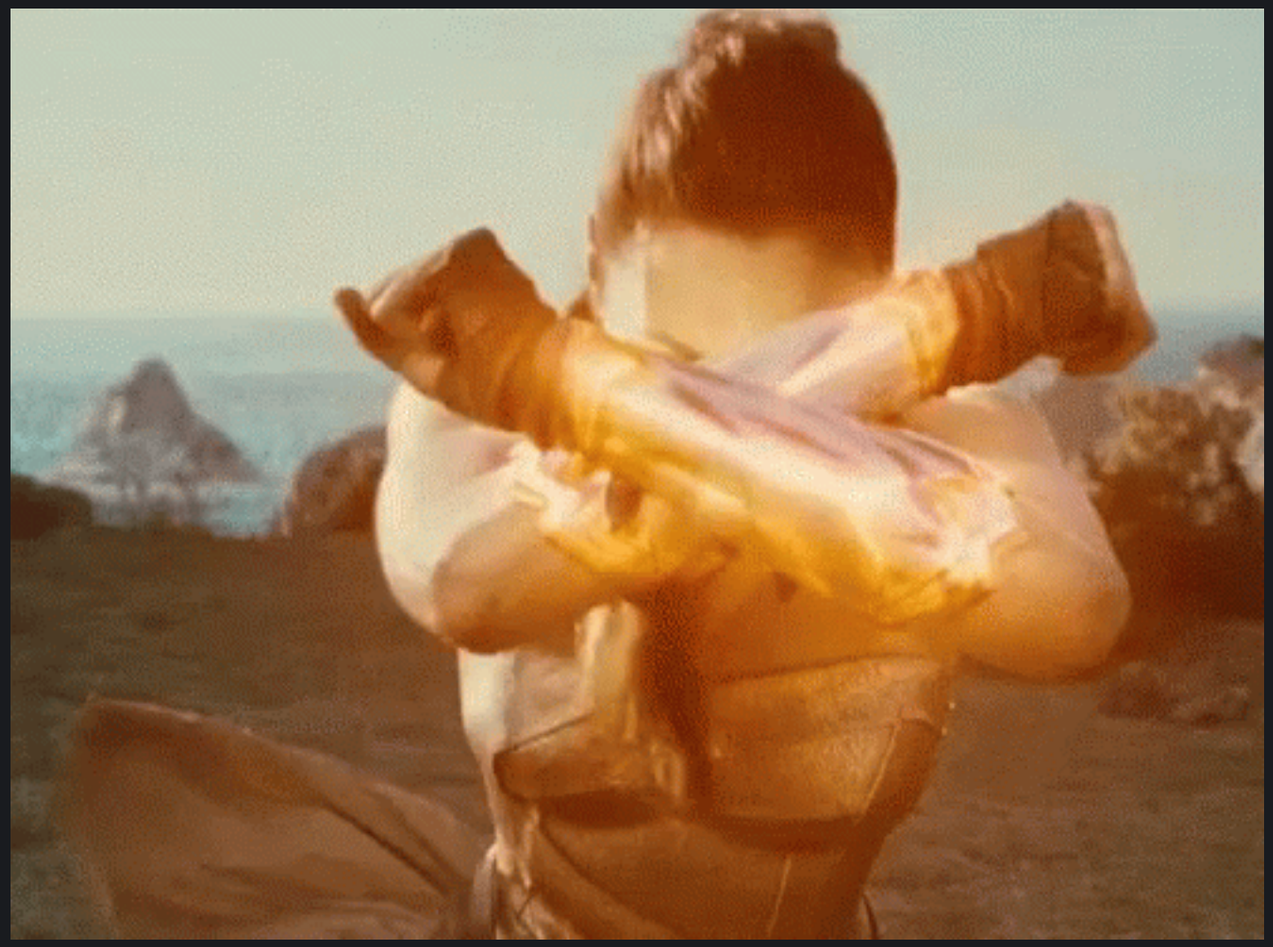
Pain Reliever

Pain Reliever



Pain Reliever

Pain Reliever

Pain Reliever

Pain Reliever

Don't mess with recruiters

Atomic

extension NSLocking

Atomic

```
func `do`<Result>(
    action: () -> Result
)
    -> Result
{
    self.lock()
    defer { self.unlock() }

    return action()
}
```

```
protocol Lockable: NSLocking { }
extension NSLock: Lockable { }
extension NSRecursiveLock: Lockable { }
```


```
typealias Value = Wrapped
typealias Observer = ((old: Value, new: Value)) -> ()
```

```
func modify<Result>(
    _ action: (inout Value) -> Result
)
    -> Result
{
    return self.lock.do {
        let oldValue = self.mutableValue
        defer {
            self.observer?((oldValue, self.mutableValue))
        }

        return action(&self.mutableValue)
    }
}
```


```
func transform<Result>(
    _ action: (Value) -> Result
)
    -> Result
{
    return self.lock.do {
        action(self.mutableValue)
    }
}
```

```
static func recursive(_ value: Value, observer: Observer? = nil)
    -> Atomic<Value>
```

```
static func lock(_ value: Value, observer: Observer? = nil)
    -> Atomic<Value>
```


```
var value: Value {
    get { return self.transform { $0 } }
    set { self.modify { $0 = newValue } }
}
```

