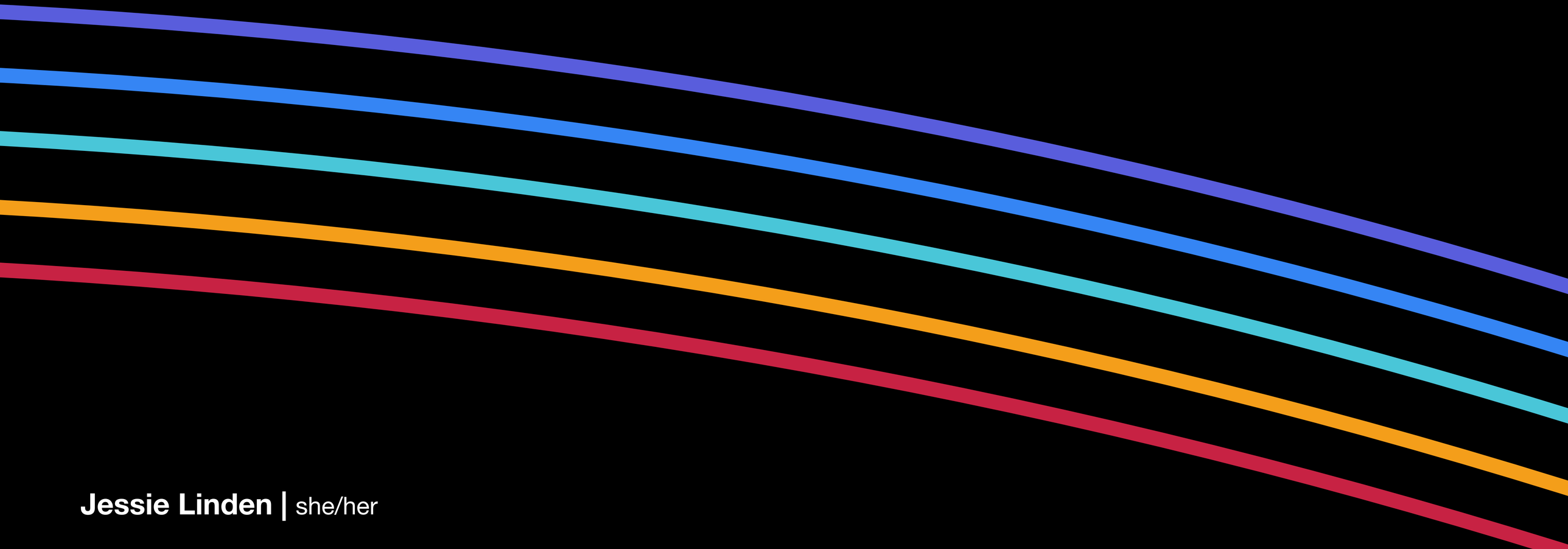
ICYMI: Enums Are The Sh*t

Swift Leeds 2023



Jessie Linden | she/her

Why enums?

“Depending on who you ask, enums are either under-utilized, or over-used in Swift.”

