any IDEA HOW TO USE some GENERICS?

Antoine van der Lee - Staff iOS Engineer at WeTransfer, Founder of SwiftLee

FrenchKit, September 29th 2022, Paris, France

4 of 4

Congrats on your first PaperCall submission!

Inbox ×

support@papercall.io
to me

Fri, 8 Jun 2018, 23:34

Just wanted to say congrats on your first talk submission!

But don't stop now, typically speakers have a 1 in 10 chance of getting accepted to speak at events. Find more events at papercall.io/cfps or subscribe to TheWeeklyCFP for all open CFPs.

Talk Info:

You submitted to FrenchKit 2018 with your talk Developer Tools to Get Things Done.

Talk Description:

Saving time and reducing stress go hand in hand. Antoine shares highly valuable insights from his own work at WeTransfer as well as the development community and reveals tools, workflows and best practices for any developer, from starters to seasoned veterans.

View all of your submissions

Good luck!


WWDC22

WWDC 2022

Swift Specific sessions

- Embrace Swift Generics
- Design protocol interfaces in Swift

WWDC 2022

Session related proposals

[SE 244] Opaque Result Types

[SE 309] Unlock existentials for all protocols

[SE 335] Introduce existential any

[SE 341] Opaque Parameter Declarations

[SE 346] Lightweight same-type requirements for primary associated types

[SE 347] Type inference from default expressions

[SE 352] Implicitly Opened Existentials

[SE 353] Constrained Existential Types

[SE 358] Primary Associated Types in the Standard Library

[SE 360] Opaque result types with limited availability

WWDC 2022

Session related proposals

[SE 244] Opaque Result Types

[SE 309] Unlock existentials for all protocols

[SE 335] Introduce existential any

[SE 341] Opaque Parameter Declarations

[SE 346] Lightweight same-type requirements for primary associated types

[SE 347] Type inference from default expressions

[SE 352] Implicitly Opened Existentials

[SE 353] Constrained Existential Types

[SE 358] Primary Associated Types in the Standard Library

[SE 360] Opaque result types with limited availability

Turn off the GenericSignatureBuilder #42113

Code

Merged **slavapestov** merged 1 commit into `apple:main` from `slavapestov:gsb-off` on 2 Apr

Conversation 3 | Commits 1 | Checks 0 | Files changed 2 | +2 −7

slavapestov commented on 31 Mar • edited Member

The Requirement Machine has been running in 'verify' mode for a while, where we run both the GenericSignatureBuilder and Requirement Machine minimization algorithm and compare the results, with the GenericSignatureBuilder being used to emit diagnostics.

Now, it's time to flip the flags to 'enabled' mode, where the GenericSignatureBuilder doesn't run at all, and the Requirement Machine emits diagnostics. This finally allows us to realize the correctness and performance gains from using the Requirement Machine.

See https://forums.swift.org/t/the-requirement-machine-a-new-generics-implementation-based-on-term-rewriting/55601/ for details.

Resolves rdar://88134788.

Correctness:

- https://bugs.swift.org/browse/SR-7353
- https://bugs.swift.org/browse/SR-9595
- https://bugs.swift.org/browse/SR-10532
- https://bugs.swift.org/browse/SR-10752
- https://bugs.swift.org/browse/SR-11100
- https://bugs.swift.org/browse/SR-11532
- https://bugs.swift.org/browse/SR-11997
- https://bugs.swift.org/browse/SR-12120
- https://bugs.swift.org/browse/SR-12736
- https://bugs.swift.org/browse/SR-12980
- https://bugs.swift.org/browse/SR-13018
- https://bugs.swift.org/browse/SR-13491
- https://bugs.swift.org/browse/SR-13502
- https://bugs.swift.org/browse/SR-14484

Reviewers
No reviews

Assignees
No one assigned

Labels
None yet

Projects
None yet

Milestone
No milestone

Development
Successfully merging this pull request may close these issues.
None yet

Notifications Customize
Subscribe
You're not receiving notifications from this thread.

2 participants

Generics, Protocols, Opaque Types, Existentials

“Make generics work naturally, the way people expect it to work”

Ben Cohen in “Swift by Sundell Podcast E117”

Manager of the Swift team at Apple

Introduction

The "Complete Generics" goal for Swift 3 has been fairly ill-defined thus far, with just this short blurb in the list of goals:

> *Complete generics*: Generics are used pervasively in a number of Swift libraries, especially the standard library. However, there are a number of generics features the standard library requires to fully realize its vision, including recursive protocol constraints, the ability to make a constrained extension conform to a new protocol (i.e., an array of Equatable elements is Equatable), and so on. Swift 3.0 should provide those generics features needed by the standard library, because they affect the standard library's ABI.

This message expands upon the notion of "completing generics". It is not a plan for Swift 3, nor an official core team communication, but it collects the results of numerous discussions among the core team and Swift developers, both of the compiler and the standard library. I hope to achieve several things:

- **Communicate a vision for Swift generics**, building on the original generics design document, so we have something concrete and comprehensive to discuss.
- **Establish some terminology** that the Swift developers have been using for these features, so our discussions can be more productive ("oh, you're proposing what we refer to as 'conditional conformances'; go look over at this thread").
- **Engage more of the community in discussions** of specific generics features, so we can coalesce around designs for public review. And maybe even get some of them implemented.

