The underestimated power of KeyPaths

Vincent Pradeilles (@v_pradeilles) – Worldline

Reminder

KeyPaths

KeyPaths were introduced with Swift 4

KeyPaths

KeyPaths were introduced with Swift 4

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

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
```

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"
```

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
```

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

How can they be efficiently leveraged?

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 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]
        })
    }
}
```

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…

Use an operator!

Use an operator!

```
prefix operator ^
```

Use an operator!

```
prefix operator ^

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

    return { element in element[keyPath: keyPath] }

}
```

Use an operator!

```
prefix operator ^

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

    return { element in element[keyPath: keyPath] }

}

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

Let’s take it one step further: DSLs

Predicates

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

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

Predicates

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

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

Predicates

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

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

To do so, we need ways to:

Predicates

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

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

To do so, we need ways to:

- Define what predicates are

Predicates

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

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

To do so, we need ways to:

- Define what predicates are
- Construct them

Predicates

We want to be able to write complex expressions, 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

Predicates

We want to be able to write complex expressions, 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)
```

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 })
}
```

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)
```

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!

That’s all!

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

That’s all!

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

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

That’s all!

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

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

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

That’s all!

```
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)
```

Recap

Recap

Recap

KeyPaths offer deferred reading/writing to a property

Recap

KeyPaths offer deferred reading/writing to a property

They allow for a very clean and expressive syntax

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

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

Conclusion

Conclusion

Conclusion

KeyPaths are great, use them!

Conclusion

KeyPaths are great, use them!

If you liked the predicates, go watch: https://vimeo.com/290272240

Conclusion

KeyPaths are great, use them!

If you liked the predicates, go watch: https://vimeo.com/290272240

Big thank you to Jérôme Alves for his valuable input.

Conclusion

KeyPaths are great, use them!

If you liked the predicates, go watch: https://vimeo.com/290272240

Big thank you to Jérôme Alves for his valuable input.

You liked what you saw, and you want it in your app?

Conclusion

KeyPaths are great, use them!

If you liked the predicates, go watch: https://vimeo.com/290272240

Big thank you to Jérôme Alves for his valuable input.

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