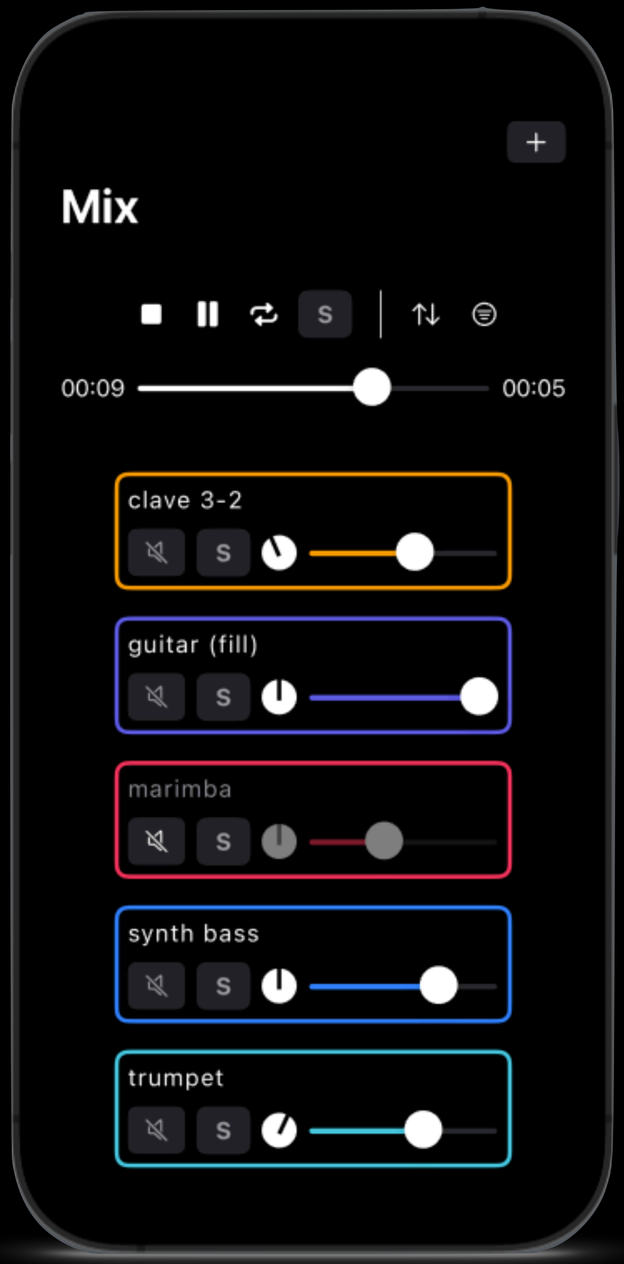
John Sundell

over-used

---

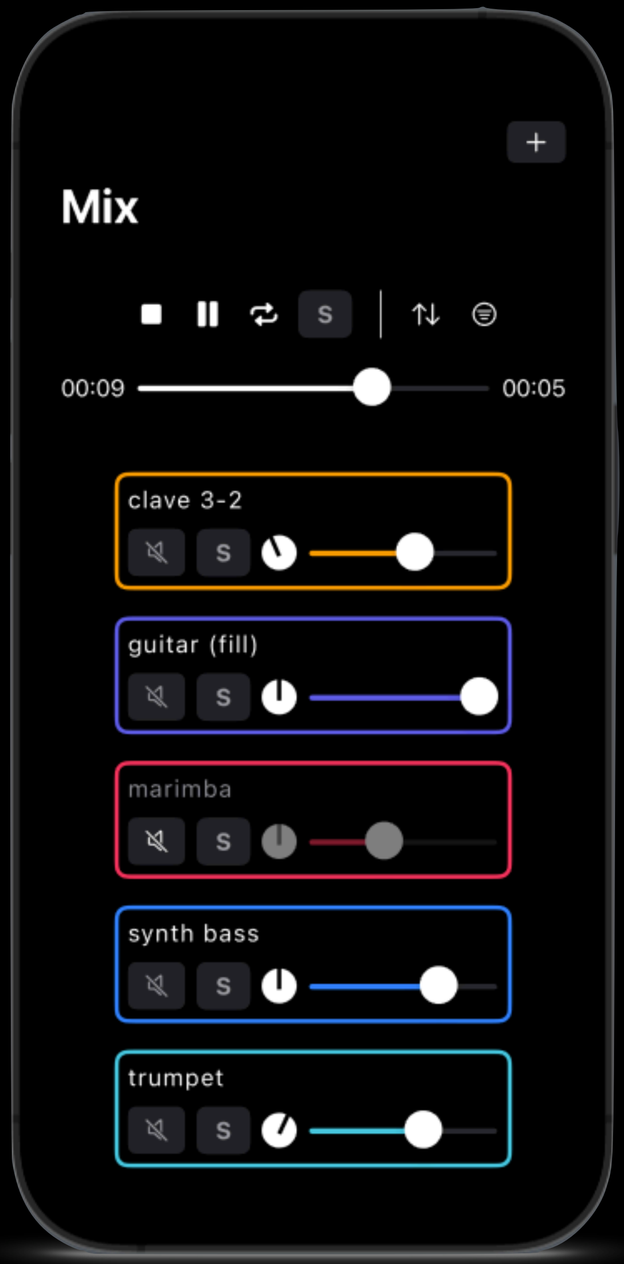
under-utilized

PocketPerc


Mix
S
00:09
00:05
clave 3-2
S
guitar (fill)
S
marimba
S
synth bass
S
trumpet
S

PocketPerc

1	2	3	4 fill



Category TrackControls SortStyle

Fill SheetState PlaybackControls

Mallets ViewState Rhythm Mix

SoloMode SelectionState SheetType

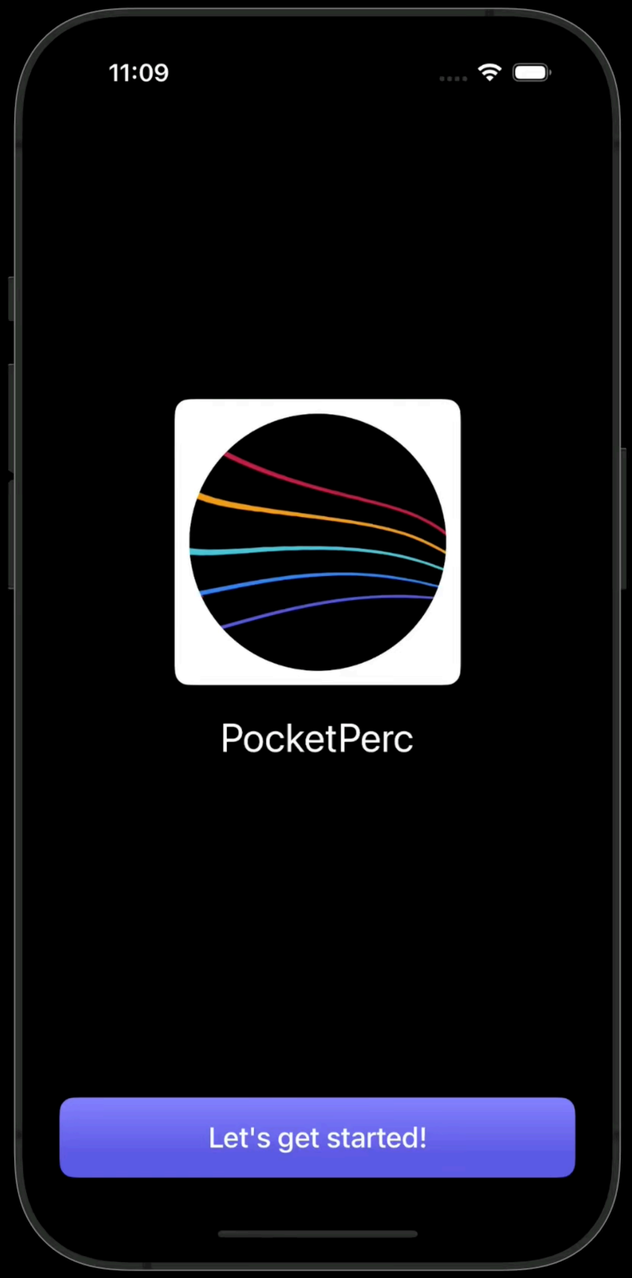
Horns Track Latin RepeatMode

TimeConstant PlaybackState

20

PocketPerc

- app demo
- code
- make a mix




11:09
PocketPerc
Let's get started!

The Basics

The Basics

category

finite list of options

```
let sortStyles = ["alphabetical", "category"]
@State private var sortStyle = "alphabetical"
```

The Basics

```
let sortStyles = ["alphabetical", "category"]
@State private var sortStyle = "alphabetical"
```

```
enum SortStyle: String {
    case alphabetical, category
}

@State private var sortStyle: SortStyle = .alphabetical
```

Raw Type

```
enum SortStyle: String {
    case alphabetical, category
}

@State private var sortStyle: SortStyle = .alphabetical
```

Raw Type
with Associated Values

```
enum SortStyle: String {
    case alphabetical, category
}

@State private var sortStyle: SortStyle =
```

```
enum SortStyle {
    case alphabetical
    case category
    case dateAdded(ascending: Bool = true)
}

extension SortStyle: RawRepresentable { ••• }
```

Associated Values

```
case dateAdded(ascending: Bool = true)
```

Associated Values

```
case dateAdded(ascending: Bool = true)
case dateAdded(ascending: Bool)
```

Associated Values

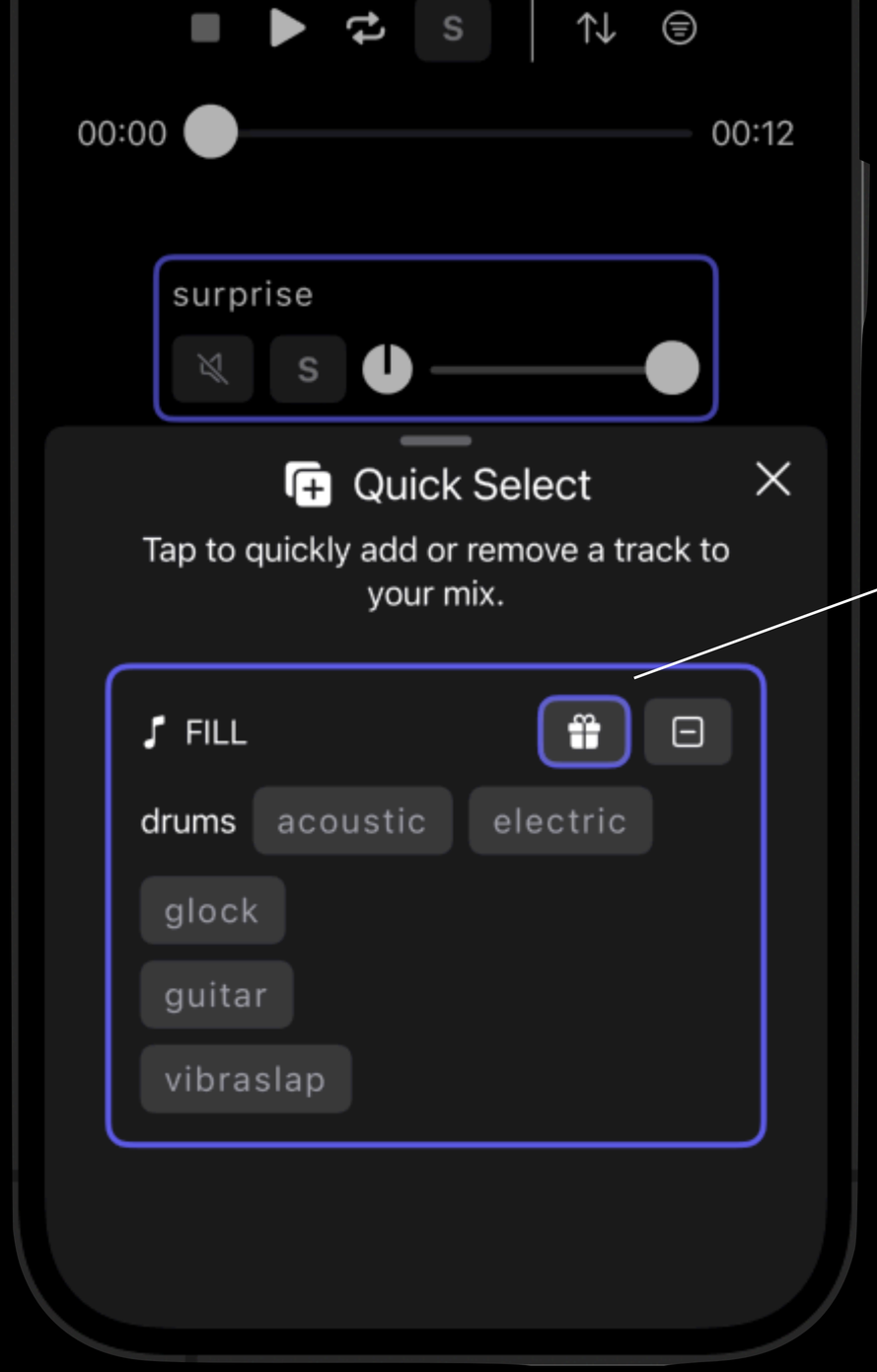
```
case dateAdded(ascending: Bool = true)
case dateAdded(ascending: Bool)
case dateAdded(Bool)
```