A message like this can easily turn into a centithread. To separate concerns in our discussion, I ask that replies to this specific thread be limited to discussions of the vision as a whole: how the pieces fit together, what pieces are missing, whether this is the right long-term vision for Swift, and so on. For discussions of specific language features, e.g., to work out the syntax and semantics of conditional conformances or discuss the implementation in compiler or use in the standard library, please start a new thread based on the feature names I'm using.

This message covers a lot of ground; I've attempted a rough categorization of the various features, and kept the descriptions brief to limit the overall length. Most of these aren't my ideas, and any syntax I'm providing is simply a way to express these ideas in code and is subject to change. Not all of these features will happen, either soon or ever, but they are intended to be a fairly complete whole that should mesh together. I've put a * next to features that I think are important in the nearer term vs. being interesting "some day". Mostly, the *'s reflect features that will have a significant impact on the Swift standard library's design and implementation.

Enough with the disclaimers; it's time to talk features.

Removing unnecessary restrictions

There are a number of restrictions to the use of generics that fall out of the implementation in the Swift compiler. Removal of these restrictions is a matter of implementation only; one need not introduce new syntax or semantics to realize them. I'm listing them for two reasons: first, it's an acknowledgment that these features are intended to exist in the model we have today, and, second, we'd love help with the implementation of these features.

Generics Manifesto https://github.com/apple/swift/blob/main/docs/GenericsManifesto.md

---

Removing unnecessary restrictions

Generics and protocol extensions

Swift 2 generics improvements

1.2

```
protocol Content {
    var id: UUID { get }
    var url: URL { get }

    func makeFavorite()
}
```

Generics and protocol extensions

Swift 2 generics improvements

1.2

```
protocol Content {
    var id: UUID { get }
    var url: URL { get }

    func makeFavorite()
}

struct ImageContent: Content {
    func makeFavorite() {
        // ..
    }
}

struct VideoContent: Content {
    func makeFavorite() {
        // ..
    }
}
```

Generics and protocol extensions

Swift 2 generics improvements

1.2

```
protocol Content {
    var id: UUID { get }
    var url: URL { get }

    func makeFavorite()
}

struct ImageContent: Content {
    func makeFavorite() {
        // ..
    }
}

struct VideoContent: Content {
    func makeFavorite() {
        // ..
    }
}
```

Generics and protocol extensions

Swift 2 removing unnecessary generics restrictions

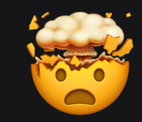
2.0

```
protocol Content {
    var id: UUID { get }
    var url: URL { get }

    func makeFavorite()
}

struct ImageContent: Content { }
struct VideoContent: Content { }

extension Content {
    func makeFavorite() {
        // ...
    }
}
```

“Swift’s goal is to make your life as a developer easier”

Angela Laar in “What’s new in Swift WWDC 2022”

Software Engineer, Swift team at Apple



Opaque types Existential types
Generics Protocols Type erasure
Associated types
Generalized Existentials
Constrained Opaque Result Types
Constrained Existential Types

LET'S DIVE IN

Generics

```
struct IntStack {
    var items: [Int] = □
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() → Int {
        return items.removeLast()
    }
}
```

```
struct Stack<Element> {
    var items: [Element] = □
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() → Element {
        return items.removeLast()
    }
}
```

```
let arrayOfStrings = ["Bernie", "Jaap", "Lady"]
```

```
let arrayOfStrings: Array<String> = ["Bernie", "Jaap", "Lady"]
```

```
let arrayOfStrings: Array<Int> = [0, 1, 2]
```

Generics and type constraints

```
func findIndex<T: Equatable>(of valueToFind: T, in array: [T]) → Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

Generics and type constraints

```
func findIndex<T: Equatable>(of valueToFind: T, in array: [T]) → Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

Generics and type constraints

```
extension Array where Element: Equatable {
    func findIndex(of valueToFind: Element) → Int? {
        for (index, value) in self.enumerated() {
            if value == valueToFind {
                return index
            }
        }
        return nil
    }
}
```

Generics and type constraints

```
let elements = [1, 2, 3]

findIndex(of: 2, in: elements) // Results in: 1
elements.findIndex(of: 2) // Results in: 1

// Default available Swift API:
elements.firstIndex(of: 2) // Results in: 1
```

Generics and type constraints

```
let elements = ["Lady", "Jaap", "Bernie"]

findIndex(of: "Bernie", in: elements) // Results in: 2
elements.findIndex(of: "Bernie") // Results in: 2

// Default available Swift API:
elements.firstIndex(of: "Bernie") // Results in: 2
```

How about Opaque Types?

Opaque Types

You’re already using them today

```
var body: some View { ••• }
```

Opaque Types

You're already using them today

```
var body: some View { ... }

// Equals to:

var body: VStack { ... }

// Or:

var body: Text { ... }
```

Opaque Types

Just like with generics, the underlying type is fixed for the scope of the value.

```
func makeFooterView(isPro: Bool) → some View {
    if isPro {
        return Text("Hi the
    } else {
        return VStack {
            Text("How about becoming PRO?")
            Button("Become PRO", action: {
                // ..
            })
        }
    }
}
```

Function declares an opaque return type 'some View', but the return statements in its body do not have matching underlying types

Opaque Types

Just like with generics, the underlying type is fixed for the scope of the value.

