Swift 4

Operadores avanzados

Operadores: de bit

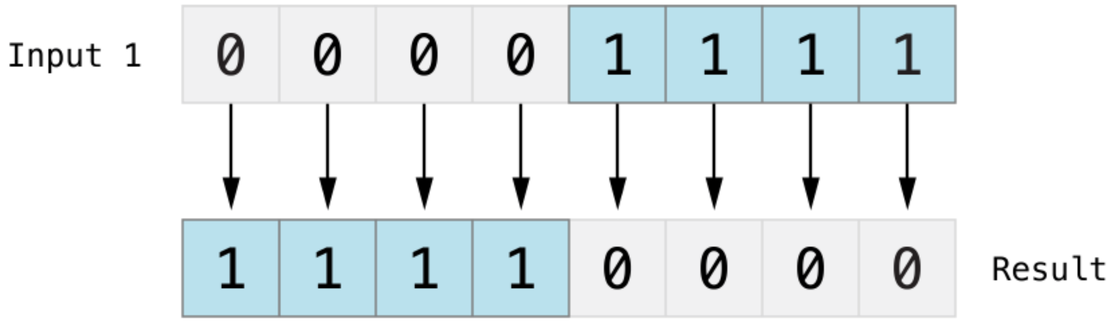
Operadores: de bit

- Permiten manipular los bits en bruto dentro de una estructura de datos
- Se utilizan normalmente en programación de bajo nivel, como programación gráfica y desarrollo de drivers o protocolos de comunicación
- Los operadores que soporta Swift son los mismos de C

Operadores: de bit

Operador	Operación
~	NOT de bit
&	AND de bit
|	OR de bit
^	XOR de bit

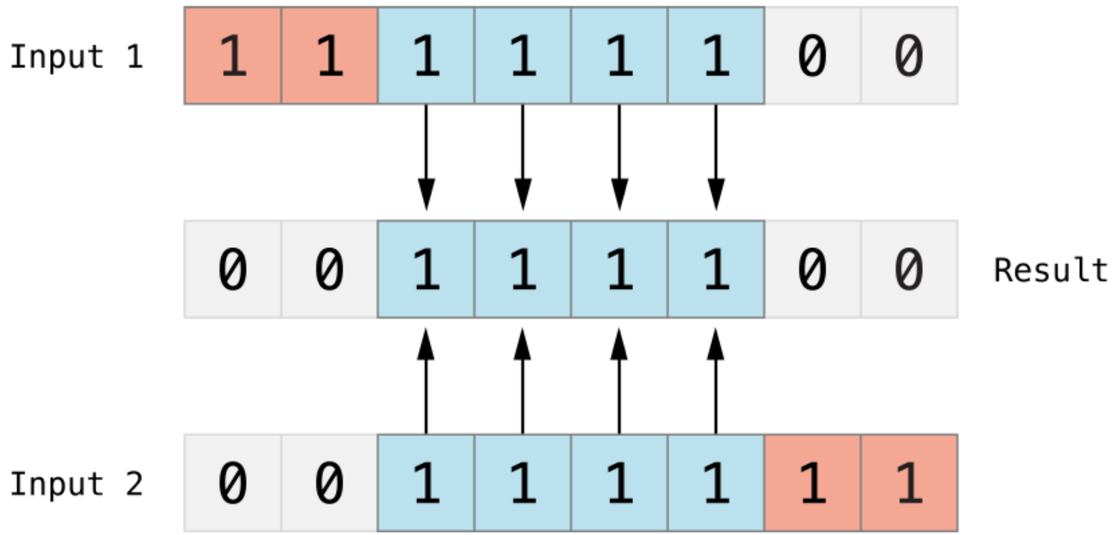
NOT de bit (~)



NOT de bit (~)

```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits  // equals 11110000
```