Associated Values

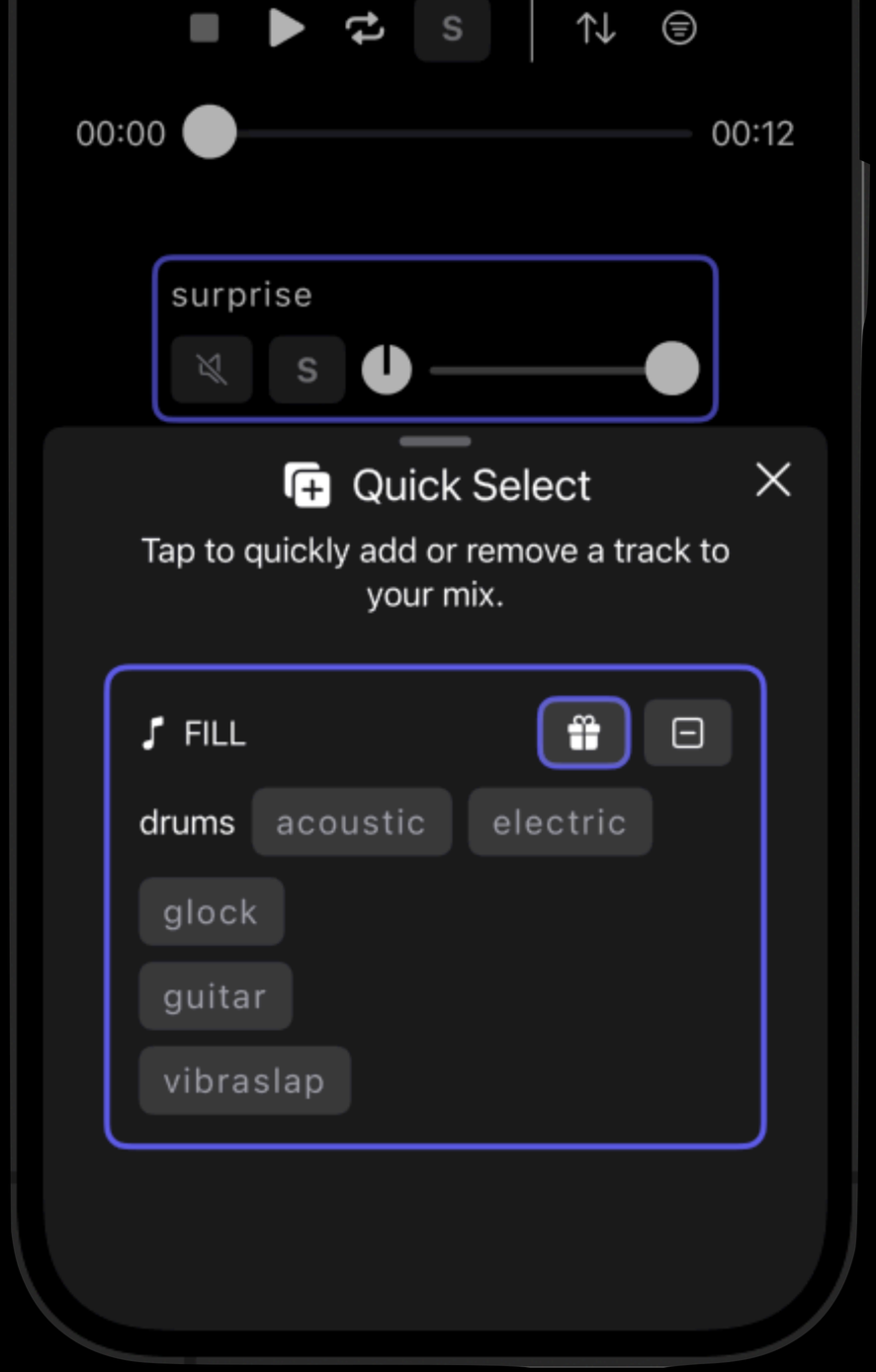
```
case dateAdded(ascending: Bool = true)
case dateAdded(ascending: Bool)
case dateAdded(Bool)
case dateAdded(T)
```

Associated Values

```
case dateAdded(ascending: Bool = true)
case dateAdded(ascending: Bool)
case dateAdded(Bool)
case dateAdded(T)
case dateAdded(Direction)
```



surprise fill!




```
enum Fill {
    case drums(DrumFillType)
    case glock
    case guitar
    case vibraslap
    indirect case surprise(Fill)
}
```

Pattern Matching With Associated Values

```
var systemName: String {
    switch self {
    case .alphabetical: "A.circle"
    case .category: "l"
    case .dateAdded(let direction):
        direction == .ascending ? "arrow.up" : "arrow.down"
    }
}
```

Pattern Matching With Associated Values

```
case .dateAdded(let direction)
    where direction == .ascending: "arrow.up"
```

Pattern Matching With Associated Values

```
case .dateAdded: "calendar"
```

Pattern Matching With Associated Values

```
if case .dateAdded(let direction) = sortStyle {

}
```

```
guard case .dateAdded(let direction) = sortStyle else { return }
```

Protocol Conformance

```
enum SortStyle: CaseIterable, Identifiable, Equatable, Comparable, Hashable {
    case alphabetical
    case category
    case dateAdded(Direction)

    enum Direction: String {
        case ascending, descending
    }

    var id: String { rawValue } // Identifiable

    static var allCases: [SortStyle] = [ ••• ] // CaseIterable

    static func <(lhs: SortStyle, rhs: SortStyle) -> Bool { ••• } // Comparable
}
```