Atomic

```
struct ID {

    let value: Int

    init(_ value: Int) {
        self.value = value
    }

    init?(string: String) {
        guard
            let value = Int(string)
        else {
            return nil
        }

        self.init(value)
    }
}
```


Atomic

```
extension ID: Comparable
extension ID: Hashable
extension ID: CustomStringConvertible
extension ID: ExpressibleByIntegerLiteral
```


Atomic

```
extension ID {
    typealias Provider = () -> ID
}
```

Atomic

```
func autoincrementedID(
    _ start: Int,
    didSet: ((_ newValue: Int) -> ())? = nil
)
    -> ID.Provider
{
    let value = Atomic.lock(start)

    return {
        value.modify {
            let result = $0
            $0 += 1
            didSet?($0)

            return ID(result)
        }
    }
}
```


Atomic

```
func autoincrementedID(start: Int) -> ID.Provider {
    return autoincrementedID(start)
}
```

Atomic

```
let provider = autoincrementedID(10)
provider() // ID(10)
provider() // ID(11)
```

Atomic

```
extension UserDefaults {

    func write(_ action: (UserDefaults) -> ()) {
        action(self)
        self.synchronize()
    }
}
```

Atomic

```
fileprivate let persistentProviders = Atomic.lock(
    [String: ID.Provider]()
)

func autoincrementedID(key: String) -> ID.Provider {
    return persistentProviders.modify { storage in
        storage[key] ?? call {
            let defaults = UserDefaults.standard
            let start = defaults.integer(forKey: key)
            let result = autoincrementedID(start) { id in
                defaults.write { $0.set(id, forKey: key) }
            }

            storage[key] = result

            return result
        }
    }
}
```


Atomic

```
let provider = autoincrementedID(key: "mama")
provider() // ID(0)
provider() // ID(1)
```

Atomic

Cancellable

Cancellable

```
protocol Cancellable {
    func cancel()
}
```

Cancellable

```
class CancellableProperty<Value: Cancellable> {

    public var value: Value? {
        willSet { value?.cancel() }
    }
}
```


Cancellable

```
struct Request: Cancellable

struct Service {
    let request = CancellableProperty<Request>()
}

service.request.value = Request()
```

Cancellable

```
infix operator <~ : AssignmentPrecedence
prefix operator ^
postfix operator ^
```

Cancellable

```
protocol Wrappable: AnyObject {
    associatedtype Wrapped

    var value: Wrapped { get set }

    static func <~(wrapper: Self, value: Self.Wrapped)
    prefix static func ^(wrapper: Self) -> Self.Wrapped
    postfix static func ^(wrapper: Self) -> Self.Wrapped
}
```


```
static func <~(wrapper: Self, value: Self.Wrapped) {
    wrapper.value = value
}

prefix static func ^(wrapper: Self) -> Self.Wrapped {
    return wrapper.value
}

postfix static func ^(wrapper: Self) -> Self.Wrapped {
    return wrapper.value
}
```


Cancellable

```
extension CancellableProperty: Wrappable { }
service.request <~ Request()
let request = ^service.request
service.request^.do { $0.cancel() }
```

Cancellable

```
extension Wrappable where Self.Wrapped: Wrappable {

    static func <~(wrapper: Self, value: Wrapped.Wrapped) {
        wrapper.value <~ value
    }

    prefix static func ^(wrapper: Self) -> Wrapped.Wrapped {
        return ^wrapper.value
    }

    postfix static func ^(wrapper: Self) -> Wrapped.Wrapped {
        return ^wrapper.value
    }
}
```

Cancellable

```
extension Atomic: Wrappable { }
```

```
let request = Atomic.lock(CancellableProperty<Request>())
request <~ Request()
```


Cancellable

Conventions

Conventions

```
enum Image { }

extension Image {
    enum Login: String, ImageRepresentable {
        case logo = "nope.jpg"
        case image
    }
}
```

Conventions

```
 // nope.jpg
imageView.image = Image.Login.logo.opaque
```

```
 // login_image.jpg
imageView.image = Image.Login.image.sized(.full)
```

Conventions

```
func identity<T>(_ value: T) -> T {
    return value
}

extension Sequence {
    var array: [Element] {
        return self.map(identity)
    }
}
```

Conventions

```
extension Sequence where Element: Hashable {
    var set: Set<Element>
    var uniqueArray: [Element]
}
```


Conventions

```
extension Sequence where SubSequence: Sequence {

    typealias Neighbours = (
        current: SubSequence.Element,
        previous: Element
    )

    func neighbourElements() -> [Neighbours] {
        return zip(self.dropFirst(), self).array
    }
}
```