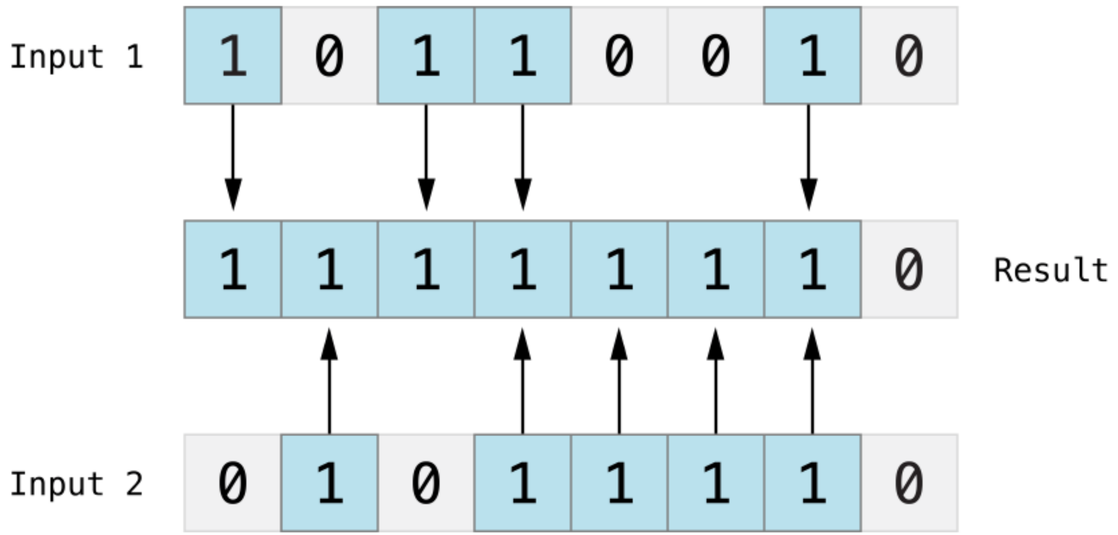
AND de bit (&)



AND de bit (&)

```
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits & lastSixBits  // equals 00111100
```

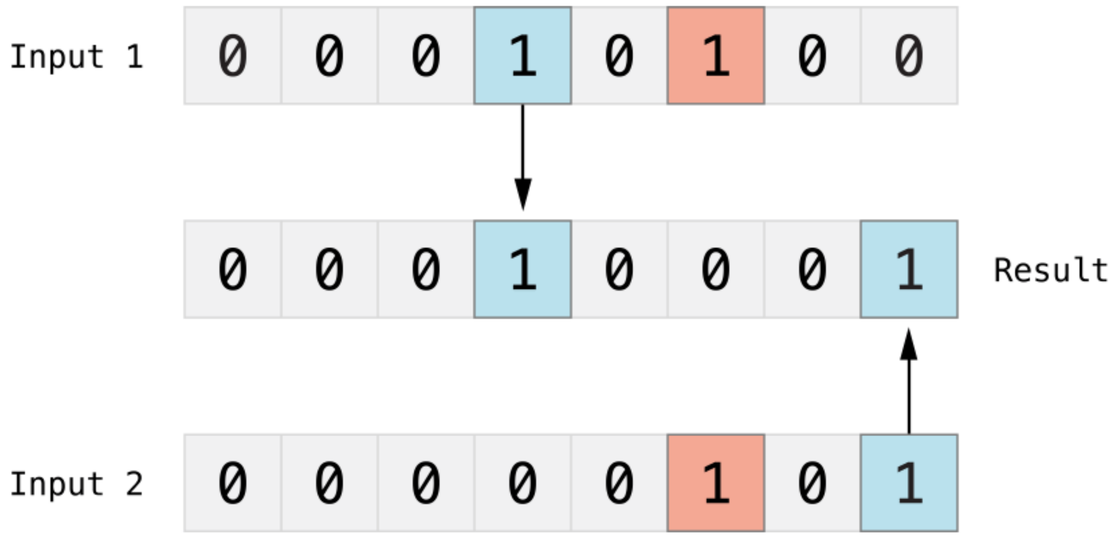
OR de bit (|)



OR de bit (|)

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits  // equals 11111110
```

XOR de bit (^)



XOR de bit (^)

