

[Swift](#) / [BinaryInteger](#)

Protocol

BinaryInteger

An integer type with a binary representation.

iOS 8.0+ | iPadOS 8.0+ | Mac Catalyst 13.0+ | macOS 10.10+ | tvOS 9.0+ | visionOS 1.0+ | watchOS 2.0+

```
protocol BinaryInteger : CustomStringConvertible, Hashable, Numeric, Strideable where Self.Magnitude : BinaryInteger, Self.Magnitude == Self.Magnitude.Magnitude
```

Overview

The `BinaryInteger` protocol is the basis for all the integer types provided by the standard library. All of the standard library's integer types, such as `Int` and `UInt32`, conform to `BinaryInteger`.

Converting Between Numeric Types

You can create new instances of a type that conforms to the `BinaryInteger` protocol from a floating-point number or another binary integer of any type. The `BinaryInteger` protocol provides initializers for four different kinds of conversion.

Range-Checked Conversion

You use the default `init(_ :)` initializer to create a new instance when you're sure that the value passed is representable in the new type. For example, an instance of `Int16` can represent the value `500`, so the first conversion in the code sample below succeeds. That same value is too large to represent as an `Int8` instance, so the second conversion fails, triggering a runtime error.

```
let x: Int = 500
let y = Int16(x)
// y == 500

let z = Int8(x)
// Error: Not enough bits to represent...
```

When you create a binary integer from a floating-point value using the default initializer, the value is rounded toward zero before the range is checked. In the following example, the value 127.75 is rounded to 127, which is representable by the `Int8` type. 128.25 is rounded to 128, which is not representable as an `Int8` instance, triggering a runtime error.

```
let e = Int8(127.75)
// e == 127

let f = Int8(128.25)
// Error: Double value cannot be converted...
```

Exact Conversion

Use the `init?(exactly:)` initializer to create a new instance after checking whether the passed value is representable. Instead of trapping on out-of-range values, using the failable `init?(exactly:)` initializer results in `nil`.

```
let x = Int16(exactly: 500)
// x == Optional(500)

let y = Int8(exactly: 500)
// y == nil
```

When converting floating-point values, the `init?(exactly:)` initializer checks both that the passed value has no fractional part and that the value is representable in the resulting type.

```
let e = Int8(exactly: 23.0)      // integral value, representable
// e == Optional(23)

let f = Int8(exactly: 23.75)    // fractional value, representable
// f == nil
```

```
let g = Int8(exactly: 500.0) // integral value, nonrepresentable
// g == nil
```

Clamping Conversion

Use the `init(clamping:)` initializer to create a new instance of a binary integer type where out-of-range values are clamped to the representable range of the type. For a type `T`, the resulting value is in the range `T.min...T.max`.

```
let x = Int16(clamping: 500)
// x == 500

let y = Int8(clamping: 500)
// y == 127

let z = UInt8(clamping: -500)
// z == 0
```

Bit Pattern Conversion

Use the `init(truncatingIfNeeded:)` initializer to create a new instance with the same bit pattern as the passed value, extending or truncating the value's representation as necessary. Note that the value may not be preserved, particularly when converting between signed to unsigned integer types or when the destination type has a smaller bit width than the source type. The following example shows how extending and truncating work for nonnegative integers:

```
let q: Int16 = 850
// q == 0b00000011_01010010

let r = Int8(truncatingIfNeeded: q) // truncate 'q' to fit in 8 bits
// r == 82
//    == 0b01010010

let s = Int16(truncatingIfNeeded: r) // extend 'r' to fill 16 bits
// s == 82
//    == 0b00000000_01010010
```

Any padding is performed by *sign-extending* the passed value. When nonnegative integers are extended, the result is padded with zeroes. When negative integers are extended, the result is

padded with ones. This example shows several extending conversions of a negative value—note that negative values are sign-extended even when converting to an unsigned type.

```
let t: Int8 = -100
// t == -100
// t's binary representation == 0b10011100

let u = UInt8(truncatingIfNeeded: t)
// u == 156
// u's binary representation == 0b10011100

let v = Int16(truncatingIfNeeded: t)
// v == -100
// v's binary representation == 0b11111111_10011100

let w = UInt16(truncatingIfNeeded: t)
// w == 65436
// w's binary representation == 0b11111111_10011100
```

Comparing Across Integer Types

You can use relational operators, such as the less-than and equal-to operators (< and ==), to compare instances of different binary integer types. The following example compares instances of the Int, UInt, and UInt8 types:

```
let x: Int = -23
let y: UInt = 1_000
let z: UInt8 = 23

if x < y {
    print("\(x) is less than \((y).")
}
// Prints "-23 is less than 1000."

if z > x {
    print("\(z) is greater than \((x).")
}
// Prints "23 is greater than -23."
```

Topics

Creating a Binary Integer

`init()`

Creates a new value equal to zero.

