The underestimated power of KeyPaths

Vincent Pradeilles (@v_pradeilles) – Worldline 🇫🇷

KeyPaths?

KeyPaths

KeyPaths were introduced with Swift 4

They are a way to defer a call to the getter/setter of a property

```
let countKeyPath: KeyPath<String, Int> = \String.count // or \.count

let string = "Foo"

string[keyPath: countKeyPath] // 3
```

They perform the same job than a closure, but with less `$0` hanging around

Underestimated?

Let’s look at some examples!

Data Manipulation

```
people.sorted(by: { $0.lastName < $1.lastName })
      .filter({ $0.isOverEighteen })
      .map({ $0.lastName })
```

```
people.sorted(by: \.lastName)
      .filter(\.isOverEighteen)
      .map(\.lastName)
```

How to implement it?

How to implement it?

```
extension Sequence {
    func map<T>(_ attribute: KeyPath<Element, T>) -> [T] {
        return map { $0[keyPath: attribute] }
    }
}
```

How to implement it?

```
extension Sequence {
    func map<T>(_ attribute: KeyPath<Element, T>) -> [T] {
        return map { $0[keyPath: attribute] }
    }
}
```

How to implement it?

```
extension Sequence {
    func map<T>(_ attribute: KeyPath<Element, T>) -> [T] {
        return map { $0[keyPath: attribute] }
    }
}
```

How to implement it?

```
extension Sequence {
    func map<T>(_ attribute: KeyPath<Element, T>) -> [T] {
        return map { $0[keyPath: attribute] }
    }
}
```

How to implement it?

```
extension Sequence {
    func map<T>(_ attribute: KeyPath<Element, T>) -> [T] {
        return map { $0[keyPath: attribute] }
    }
}
```

```
people.map(\.lastName)
```

How to implement it?

```
extension Sequence {
    func sorted<T: Comparable>(by attribute: KeyPath<Element, T>) -> [Element] {
        return sorted(by: { (elm1, elm2) -> Bool in
            return elm1[keyPath: attribute] < elm2[keyPath: attribute]
        })
    }
}
```

How to implement it?

```
extension Sequence {
    func sorted<T: Comparable>(by attribute: KeyPath<Element, T>) -> [Element] {
        return sorted(by: { (elm1, elm2) -> Bool in
            return elm1[keyPath: attribute] < elm2[keyPath: attribute]
        })
    }
}
```

How to implement it?

```
extension Sequence {
    func sorted<T: Comparable>(by attribute: KeyPath<Element, T>) -> [Element] {
        return sorted(by: { (elm1, elm2) -> Bool in
            return elm1[keyPath: attribute] < elm2[keyPath: attribute]
        })
    }
}
```

How to implement it?

```
extension Sequence {
    func sorted<T: Comparable>(by attribute: KeyPath<Element, T>) -> [Element] {
        return sorted(by: { (elm1, elm2) -> Bool in
            return elm1[keyPath: attribute] < elm2[keyPath: attribute]
        })
    }
}
```

How to implement it?

```
extension Sequence {
    func sorted<T: Comparable>(by attribute: KeyPath<Element, T>) -> [Element] {
        return sorted(by: { (elm1, elm2) -> Bool in
            return elm1[keyPath: attribute] < elm2[keyPath: attribute]
        })
    }
}
```

How to implement it?

```
extension Sequence {
    func sorted<T: Comparable>(by attribute: KeyPath<Element, T>) -> [Element] {
        return sorted(by: { (elm1, elm2) -> Bool in
            return elm1[keyPath: attribute] < elm2[keyPath: attribute]
        })
    }
}
```

How to implement it?

```
extension Sequence {
    func sorted<T: Comparable>(by attribute: KeyPath<Element, T>) -> [Element] {
        return sorted(by: { (elm1, elm2) -> Bool in
            return elm1[keyPath: attribute] < elm2[keyPath: attribute]
        })
    }
}

people.sorted(by: \.age)
```

Same idea goes for all classic functions, like `min`, `max`, `sum`, `average`, etc.