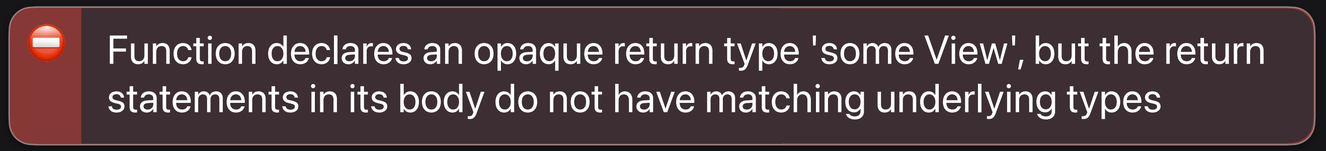
```
func makeFooterView(isPro: Bool) → some View {
    if isPro {
        return Text("Hi there, PRO!") // Return type is Text
    } else {
        return VStack { // Return type is VStack<TupleView<(Text, Button<Text>)>>
            Text("How about becoming PRO?")
            Button("Become PRO", action: {
                // ..
            })
        }
    }
}
```

⛔ Function declares an opaque return type 'some View', but the return statements in its body do not have matching underlying types

Opaque Types

Just like with generics, the underlying type is fixed for the scope of the value.

```
func makeFooterView(isPro: Bool) → some View {
    return VStack {
        if isPro {
            Text("Hi there, PRO!")
        } else {
            Text("How about becoming PRO?")
            Button("Become PRO", action: {
                // ..
            })
        }
    }
}
```



Opaque Types

Just like with generics, the underlying type is fixed for the scope of the value.

```
@ViewBuilder
func makeFooterView(isPro: Bool) → some View {
    if isPro {
        Text("Hi there, PRO!")
    } else {
        Text("How about becoming PRO?")
        Button("Become PRO", action: {
            // ..
        })
    }
}
```

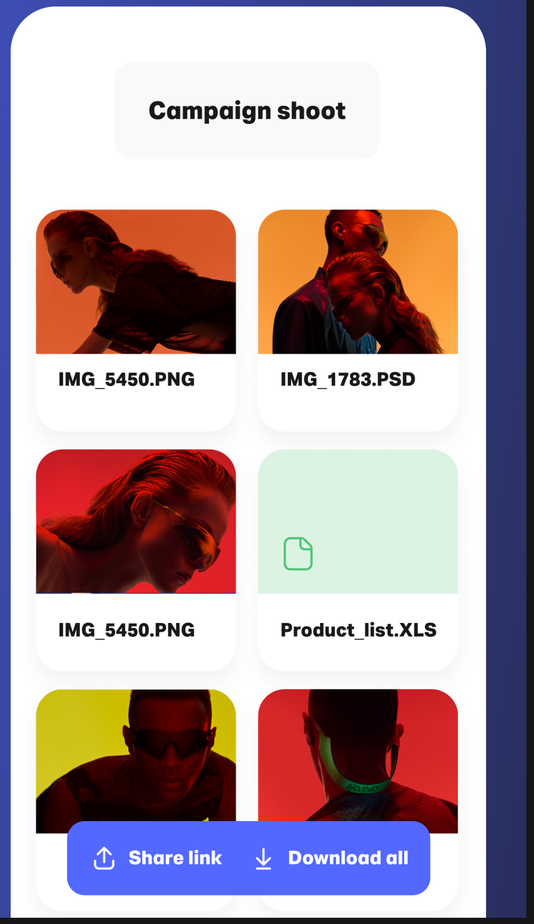
Result builders in Swift explained with code examples avanderlee.com/swift/result-builders

---

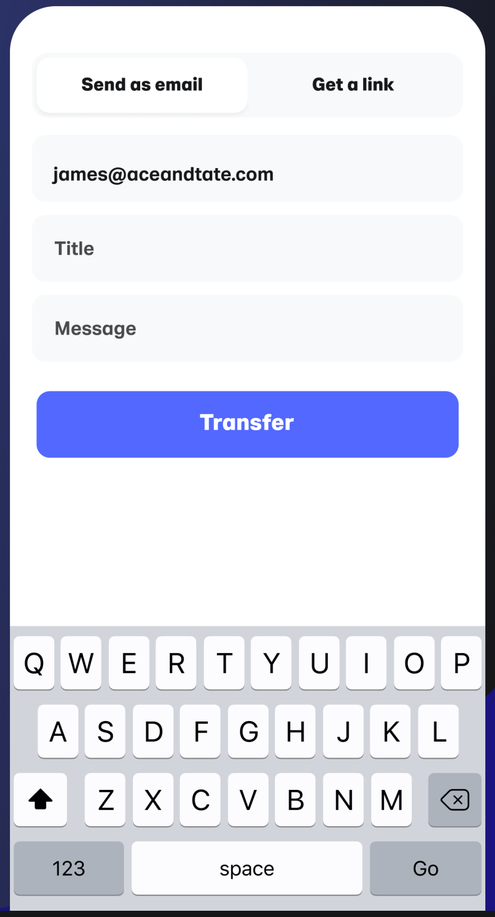
How do Opaque types relate to generics?

