```
import CoreFoundation.CFArray
import CoreFoundation.CFAttributedString
import CoreFoundation.CFAvailability
import CoreFoundation.CFBag
import CoreFoundation.CFBase
import CoreFoundation.CFBinaryHeap
import CoreFoundation.CFBitVector
import CoreFoundation.CFBundle
import CoreFoundation.CFByteOrder
import CoreFoundation.CFCGTypes
import CoreFoundation.CFCalendar
import CoreFoundation.CFCharacterSet
import CoreFoundation.CFData
import CoreFoundation.CFDate
import CoreFoundation.CFDateFormatter
import CoreFoundation.CFDictionary
import CoreFoundation.CFError
import CoreFoundation.CFFileDescriptor
import CoreFoundation.CFFileSecurity
import CoreFoundation.CFLocale
import CoreFoundation.CFMachPort
import CoreFoundation.CFMessagePort
import
CoreFoundation.CFNotificationCenter
import CoreFoundation.CFNumber
import CoreFoundation.CFNumberFormatter
import CoreFoundation.CFPlugIn
import CoreFoundation.CFPlugInCOM
import CoreFoundation.CFPreferences
import CoreFoundation.CFPropertyList
import CoreFoundation.CFRunLoop
import CoreFoundation.CFSet
import CoreFoundation.CFSocket
```

```
import CoreFoundation.CFStream
import CoreFoundation.CFString
import CoreFoundation.CFStringEncodingExt
import CoreFoundation.CFStringTokenizer
import CoreFoundation.CFTimeZone
import CoreFoundation.CFTree
import CoreFoundation.CFURL
import CoreFoundation.CFURLAccess
import CoreFoundation.CFURLEnumerator
import CoreFoundation.CFUUID
import CoreFoundation.CFUserNotification
import CoreFoundation.CFUtilities
import CoreFoundation.CFXMLNode
import CoreFoundation.CFXMLParser
import TargetConditionals
import _Builtin_float
import _Builtin_inttypes
import _Builtin_limits
import _Builtin_stdbool
import _Builtin_stdint
import _Concurrency
import _StringProcessing
import _SwiftConcurrencyShims
import _ctype
import _errno
import _math
import _setjmp
import _stdio
import _stdlib
import _string
import _time
import sys_types
```

```
public var __COREFOUNDATION__: Int32 {
get }

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
@frozen public struct CGFloat {

    /// The native type used to store the
CGFloat, which is Float on
    /// 32-bit architectures and Double
on 64-bit architectures.
    public typealias NativeType = Double

    public init()

    /// Creates a new instance from the
given value, rounded to the closest
    /// possible representation.
    ///
    /// - Parameter value: A floating-
point value to be converted.
    public init(_ value: Float)

    /// Creates a new instance from the
given value, rounded to the closest
    /// possible representation.
    ///
    /// - Parameter value: A floating-
point value to be converted.
    public init(_ value: Double)

    public init(_ value: CGFloat)
```

```
    /// The native value.
    public var native: CGFloat.NativeType
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension CGFloat : SignedNumeric {

    /// Creates a new value, rounded to
the closest possible representation.
    ///
    /// If two representable values are
equally close, the result is the value
    /// with more trailing zeros in its
significand bit pattern.
    ///
    /// - Parameter value: The integer to
convert to a floating-point value.
    public init<T>(_ source: T) where T :
BinaryInteger

    /// Creates a new instance from the
given integer, if it can be represented
    /// exactly.
    ///
    /// If the value passed as `source`
is not representable exactly, the result
    /// is `nil`. In the following
example, the constant `x` is successfully
    /// created from a value of `100`,
while the attempt to initialize the
    /// constant `y` from `1_000` fails
because the `Int8` type can represent
```

```
    /// `127` at maximum:
    ///
    ///     let x = Int8(exactly: 100)
    ///     // x == Optional(100)
    ///     let y = Int8(exactly: 1_000)
    ///     // y == nil
    ///
    /// - Parameter source: A value to
convert to this type.
    public init?<T>(exactly source: T)
where T : BinaryInteger

    /// The magnitude of this value.
    ///
    /// For any numeric value `x`,
`x.magnitude` is the absolute value of
`x`.
    /// You can use the `magnitude`
property in operations that are simpler
to
    /// implement in terms of unsigned
values, such as printing the value of an
    /// integer, which is just printing a
'-' character in front of an absolute
    /// value.
    ///
    ///     let x = -200
    ///     // x.magnitude == 200
    ///
    /// The global `abs(_:)` function
provides more familiar syntax when you
need
    /// to find an absolute value. In
```

```
addition, because `abs(_:)` always
returns
    /// a value of the same type, even in
a generic context, using the function
    /// instead of the `magnitude`
property is encouraged.
    public var magnitude: CGFloat { get }

    /// A type that represents an integer
literal.
    ///
    /// The standard library integer and
floating-point types are all valid types
    /// for `IntegerLiteralType`.
    @available(iOS 2.0, tvOS 9.0, watchOS
1.0, macOS 10.0, macCatalyst 13.0, *)
    public typealias IntegerLiteralType =
Int

    /// A type that can represent the
absolute value of any possible value of
the
    /// conforming type.