And if you don’t want to write those wrappers…

You don’t even have to!

From KeyPath to getter function

```
func get<Type, Property>(_ keyPath: KeyPath<Type, Property>)
                         -> (Type) -> Property {

    return { obj in obj[keyPath: keyPath] }
}
```

From KeyPath to getter function

```
func get<Type, Property>(_ keyPath: KeyPath<Type, Property>)
                         -> (Type) -> Property {

    return { obj in obj[keyPath: keyPath] }
}
```

From KeyPath to getter function

```
func get<Type, Property>(_ keyPath: KeyPath<Type, Property>)
                         -> (Type) -> Property {

    return { obj in obj[keyPath: keyPath] }
}
```

From KeyPath to getter function

```
func get<Type, Property>(_ keyPath: KeyPath<Type, Property>)
                         -> (Type) -> Property {

    return { obj in obj[keyPath: keyPath] }
}
```

From KeyPath to getter function

```
func get<Type, Property>(_ keyPath: KeyPath<Type, Property>)
                         -> (Type) -> Property {

    return { obj in obj[keyPath: keyPath] }
}

people.filter(get(\.isOverEighteen))
```

An operator can even make the syntax shorter!

Defining an operator

```
prefix operator ^

prefix func ^ <Element, Attribute>(_ keyPath: KeyPath<Element, Attribute>)
                                    -> (Element) -> Attribute {

    return get(keyPath)

}

people.map(^\.lastName)
```

It’s not always this simple…

```
people.sorted(by: { $0.lastName < $1.lastName })
```

Just build another helper

```
func their<Type, T: Comparable>(_ keyPath: KeyPath<Type, T>,
                                comparedWith comparator: @escaping (T, T) -> Bool = (<))
                                -> (Type, Type) -> Bool {
    return { obj1, obj2 in
        return comparator(obj1[keyPath: keyPath], obj2[keyPath: keyPath])
    }
}
```

Just build another helper

```
func their<Type, T: Comparable>(_ keyPath: KeyPath<Type, T>,
                                comparedWith comparator: @escaping (T, T) -> Bool = (<))
                                -> (Type, Type) -> Bool {
    return { obj1, obj2 in
        return comparator(obj1[keyPath: keyPath], obj2[keyPath: keyPath])
    }
}
```

Just build another helper

```
func their<Type, T: Comparable>(_ keyPath: KeyPath<Type, T>,
                                comparedWith comparator: @escaping (T, T) -> Bool = (<))
                                -> (Type, Type) -> Bool {
    return { obj1, obj2 in
        return comparator(obj1[keyPath: keyPath], obj2[keyPath: keyPath])
    }
}
```

Just build another helper

```
func their<Type, T: Comparable>(_ keyPath: KeyPath<Type, T>,
                                comparedWith comparator: @escaping (T, T) -> Bool = (<))
                                -> (Type, Type) -> Bool {
    return { obj1, obj2 in
        return comparator(obj1[keyPath: keyPath], obj2[keyPath: keyPath])
    }
}
```

Just build another helper

```
func their<Type, T: Comparable>(_ keyPath: KeyPath<Type, T>,
                                comparedWith comparator: @escaping (T, T) -> Bool = (<))
                                -> (Type, Type) -> Bool {
    return { obj1, obj2 in
        return comparator(obj1[keyPath: keyPath], obj2[keyPath: keyPath])
    }
}
```

Just build another helper

```
func their<Type, T: Comparable>(_ keyPath: KeyPath<Type, T>,
                                comparedWith comparator: @escaping (T, T) -> Bool = (<))
                                -> (Type, Type) -> Bool {
    return { obj1, obj2 in
        return comparator(obj1[keyPath: keyPath], obj2[keyPath: keyPath])
    }
}
```

