。

注釈

Rational 値を比較する演算子。Rational 値を比較して変数に割り当てる時は次の例の様に使用する。

```
Rational number1 = 945834723;
Rational number2 = 345145625;
Rational number3 = 945834724;
Console.WriteLine(number1 < number2);          // Displays False
Console.WriteLine(number1 < number3);          // Displays True
```

カスタム演算子をサポートしない言語では Compare メソッドを代わりに使用します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.LessThanOrEqual(Rational,Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値がもう 1 つの Rational 値以下かどうかを示す値を返します。

```
public static bool operator <=(WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメータ
left Rational
比較する最初の値。
right Rational
比較する 2 番目の値。

戻り値
left が right 以下の場合は true。それ以外の場合は false。

注釈

Rational 値を比較する演算子。Rational 値を比較して変数に割り当てる時は次の例の様に使用する。

```
Rational number1 = 945834723;
Rational number2 = 345145625;
Rational number3 = 945834724;
Rational number4 = 945834723;
Console.WriteLine(number1 <= number2);        // Displays False
Console.WriteLine(number1 <= number3);        // Displays True
Console.WriteLine(number1 <= number4);        // Displays True
```

カスタム演算子をサポートしない言語では Compare メソッドを代わりに使用します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Modulus(Rational,Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された 2 つの Rational 値の除算の結果生じた剰余を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational
operator %( WS.Theia.ExtremelyPrecise.Rational dividend,
WS.Theia.ExtremelyPrecise.Rational divisor);
```

パラメーター
dividend　Rational
被除数。
divisor　Rational
除数。

戻り値
Rational
除算の結果生じた剰余。

注釈

Rational 値の剰余を求める演算子です。Rational 値の剰余を求めて変数に割り当てる時は次の例の様に使用する。

```
Rational num1 = 100045632194;
Rational num2 = 90329434;
Rational remainder = num1 % num2;
Console.WriteLine(remainder);                  // Displays 50948756
```

カスタム演算子をサポートしない言語では Mod メソッドを代わりに使用します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.Multiply(Rational,Rational)

Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
2 つの Rational 値の積を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational operator
*( WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター
left　Rational
乗算対象の最初の数。
right　Rational
乗算対象の 2 番目の数。

戻り値
Rational
left と right の積。

注釈

Rational 値を乗算する演算子。Rational 値の積を求めて変数に割り当てる時は次の例の様に使用する。

```
Rational num1 = 1000456321;
Rational num2 = 90329434;
Rational result = num1 * num2;
```

カスタム演算子をサポートしない言語では Multiply メソッドを代わりに使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.OnesComplement(Rational)

Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値のビットごとの 1 の補数を返します。

```
public static WS.Theia.ExtremelyPrecise.Rational operator
~( WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
1 の補数を取得したい値。

戻り値
Rational
value のビットごとの 1 の補数。

注釈

Rational 値のビット反転をする演算子。value で 0 であるビットは 1 に、1 であるビットには 0 を設定します。

```
sing System;
using System.Numerics;

public class Example
{
    public static void Main()
```

```
    {
        Rational value, complement;

        value = Rational.Multiply(Rational.One, 9);
        complement = ~value;

        Console.WriteLine("{0,5} -- {1,-32}", value, DisplayInBinary(value));
        Console.WriteLine("{0,5} -- {1,-32}¥n", complement,
DisplayInBinary(complement));

        value = Rational.MinusOne * SByte.MaxValue;
        complement = ~value;

        Console.WriteLine("{0,5} -- {1,-32}", value, DisplayInBinary(value));
        Console.WriteLine("{0,5} -- {1,-32}¥n", complement,
DisplayInBinary(complement));
    }

    private static string DisplayInBinary(Rational number)
    {
        (_,byte[] bytes,_,_) = number.ToByteArray();
        string binaryString = string.Empty;
        foreach (byte byteValue in bytes)
        {
            string byteString = Convert.ToString(byteValue, 2).Trim();
            binaryString += byteString.Insert(0, new string('0', 8 -
byteString.Length));
        }
        return binaryString;
    }
}
// The example displays the following output:
//       9 –
0000100100000000000000000000000000000000000000000000000000000000
//-18446744073709551606 –
1111011011111111111111111111111111111111111111111111111111111111
```

```
//
// -127 –
01111111000000000000000000000000000000000000000000000000000000000
//18446744073709551488 --
10000000111111111111111111111111111111111111111111111111111111111
```

カスタム演算子をサポートしない言語では OnesComplement メソッドを代わりに使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.RightShift(Rational,Int32)

Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定されたビット数だけ Rational 値を右にシフトします。