Nested structures

Nested structures

Custom initializers

Nested structures

Custom initializers

Functions

Nested structures

Custom initializers

Functions

Mutating functions

Nested structures

Custom initializers

Functions

Mutating functions

Static functions

Nested structures

Custom initializers

Functions

Mutating functions

Static functions

Static properties

Nested structures

Custom initializers

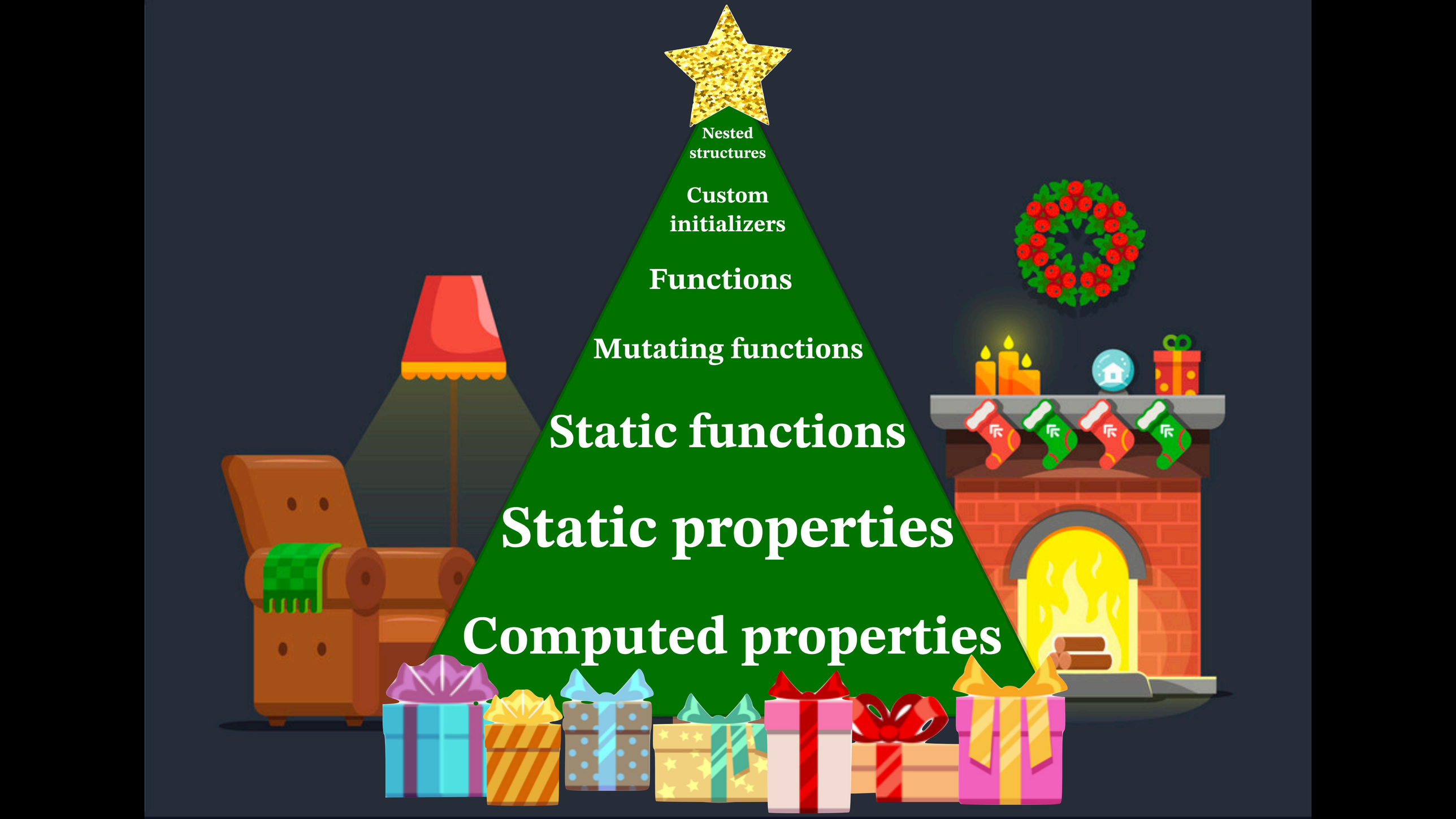
Functions

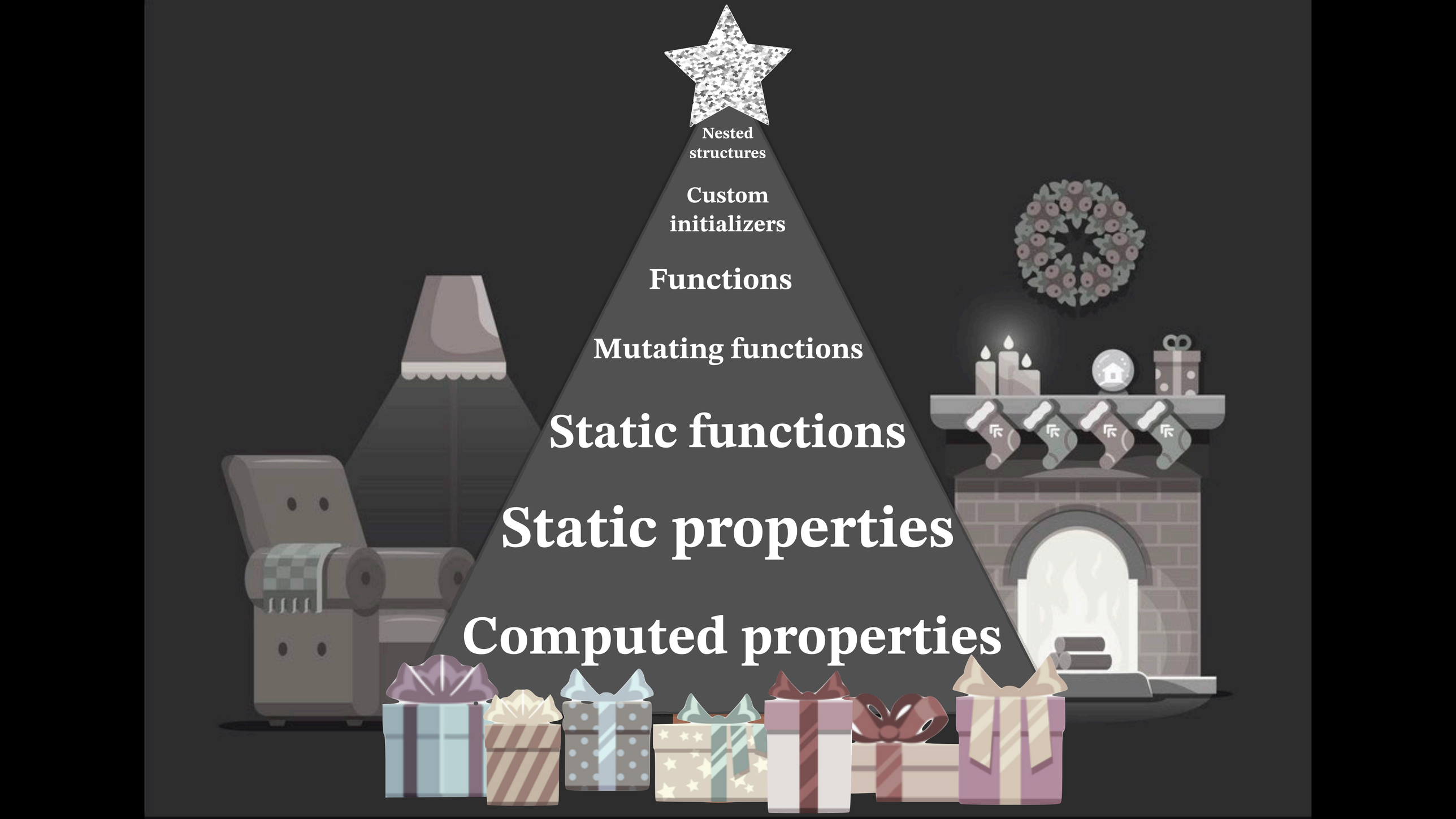
Mutating functions

Static functions

Static properties

Computed properties


Nested structures
Custom initializers
Functions
Mutating functions
Static functions
Static properties
Computed properties


Nested structures
Custom initializers
Functions
Mutating functions
Static functions
Static properties
Computed properties

Enums can’t contain stored properties.

Enums can’t contain stored properties.

Enums can’t hold state.

	Enum	Struct	Class
Nesting	✅	✅	✅
Custom initializers	✅	✅	✅
Reg. functions	✅	✅	✅
Mutating functions	✅	✅	✅
Static functions	✅	✅	✅
Static properties	✅	✅	✅
Computed properties	✅	✅	✅
Protocol conformance	✅	✅	✅
Stored properties		✅	✅
Observable			✅