Send and receive files right from your phone

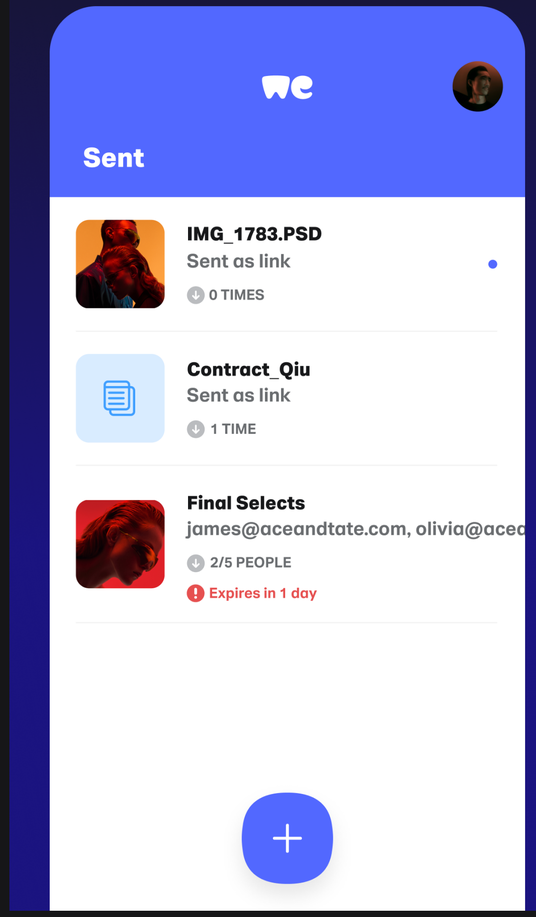
Preview and download files

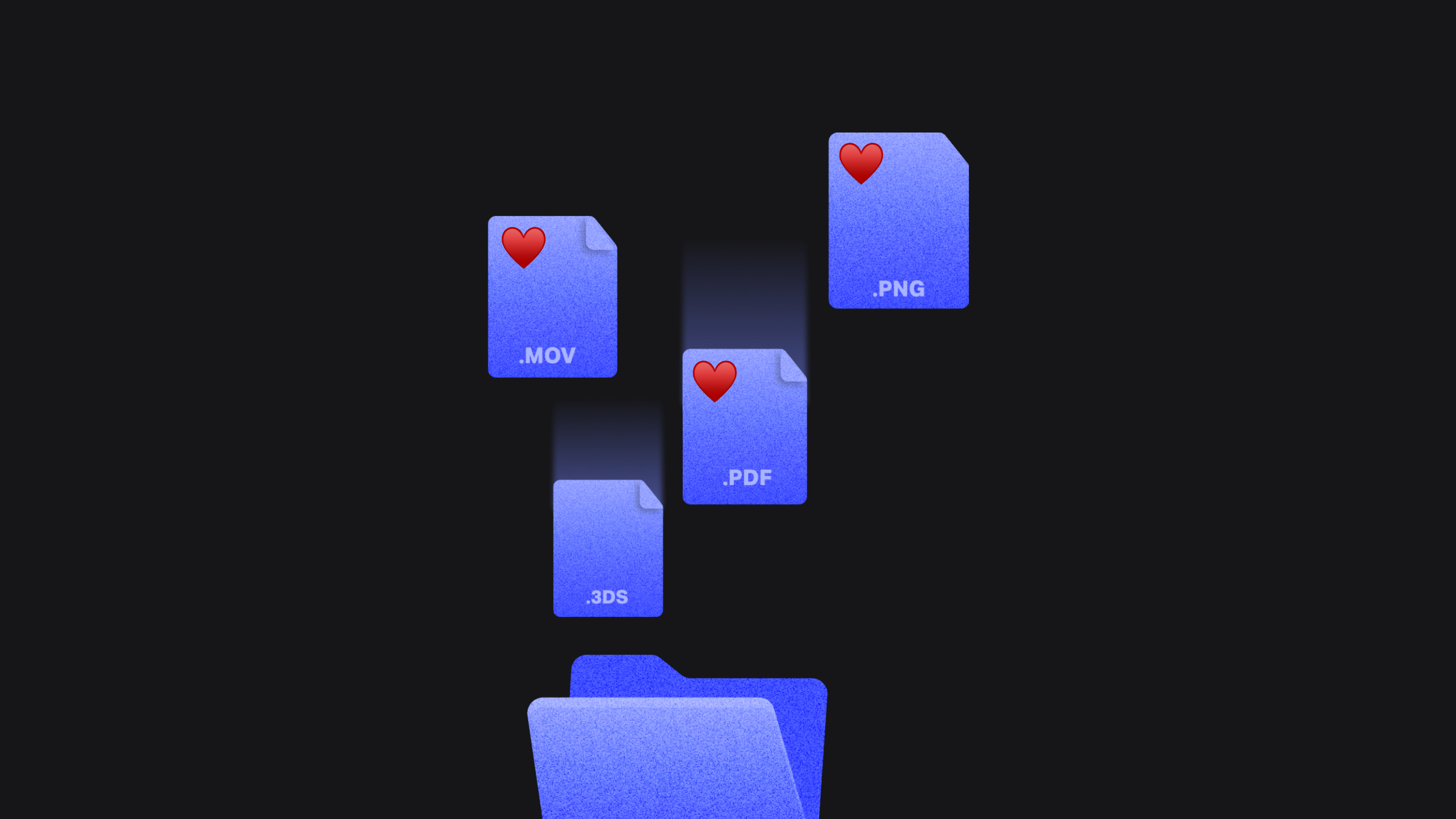


Transfer files securely with a link or email



Resend, forward, and delete transfers




.MOV
.PNG
.PDF
.3DS

```
struct ImageContent {
    let id = UUID()
    let url: URL
}
```

```swift
struct ImageContent {
    let id = UUID()
    let url: URL
}

final class FavoriteImageContentStore {

    private var favorites: [UUID] = □

    func favorite(_ contentItems: [ImageContent]) {
        for content in contentItems {
            favorites.append(content.id)
        }
    }

    func isFavorite(_ content: ImageContent) → Bool {
        return favorites.contains(content.id)
    }
}

let favoritesStore = FavoriteImageContentStore()
let imageContent = ImageContent(…)
favoritesStore.favorite([imageContent])
```

```swift
struct ImageContent {
    let id = UUID()
    let url: URL
}

struct VideoContent {
    let id = UUID()
    let url: URL
}

final class FavoriteImageContentStore {

    private var favorites: [UUID] = □

    func favorite(_ contentItems: [ImageContent]) { ••• }

    func isFavorite(_ content: ImageContent) → Bool {
        return favorites.contains(content.id)
    }
}

let favoritesStore = FavoriteImageContent
let videoContent = VideoContent(…)
favoritesStore.favorite([videoContent])
```

Cannot convert value of type 'VideoContent' to expected element type 'Array<ImageContent>.ArrayLiteralElement' (aka 'ImageContent')

```
struct ImageContent {
    let id = UUID()
    let url: URL
}

struct VideoContent {
    let id = UUID()
    let url: URL
}
```

```swift
protocol Content {
    var id: UUID { get }
    var url: URL { get }
}

struct ImageContent: Content {
    let id = UUID()
    let url: URL
}

struct VideoContent: Content {
    let id = UUID()
    let url: URL
}
```

```swift
protocol Content {
    var id: UUID { get }
    var url: URL { get }
}

struct ImageContent: Content { ··· }
struct VideoContent: Content { ··· }

final class FavoriteImageContentStore {

    private var favorites: [UUID] = □

    func favorite(_ contentItems: [ImageContent]) { ··· }

    func isFavorite(_ content: ImageContent) → Bool {
        return favorites.contains(content.id)
    }
}

let favoritesStore = FavoriteImageContent
let videoContent = VideoContent(…)
favoritesStore.favorite([videoContent])
```

Cannot convert value of type 'VideoContent' to expected element type 'Array<ImageContent>.ArrayLiteralElement' (aka 'ImageContent')

5.6

```
protocol Content {
    var id: UUID { get }
    var url: URL { get }
}

struct ImageContent: Content { ··· }
struct VideoContent: Content { ··· }

final class FavoriteContentStore {

    private var favorites: [UUID] = □