```
public static WS.Theia.ExtremelyPrecise.Rational operator
>>( WS.Theia.ExtremelyPrecise.Rational value,int shift);
```

パラメーター
value　Rational
ビットをシフトする対象の値。
shift　Int32
value を右にシフトするビット数。

戻り値
Rational
指定されたビット数だけ右にシフトされた値。

注釈

Rational 値を左シフトする演算子。Rational 値を右シフトして変数に割り当てる時は次の例の様に使用する。

```
var number = Rational.Parse("-90473216784498162499999312055");
Console.WriteLine("Shifting {0} right by:", number);
for (int ctr = 0; ctr <= 16; ctr++) {
    Rational newNumber = number >> ctr;
    Console.WriteLine(" {0,2} bits: {1,35}", ctr, newNumber);
}
// The example displays the following output:
//       Shifting -90473216784498162499999312055 right by:
//          0 bits:            -90473216784498162499999312055
//          1 bits:         -45236608392249081249999656027.5
//          2 bits:       -22618304196124540624999828013.75
//          3 bits:     -11309152098062270312499914006.875
//          4 bits:     -5654576049031135156249957003.4375
//          5 bits:    -2827288024515567578124978501.71875
//          6 bits: -1413644012257783789062392500.859375
//          7 bits: -706822006128891894531196250.4296875
//          8 bits: -353411003064445947265598125.71484375
//          9 bits: -176705501532222973632799062.857421875
//         10 bits: -88352750766111486816399531.1787109375
//         11 bits: -44176375383055743408199765.58935546875
//         12 bits: -22088187691527871704099882.294677734375
//         13 bits: -11044093845763935852049941.1473388671875
//         14 bits: -5522046922881967926024970.07366943359375
//         15 bits: -2761023461440983963012485.536834716796875
//         16 bits: -1380511730720491981506242.2684173583984375
```

カスタム演算子をサポートしない言語では RightShift メソッドを代わりに使用します。

> ⚠注意
> プリミティブ型の右シフト演算と異なり、Rational の RightShift 演算子は小数点以下の値を切り捨てません。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.Subtraction(Rational,Rational)

Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational 値をもう 1 つの Rational 値から減算します。

```
public static WS.Theia.ExtremelyPrecise.Rational operator -
( WS.Theia.ExtremelyPrecise.Rational left,
WS.Theia.ExtremelyPrecise.Rational right);
```

パラメーター
left　Rational
減算される値（被減数）。
right　Rational
減算する値（減数）。

戻り値
Rational
left から right を減算した結果。

注釈

Rational 値を減算する演算子。Rational 値を減算して変数に割り当てる時は次の例の様に使用する。

```
Rational num1 = 100045632194;
Rational num2 = 90329434;
Rational result = num1 - num2;
```

カスタム演算子をサポートしない言語では Subtract メソッドを代わりに使用します。

適用対象

.NET Core
2.0
.NET Framework
4.6.1
.NET Standard
2.0
UWP
10.0.16299
Xamarin.Android
8.0
Xamarin.iOS
10.14
Xamarin.Mac
3.8

Rational.UnaryNegation(Rational)

Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
指定された Rational 値の符号を反転します。

```
public static WS.Theia.ExtremelyPrecise.Rational operator -
( WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
符号を反転させる値。

戻り値
Rational
value パラメーターに -1 を乗算した結果。

注釈

Rational 値の符号反転する演算子。Rational 値の符号を反転して変数に割り当てる手順は次の例で示します。

```
Rational number = 12645002;
Console.WriteLine(Rational.Negate(number));        // Displays -12645002
Console.WriteLine(-number);                        // Displays -12645002
Console.WriteLine(number * Rational.MinusOne);     // Displays -12645002
```

カスタム演算子をサポートしない言語では Negate メソッドを代わりに使用します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8

Rational.UnaryPlus(Rational) Operator

名前空間: WS.Theia.ExtremelyPrecise
アセンブリ: ExtremelyPrecise.dll
Rational オペランドの値を返します。 オペランドの符号は変更されません。

```
public static WS.Theia.ExtremelyPrecise.Rational operator
+( WS.Theia.ExtremelyPrecise.Rational value);
```

パラメーター
value　Rational
符号を反転させない値。

戻り値
Rational
value パラメーターと等価な値。

注釈

Rational 値の単項プラス演算子。カスタム演算子をサポートしない言語では Plus メソッドを代わりに使用します。

適用対象

.NET Core

2.0

.NET Framework

4.6.1

.NET Standard

2.0

UWP

10.0.16299

Xamarin.Android

8.0

Xamarin.iOS

10.14

Xamarin.Mac

3.8