Representing State

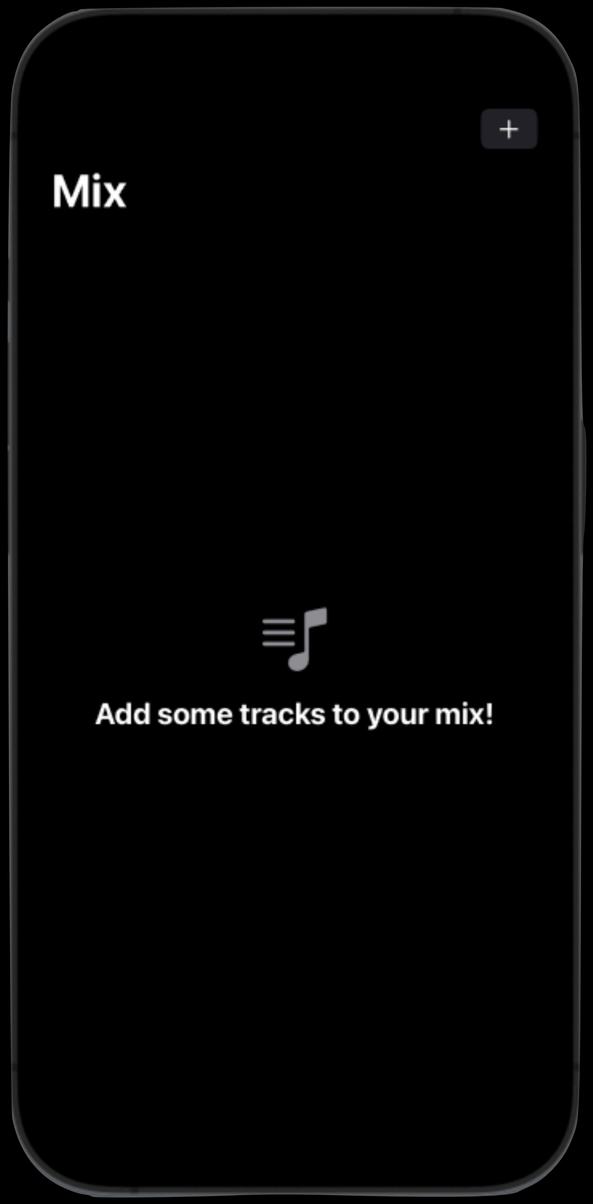
Representing State

Representing A Collection

Representing A Collection

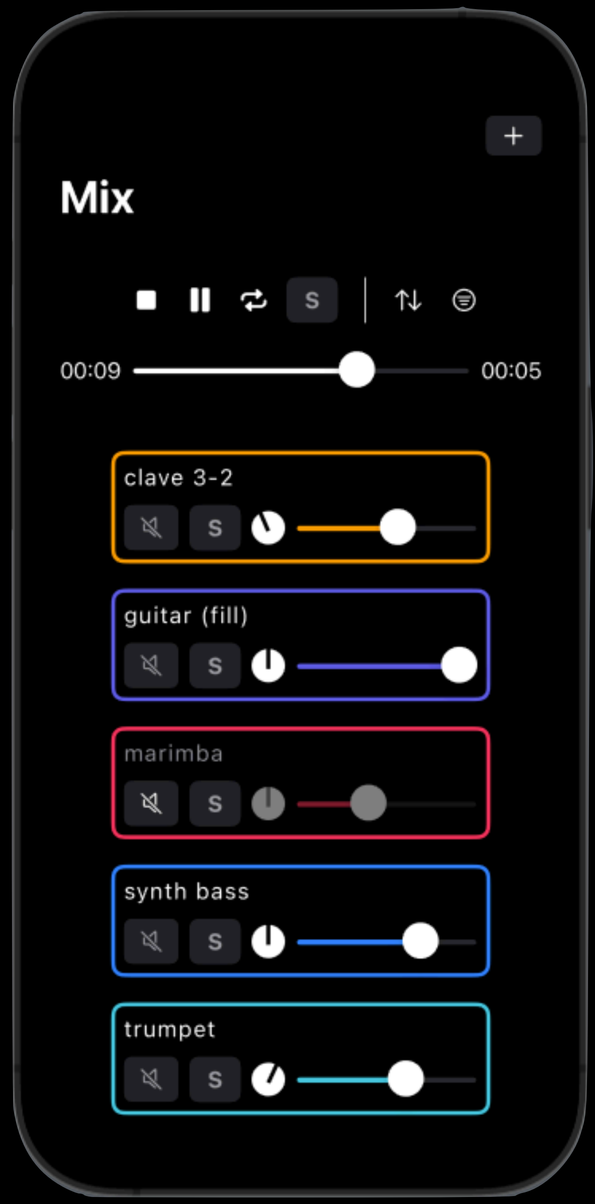
```
class Mix: ObservableObject {
    @Published var tracks: Set<Track> = []
}
```

Representing A Collection




```
class Mix: ObservableObject {
    @Published var tracks: Set<Track> = []
}
```

```
struct MixingView: View {
    @StateObject var mix = Mix()
    var body: some View {
        if mix.tracks.isEmpty {
            Text("Add some tracks to your mix!")
        } else {
            VStack {
                // playback controls
                // list of tracks
            }
        }
    }
}
```



Representing A Collection

```
class Mix: ObservableObject {
    @Published var tracks: Set<Track> = []
}
```

Using Enums to Increase Coherence

```
var isEmpty: Boolean
```

0 & 1

```
var isEmpty: Boolean
```

0 & 1 → true & false

~~Boolean~~

two-cased enum

two-cased enum benefits

- ★ Add additional cases later
- ★ Switch statement protection
- ★ More readable

Representing A Collection

```
enum Mix: Equatable {
    case empty
    case contains(Set<Track>)









}
```

Representing A Collection

```
enum Mix: Equatable {
    case empty
    case contains(Set<Track>)

    var tracks: Set<Track> {
        get {
            switch self {
            case .empty: []
            case .contains(let tracks): tracks
            }
        }
        set {
            if newValue.isEmpty {
                self = .empty
            } else {
                self = .contains(newValue)
            }
        }
    }
}
```

Representing A Collection

```
enum Mix: Equatable {
    case empty
    case contains(Set<Track>)

    var tracks: Set<Track> {
        get {
            switch self {
            case .empty: []
            case .contains(let tracks): tracks
            }
        }
        set {
            if newValue.isEmpty {
                self = .empty
            } else {
                self = .contains(newValue)
            }
        }
    }
}
```

```
struct MixingView: View {
    @State private var mix = Mix.empty
    var body: some View {
        switch mix {
        case .empty:
            Text("Add some tracks to your mix!")
        case .contains:
            VStack {
                // playback controls
                // list of tracks
            }
        }
    }
}
```

Using Enums to Increase Coherence

Part 2

Using Enums to Increase Coherence

Part 2

```
player.numberOfLoops = 0 (don't loop)
                     = 1 (loop once)
                     = -1 (forever)
```

Using Enums to Increase Coherence

```
player.numberOfLoops = 0 (don't loop)
                     = 1 (loop once)
                     = -1 (forever)
```

```
struct RepeatButton: View {
    let value: Int
    var body: some View {
        Image(systemName: value == 1 ? "repeat.1" : "repeat")
            .padding(8)
            .background(
                RoundedRectangle(cornerRadius: 8)
                    .fill(value == 0 ? .clear : .primary.opacity(0.2)))
    }
}
```

Using Enums to Increase Coherence

```
enum RepeatMode {
    case none
    case once
    case forever
}
```

Using Enums to Increase Coherence

```
enum RepeatMode {
    case none
    case once
    case forever
}
```

```
enum RepeatMode {
    case none(times: Int = 0)
    case once(times: Int = 1)
    case forever
}
```

Using Enums to Increase Coherence

```
enum RepeatMode {
    case none(times: Int = 0)
    case once(times: Int = 1)
    case forever
}
```

```
enum RepeatMode: Equatable, CaseIterable {
    case times(_ total: Int)
    case forever

    static var none: Self { .times(0) }
    static var once: Self { .times(1) }

    static var allCases: [RepeatMode] = [.none, .forever, .once]

    var systemName: String {
        switch self {
        case .times(let times) where times == 1: "repeat.1"
        default: "repeat"
        }
    }

    var value: Int {
        switch self {
        case .none: 0
        case .once: 1
        case .times(let times): times
        case .forever: -1
        }
    }
}
```

Using Enums to Increase Coherence

```swift
enum RepeatMode: Equatable, CaseIterable {
    case times(_ total: Int)
    case forever

    static var none: Self { .times(0) }
    static var once: Self { .times(1) }

    static var allCases: [RepeatMode] = [.none, .forever, .once]

    var systemName: String {
        switch self {
        case .times(let times) where times == 1: "repeat.1"
        default: "repeat"
        }
    }

    var value: Int {
        switch self {
        case .none: 0
        case .once: 1
        case .times(let times): times
        case .forever: -1
        }
    }
}
```

```swift
init(from playerValue: Int) {
    switch playerValue {
    case -1: self = .forever
    default: self = .times(playerValue)
    }
}
```

Using Enums to Increase Coherence

```
init(from playerValue: Int) {
    switch playerValue {
    case -1: self = .forever
    default: self = .times(playerValue)
    }
}
```

```
var repeatMode: RepeatMode {
    get {
        if let player {
            return RepeatMode(from: player.numberOfLoops)
        } else {
            return .none
        }
    }
    set {
        if let player {
            player.numberOfLoops = newValue.value
        }
    }
}
```

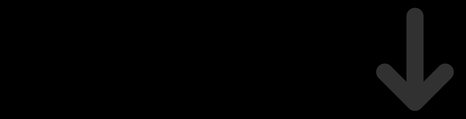
Using Enums to Increase Coherence

```
init(from playerValue: Int) {
    switch playerValue {
    case -1: self = .forever
    default: self = .times(playerValue)
    }
}
```

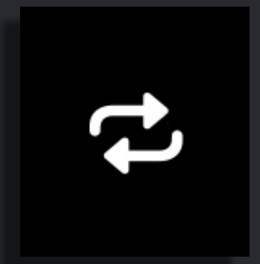
```
var repeatMode: RepeatMode {
    get {
        if let player {
            return RepeatMode(from: player.numberOfLoops)
        } else {
            return .none
        }
    }
    set {
        if let player {
            player.numberOfLoops = newValue.value
        }
    }
}
```