    func favorite(_ contentItems: [Content]) { ··· }

    func isFavorite(_ content: Content) → Bool {
        return favorites.contains(content.id)
    }
}

let favoritesStore = FavoriteContentStore()
let videoContent = VideoContent(…)
favoritesStore.favorite([videoContent])
```

Opaque types vs. Generics

The differences between Swift 5.6 and 5.7

```
protocol Content {
    var id: UUID { get }
    var url: URL { get }
}
```

Opaque types vs. Generics

The differences between Swift 5.6 and 5.7

```
protocol Content: Identifiable where ID == UUID {
    var url: URL { get }
}
```

Opaque types vs. Generics

The differences between Swift 5.6 and 5.7

5.6

```
protocol Content: Identifiable where ID == UUID {
    var url: URL { get }
}

extension FavoriteImageContentStore {
    func isFavorite(_ content: Content) -> Bool {
        favorites.contains(content.id)
    }
}
```

Protocol 'Content' can only be used as a generic constraint because it has Self or associated type requirements

Opaque types vs. Generics

The differences between Swift 5.6 and 5.7

5.6

```
protocol Content: Identifiable where ID == UUID {
    var url: URL { get }
}

extension FavoriteImageContentStore {
    func isFavorite<T: Content>(_ content: T) -> Bool {
        favorites.contains(content.id)
    }
}
```

Opaque types vs. Generics

The differences between Swift 5.6 and 5.7

5.6

```
protocol Content: Identifiable where ID == UUID {
    var url: URL { get }
}

extension FavoriteImageContentStore {
    func isFavorite<T>(_ content: T) -> Bool where T: Content {
        favorites.contains(content.id)
    }
}
```

Opaque types vs. Generics

The differences between Swift 5.6 and 5.7

5.7

```
protocol Content: Identifiable where ID == UUID {
    var url: URL { get }
}

extension FavoriteImageContentStore {
    func isFavorite(_ content: some Content) -> Bool {
        favorites.contains(content.id)
    }
}
```

Opaque Parameter Declarations

SE-341

Opaque types vs. Generics

The differences between Swift 5.6 and 5.7

5.7

```swift
// Swift 5.7
func isFavorite(_ content: some Content) -> Bool { }

// Swift 5.6
func isFavorite<T: Content>(_ content: T) -> Bool { }
func isFavorite<T>(_ content: T) -> Bool where T: Content { }
```

Opaque Parameter Declarations

SE-341

some Protocol *as shorthand for* T where T: Protocol

All we know is that there's going to be some value of type Content

Opaque types
In summary

Just like with generics, the underlying type is fixed for the scope of the value

If a generic parameter is only used in one place, you can now write it with the `some` keyword as a shorthand

`some Protocol` *as shorthand for* `T where T: Protocol`

Code becomes easier to read

Generics <T>, Opaque Type some,
how about existential types?