    @available(iOS 2.0, tvOS 9.0, watchOS
1.0, macOS 10.0, macCatalyst 13.0, *)
    public typealias Magnitude = CGFloat
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension CGFloat : BinaryFloatingPoint {

    /// A type that represents the
```

```
encoded significand of a value.
    public typealias RawSignificand =
UInt

    /// A type that can represent any
written exponent.
    public typealias Exponent = Int

    /// The number of bits used to
represent the type's exponent.
    ///
    /// A binary floating-point type's
`exponentBitCount` imposes a limit on the
    /// range of the exponent for normal,
finite values. The *exponent bias* of
    /// a type `F` can be calculated as
the following, where `**` is
    /// exponentiation:
    ///
    ///     let bias = 2 **
(F.exponentBitCount - 1) - 1
    ///
    /// The least normal exponent for
values of the type `F` is `1 - bias`, and
    /// the largest finite exponent is
`bias`. An all-zeros exponent is reserved
    /// for subnormals and zeros, and an
all-ones exponent is reserved for
    /// infinity and NaN.
    ///
    /// For example, the `Float` type has
an `exponentBitCount` of 8, which gives
    /// an exponent bias of `127` by the
```

```
calculation above.
    ///
    ///     let bias = 2 **
(Float.exponentBitCount - 1) - 1
    ///     // bias == 127
    ///
print(Float.greatestFiniteMagnitude.expon
ent)
    ///     // Prints "127"
    ///
print(Float.leastNormalMagnitude.exponent
)
    ///     // Prints "-126"
    public static var exponentBitCount:
Int { get }

    /// The available number of
fractional significand bits.
    ///
    /// For fixed-width floating-point
types, this is the actual number of
    /// fractional significand bits.
    ///
    /// For extensible floating-point
types, `significandBitCount` should be
the
    /// maximum allowed significand width
(without counting any leading integral
    /// bit of the significand). If there
is no upper limit, then
    /// `significandBitCount` should be
`Int.max`.
    ///
```

```
    /// Note that
`Float80.significandBitCount` is 63, even
though 64 bits are
    /// used to store the significand in
the memory representation of a
    /// `Float80` (unlike other floating-
point types, `Float80` explicitly
    /// stores the leading integral
significand bit, but the
    /// `BinaryFloatingPoint` APIs
provide an abstraction so that users
don't
    /// need to be aware of this detail).
    public static var
significandBitCount: Int { get }

    public var bitPattern: UInt { get }

    public init(bitPattern: UInt)

    /// The sign of the floating-point
value.
    ///
    /// The `sign` property is `.minus`
if the value's signbit is set, and
    /// `.plus` otherwise. For example:
    ///
    ///     let x = -33.375
    ///     // x.sign == .minus
    ///
    /// Don't use this property to check
whether a floating point value is
    /// negative. For a value `x`, the
```

```
comparison `x.sign == .minus` is not
    /// necessarily the same as `x < 0`.
In particular, `x.sign == .minus` if
    /// `x` is -0, and while `x < 0` is
always `false` if `x` is NaN, `x.sign`
    /// could be either `.plus` or
`.minus`.
    public var sign: FloatingPointSign {
get }

    /// The raw encoding of the value's
exponent field.
    ///
    /// This value is unadjusted by the
type's exponent bias.
    public var exponentBitPattern: UInt {
get }

    /// The raw encoding of the value's
significand field.
    ///
    /// The `significandBitPattern`
property does not include the leading
    /// integral bit of the significand,
even for types like `Float80` that
    /// store it explicitly.
    public var significandBitPattern:
UInt { get }

    /// Creates a new instance from the
specified sign and bit patterns.
    ///
    /// The values passed as
```

```
`exponentBitPattern` and
`significandBitPattern` are
    /// interpreted in the binary
interchange format defined by the [IEEE
754
    /// specification][spec].
    ///
    /// [spec]:
http://ieeexplore.ieee.org/servlet/opac?
punumber=4610933
    ///
    /// - Parameters:
    ///   - sign: The sign of the new
value.
    ///   - exponentBitPattern: The bit
pattern to use for the exponent field of
    ///     the new value.
    ///   - significandBitPattern: The
bit pattern to use for the significand
    ///     field of the new value.
    public init(sign: FloatingPointSign,
exponentBitPattern: UInt,
significandBitPattern: UInt)

    public init(nan payload:
CGFloat.RawSignificand, signaling: Bool)

    /// Positive infinity.
    ///
    /// Infinity compares greater than
all finite numbers and equal to other
    /// infinite values.
    ///
```

```
    ///     let x =
Double.greatestFiniteMagnitude
    ///     let y = x * 2
    ///     // y == Double.infinity
    ///     // y > x
    public static var infinity: CGFloat {
get }

    /// A quiet NaN ("not a number").
    ///
    /// A NaN compares not equal, not
greater than, and not less than every
    /// value, including itself. Passing
a NaN to an operation generally results
    /// in NaN.
    ///
    ///     let x = 1.21
    ///     // x > Double.nan == false
    ///     // x < Double.nan == false
    ///     // x == Double.nan == false
    ///
    /// Because a NaN always compares not
equal to itself, to test whether a
    /// floating-point value is NaN, use
its `isNaN` property instead of the
    /// equal-to operator (`==`). In the
following example, `y` is NaN.
    ///
    ///     let y = x + Double.nan
    ///     print(y == Double.nan)
    ///     // Prints "false"
    ///     print(y.isNaN)
    ///     // Prints "true"
```

```
    public static var nan: CGFloat {
get }

    /// A signaling NaN ("not a number").