```
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
let outputBits = firstBits ^ otherBits  // equals 00010001
```

Operadores: de desplazamiento

Operadores: de desplazamiento

- Los operadores de desplazamiento a izquierda y derecha mueven una serie de posiciones todos los bits de un número
- El efecto resultante es el de multiplicar (izquierda) o dividir (derecha) por un factor de dos el número
- El comportamiento varía dependiendo de si operamos con enteros con signo o sin signo

Operadores: de desplazamiento

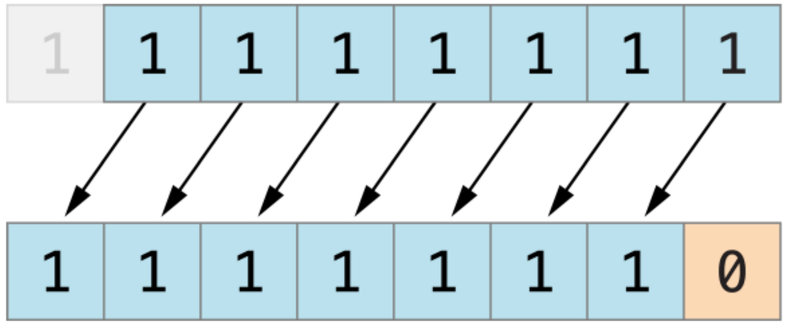
Operador	Operación
>>	Desplazamiento de bits a la derecha
<<	Desplazamiento de bits a la izquierda

Desplazamiento de enteros sin signo

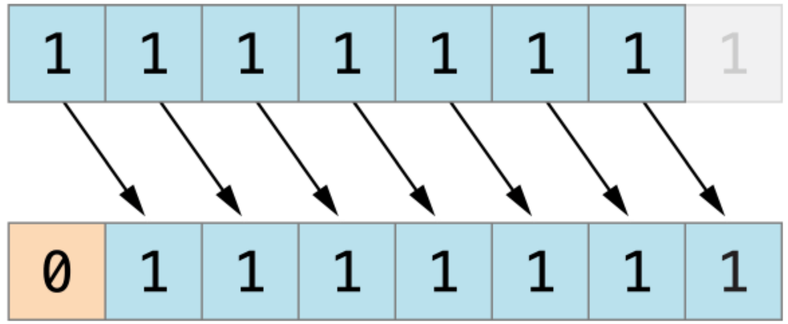
1. Los bits existentes se desplazan las posiciones requeridas
2. Los bits que quedan fuera de los límites del tipo, se descartan
3. Se insertan ceros en los bits necesarios para rellenar el dato

Desplazamiento de enteros sin signo

`11111111 << 1`



`11111111 >> 1`



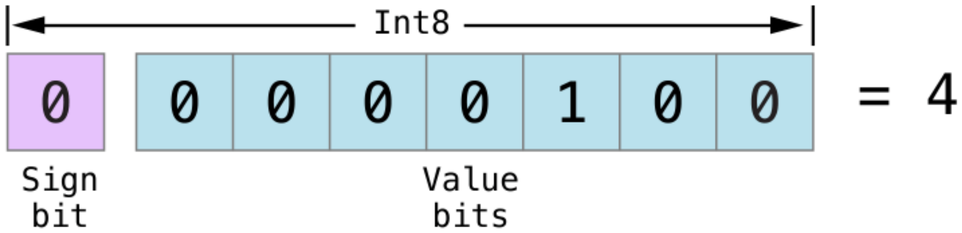
Desplazamiento de enteros sin signo

```
let shiftBits: UInt8 = 4    // 00000100 in binary
shiftBits << 1              // 00001000
shiftBits << 2              // 00010000
shiftBits << 5              // 10000000
shiftBits << 6              // 00000000
shiftBits >> 2              // 00000001
```

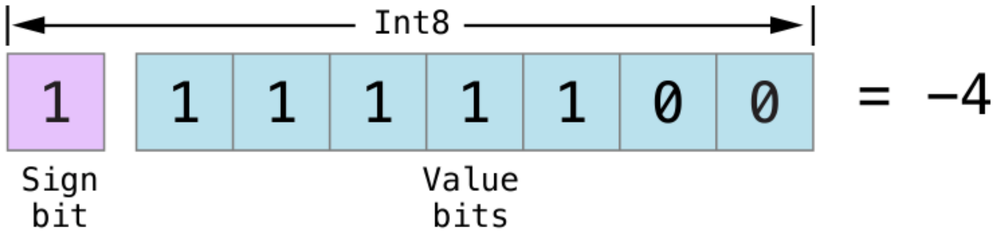
Desplazamiento de enteros sin signo