Existential any

```
final class FavoriteImageContentStore {

    private var favorites: [UUID] = □

    func favorite(_ contentItems: [ImageContent]) {
        for content in contentItems {
            favorites.append(content.id)
        }
    }

    func isFavorite(_ content: ImageContent) → Bool {
        return favorites.contains(content.id)
    }
}
```

Existential any

```
func favorite(_ contentItems: [ImageContent]) {
    for content in contentItems {
        favorites.append(content.id)
    }
}
```

Existential any

5.6

```
func favorite(_ contentItems: [Content]) {
    for content in contentItems {
        favorites.append(content.id)
    }
}
```

Protocol 'Content' can only be used as a generic constraint because it has Self or associated type requirements

Existential any

5.6

```
func favorite<T: Collection>(_ contentItems: T) where T.Element: Content {
    for content in contentItems {
        favorites.append(content.id)
    }
}
```

Existential any

5.6

```
func favorite<T: Collection>(_ contentItems: T) where T.Element: Content {
    for content in contentItems {
        favorites.append(content.id)
    }
}

let store = FavoriteContentStore()
store.favorite([
    ImageContent(…),
    ImageContent(…),
    ImageContent(…)
])
```

Existential any

5.6

```
func favorite<T: Collection>(_ contentItems: T) where T.Element: Content {
    for content in contentItems {
        favorites.append(content.id)
    }
}

let store = FavoriteContentStore()
store.favorite([
    ImageContent(…),
    VideoContent(…),
    ImageContent(…)
])
```

Cannot convert value of type 'VideoContent' to expected element type 'ImageContent'

Existential any

```
func favorite(_ contentItems: [some Content]) {
    for content in contentItems {
        favorites.append(content.id)
    }
}

let store = FavoriteContentStore()
store.favorite([
    ImageContent(…),
    VideoContent(…),
    ImageContent(…)
])
```

Cannot convert value of type 'VideoContent' to expected element type 'ImageContent'

Existential any

5.7

```
func favorite(_ contentItems: [any Content]) {
    for content in contentItems {
        favorites.append(content.id)
    }
}

let store = FavoriteContentStore()
store.favorite([
    ImageContent(…),
    VideoContent(…),
    ImageContent(…)
])
```

A collection of any kind of Content

A collection of any kind of Content

Understanding further with alternatives

```
func favorite(_ contentItems: [any Content]) {
    for content in contentItems {
        favorites.append(content.id)
    }
}
```

```
func favorite(_ contentItems: [ContentBox]) {
    for content in contentItems {
        favorites.append(content.innerValue.id)
    }
}

struct ContentBox {
    var innerValue: any Content
}
```

```swift
func favorite(_ contentItems: [any Content]) {
    for content in contentItems {
        favorites.append(content.id)
    }
}

func favorite(_ contentItems: [AnyContent]) {
    for content in contentItems {
        favorites.append(content.id)
    }
}

struct AnyContent: Content {
    let id: UUID
    let url: URL

    init<T: Content>(content: T) {
        self.id = content.id
        self.url = content.url
    }
}
```

any value, but it conforms to Content

Existential types and performance

```
var anyContent: any Content = ImageContent(…)
anyContent = VideoContent(…)
```

Existential types and performance

```
var anyContent: any Content = ImageContent(…)
anyContent = VideoContent(…)

var someContent: some Content = ImageContent(…)
someContent = VideoContent(…)
```

Cannot assign value of type 'VideoContent' to type 'some Content'

Existential any type

In summary

- `any Protocol` stands for “any value conforming to `Protocol`"
- Swift 5.7 adds support `'Self'` or associated type requirements
- Enforced when using protocol types directly to indicate performance impact (starting from Swift 6)
- Unpredictable for the compiler

Comparing some and any

some

```
let someContent: [some Content] = [
    ImageContent(…),
    ImageContent(…)
]
```

Holds a fixed concrete type

Guarantees type relationships

any

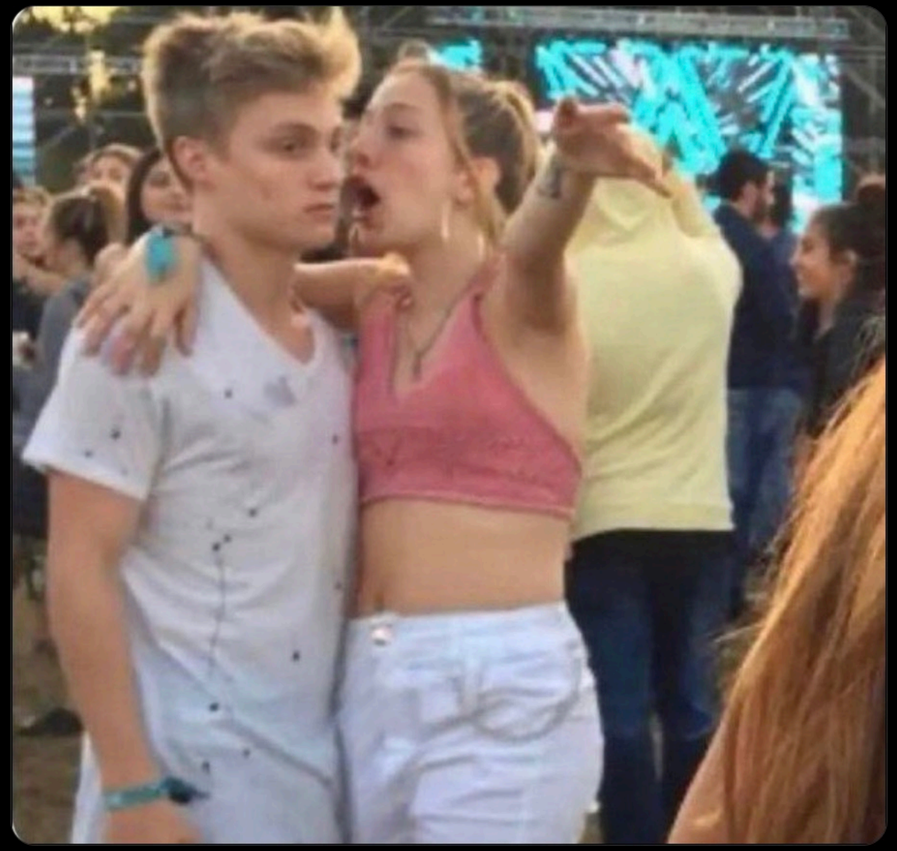
```
let anyContent: [any Content] = [
    ImageContent(…),
    VideoContent(…)
]
```