    ///
    /// The default IEEE 754 behavior of
operations involving a signaling NaN is
    /// to raise the Invalid flag in the
floating-point environment and return a
    /// quiet NaN.
    ///
    /// Operations on types conforming to
the `FloatingPoint` protocol should
    /// support this behavior, but they
might also support other options. For
    /// example, it would be reasonable
to implement alternative operations in
    /// which operating on a signaling
NaN triggers a runtime error or results
    /// in a diagnostic for debugging
purposes. Types that implement
alternative
    /// behaviors for a signaling NaN
must document the departure.
    ///
    /// Other than these signaling
operations, a signaling NaN behaves in
the
    /// same manner as a quiet NaN.
    public static var signalingNaN:
CGFloat { get }

    /// The greatest finite number
```

```
representable by this type.
    ///
    /// This value compares greater than
or equal to all finite numbers, but less
    /// than `infinity`.
    ///
    /// This value corresponds to type-
specific C macros such as `FLT_MAX` and
    /// `DBL_MAX`. The naming of those
macros is slightly misleading, because
    /// `infinity` is greater than this
value.
    public static var
greatestFiniteMagnitude: CGFloat { get }

    /// The mathematical constant pi (π),
approximately equal to 3.14159.
    ///
    /// When measuring an angle in
radians, π is equivalent to a half-turn.
    ///
    /// This value is rounded toward zero
to keep user computations with angles
    /// from inadvertently ending up in
the wrong quadrant. A type that conforms
    /// to the `FloatingPoint` protocol
provides the value for `pi` at its best
    /// possible precision.
    ///
    ///     print(Double.pi)
    ///     // Prints "3.141592653589793"
    public static var pi: CGFloat { get }
```

```
    /// The unit in the last place of
this value.
    ///
    /// This is the unit of the least
significant digit in this value's
    /// significand. For most numbers
`x`, this is the difference between `x`
    /// and the next greater (in
magnitude) representable number. There
are some
    /// edge cases to be aware of:
    ///
    /// - If `x` is not a finite number,
then `x.ulp` is NaN.
    /// - If `x` is very small in
magnitude, then `x.ulp` may be a
subnormal
    ///   number. If a type does not
support subnormals, `x.ulp` may be
rounded
    ///   to zero.
    /// - `greatestFiniteMagnitude.ulp`
is a finite number, even though the next
    ///   greater representable value is
`infinity`.
    ///
    /// See also the `ulpOfOne` static
property.
    public var ulp: CGFloat { get }

    /// The least positive normal number.
    ///
    /// This value compares less than or
```

```
equal to all positive normal numbers.
    /// There may be smaller positive
numbers, but they are *subnormal*,
meaning
    /// that they are represented with
less precision than normal numbers.
    ///
    /// This value corresponds to type-
specific C macros such as `FLT_MIN` and
    /// `DBL_MIN`. The naming of those
macros is slightly misleading, because
    /// subnormals, zeros, and negative
numbers are smaller than this value.
    public static var
leastNormalMagnitude: CGFloat { get }

    /// The least positive number.
    ///
    /// This value compares less than or
equal to all positive numbers, but
    /// greater than zero. If the type
supports subnormal values,
    /// `leastNonzeroMagnitude` is
smaller than `leastNormalMagnitude`;
    /// otherwise they are equal.
    public static var
leastNonzeroMagnitude: CGFloat { get }

    /// The exponent of the floating-
point value.
    ///
    /// The *exponent* of a floating-
point value is the integer part of the
```

```
    /// logarithm of the value's
magnitude. For a value `x` of a floating-
point
    /// type `F`, the magnitude can be
calculated as the following, where `**`
    /// is exponentiation:
    ///
    ///     x.significand * (F.radix **
x.exponent)
    ///
    /// In the next example, `y` has a
value of `21.5`, which is encoded as
    /// `1.34375 * 2 ** 4`. The
significand of `y` is therefore 1.34375.
    ///
    ///     let y: Double = 21.5
    ///     // y.significand == 1.34375
    ///     // y.exponent == 4
    ///     // Double.radix == 2
    ///
    /// The `exponent` property has the
following edge cases:
    ///
    /// - If `x` is zero, then
`x.exponent` is `Int.min`.
    /// - If `x` is +/-infinity or NaN,
then `x.exponent` is `Int.max`
    ///
    /// This property implements the
`logB` operation defined by the [IEEE 754
    /// specification][spec].
    ///
    /// [spec]:
```

```
http://ieeexplore.ieee.org/servlet/opac?
punumber=4610933
    public var exponent: Int { get }

    /// The significand of the floating-
point value.
    ///
    /// The magnitude of a floating-point
value `x` of type `F` can be calculated
    /// by using the following formula,
where `**` is exponentiation:
    ///
    ///     x.significand * (F.radix **
x.exponent)
    ///
    /// In the next example, `y` has a
value of `21.5`, which is encoded as
    /// `1.34375 * 2 ** 4`. The
significand of `y` is therefore 1.34375.
    ///
    ///     let y: Double = 21.5
    ///     // y.significand == 1.34375
    ///     // y.exponent == 4
    ///     // Double.radix == 2
    ///
    /// If a type's radix is 2, then for
finite nonzero numbers, the significand
    /// is in the range `1.0 ..< 2.0`.