Converting Integers

`init<T>(T)`

Creates a new instance from the given integer.

Required Default implementations provided.

`init<T>(clamping: T)`

Creates a new instance with the representable value that's closest to the given integer.

Required Default implementation provided.

`init<T>(truncatingIfNeeded: T)`

Creates a new instance from the bit pattern of the given instance by sign-extending or truncating to fit this type.

Required Default implementation provided.

Converting Floating-Point Values

`init<T>(T)`

Creates an integer from the given floating-point value, rounding toward zero.

Required Default implementations provided.

Converting with No Loss of Precision

`init?<T>(exactly: T)`

Creates an integer from the given floating-point value, if it can be represented exactly.

Required Default implementations provided.

Performing Calculations

 Binary Integer Operators

Perform arithmetic and bitwise operations or compare values.

```
func quotientAndRemainder(dividingBy: Self) -> (quotient: Self, remainder: Self)
```

Returns the quotient and remainder of this value divided by the given value.

Required Default implementation provided.

```
func isMultiple(of: Self) -> Bool
```

Returns true if this value is a multiple of the given value, and false otherwise.

Required Default implementations provided.

Finding the Sign and Magnitude

```
func signum() -> Self
```

Returns -1 if this value is negative and 1 if it's positive; otherwise, 0 .

Required Default implementation provided.

Accessing Numeric Constants

```
static var isSigned: Bool
```

A Boolean value indicating whether this type is a signed integer type.

Required Default implementations provided.

Working with Binary Representation

```
var bitWidth: Int
```

The number of bits in the current binary representation of this value.

Required Default implementation provided.

```
var trailingZeroBitCount: Int
```

The number of trailing zeros in this value's binary representation.

Required

```
var words: Self.Words
```

A collection containing the words of this value's binary representation, in order from the least significant to most significant.

Required

```
associatedtype Words : RandomAccessCollection
```

A type that represents the words of a binary integer.

Required

Operators

```
static func % (Self, Self) -> Self
```

Returns the remainder of dividing the first value by the second.

Required

```
static func ^ (Self, Self) -> Self
```

Returns the result of performing a bitwise XOR operation on the two given values.

Required Default implementation provided.

```
static func | (Self, Self) -> Self
```

Returns the result of performing a bitwise OR operation on the two given values.

Required Default implementation provided.

```
static func ^= (inout Self, Self)
```

Stores the result of performing a bitwise XOR operation on the two given values in the left-hand-side variable.

Required

```
static func |= (inout Self, Self)
```

Stores the result of performing a bitwise OR operation on the two given values in the left-hand-side variable.

Required

```
static func %= (inout Self, Self)
```

Divides the first value by the second and stores the remainder in the left-hand-side variable.

Required

```
static func << <RHS>(Self, RHS) -> Self
```

Returns the result of shifting a value's binary representation the specified number of digits to the left.

Required Default implementations provided.

```
static func >> <RHS>(Self, RHS) -> Self
```

Returns the result of shifting a value's binary representation the specified number of digits to the right.

Required Default implementations provided.

```
static func >>= <RHS>(inout Self, RHS)
```

Stores the result of shifting a value's binary representation the specified number of digits to the right in the left-hand-side variable.

Required Default implementation provided.

```
static func <=< <RHS>(inout Self, RHS)
```

Stores the result of shifting a value's binary representation the specified number of digits to the left in the left-hand-side variable.

Required Default implementation provided.

Initializers

```
init(String, format: IntegerFormatStyle<Self>, lenient: Bool) throws
```

```
init(String, format: IntegerFormatStyle<Self>.Currency, lenient: Bool) throws
```

```
init(String, format: IntegerFormatStyle<Self>.Percent, lenient: Bool) throws
```

```
init<S>(S.ParseInput, strategy: S) throws
```

Initialize an instance by parsing value with the given strategy.

Instance Methods

```
func formatted() -> String
```

Format self using IntegerFormatStyle()

```
func formatted<S>(S) -> S.FormatOutput
```

Format self with the given format.

```
func formatted<S>(S) -> S.FormatOutput
```

Format self with the given format. self is first converted to S.FormatInput type, then format with the given format.

Default Implementations

☰ BinaryInteger Implementations

☰ Equatable Implementations

Relationships

Inherits From

AdditiveArithmetic
Comparable
CustomStringConvertible
Equatable
ExpressibleByIntegerLiteral
Hashable
Numeric
Strideable

Inherited By

FixedWidthInteger, SignedInteger, UnsignedInteger

Conforming Types

Int
Int128
Int16
Int32
Int64
Int8
UInt
UInt128
UInt16
UInt32
UInt64
UInt8

See Also

Integer

`protocol` FixedWidthInteger

An integer type that uses a fixed size for every instance.

`protocol` SignedInteger

An integer type that can represent both positive and negative values.

`protocol` UnsignedInteger

An integer type that can represent only nonnegative values.