Using Enums to Increase Coherence

```
struct RepeatButton: View {
    let value: Int
    var body: some View {
        Image(systemName: value == 1 ? "repeat.1" : "repeat")
            .padding(8)
            .background(
                RoundedRectangle(cornerRadius: 8)
                    .fill(value == 0 ? .clear : .primary.opacity(0.2)))
    }
}
```



```
struct RepeatButton: View {
    let repeatMode: RepeatMode
    var body: some View {
        Image(systemName: repeatMode.systemName)
            .padding(8)
            .background(
                RoundedRectangle(cornerRadius: 8)
                    .fill(repeatMode == .none ? .clear : .primary.opacity(0.2)))
    }
}
```



Using Enums to Increase Coherence

```
struct RepeatButton: View {
    let repeatMode: RepeatMode
    var body: some View {
        Image(systemName: repeatMode.systemName)
            .padding(8)
            .background(
                RoundedRectangle(cornerRadius: 8)
                    .fill(repeatMode == .none ? .clear : .primary.opacity(0.2)))
    }
}
```

```
Button {
    repeatMode.stepForward()
} label: {
    RepeatButton(repeatMode: repeatMode)
}
```


Quick Select
Tap to quickly add or remove a track to your mix.
FILL
drums
acoustic
electric
glock
guitar
vibraslap
Browse
Explore the catalog.
MALLETS
marimba
vibes
LATIN
agogo
bongos
cabasa
clave 2-3
clave 3-2
cowbell
guiro
shaker

CaseIterable Bonus

```swift
extension CaseIterable where Self: Equatable, AllCases.Index == Int {
    mutating func stepForward() {
        guard let nextIndex else { return }
        self = Self.allCases[nextIndex]
    }

    mutating func stepBackward() {
        guard let previousIndex else { return }
        self = Self.allCases[previousIndex]
    }

    func next() -> Self? {
        guard let nextIndex else { return nil }
        return Self.allCases[nextIndex]
    }

    func previous() -> Self? {
        guard let previousIndex else { return nil }
        return Self.allCases[previousIndex]
    }
```

CaseIterable Bonus

```swift
extension CaseIterable where Self: Equatable, AllCases.Index == Int {
    mutating func stepForward() {
        guard let nextIndex else { return }
        self = Self.allCases[nextIndex]
    }

    mutating func stepBa
        guard let previo
        self = Self.allC
    }

    func next() -> Self?
        guard let nextIn
        return Self.allC
    }

    func previous() -> S
        guard let previo
        return Self.allC
    }
```

```swift
var nextIndex: Int? {
    let allCases = Self.allCases
    guard let currentIndex = allCases.firstIndex(of: self) else {
        return nil }
    return (currentIndex + 1) % allCases.count
}

var previousIndex: Int? {
    let allCases = Self.allCases
    guard let currentIndex = allCases.firstIndex(of: self) else {
        return nil }
    return (currentIndex - 1 + allCases.count) % allCases.count
}
```

Streamlining Multiple States

Streamlining Multiple States

i.e. mitigating single source of
truth violations

Streamlining Multiple States

```
var stopButton: some View {
    Button {
        player?.stop()
    } label: {
        Image(systemName: "stop.fill")
    }
    .disabled(!(player?.isPlaying ?? false) == false &&
              (player?.currentTime ?? 0) == 0)
}
```

🚩 Multiple Conditional Checks…

```
var stopButton: some View {
    Button {
        player?.stop()
    } label: {
        Image(systemName: "stop.fill")
    }
    .disabled(!(player?.isPlaying ?? false) == false &&
              (player?.currentTime ?? 0) == 0)
}
```

🚩 Multiple Conditional Checks…

```
var stopButton: some View {
    Button {
        player?.stop()
    } label: {
        Image(systemName: "stop.fill")
    }
    .disabled(!(player?.isPlaying ?? false) == false &&
              (player?.currentTime ?? 0) == 0)
}
```

…Can Mean Multiple States 🚩

Streamlining Multiple States

```
if !isPlaying && currentTime == 0 {
    // disable
} else {
    // enable
}
```

Streamlining Multiple States

```
if !isPlaying && currentTime == 0 {
    // disable
} else {
    // enable
}
```

	default		
isPlaying	false		
currentTime	0		
disabled	true		

Streamlining Multiple States

```
if !isPlaying && currentTime == 0 {
    // disable
} else {
    // enable
}
```

	default	▶	
isPlaying	false	true	
currentTime	0	??:??	
disabled	true	false	

Streamlining Multiple States

```
if !isPlaying && currentTime == 0 {
    // disable
} else {
    // enable
}
```

	default	▶	⏹
isPlaying	false	true	false
currentTime	0	??:??	5:24… or whatever
disabled	true	false	false

Streamlining Multiple States

```
if !isPlaying && currentTime == 0 {
    // disable
} else {
    // enable
}
```

	default	▶	⏸
isPlaying	false	true	false
currentTime	0	??:??	5:24… or whatever
disabled	true	false	false

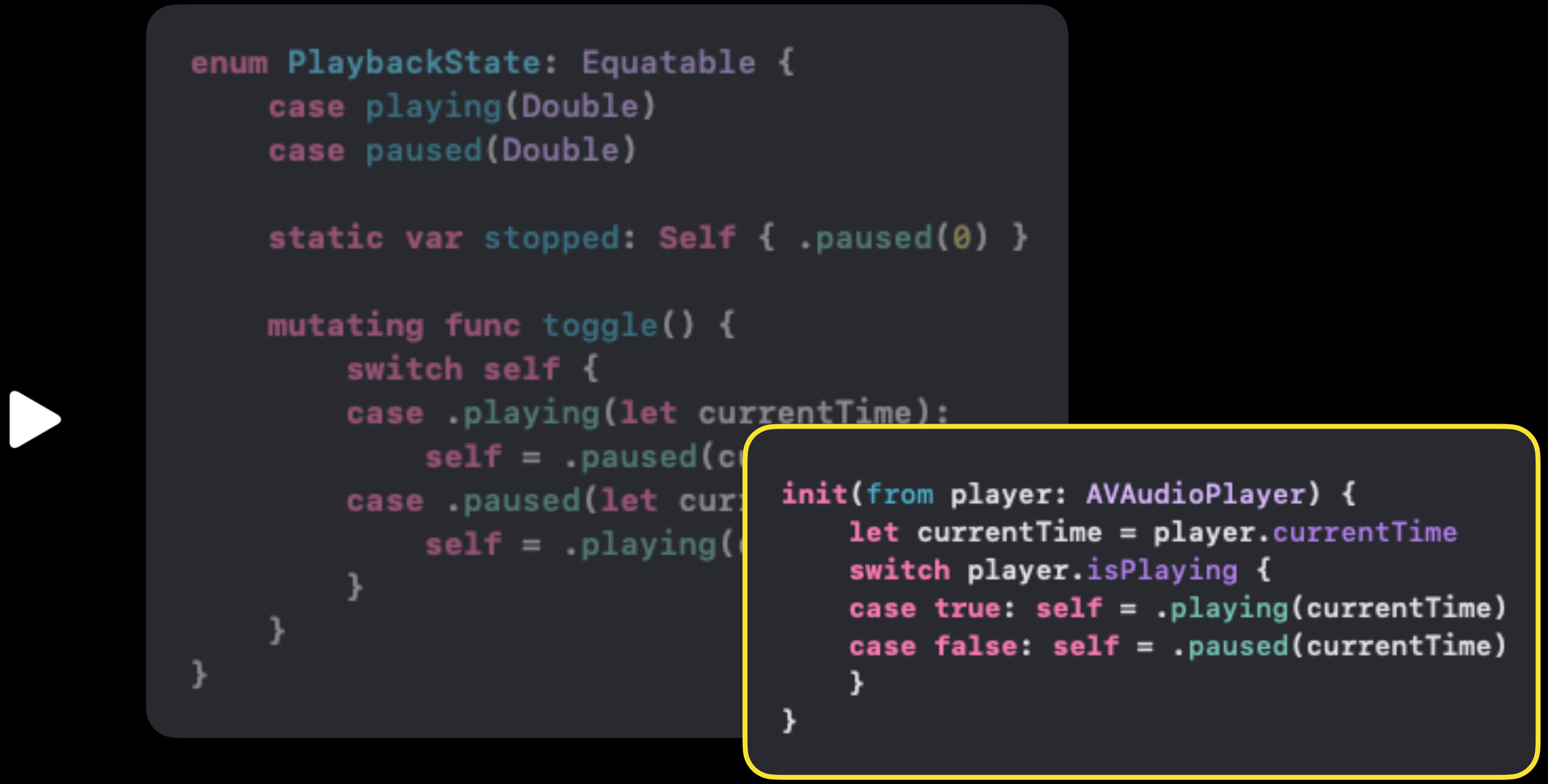
🚩 Multiple Conditional Checks 🚩

```
var stopButton: some View {
    Button {
        player?.stop()
    } label: {
        Image(systemName: "stop.fill")
    }
    .disabled(!(player?.isPlaying ?? false) == false &&
              (player?.currentTime ?? 0) == 0)
}
```