For other values of `x`, `x.significand`
    /// is defined as follows:
    ///
    /// - If `x` is zero, then
`x.significand` is 0.0.
```

```
    /// - If `x` is infinite, then
`x.significand` is infinity.
    /// - If `x` is NaN, then
`x.significand` is NaN.
    /// - Note: The significand is
frequently also called the *mantissa*,
but
    ///   significand is the preferred
terminology in the [IEEE 754
    ///   specification][spec], to allay
confusion with the use of mantissa for
    ///   the fractional part of a
logarithm.
    ///
    /// [spec]:
http://ieeexplore.ieee.org/servlet/opac?
punumber=4610933
    public var significand: CGFloat { get
}

    /// Creates a new value from the
given sign, exponent, and significand.
    ///
    /// The following example uses this
initializer to create a new `Double`
    /// instance. `Double` is a binary
floating-point type that has a radix of
    /// `2`.
    ///
    ///     let x = Double(sign: .plus,
exponent: -2, significand: 1.5)
    ///     // x == 0.375
    ///
```

```
    /// This initializer is equivalent to
the following calculation, where `**`
    /// is exponentiation, computed as if
by a single, correctly rounded,
    /// floating-point operation:
    ///
    ///     let sign: FloatingPointSign =
.plus
    ///     let exponent = -2
    ///     let significand = 1.5
    ///     let y = (sign == .minus ?
-1 : 1) * significand * Double.radix **
exponent
    ///     // y == 0.375
    ///
    /// As with any basic operation, if
this value is outside the representable
    /// range of the type, overflow or
underflow occurs, and zero, a subnormal
    /// value, or infinity may result. In
addition, there are two other edge
    /// cases:
    ///
    /// - If the value you pass to
`significand` is zero or infinite, the
result
    ///   is zero or infinite, regardless
of the value of `exponent`.
    /// - If the value you pass to
`significand` is NaN, the result is NaN.
    ///
    /// For any floating-point value `x`
of type `F`, the result of the following
```

```
    /// is equal to `x`, with the
distinction that the result is
canonicalized
    /// if `x` is in a noncanonical
encoding:
    ///
    ///     let x0 = F(sign: x.sign,
exponent: x.exponent, significand:
x.significand)
    ///
    /// This initializer implements the
`scaleB` operation defined by the [IEEE
    /// 754 specification][spec].
    ///
    /// [spec]:
http://ieeexplore.ieee.org/servlet/opac?
punumber=4610933
    ///
    /// - Parameters:
    ///   - sign: The sign to use for the
new value.
    ///   - exponent: The new value's
exponent.
    ///   - significand: The new value's
significand.
    public init(sign: FloatingPointSign,
exponent: Int, significand: CGFloat)

    /// Rounds the value to an integral
value using the specified rounding rule.
    ///
    /// The following example rounds a
value using four different rounding
```

```
rules:
    ///
    ///     // Equivalent to the C
'round' function:
    ///     var w = 6.5
    ///
w.round(.toNearestOrAwayFromZero)
    ///     // w == 7.0
    ///
    ///     // Equivalent to the C
'trunc' function:
    ///     var x = 6.5
    ///     x.round(.towardZero)
    ///     // x == 6.0
    ///
    ///     // Equivalent to the C 'ceil'
function:
    ///     var y = 6.5
    ///     y.round(.up)
    ///     // y == 7.0
    ///
    ///     // Equivalent to the C
'floor' function:
    ///     var z = 6.5
    ///     z.round(.down)
    ///     // z == 6.0
    ///
    /// For more information about the
available rounding rules, see the
    /// `FloatingPointRoundingRule`
enumeration. To round a value using the
    /// default "schoolbook rounding",
you can use the shorter `round()` method
```

```
    /// instead.
    ///
    ///     var w1 = 6.5
    ///     w1.round()
    ///     // w1 == 7.0
    ///
    /// - Parameter rule: The rounding
rule to use.
    public mutating func round(_ rule:
FloatingPointRoundingRule)

    /// The least representable value
that compares greater than this value.
    ///
    /// For any finite value `x`,
`x.nextUp` is greater than `x`. For `nan`
or
    /// `infinity`, `x.nextUp` is `x`
itself. The following special cases also
    /// apply:
    ///
    /// - If `x` is `-infinity`, then
`x.nextUp` is `-greatestFiniteMagnitude`.
    /// - If `x` is `-
leastNonzeroMagnitude`, then `x.nextUp`
is `-0.0`.
    /// - If `x` is zero, then `x.nextUp`
is `leastNonzeroMagnitude`.
    /// - If `x` is
`greatestFiniteMagnitude`, then
`x.nextUp` is `infinity`.
    public var nextUp: CGFloat { get }
```

```
    /// Replaces this value with its
additive inverse.
    ///
    /// The following example uses the
`negate()` method to negate the value of
    /// an integer `x`:
    ///
    ///     var x = 21
    ///     x.negate()
    ///     // x == -21
    ///
    /// The resulting value must be
representable within the value's type. In
    /// particular, negating a signed,
fixed-width integer type's minimum
    /// results in a value that cannot be
represented.
    ///
    ///     var y = Int8.min
    ///     y.negate()
    ///     // Overflow error
    public mutating func negate()

    /// Adds two values and stores the
result in the left-hand-side variable.
    ///
    /// - Parameters:
    ///   - lhs: The first value to add.