```
let pink: UInt32 = 0xCC6699
let redComponent = (pink & 0xFF0000) >> 16    // redComponent is 0xCC, or 204
let greenComponent = (pink & 0x00FF00) >> 8   // greenComponent is 0x66, or 102
let blueComponent = pink & 0x0000FF           // blueComponent is 0x99, or 153
```

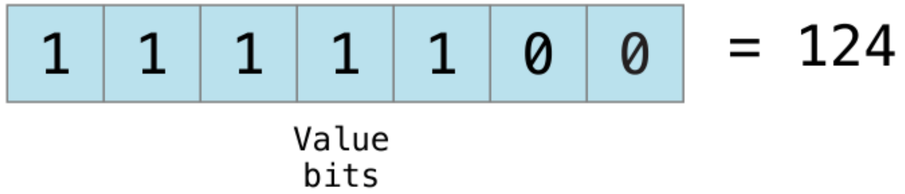
Representación en complemento a dos



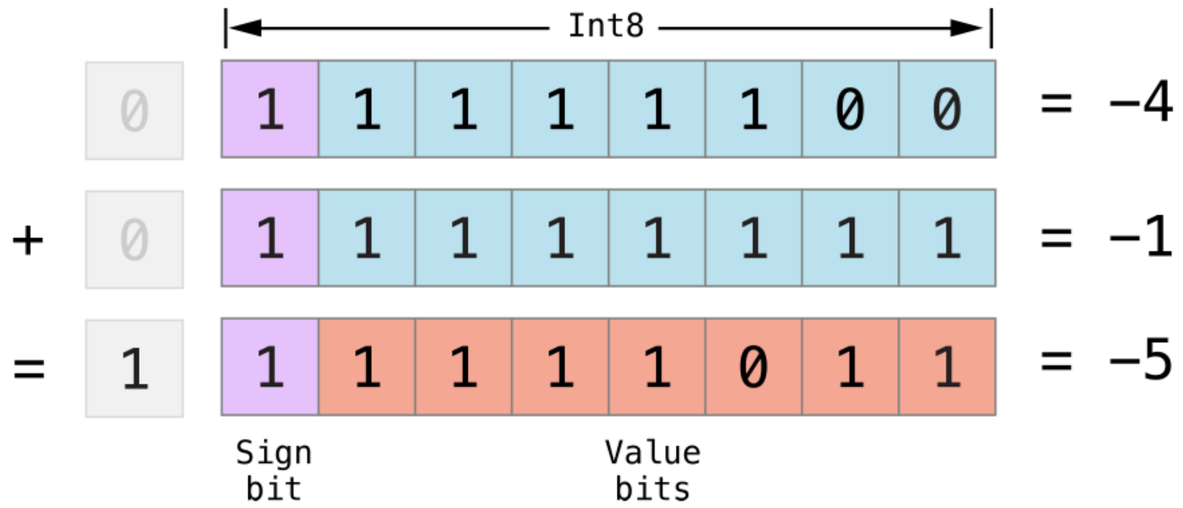
Representación en complemento a dos



Representación en complemento a dos



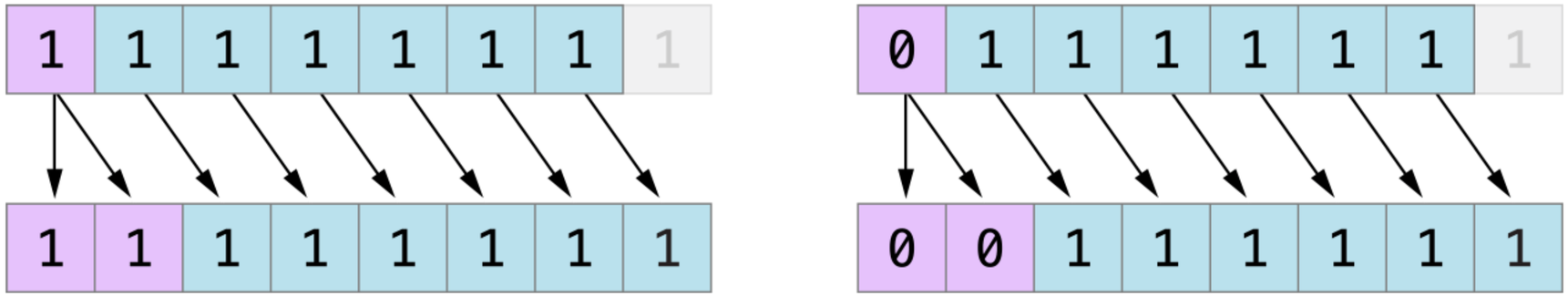
Representación en complemento a dos



Desplazamiento de enteros con signo

4. Al desplazar enteros con signo a la derecha, rellenar por la izquierda con el valor del bit de signo, en vez de ceros

Desplazamiento de enteros con signo



Operadores: aritméticos con overflow

Operadores: aritméticos con overflow

- Si intentamos insertar un valor en una variable entera que no pueda almacenar dicho valor, Swift por defecto produce un error de desbordamiento que evita que creemos un valor no válido

- Podemos forzar la operación usando los operadores con overflow de modo que se trunque el dato y no haya error de desbordamiento

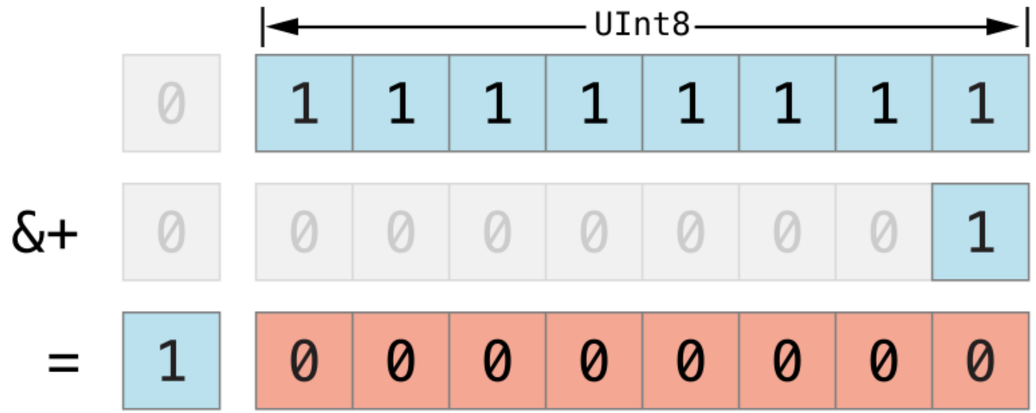
Operadores: aritméticos con overflow

Operador	Operación
&+	Suma
&−	Resta
&∗	Multiplicación

Overflow: superior