Just build another helper

```
func their<Type, T: Comparable>(_ keyPath: KeyPath<Type, T>,
                                comparedWith comparator: @escaping (T, T) -> Bool = (<))
                                -> (Type, Type) -> Bool {
    return { obj1, obj2 in
        return comparator(obj1[keyPath: keyPath], obj2[keyPath: keyPath])
    }
}
```

Just build another helper

```
func their<Type, T: Comparable>(_ keyPath: KeyPath<Type, T>,
                                comparedWith comparator: @escaping (T, T) -> Bool = (<))
                                -> (Type, Type) -> Bool {
    return { obj1, obj2 in
        return comparator(obj1[keyPath: keyPath], obj2[keyPath: keyPath])
    }
}
```

```
people.sorted(by: { $0.lastName < $1.lastName })
```

```
people.sorted(by: their(\.lastName))
```

```
people.sorted(by: their(\.lastName, comparedWith: >))
```

Let’s take it one step further: DSLs

Predicates

NSPredicate is great, but it’s a stringly-typed API.

```
NSPredicate(format: "firstName = 'Bob'")

NSPredicate(format: "lastName = %@", "Smith")
```

Predicates

We want to be able to write complex predicates, like this one:

```
people.select(where: { $0.age <= 22 } && { $0.lastName.count > 10 })
```

Predicates

We want to be able to write complex predicates, like this one:

```
people.select(where: \.age <= 22 && \.lastName.count > 10)
```

To do so, we need ways to:

- Define what predicates are
- Construct them
- Combine them
- Evaluate them

Defining Predicates

```
struct Predicate<Element> {

    private let condition: (Element) -> Bool

    func evaluate(for element: Element) -> Bool {
        return condition(element)
    }

}
```

Constructing Predicates

```
contacts.select(where: \.age <= 22 && \.lastName.count > 10)


func <= <Element, T: Comparable>(_ attribute: KeyPath<Element, T>,
                                 _ constant: T)
                                 -> Predicate<Element> {

    return Predicate(condition: { value in value[keyPath: attribute] <= constant })
}
```

Constructing Predicates

```
contacts.select(where: \.age <= 22 && \.lastName.count > 10)


func <= <Element, T: Comparable>(_ attribute: KeyPath<Element, T>,
                                 _ constant: T)
                                 -> Predicate<Element> {

    return Predicate(condition: { value in value[keyPath: attribute] <= constant })
}
```

Constructing Predicates

```
contacts.select(where: \.age <= 22 && \.lastName.count > 10)


func <= <Element, T: Comparable>(_ attribute: KeyPath<Element, T>,
                                 _ constant: T)
                                 -> Predicate<Element> {

    return Predicate(condition: { value in value[keyPath: attribute] <= constant })
}
```

Constructing Predicates

```
contacts.select(where: \.age <= 22 && \.lastName.count > 10)


func <= <Element, T: Comparable>(_ attribute: KeyPath<Element, T>,
                                 _ constant: T)
                                 -> Predicate<Element> {

    return Predicate(condition: { value in value[keyPath: attribute] <= constant })
}
```

Constructing Predicates

```
contacts.select(where: \.age <= 22 && \.lastName.count > 10)


func <= <Element, T: Comparable>(_ attribute: KeyPath<Element, T>,
                                 _ constant: T)
                                 -> Predicate<Element> {

    return Predicate(condition: { value in value[keyPath: attribute] <= constant })
}
```

Constructing Predicates

```
contacts.select(where: \.age <= 22 && \.lastName.count > 10)


func <= <Element, T: Comparable>(_ attribute: KeyPath<Element, T>,
                                 _ constant: T)
                                 -> Predicate<Element> {

    return Predicate(condition: { value in value[keyPath: attribute] <= constant })
}
```