    ///   - rhs: The second value to add.
    public static func += (lhs: inout
CGFloat, rhs: CGFloat)

    /// Subtracts the second value from
```

```
the first and stores the difference in
the
    /// left-hand-side variable.
    ///
    /// - Parameters:
    ///   - lhs: A numeric value.
    ///   - rhs: The value to subtract
from `lhs`.
    public static func -= (lhs: inout
CGFloat, rhs: CGFloat)

    /// Multiplies two values and stores
the result in the left-hand-side
    /// variable.
    ///
    /// - Parameters:
    ///   - lhs: The first value to
multiply.
    ///   - rhs: The second value to
multiply.
    public static func *= (lhs: inout
CGFloat, rhs: CGFloat)

    /// Divides the first value by the
second and stores the quotient in the
    /// left-hand-side variable, rounding
to a representable value.
    ///
    /// - Parameters:
    ///   - lhs: The value to divide.
    ///   - rhs: The value to divide
`lhs` by.
    public static func /= (lhs: inout
```

```
CGFloat, rhs: CGFloat)

    /// Replaces this value with the
remainder of itself divided by the given
    /// value using truncating division.
    ///
    /// Performing truncating division
with floating-point values results in a
    /// truncated integer quotient and a
remainder. For values `x` and `y` and
    /// their truncated integer quotient
`q`, the remainder `r` satisfies
    /// `x == y * q + r`.
    ///
    /// The following example calculates
the truncating remainder of dividing
    /// 8.625 by 0.75:
    ///
    ///     var x = 8.625
    ///     print(x / 0.75)
    ///     // Prints "11.5"
    ///
    ///     let q = (x /
0.75).rounded(.towardZero)
    ///     // q == 11.0
    ///
x.formTruncatingRemainder(dividingBy:
0.75)
    ///     // x == 0.375
    ///
    ///     let x1 = 0.75 * q + x
    ///     // x1 == 8.625
    ///
```

```
    /// If this value and `other` are
both finite numbers, the truncating
    /// remainder has the same sign as
this value and is strictly smaller in
    /// magnitude than `other`. The
`formTruncatingRemainder(dividingBy:)`
    /// method is always exact.
    ///
    /// - Parameter other: The value to
use when dividing this value.
    public mutating func
formTruncatingRemainder(dividingBy other:
CGFloat)

    /// Replaces this value with the
remainder of itself divided by the given
    /// value.
    ///
    /// For two finite values `x` and
`y`, the remainder `r` of dividing `x` by
    /// `y` satisfies `x == y * q + r`,
where `q` is the integer nearest to
    /// `x / y`. If `x / y` is exactly
halfway between two integers, `q` is
    /// chosen to be even. Note that `q`
is *not* `x / y` computed in
    /// floating-point arithmetic, and
that `q` may not be representable in any
    /// available integer type.
    ///
    /// The following example calculates
the remainder of dividing 8.625 by 0.75:
    ///
```

```
    ///     var x = 8.625
    ///     print(x / 0.75)
    ///     // Prints "11.5"
    ///
    ///     let q = (x /
0.75).rounded(.toNearestOrEven)
    ///     // q == 12.0
    ///     x.formRemainder(dividingBy:
0.75)
    ///     // x == -0.375
    ///
    ///     let x1 = 0.75 * q + x
    ///     // x1 == 8.625
    ///
    /// If this value and `other` are
finite numbers, the remainder is in the
    /// closed range `-abs(other /
2)...abs(other / 2)`. The
    /// `formRemainder(dividingBy:)`
method is always exact.
    ///
    /// - Parameter other: The value to
use when dividing this value.
    public mutating func
formRemainder(dividingBy other: CGFloat)

    /// Replaces this value with its
square root, rounded to a representable
    /// value.
    public mutating func formSquareRoot()

    /// Adds the product of the two given
values to this value in place, computed
```

```
    /// without intermediate rounding.
    ///
    /// - Parameters:
    ///   - lhs: One of the values to
multiply before adding to this value.
    ///   - rhs: The other value to
multiply.
    public mutating func addProduct(_
lhs: CGFloat, _ rhs: CGFloat)

    /// Returns a Boolean value
indicating whether this instance is equal
to the
    /// given value.
    ///
    /// This method serves as the basis
for the equal-to operator (`==`) for
    /// floating-point values. When
comparing two values with this method,
`-0`
    /// is equal to `+0`. NaN is not
equal to any value, including itself. For
    /// example:
    ///
    ///     let x = 15.0
    ///     x.isEqual(to: 15.0)
    ///     // true
    ///     x.isEqual(to: .nan)
    ///     // false
    ///     Double.nan.isEqual(to: .nan)
    ///     // false
    ///
    /// The `isEqual(to:)` method
```

```
implements the equality predicate defined
by
    /// the [IEEE 754 specification]
[spec].
    ///
    /// [spec]:
http://ieeexplore.ieee.org/servlet/opac?
punumber=4610933
    ///
    /// - Parameter other: The value to
compare with this value.
    /// - Returns: `true` if `other` has
the same value as this instance;
    ///   otherwise, `false`. If either
this value or `other` is NaN, the result
    ///   of this method is `false`.
    public func isEqual(to other:
CGFloat) -> Bool

    /// Returns a Boolean value
indicating whether this instance is less
than the
    /// given value.
