```swift
//: ## Inspiration for Swift Playgrounds:
//: * Bret Victor's [**Learnable Programming**](http://worrydream.com/#!/LearnableProgramming) concept (see also the talk [**Inventing on Principle**](http://worrydream.com/#!/InventingOnPrinciple))
//: * [**Light Table**](http://lighttable.com/)

import Cocoa

//: ## Curried functions

// Function currying can come in handy when adopting a functional style of programming (eg: when you have chains of unary functions)
func offer(a thing: String)(to person: String) -> String {
    return "\(person): take a \(thing)"
}

offer(a: "Beer")(to: "Alice")

// The following example shows Swift's first-class functions in action!
let 🍻 = offer(a: "beer")

🍻(to: "Alice")

🍻(to: "Mario")




//: ## Class extension

// Methods and properties of the class extension will be available
// anywhere the module declaring this extension is imported
extension String {

    var offerBeerTo: (String) -> String {
        let offerBeer = offer(a: "🍺")
        return { recipient in
            return "\(self): <\(offerBeer(to: recipient))>"
        }
    }

}

"Matteo".offerBeerTo("Bob")





//: ## Custom operators (❤)

// Swift let us define custom operators, and even overload existing ones.
// Every operator in Swift is just a function, with a UTF-8 symbol as name
infix operator ❤ { associativity left precedence 140 }
func ❤ (lover1: String, lover2: String) -> String {
    return "\(lover1) ❤️ \(lover2)"
}
```

```swift
"I" ❤ "Swift!"




//: ## The power of Playground in action

// Playground's value history is a handy way to see values of expressions
    across time.
// It can be really useful for debugging
func fibonacci(n: Int) -> Int {
    return n <= 1 ? n : fibonacci(n-1) + fibonacci(n-2)
}

fibonacci(5)




//: ## Map, Filter, Reduce

struct Attendee {

    let name: String
    let age: Int

}

let attendees = [Attendee(name: "Marco", age: 27), Attendee(name: "Alice",
    age: 17), Attendee(name: "Giuseppe", age: 35)]

let maggiorenni = attendees.map { attendee in
    attendee.age
}.filter { age in
    age >= 18
}

let avg = maggiorenni.reduce(0, combine: { sum, age in
    sum + age
}) / maggiorenni.count

// This is equivalent to the above form: remember that '+' is just a binary
    function, thus, it matches parameter combine's type.
maggiorenni.reduce(0, combine: +) / maggiorenni.count




//: ## Functional compositions

// Box is just a workaround to the Swift compiler not allowing enums with
    multiple variable-size associated values.
// Hopefully the Swift team will address the issue soon
class Box<T> {
    let value: T

    init(_ value: T) {
```

```
        self.value = value
    }
}

enum Result<T> {
    case Failure(Box<NSError>)
    case Success(Box<T>)
}

func description<T>(value: Result<T>) -> String {
    switch value {
    case .Failure(let error):
        return "Error: \(error.value)"
    case .Success(let result):
        return "Success: \(result.value)"
    }
}

func inverse(num: Double) -> Result<Double> {
    if !num.isZero {
        return .Success(Box(1.0/num))
    } else {
        return .Failure(Box(NSError(domain: "Division by zero!", code: 1,
            userInfo: nil)))
    }
}

func squareRoot(num: Double) -> Result<Double> {
    if num >= 0 {
        return .Success(Box(sqrt(num)))
    } else {
        return .Failure(Box(NSError(domain: "Can't square root a negative
            number!", code: 2, userInfo: nil)))
    }
}

func logarithm(num: Double) -> Result<Double> {
    if num > 0 {
        return .Success(Box(log(num)))
    } else {
        return .Failure(Box(NSError(domain: "Can't do the logarithm of a
            negative number!", code: 3, userInfo: nil)))
    }
}

let n = 10.0

// This is usually known as 'flatMap' or 'bind', and is very common among
    funcional languages.
// Swift does actually provide implementations of flatMap for sequences,
    collections, and optionals
func when<T,U>(optionalValue: Result<T>, apply: (T) -> Result<U>) -> Result<U>
    {
    switch optionalValue {
    case .Failure(let error):
        return .Failure(error)
    case .Success(let value):
        return apply(value.value)
    }
}
```

```
let result = when(when(inverse(n)) {
        squareRoot($0)
    }) {
        logarithm($0)
}

println(description(result))

infix operator >>> { associativity left }
func >>> <T,U>(optionalValue: Result<T>, apply: (T) -> Result<U>) -> Result<U>
    {
    return when(optionalValue, apply)
}

// Hides away all the boilerplate, leaving visible only what really matters
    ("what" our code does, not "how" to do it)
let result1 = inverse(n) >>> squareRoot >>> logarithm >>> inverse
println(description(result1))

func lift(num: Double) -> Result<Double> {
    return .Success(Box(num))
}

let result2 = lift(n) >>> inverse >>> squareRoot >>> logarithm >>> inverse
println(description(result2))
```