Constructing Predicates

```
contacts.select(where: \.age <= 22 && \.lastName.count > 10)
```

```
func <= <Element, T: Comparable>(_ attribute: KeyPath<Element, T>,
                                 _ constant: T)
                                 -> Predicate<Element> {

    return Predicate(condition: { value in value[keyPath: attribute] <= constant })
}
```

```
func > <Element, T: Comparable>(_ attribute: KeyPath<Element, T>,
                                _ constant: T)
                                -> Predicate<Element> {

    return Predicate(condition: { value in value[keyPath: attribute] > constant })
}
```

Combining Predicates

```
contacts.select(where: \.age <= 22 && \.lastName.count > 10)


func && <Element> (_ leftPredicate: Predicate<Element>,
                   _ rightPredicate: Predicate<Element>)
                   -> Predicate<Element> {

    return Predicate(condition: { value in
        leftPredicate.evaluate(for: value) && rightPredicate.evaluate(for: value)
    })
}
```

Evaluating Predicates

```
extension Sequence {

    func select(where predicate: Predicate<Element>) -> [Element] {
        return filter { element in predicate.evaluate(for: element) }
    }

}
```

That’s all you need!

```
people.select(where: \.age <= 22 && \.lastName.count > 10)

people.select(where: 4...18 ~= \.age)

people.first(where: \.age < 18)

people.contains(where: \.lastName.count > 10)
```

You can even go for more fancy stuff…

Switching with Predicates

```
func ~= <Element>(_ lhs: KeyPathPredicate<Element>, rhs: Element) -> Bool {
    return lhs.evaluate(for: rhs)
}
```

Switching with Predicates

```
func ~= <Element>(_ lhs: KeyPathPredicate<Element>, rhs: Element) -> Bool {
    return lhs.evaluate(for: rhs)
}

switch person {
case \.firstName == "Charlie":
    print("I'm Charlie!")
    fallthrough
case \.age < 18:
    print("I'm not an adult...")
    fallthrough
default:
    break
}
```

Now, let’s look at other possible applications

Builder Pattern

Builder Pattern

Provides separation between building and using an object

Builder Pattern

```
let label = UILabel()
label.textColor = .red
label.text = "Hello iOS Conf SG!"
label.textAlignment = .center
label.layer.cornerRadius = 5

// do something with label
view.addSubview(label)
```

Builder Pattern

```
protocol Buildable {}
```

Builder Pattern

```
protocol Buildable {}

extension Buildable where Self: AnyObject {
    func with<T>(_ property: ReferenceWritableKeyPath<Self, T>,
                 setTo value: T) -> Self {
        self[keyPath: property] = value
        return self
    }
}
```

Builder Pattern

```
protocol Buildable {}

extension Buildable where Self: AnyObject {
    func with<T>(_ property: ReferenceWritableKeyPath<Self, T>,
                 setTo value: T) -> Self {
        self[keyPath: property] = value
        return self
    }
}
```

Builder Pattern

```
protocol Buildable {}

extension Buildable where Self: AnyObject {
    func with<T>(_ property: ReferenceWritableKeyPath<Self, T>,
                 setTo value: T) -> Self {
        self[keyPath: property] = value
        return self
    }
}
```

Builder Pattern

```
protocol Buildable {}

extension Buildable where Self: AnyObject {
    func with<T>(_ property: ReferenceWritableKeyPath<Self, T>,
                 setTo value: T) -> Self {
        self[keyPath: property] = value
        return self
    }
}
```

Builder Pattern

```
protocol Buildable {}

extension Buildable where Self: AnyObject {
    func with<T>(_ property: ReferenceWritableKeyPath<Self, T>,
                 setTo value: T) -> Self {
        self[keyPath: property] = value
        return self
    }
}
```

Builder Pattern

```
protocol Buildable {}

extension Buildable where Self: AnyObject {
    func with<T>(_ property: ReferenceWritableKeyPath<Self, T>,
                 setTo value: T) -> Self {
        self[keyPath: property] = value
        return self
    }
}
```