```
var unsignedOverflow = UInt8.max
// unsignedOverflow equals 255, which is the maximum value a UInt8 can hold
unsignedOverflow = unsignedOverflow &+ 1
// unsignedOverflow is now equal to 0
```

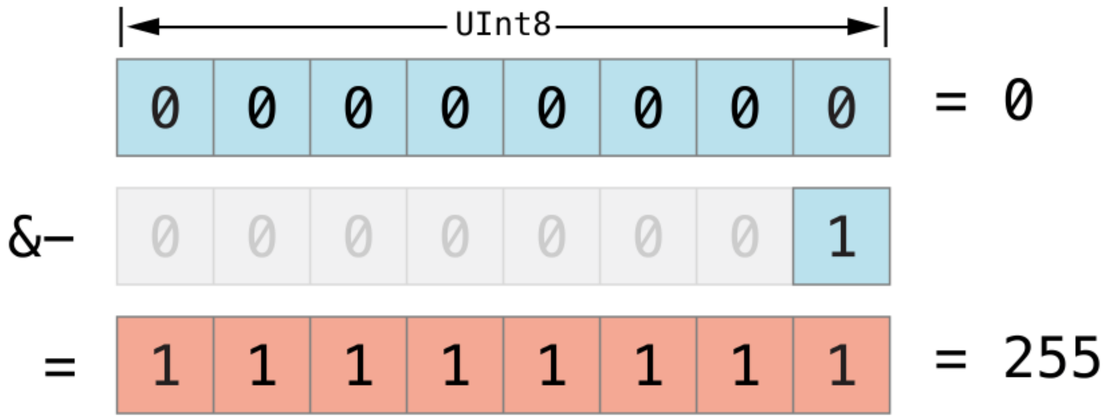
Overflow: superior



Overflow: inferior