    ///
    /// This method serves as the basis
for the less-than operator (`<`) for
    /// floating-point values. Some
special cases apply:
    ///
    /// - Because NaN compares not less
than nor greater than any value, this
    ///   method returns `false` when
called on NaN or when NaN is passed as
```

```
    ///   `other`.
    /// - `-infinity` compares less than
all values except for itself and NaN.
    /// - Every value except for NaN and
`+infinity` compares less than
    ///   `+infinity`.
    ///
    ///     let x = 15.0
    ///     x.isLess(than: 20.0)
    ///     // true
    ///     x.isLess(than: .nan)
    ///     // false
    ///     Double.nan.isLess(than: x)
    ///     // false
    ///
    /// The `isLess(than:)` method
implements the less-than predicate
defined by
    /// the [IEEE 754 specification]
[spec].
    ///
    /// [spec]:
http://ieeexplore.ieee.org/servlet/opac?
punumber=4610933
    ///
    /// - Parameter other: The value to
compare with this value.
    /// - Returns: `true` if this value
is less than `other`; otherwise, `false`.
    ///   If either this value or `other`
is NaN, the result of this method is
    ///   `false`.
    public func isLess(than other:
```

```
CGFloat) -> Bool

    /// Returns a Boolean value
indicating whether this instance is less
than or
    /// equal to the given value.
    ///
    /// This method serves as the basis
for the less-than-or-equal-to operator
    /// (`<=`) for floating-point values.
Some special cases apply:
    ///
    /// - Because NaN is incomparable
with any value, this method returns
`false`
    ///   when called on NaN or when NaN
is passed as `other`.
    /// - `-infinity` compares less than
or equal to all values except NaN.
    /// - Every value except NaN compares
less than or equal to `+infinity`.
    ///
    ///     let x = 15.0
    ///     x.isLessThanOrEqualTo(20.0)
    ///     // true
    ///     x.isLessThanOrEqualTo(.nan)
    ///     // false
    ///
Double.nan.isLessThanOrEqualTo(x)
    ///     // false
    ///
    /// The `isLessThanOrEqualTo(_:)`
method implements the less-than-or-equal
```

```
    /// predicate defined by the [IEEE
754 specification][spec].
    ///
    /// [spec]:
http://ieeexplore.ieee.org/servlet/opac?
punumber=4610933
    ///
    /// - Parameter other: The value to
compare with this value.
    /// - Returns: `true` if `other` is
greater than this value; otherwise,
    ///   `false`. If either this value
or `other` is NaN, the result of this
    ///   method is `false`.
    public func isLessThanOrEqualTo(_
other: CGFloat) -> Bool

    /// A Boolean value indicating
whether this instance is normal.
    ///
    /// A *normal* value is a finite
number that uses the full precision
    /// available to values of a type.
Zero is neither a normal nor a subnormal
    /// number.
    public var isNormal: Bool { get }

    /// A Boolean value indicating
whether this instance is finite.
    ///
    /// All values other than NaN and
infinity are considered finite, whether
    /// normal or subnormal.  For NaN,
```

```
both `isFinite` and `isInfinite` are
false.
    public var isFinite: Bool { get }

    /// A Boolean value indicating
whether the instance is equal to zero.
    ///
    /// The `isZero` property of a value
`x` is `true` when `x` represents either
    /// `-0.0` or `+0.0`. `x.isZero` is
equivalent to the following comparison:
    /// `x == 0.0`.
    ///
    ///     let x = -0.0
    ///     x.isZero        // true
    ///     x == 0.0        // true
    public var isZero: Bool { get }

    /// A Boolean value indicating
whether the instance is subnormal.
    ///
    /// A *subnormal* value is a nonzero
number that has a lesser magnitude than
    /// the smallest normal number.
Subnormal values don't use the full
    /// precision available to values of
a type.
    ///
    /// Zero is neither a normal nor a
subnormal number. Subnormal numbers are
    /// often called *denormal* or
*denormalized*---these are different
names
```

```
    /// for the same concept.
    public var isSubnormal: Bool { get }

    /// A Boolean value indicating
whether the instance is infinite.
    ///
    /// For NaN, both `isFinite` and
`isInfinite` are false.
    public var isInfinite: Bool { get }

    /// A Boolean value indicating
whether the instance is NaN ("not a
number").
    ///
    /// Because NaN is not equal to any
value, including NaN, use this property
    /// instead of the equal-to operator
(`==`) or not-equal-to operator (`!=`)
    /// to test whether a value is or is
not NaN. For example:
    ///
    ///     let x = 0.0
    ///     let y = x * .infinity
    ///     // y is a NaN
    ///
    ///     // Comparing with the equal-
to operator never returns 'true'
    ///     print(x == Double.nan)
    ///     // Prints "false"
    ///     print(y == Double.nan)
    ///     // Prints "false"
    ///
    ///     // Test with the 'isNaN'
```

```
property instead
    ///     print(x.isNaN)
    ///     // Prints "false"
    ///     print(y.isNaN)
    ///     // Prints "true"
    ///
    /// This property is `true` for both
quiet and signaling NaNs.
    public var isNaN: Bool { get }

    /// A Boolean value indicating
whether the instance is a signaling NaN.
    ///
    /// Signaling NaNs typically raise
the Invalid flag when used in general
    /// computing operations.
    public var isSignalingNaN: Bool { get
}

    /// A Boolean value indicating
whether the instance's representation is
in
    /// its canonical form.