Builder Pattern

```
protocol Buildable {}

extension Buildable where Self: AnyObject {
    func with<T>(_ property: ReferenceWritableKeyPath<Self, T>,
                 setTo value: T) -> Self {
        self[keyPath: property] = value
        return self
    }
}
```

Builder Pattern

```
protocol Buildable {}

extension Buildable where Self: AnyObject {
    func with<T>(_ property: ReferenceWritableKeyPath<Self, T>,
                 setTo value: T) -> Self {
        self[keyPath: property] = value
        return self
    }
}

extension NSObject: Buildable { }
```

Builder Pattern

```
let label = UILabel()
    .with(\.textColor, setTo: .red)
    .with(\.text, setTo: "Hello iOS Conf SG!")
    .with(\.textAlignment, setTo: .center)
    .with(\.layer.cornerRadius, setTo: 5)

// do something with label
view.addSubview(label)
```

Identifiable Pattern

Identifiable Pattern

Enables equality based on an ID

Identifiable Pattern

```
protocol Identifiable: Equatable {
    associatedtype ID: Equatable
    static var idKeyPath: KeyPath<Self, ID> { get }
}
```

Identifiable Pattern

```
protocol Identifiable: Equatable {
    associatedtype ID: Equatable
    static var idKeyPath: KeyPath<Self, ID> { get }
}

extension Identifiable {
    static func == (lhs: Self, rhs: Self) -> Bool {
        return lhs[keyPath: Self.idKeyPath] == rhs[keyPath: Self.idKeyPath]
    }
}
```

Identifiable Pattern

```
struct User: Identifiable {
    static let idKeyPath = \User.id
    let id: String

    let firstName: String
    let lastName: String
}
```

Identifiable Pattern

```
let user1 = User(id: "1", firstName: "John", lastName: "Lennon")
let user2 = User(id: "2", firstName: "Ringo", lastName: "Starr")

user1 == user2 //  false

let user3 = User(id: "2", firstName: "Paul", lastName: "McCartney")

user2 == user3 // true
```

Validable Pattern

Validable Pattern

Performs property-wise checks

Validable Pattern

```
struct Validator<T> {
    let validee: T
    let validator: (T) -> Bool

    func validate() -> Bool {
        return validator(validee)
    }
}
```

Validable Pattern

```
protocol Validable { }
```

Validable Pattern

```
protocol Validable { }

extension Validable {
    func add<Property>(validation: @escaping (Property) -> Bool,
                       for property: KeyPath<Self, Property>)
                       -> Validator<Self> {

        return Validator<Self>(validee: self,
                               validator: { validation($0[keyPath: property]) })

    }
}
```

Validable Pattern

```
protocol Validable { }

extension Validable {
    func add<Property>(validation: @escaping (Property) -> Bool,
                       for property: KeyPath<Self, Property>)
                       -> Validator<Self> {

        return Validator<Self>(validee: self,
                               validator: { validation($0[keyPath: property]) })

    }
}
```

Validable Pattern

```
protocol Validable { }

extension Validable {
    func add<Property>(validation: @escaping (Property) -> Bool,
                       for property: KeyPath<Self, Property>)
                       -> Validator<Self> {

        return Validator<Self>(validee: self,
                               validator: { validation($0[keyPath: property]) })

    }
}
```

Validable Pattern

```
protocol Validable { }

extension Validable {
    func add<Property>(validation: @escaping (Property) -> Bool,
                       for property: KeyPath<Self, Property>)
                       -> Validator<Self> {

        return Validator<Self>(validee: self,
                               validator: { validation($0[keyPath: property]) })

    }
}
```

Validable Pattern

```
protocol Validable { }

extension Validable {
    func add<Property>(validation: @escaping (Property) -> Bool,
                       for property: KeyPath<Self, Property>)
                       -> Validator<Self> {

        return Validator<Self>(validee: self,
                               validator: { validation($0[keyPath: property]) })

    }
}
```