Holds an arbitrary concrete type

Erases type relationships

Ben Cohen
@AirspeedSwift

SO EVEN THOUGH A FUNCTION TAKING ANY P AND A FUNCTION TAKING SOME P ARE FUNCTIONALLY SIMILAR TO THE CALLER THEY ARE ACTUALLY DIFFERENT THINGS



4:16 PM · Aug 18, 2022 · Twitter for iPad

4 Retweets **1** Quote Tweet **63** Likes

Practical Examples

```
func printElement<T: CustomStringConvertible>(_ element: T) {
    print(element)
}
```

```
func printElement(_ element: some CustomStringConvertible) {
    print(element.description)
}
```

```
public struct RemoteImageFetcher {
    func fetchImage() → UIImage {
        ...
    }
}

public struct ImageFetcherFactory {
    public func imageFetcher(for url: URL) → RemoteImageFetcher {
        ...
    }
}
```

```
public protocol ImageFetching {
    func fetchImage() → UIImage
}

public struct RemoteImageFetcher {
    func fetchImage() → UIImage {
        ...
    }
}

public struct ImageFetcherFactory {
    public func imageFetcher(for url: URL) → RemoteImageFetcher {
        ...
    }
}
```

```
public protocol ImageFetching {
    func fetchImage() → UIImage
}

struct RemoteImageFetcher: ImageFetching {
    func fetchImage() → UIImage {
        ...
    }
}

public struct ImageFetcherFactory {
    public func imageFetcher(for url: URL) → ImageFetching {
        ...
    }
}
```

```swift
public protocol ImageFetching {
    associatedtype Image
    func fetchImage() -> Image
}

struct RemoteImageFetcher: ImageFetching {
    func fetchImage() -> UIImage {
        ...
    }
}

public struct ImageFetcherFactory {
    public func imageFetcher(for url: URL) -> ImageFetching {
        ...
    }
}
```

⛔ Use of protocol 'ImageFetching' as a type must be written 'any ImageFetching'

```swift
public protocol ImageFetching {
    associatedtype Image
    func fetchImage() → Image
}

struct RemoteImageFetcher: ImageFetching {
    func fetchImage() → UIImage {
        ...
    }
}

public struct ImageFetcherFactory {
    public func imageFetcher(for url: URL) → any ImageFetching {
        ...
    }
}
```

```
public func imageFetcher(for url: URL) -> any ImageFetching {
    if url.isFileURL {
        return LocalImageFetcher(url: url)
    } else {
        return RemoteImageFetcher(url: url)
    }
}
```

```
public func imageFetcher(for url: URL) → some ImageFetching {
    return RemoteImageFetcher(url: url)
}
```

```swift
public protocol ImageFetching {
    associatedtype Image
    func fetchImage() → Image
}

public extension UIImageView {
    func configureImage(with imageFetcher: any ImageFetching) {
        image = imageFetcher.fetchImage()
    }
}
```

Cannot assign value of type 'Any' to type 'UIImage'

```swift
public protocol ImageFetching<Image> {
    associatedtype Image
    func fetchImage() → Image
}

public extension UIImageView {
    func configureImage(with imageFetcher: any ImageFetching) {
        image = imageFetcher.fetchImage()
    }
}
```

Cannot assign value of type 'Any' to type 'UIImage'

```
public protocol ImageFetching<Image> {
    associatedtype Image
    func fetchImage() → Image
}

public extension UIImageView {
    func configureImage(with imageFetcher: any ImageFetching<UIImage>) {
        image = imageFetcher.fetchImage()
    }
}
```

```swift
public protocol ImageFetching<Image> {
    associatedtype Image
    func fetchImage() → Image
}

public extension UIImageView {
    func configureImage(with imageFetcher: some ImageFetching<UIImage>) {
        image = imageFetcher.fetchImage()
    }
}
```

Lightweight same-type requirements for primary associated types SE-346

Do we still need AnyView or can we use any View now?

- @ViewBuilder and generics should solve most cases of AnyView
- Existential any is just defining an existential container box
- You can’t instantiate existentials
  any View(...) is not possible while AnyView(...) is

Does any relate to Any or AnyObject?

AnyObject, Any, and any: When to use which? https://www.avanderlee.com/swift/anyobject-any

Wrap up

When to use some, any, or generics?

- Start with some if a generic parameter is only used in one place
- Change some to any when you know you need to store arbitrary (random) values
- Use generics if you have multiple type constraints

Opaque types, existential types, generics, protocols, type erasure, associated types, Generalized Existentials, constrained Opaque Result Types, Constrained Existential Types

Still feeling overwhelmed?