Streamlining Multiple States

```
enum PlaybackState: Equatable {
    case playing(Double)
    case paused(Double)

    static var stopped: Self { .paused(0) }

    mutating func toggle() {
        switch self {
        case .playing(let currentTime):
            self = .paused(currentTime)
        case .paused(let currentTime):
            self = .playing(currentTime)
        }
    }
}
```

Streamlining Multiple States



Streamlining Multiple States

```
enum PlaybackState: Equatable {
    case playing(Double)
    case paused(Double)

    static var stopped: Self { .paused(0) }

    mutating func toggle() {
        switch self {
        case .playing(let currentTime):
            self = .paused(currentTime)
        case .paused(let currentTime):
            self = .playing(currentTime)
        }
    }
}
```

```
    init(from player: AVAudioPlayer) {
        let currentTime = player.currentTime
        switch player.isPlaying {
        case true: self = .playing(currentTime)
        case false: self = .paused(currentTime)
        }
    }
```

```
var playbackState: PlaybackState {
    get {
        if let player {
            return PlaybackState(from: player)
        } else {
            return .stopped
        }
    }
    set {
        if let player {
            update(player, from: newValue)
        }
    }
}
```

Streamlining Multiple States

```
enum PlaybackState: Equatable {
    case playing(Double)
    case paused(Double)

    static var stopped: Self { .paused(0) }

    mutating func toggle() {
        switch self {
        case .playing(let currentTime):
            self = .paused(currentTime)
        case .paused(let currentTime):
            self = .playing(currentTime)
        }
    }
}
```

```
    init(from player: AVAudioPlayer) {
        let currentTime = player.currentTime
        switch player.isPlaying {
        case true: self = .playing(currentTime)
        case false: self = .paused(currentTime)
        }
    }
```

```
var playbackState: PlaybackState {
    get {
        if let player {
            return PlaybackState(from: player)
        } else {
            return .stopped
        }
    }
    set {
        if let player {
            update(player, from: newValue)
        }
    }
}
```

Streamlining Multiple States

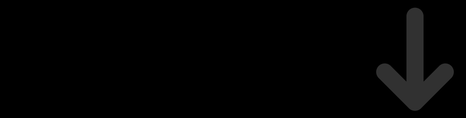
```
var playbackState: PlaybackState {
    get {
        if let player {
            return PlaybackState(from: player)
        } else {
            return .stopped
        }
    }
    set {
        if let player {
            update(player, from: newValue)
        }
    }
}
```



```
func update(
    _ player: AVAudioPlayer, from pbState: PlaybackState) {
    switch pbState {
    case .playing:
        player.play()
    case .stopped:
        player.stop()
        player.currentTime = 0
    case .paused:
        player.pause()
    }
}
```

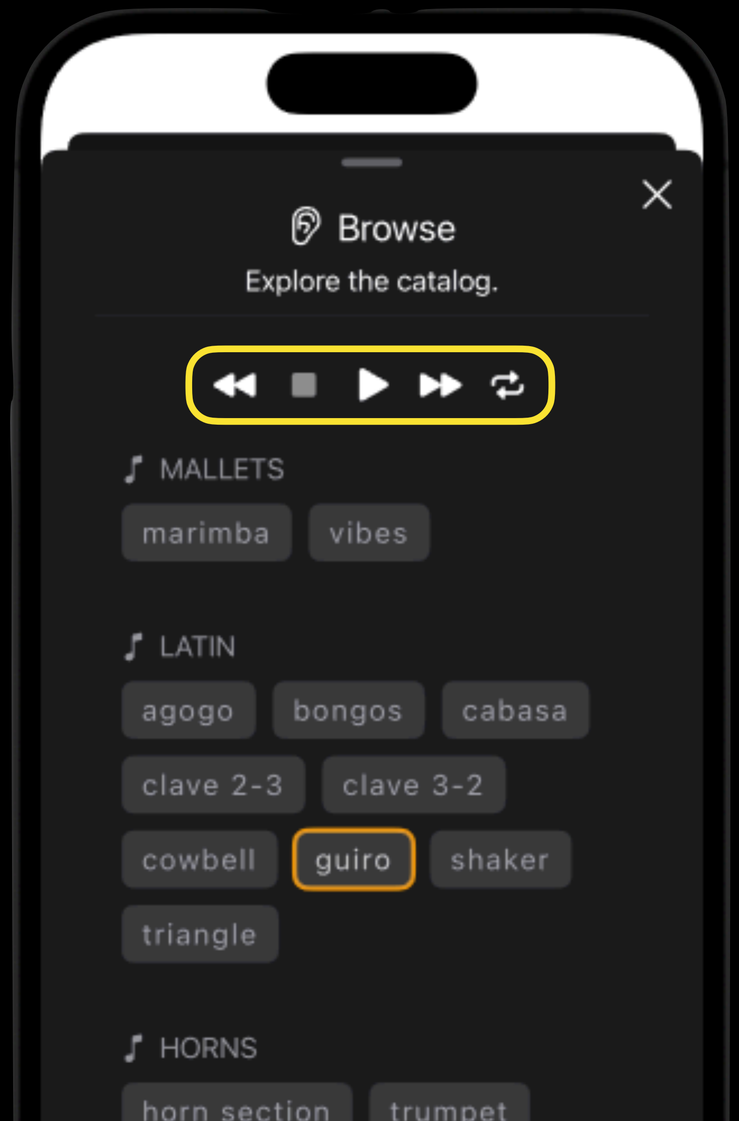
Streamlining Multiple States

```
var stopButton: some View {
    Button {
        player?.stop()
    } label: {
        Image(systemName: "stop.fill")
    }
    .disabled(!(player?.isPlaying ?? false) == false &&
              (player?.currentTime ?? 0) == 0)
}
```



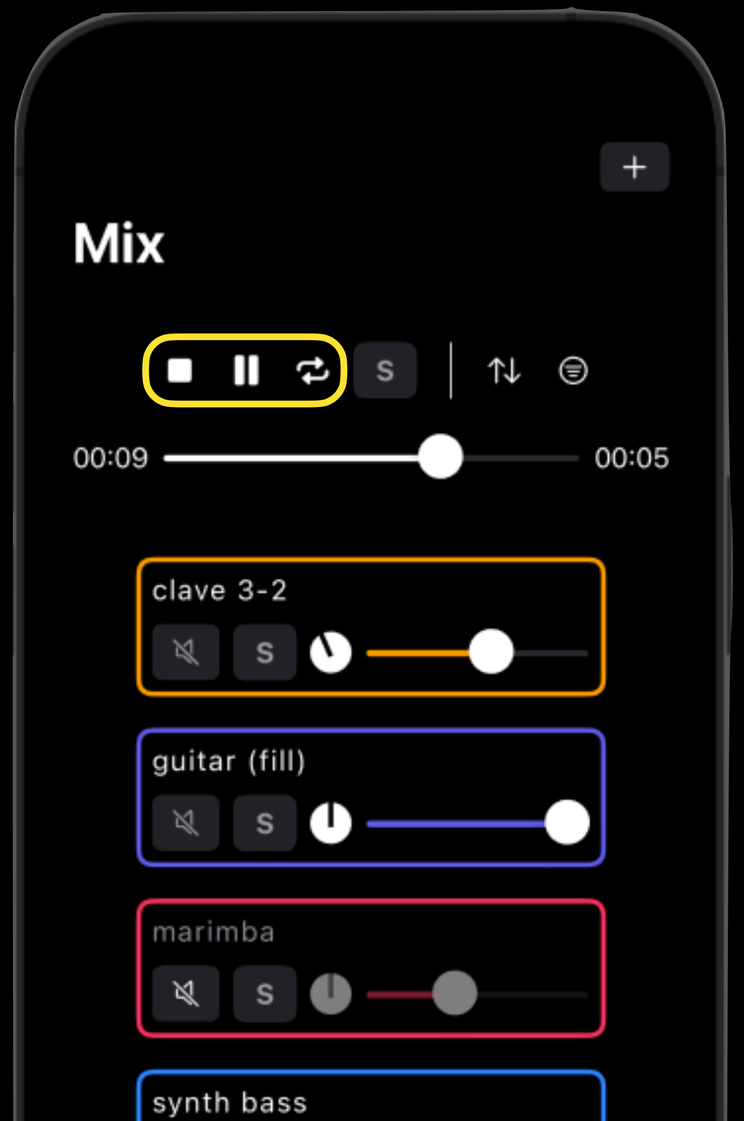
```
var stopButton: some View {
    Button {
        playbackState = .stopped
    } label: {
        Image(systemName: "stop.fill")
    }
    .disabled(playbackState == .stopped)
}
```