Validable Pattern

```
protocol Validable { }

extension Validable {
    func add<Property>(validation: @escaping (Property) -> Bool,
                       for property: KeyPath<Self, Property>)
                       -> Validator<Self> {

        return Validator<Self>(validee: self,
                               validator: { validation($0[keyPath: property]) })

    }
}
```

Validable Pattern

```
protocol Validable { }

extension Validable {
    func add<Property>(validation: @escaping (Property) -> Bool,
                       for property: KeyPath<Self, Property>)
                       -> Validator<Self> {

        return Validator<Self>(validee: self,
                               validator: { validation($0[keyPath: property]) })

    }
}
```

Validable Pattern

```
extension Validator {
    func add<Property>(validation: @escaping (Property) -> Bool,
                       for property: KeyPath<T, Property>)
                       -> Validator<T> {

        return Validator<T>(validee: self.validee,
                            validator: { self.validate() &&
                                         validation($0[keyPath: property]) })

    }
}
```

Validable Pattern

```
extension Validator {
    func add<Property>(validation: @escaping (Property) -> Bool,
                       for property: KeyPath<T, Property>)
                       -> Validator<T> {

        return Validator<T>(validee: self.validee,
                            validator: { self.validate() &&
                                         validation($0[keyPath: property]) })

    }
}
```

Validable Pattern

```
extension Validator {
    func add<Property>(validation: @escaping (Property) -> Bool,
                       for property: KeyPath<T, Property>)
                       -> Validator<T> {

        return Validator<T>(validee: self.validee,
                            validator: { self.validate() &&
                                         validation($0[keyPath: property]) })

    }
}
```

Validable Pattern

```
extension Validator {
    func add<Property>(validation: @escaping (Property) -> Bool,
                       for property: KeyPath<T, Property>)
                       -> Validator<T> {

        return Validator<T>(validee: self.validee,
                            validator: { self.validate() &&
                                         validation($0[keyPath: property]) })

    }
}
```

Validable Pattern

```
let user1 = User(id: "1", firstName: "John", lastName: "Lennon")
let user2 = User(id: "2", firstName: "Ringo", lastName: "Starr")
```

Validable Pattern

```
let user1 = User(id: "1", firstName: "John", lastName: "Lennon")
let user2 = User(id: "2", firstName: "Ringo", lastName: "Starr")

extension User: Validable { }
```

Validable Pattern

```
let user1 = User(id: "1", firstName: "John", lastName: "Lennon")
let user2 = User(id: "2", firstName: "Ringo", lastName: "Starr")

extension User: Validable { }

user1.add(validation: { !$0.isEmpty }, for: \.id)
     .add(validation: { $0 == "Lennon" }, for: \.lastName)
     .validate() // true
```

Validable Pattern

```
let user1 = User(id: "1", firstName: "John", lastName: "Lennon")
let user2 = User(id: "2", firstName: "Ringo", lastName: "Starr")

extension User: Validable { }

user1.add(validation: { !$0.isEmpty }, for: \.id)
     .add(validation: { $0 == "Lennon" }, for: \.lastName)
     .validate() // true

user2.add(validation: { !$0.isEmpty }, for: \.id)
     .add(validation: { $0 == "Lennon" }, for: \.lastName)
     .validate() // false
```

(Of course, you could also use predicates instead of closures 😉)

Recap

Recap

KeyPaths offer deferred reading/writing to a property

They allow for a very clean and expressive syntax

They work very well with generic algorithms

They are a great tool to build DSLs

They offer nice sugar syntax to many patterns

Conclusion

Conclusion

KeyPaths are great, use them!

Big thank you to Marion Curtil and Jérôme Alves for their valuable inputs.

You liked what you saw, and you want it in your app?
https://github.com/vincent-pradeilles/KeyPathKit