Swift Jun 21, 2022 • 6 min read

Some keyword in Swift: Opaque types explained with code examples

The some keyword in Swift declares opaque types, and Swift 5.1 introduced it with support for opaque result types. Many engineers experience working with opaque types for the first time when writing a body of a SwiftUI view. Though, it's often unclear what some keyword does and when to use them in other places.

With the introduction of Opaque Parameter Declarations in SE-0341, there are many more places where you can start adopting the some keyword. In this article, I'll explain what opaque types are and when you should use them.

In this article

SwiftLee Jobs

BRAND NEW UPDATE SOON
GET IN TOUCH IF YOU'RE OPEN FOR JOB POSITIONS!

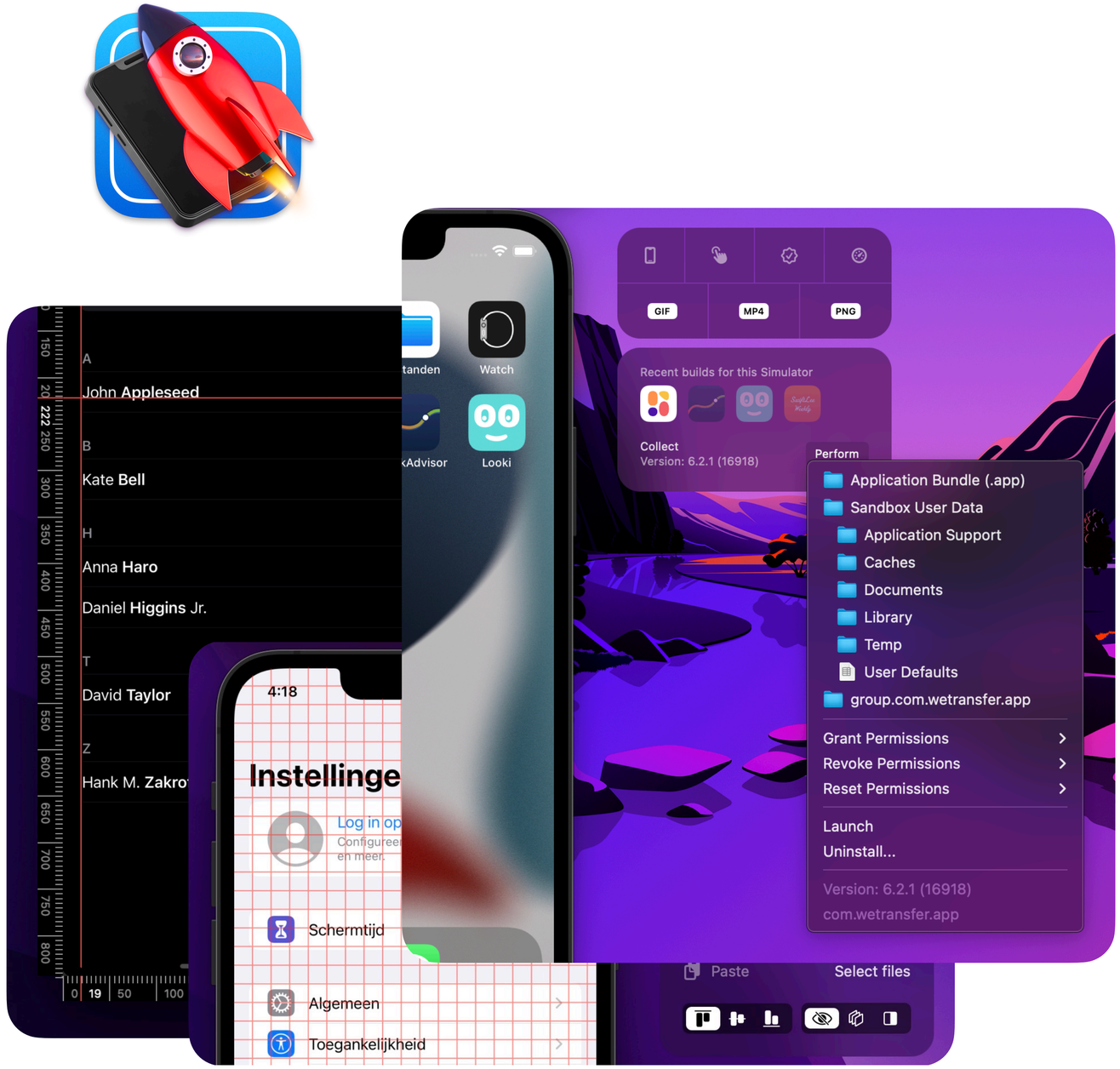
swiftleejobs.com

SwiftLee

avanderlee.com

SwiftLee Weekly

avanderlee.com/swiftlee-weekly


A
John Appleseed
B
Kate Bell
H
Anna Haro
Daniel Higgins Jr.
T
David Taylor
Z
Hank M. Zakro
Watch
Looki
GIF
MP4
PNG
Recent builds for this Simulator
Collect
Version: 6.2.1 (16918)
Perform
Application Bundle (.app)
Sandbox User Data
Application Support
Caches
Documents
Library
Temp
User Defaults
group.com.wetransfer.app
Grant Permissions
Revoke Permissions
Reset Permissions
Launch
Uninstall...
Version: 6.2.1 (16918)
com.wetransfer.app
4:18
Instellinge
Schermtijd
Algemeen
Toegankelijkheid
Paste
Select files

SwiftLee

THANKS

@TWANNL
AVANDERLEE.COM