    ///
    /// The [IEEE 754 specification]
[spec] defines a *canonical*, or
preferred,
    /// encoding of a floating-point
value. On platforms that fully support
    /// IEEE 754, every `Float` or
`Double` value is canonical, but
    /// non-canonical values can exist on
other platforms or for other types.
```

```
    /// Some examples:
    ///
    /// - On platforms that flush
subnormal numbers to zero (such as armv7
    ///   with the default floating-point
environment), Swift interprets
    ///   subnormal `Float` and `Double`
values as non-canonical zeros.
    ///   (In Swift 5.1 and earlier,
`isCanonical` is `true` for these
    ///   values, which is the incorrect
value.)
    ///
    /// - On i386 and x86_64, `Float80`
has a number of non-canonical
    ///   encodings. "Pseudo-NaNs",
"pseudo-infinities", and "unnormals" are
    ///   interpreted as non-canonical
NaN encodings. "Pseudo-denormals" are
    ///   interpreted as non-canonical
encodings of subnormal values.
    ///
    /// - Decimal floating-point types
admit a large number of non-canonical
    ///   encodings. Consult the IEEE 754
standard for additional details.
    ///
    /// [spec]:
http://ieeexplore.ieee.org/servlet/opac?
punumber=4610933
    public var isCanonical: Bool { get }

    /// The classification of this value.
```

```
    ///
    /// A value's `floatingPointClass` property describes its "class" as
    /// described by the [IEEE 754 specification][spec].
    ///
    /// [spec]: http://ieeexplore.ieee.org/servlet/opac?punumber=4610933
    public var floatingPointClass: FloatingPointClassification { get }

    /// The floating-point value with the same sign and exponent as this value,
    /// but with a significand of 1.0.
    ///
    /// A *binade* is a set of binary floating-point values that all have the
    /// same sign and exponent. The `binade` property is a member of the same
    /// binade as this value, but with a unit significand.
    ///
    /// In this example, `x` has a value of `21.5`, which is stored as
    /// `1.34375 * 2**4`, where `**` is exponentiation. Therefore, `x.binade` is
    /// equal to `1.0 * 2**4`, or `16.0`.
    ///
    ///     let x = 21.5
    ///     // x.significand == 1.34375
    ///     // x.exponent == 4
    ///
```

```
    ///     let y = x.binade
    ///     // y == 16.0
    ///     // y.significand == 1.0
    ///     // y.exponent == 4
    public var binade: CGFloat { get }

    /// The number of bits required to
represent the value's significand.
    ///
    /// If this value is a finite nonzero
number, `significandWidth` is the
    /// number of fractional bits
required to represent the value of
    /// `significand`; otherwise,
`significandWidth` is -1. The value of
    /// `significandWidth` is always -1
or between zero and
    /// `significandBitCount`. For
example:
    ///
    /// - For any representable power of
two, `significandWidth` is zero, because
    ///   `significand` is `1.0`.
    /// - If `x` is 10, `x.significand`
is `1.01` in binary, so
    ///   `x.significandWidth` is 2.
    /// - If `x` is Float.pi,
`x.significand` is
`1.10010010000111111011011` in
    ///   binary, and
`x.significandWidth` is 23.
    public var significandWidth: Int {
get }
```

```
    /// Create an instance initialized to
`value`.
    public init(floatLiteral value:
CGFloat.NativeType)

    /// Create an instance initialized to
`value`.
    public init(integerLiteral value:
Int)

    /// A type that represents a
floating-point literal.
    ///
    /// Valid types for
`FloatLiteralType` are `Float`, `Double`,
and `Float80`
    /// where available.
    @available(iOS 2.0, tvOS 9.0, watchOS
1.0, macOS 10.0, macCatalyst 13.0, *)
    public typealias FloatLiteralType =
CGFloat.NativeType

    /// A type that represents the
encoded exponent of a value.
    @available(iOS 2.0, tvOS 9.0, watchOS
1.0, macOS 10.0, macCatalyst 13.0, *)
    public typealias RawExponent = UInt
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension CGFloat : CustomReflectable {
```

```
    /// Returns a mirror that reflects
`self`.
    public var customMirror: Mirror { get
}
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension CGFloat :
CustomStringConvertible {

    /// A textual representation of
`self`.
    public var description: String {
get }
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension CGFloat : Hashable {

    /// The hash value.
    ///
    /// **Axiom:** `x == y` implies
`x.hashValue == y.hashValue`
    ///
    /// - Note: the hash value is not
guaranteed to be stable across
    ///   different invocations of the
same program.  Do not persist the
    ///   hash value across program runs.
    public var hashValue: Int { get }
```

```
    /// Hashes the essential components
of this value by feeding them into the
    /// given hasher.
    ///
    /// - Parameter hasher: The hasher to
use when combining the components
    ///   of this instance.
    @inlinable public func hash(into
hasher: inout Hasher)
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension CGFloat {

    /// Adds two values and produces
their sum.
    ///
    /// The addition operator (`+`)
calculates the sum of its two arguments.
For
    /// example:
    ///
    ///     1 + 2                   // 3
    ///     -10 + 15                // 5
    ///     -15 + -5                //
-20
    ///     21.5 + 3.25             //
24.75
    ///
    /// You cannot use `+` with arguments
of different types. To add values of
```

```
    /// different types, convert one of
the values to the other value's type.