```
var unsignedOverflow = UInt8.min
// unsignedOverflow equals 0, which is the minimum value a UInt8 can hold
unsignedOverflow = unsignedOverflow &- 1
// unsignedOverflow is now equal to 255
```

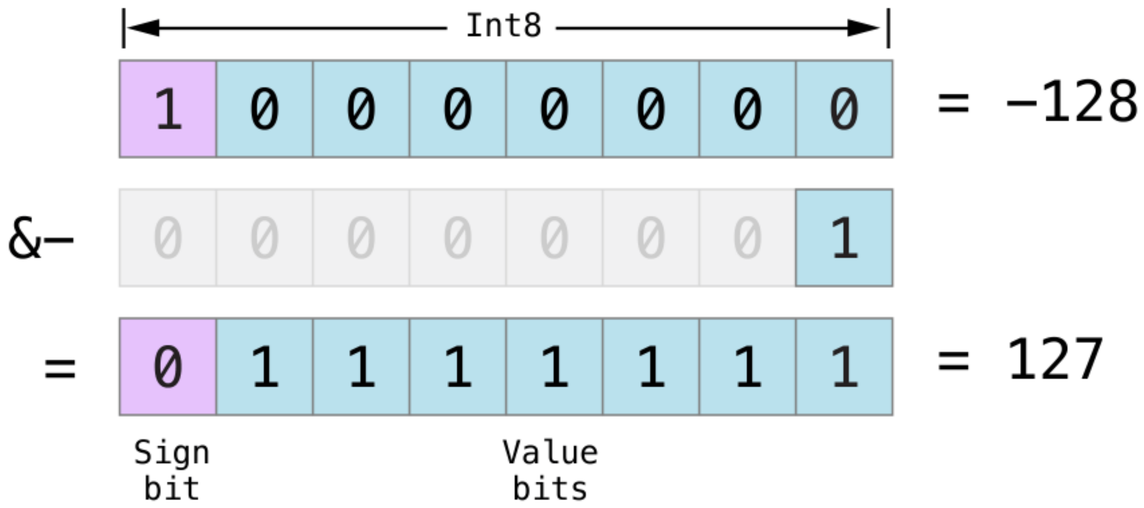
Overflow: inferior



Overflow: inferior

```
var signedOverflow = Int8.min
// signedOverflow equals -128, which is the minimum value an Int8 can hold
signedOverflow = signedOverflow &- 1
// signedOverflow is now equal to 127
```

Overflow: inferior



Precedencia de operadores

Precedencia de operadores

Precedencia	Descripción	Operadores
Mayor	Exponenciales	`<<, >>`
	Multiplicativos	`*, /, %, &*, &/, &*, &`
	Aditivos	`+, -, &+, &-, |, ^`
	Rangos	`--<, ...`
	Comprobación de tipo	`is, as`
	Relacionales	`<, <=, >, >=, ==, !=, ===, !==, ~=`
	Conjunción	`&&`
	Disyunción	`||`
	Condicional ternario	`?:`
Menor	Asignación	`=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=`

https://developer.apple.com/documentation/swift/operator_declarations

Sobrecarga de operadores

Sobrecarga de operadores

- Es la capacidad de clases y estructuras para proporcionar sus propias implementaciones de los operadores
- Permite crear nuevos operadores o redefinir el comportamiento de uno existente
- Podemos crear operadores infijos, prefijos o postfijos

Operador infijo

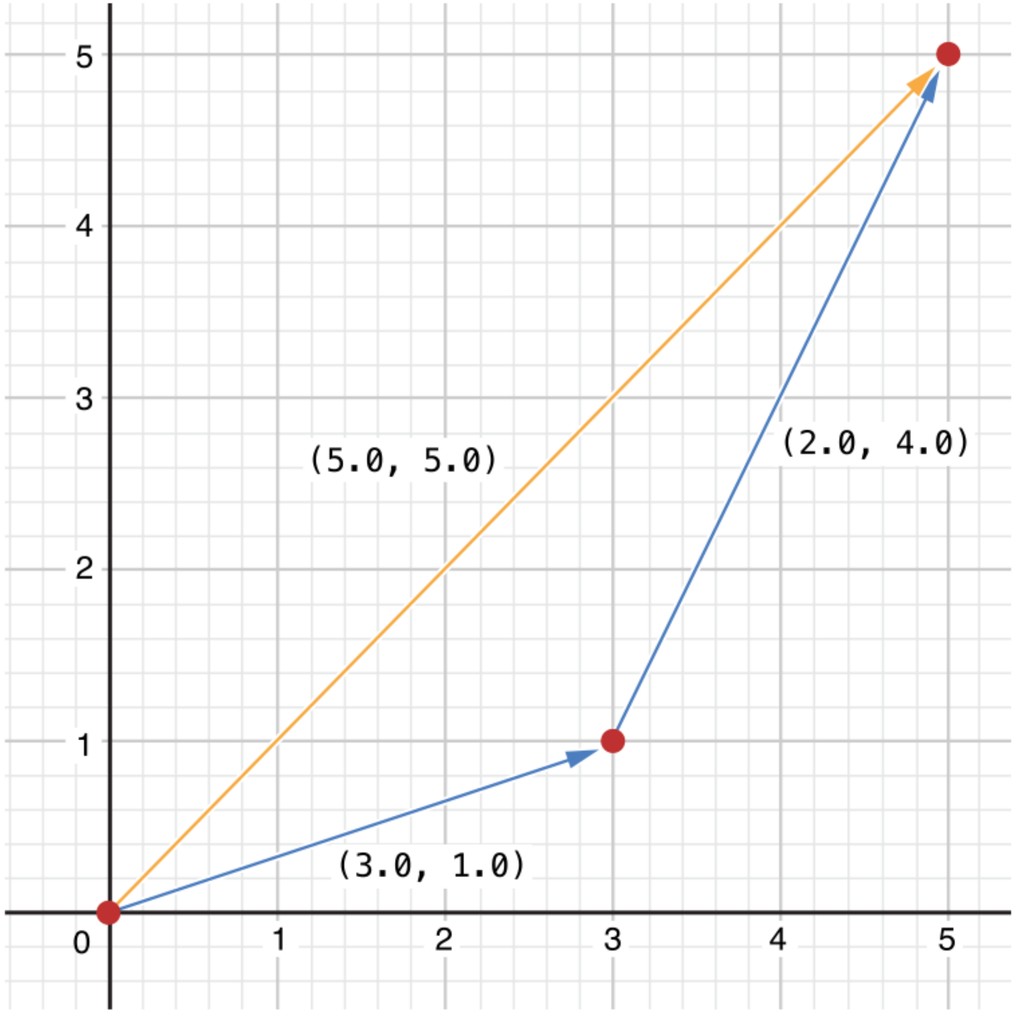
```
struct Vector2D {
    var x = 0.0, y = 0.0
}