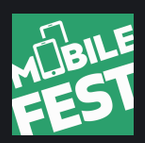
```
func split(by comparator: Character.Comparator) -> [String] {
    let characterIndices = self
        .filter(comparator)
        .flatMap(self.index)

    let border = [self.startIndex, self.endIndex]
    let indices = border + characterIndices

    return indices
        .uniqueArray
        .sorted()
        .neighbourElements()
        .map { String(self[$0.previous..<$0.current]) }
}
```


Conventions

```
extension Character {

    typealias Comparator = (Character) -> Bool

    func isUppercase() -> Bool {
        let string = String(self)

        return string == string.uppercased()
    }
}
```

Conventions

```
"mamaPapaDedaBaba".split { $0.isUppercase() }
```

Conventions

```
infix operator §: ApplicationPrecedence

func § <Value, Result>(
    function: (Value) -> Result,
    value: Value
)
    -> Result
{
    return function(value)
}
```

Conventions

```
prefix operator ∑->

prefix func ∑-> <Type, Result>(
    _ function: @escaping (Type) -> () -> Result
)
    -> (Type)
    -> Result
{
    return { function § $0 § () }
}
```

Conventions

```
func • <A, B, C>(
    lhs: @escaping (A) -> B,
    rhs: @escaping (B) -> C
)
    -> (A)
    -> C
{
    return { rhs(lhs § $0) }
}
```

Conventions

```
["mama", "papa"].map(
    ∑->String.uppercased • ∑->String.dropFirst • String.init
)
```


Conventions

```
infix operator §-> : CompactPrecedence

func §-> <Type, Arguments, Result>(
    _ function: @escaping (Type) -> (Arguments) -> Result,
    arguments: Arguments
)
    -> (Type)
    -> Result
{
    return { function § $0 § arguments }
}
```

Conventions

```
protocol ImageRepresentable {
    var opaque: UIImage? { get }
    func sized(_ size: Image.Size) -> UIImage?
}
```


Conventions

```
extension ImageRepresentable
    where
    Self: RawRepresentable,
    Self.RawValue == String
{

    var opaque: UIImage? {
        return UIImage(named: self.name)
    }

    func sized(_ size: Image.Size) -> UIImage? {
        return self.opaque.flatMap(UIImage.resize §-> size)
    }
}
```


Conventions

```
extension ImageRepresentable {

    private var name: String {
        return self.shouldOverrideConvention
            ? self.conventionName
            : self.rawValue
    }

    private var conventionName: String {
        return [typeString § self, self.rawValue]
            .flatMap(String.split §-> ∑->Character.isUppercase)
            .map(∑->String.lowercased)
            .joined(separator: "_")
    }

    private var shouldOverrideConvention: Bool {
        return "\(self)" == self.rawValue
    }
}
```


Conventions

```
extension CGRect {

    var x: CGFloat
    var y: CGFloat
    var width: CGFloat
    var height: CGFloat
}

extension CGSize {

    var min: CGFloat
    func scaled(by scale: CGFloat) -> CGSize
}
```

Conventions

```
extension Image {
    enum Size: String {
        case thumb
        case small
        case medium
        case large
        case full
    }
}
```

```
static var all: [Image.Size] {
    return [.thumb, .small, .medium, .large, .full]
}
```

```
extension Image.Size

var dimension: CGFloat? {
    let screen = UIScreen.main
    let dimension = screen.bounds.size.min * screen.scale

    switch self {
    case .thumb: return dimension / 8
    case .small: return dimension / 4
    case .medium: return dimension / 2
    case .large: return dimension
    case .full: return nil
    }
}

func multiplier(for size: CGSize) -> CGFloat {
    return self.dimension.map { $0 / size.min } ?? 1
}

func size(for size: CGSize) -> CGSize {
    return size.scaled § self.multiplier(for: size)
}
```

Conventions

```
label.text = NSLocalizedString(
    "label_content_identifier",
    comment: "localized"
)

extension String {
    var localized: String {
        return NSLocalizedString(self, comment: "")
    }
}

label.text = "label_content_identifier".localized
```

Conventions

```
enum Strings: String, LocalizedStringConvertible {
    case ok
    case cancel

    enum Login: String, LocalizedStringConvertible {
        case login = "signin"
    }
}
```

Conventions

```
protocol RawStringRepresentable: CustomStringConvertible { }

extension RawStringRepresentable
    where
    Self: RawRepresentable,
    Self.RawValue: CustomStringConvertible
{
    var description: String {
        return self.rawValue.description
    }
}
```


Conventions

```
protocol LocalizedStringConvertible: RawStringRepresentable {
    var localizedDescription: String { get }
    var localizationIdentifier: String { get }
}

extension LocalizedStringConvertible {

    var localizedDescription: String {
        return NSLocalizedString(
            self.localizationIdentifier,
            comment: ""
        )
    }

    var localizationIdentifier: String {
        return "\(typeString § self).\(self)".lowercased()
    }
}
```


Conventions

```
label.text = Strings.cancel.localizedDescription
```


Conventions

Thanks!

https://www.idapgroup.com/blog

https://github.com/trimmurrti/Talks