    ///
    ///     let x: Int8 = 21
    ///     let y: Int = 1000000
    ///     Int(x) + y              //
1000021
    ///
    /// - Parameters:
    ///   - lhs: The first value to add.
    ///   - rhs: The second value to add.
    public static func + (lhs: CGFloat,
rhs: CGFloat) -> CGFloat

    /// Subtracts one value from another
and produces their difference.
    ///
    /// The subtraction operator (`-`)
calculates the difference of its two
    /// arguments. For example:
    ///
    ///     8 - 3                   // 5
    ///     -10 - 5                 //
-15
    ///     100 - -5                //
105
    ///     10.5 - 100.0            //
-89.5
    ///
    /// You cannot use `-` with arguments
of different types. To subtract values
    /// of different types, convert one
of the values to the other value's type.
```

```
    ///
    ///     let x: UInt8 = 21
    ///     let y: UInt = 1000000
    ///     y - UInt(x)             //
999979
    ///
    /// - Parameters:
    ///   - lhs: A numeric value.
    ///   - rhs: The value to subtract
from `lhs`.
    public static func - (lhs: CGFloat,
rhs: CGFloat) -> CGFloat

    /// Multiplies two values and
produces their product.
    ///
    /// The multiplication operator (`*`)
calculates the product of its two
    /// arguments. For example:
    ///
    ///     2 * 3                   // 6
    ///     100 * 21                //
2100
    ///     -10 * 15                //
-150
    ///     3.5 * 2.25              //
7.875
    ///
    /// You cannot use `*` with arguments
of different types. To multiply values
    /// of different types, convert one
of the values to the other value's type.
    ///
```

```
    ///     let x: Int8 = 21
    ///     let y: Int = 1000000
    ///     Int(x) * y             //
21000000
    ///
    /// - Parameters:
    ///   - lhs: The first value to
multiply.
    ///   - rhs: The second value to
multiply.
    public static func * (lhs: CGFloat,
rhs: CGFloat) -> CGFloat

    /// Returns the quotient of dividing
the first value by the second, rounded
    /// to a representable value.
    ///
    /// The division operator (`/`)
calculates the quotient of the division
if
    /// `rhs` is nonzero. If `rhs` is
zero, the result of the division is
    /// infinity, with the sign of the
result matching the sign of `lhs`.
    ///
    ///     let x = 16.875
    ///     let y = x / 2.25
    ///     // y == 7.5
    ///
    ///     let z = x / 0
    ///     // z.isInfinite == true
    ///
    /// The `/` operator implements the
```

```
division operation defined by the [IEEE
    /// 754 specification][spec].
    ///
    /// [spec]:
http://ieeexplore.ieee.org/servlet/opac?
punumber=4610933
    ///
    /// - Parameters:
    ///   - lhs: The value to divide.
    ///   - rhs: The value to divide
`lhs` by.
    public static func / (lhs: CGFloat,
rhs: CGFloat) -> CGFloat
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension CGFloat : Strideable {

    /// Returns a stride `x` such that
`self.advanced(by: x)` approximates
    /// `other`.
    ///
    /// - Complexity: O(1).
    public func distance(to other:
CGFloat) -> CGFloat

    /// Returns a `Self` `x` such that
`self.distance(to: x)` approximates
    /// `n`.
    ///
    /// - Complexity: O(1).
    public func advanced(by amount:
```

```
CGFloat) -> CGFloat

    /// A type that represents the
distance between two values.
    @available(iOS 2.0, tvOS 9.0, watchOS
1.0, macOS 10.0, macCatalyst 13.0, *)
    public typealias Stride = CGFloat
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension CGFloat : Codable {

    /// Creates a new instance by
decoding from the given decoder.
    ///
    /// This initializer throws an error
if reading from the decoder fails, or
    /// if the data read is corrupted or
otherwise invalid.
    ///
    /// - Parameter decoder: The decoder
to read data from.
    public init(from decoder: any
Decoder) throws

    /// Encodes this value into the given
encoder.
    ///
    /// If the value fails to encode
anything, `encoder` will encode an empty
    /// keyed container in its place.
    ///
```

```
    /// This function throws an error if
any values are invalid for the given
    /// encoder's format.
    ///
    /// - Parameter encoder: The encoder
to write data to.
    public func encode(to encoder: any
Encoder) throws
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension CGFloat : Sendable {
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension CGFloat : BitwiseCopyable {
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension UInt8 {

    public init(_ value: CGFloat)
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension Int8 {

    public init(_ value: CGFloat)
}
```

```
@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension UInt16 {

    public init(_ value: CGFloat)
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension Int16 {

    public init(_ value: CGFloat)
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension UInt32 {

    public init(_ value: CGFloat)
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension Int32 {

    public init(_ value: CGFloat)
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension UInt64 {
```

```
    public init(_ value: CGFloat)
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension Int64 {

    public init(_ value: CGFloat)
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension UInt {

    public init(_ value: CGFloat)
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension Int {

    public init(_ value: CGFloat)
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
extension Double {

    public init(_ value: CGFloat)
}

@available(macOS 10.0, macCatalyst 13.0,
iOS 2.0, watchOS 1.0, tvOS 9.0, *)
```

```
extension Float {

    public init(_ value: CGFloat)
}
```