extension Vector2D {
    static func + (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y + right.y)
    }
}
```

Operador infijo

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector is a Vector2D instance with values of (5.0, 5.0)
```

Operador infijo



Operador prefijo y postfijo

```
extension Vector2D {
    static prefix func - (vector: Vector2D) -> Vector2D {
        return Vector2D(x: -vector.x, y: -vector.y)
    }
}

let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative is a Vector2D instance with values of (-3.0, -4.0)
let alsoPositive = -negative
// alsoPositive is a Vector2D instance with values of (3.0, 4.0)
```

Operadores de asignación compuesta

```
extension Vector2D {
    static func += (left: inout Vector2D, right: Vector2D) {
        left = left + right
    }
}

var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original now has values of (4.0, 6.0)
```

Operadores de asignación compuesta

- No se pueden sobrecargar el operador `=` ni el `?:`

Operadores de equivalencia

```
extension Vector2D {
    static func == (left: Vector2D, right: Vector2D) -> Bool {
        return (left.x == right.x) && (left.y == right.y)
    }
    static func != (left: Vector2D, right: Vector2D) -> Bool {
        return !(left == right)
    }
}
```

Operadores de equivalencia

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    print("These two vectors are equivalent.")
}
// Prints "These two vectors are equivalent."
```

Operadores personalizados

```
prefix operator +++ // declarado a nivel global

extension Vector2D {
    static prefix func +++ (vector: inout Vector2D) -> Vector2D {
        vector += vector
        return vector
    }
}

var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled now has values of (2.0, 8.0)
// afterDoubling also has values of (2.0, 8.0)
```

Precedencia de operadores personalizados

- Los operadores infijos personalizados pueden especificar un grupo de precedencia
- Si no lo especifican, se sitúan un peldaño por encima del operador ternario
- Los prefijos y postfijos no pueden especificarla, el postfijo siempre se aplica primero

https://developer.apple.com/documentation/swift/operator_declarations

Precedencia de operadores personalizados

```
infix operator +-: AdditionPrecedence

extension Vector2D {
    static func +- (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y - right.y)
    }
}

let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +- secondVector
// plusMinusVector is a Vector2D instance with values of (4.0, -2.0)
```