PlaybackState Bonus!



AVAudioPlayer?

AVAudioPlayerNode



PlaybackState Bonus!

AVAudioPlayer?

```
enum PlaybackState {
    case playing(Double)
    case paused(Double)

    static var stopped: Self { .paused(0) }

    mutating func toggle() {•••}
}
```

AVAudioPlayerNode

```
enum PlaybackState {
    case playing(AVAudioFramePosition)
    case paused(AVAudioFramePosition)

    static var stopped: Self { .paused(0) }

    mutating func toggle() {•••}
}
```

PlaybackState Bonus!

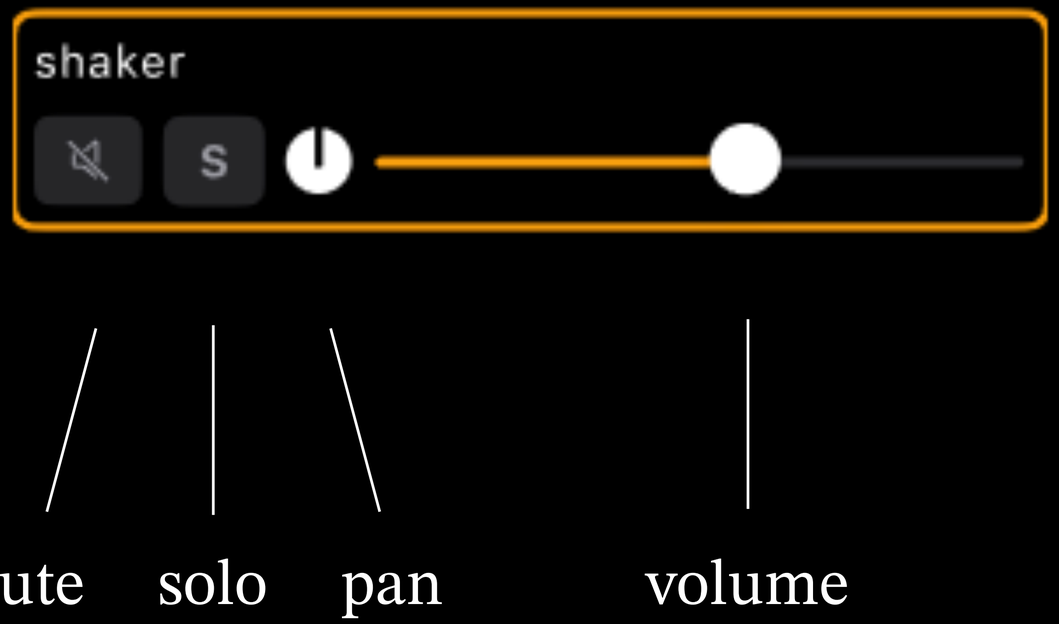
```
enum PlaybackState<T: Numeric> {
    case playing(T)
    case paused(T)

    static var stopped: Self { .paused(0) }

    mutating func toggle() {•••}
}
```

The nugget that started it all…

TrackView

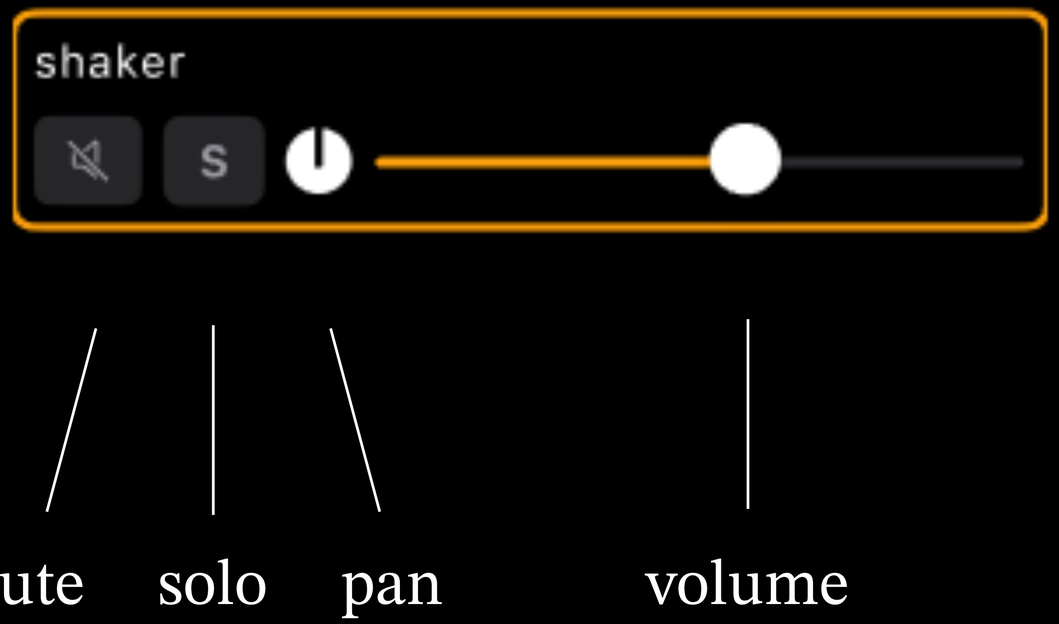
mute solo pan volume

TrackView




```
enum TrackControls {
    static let mute = MuteButton()
    static let solo = SoloButton()
    static let pan = PanKnob()
    static let volume = VolumeSlider()
}
```

TrackView




```
enum TrackControls {
    case mute(Binding<Bool>)
    case solo(Binding<Bool>)
    case pan(Binding<CGFloat>)
    case volume(Binding<Float>)
}
```

Enums That Conform To View!!!!!

TrackView

mute solo pan volume

```
enum TrackControls: View {
    case mute(Binding<Bool>)
    case solo(Binding<Bool>)
    case pan(Binding<CGFloat>)
    case volume(Binding<Float>)
}
```

Enums That Conform To View!!!!!

```
enum TrackControls: View {
    case mute(Binding<Bool>)
    case solo(Binding<Bool>)
    case pan(Binding<CGFloat>)
    case volume(Binding<Float>)
}
```


```
var body: some View {
    switch self {
    case .mute(let isMuted): Button {•••} label: {•••}
    case .solo(let soloed): Button {•••} label: {•••}
    case .pan(let boundValue):
        Circle()
            .frame(width: 25)
            .overlay(alignment: .top) {
                Rectangle()
                    .fill(.black)
                    .frame(width: 3, height: 15)
                    .rotationEffect(.degrees(boundValue.wrappedValue), anchor:
                        .bottom)
                    .gesture(
                        DragGesture()
                            .onChanged { value in
                                guard (-135...135).contains(value.location.x)
                                    else { return }
                                boundValue.wrappedValue = value.location.x +
                                    value.location.y
                            }
                    )
            }
    case .volume(let value):
        Slider(value: value, in: 0...1)
    }
}
```

Enums That Conform To View!!!!!

```
enum TrackControls: View {
    case mute(Binding<Bool>)
    case solo(Binding<Bool>)
    case pan(Binding<CGFloat>)
    case volume(Binding<Float>)
}
```

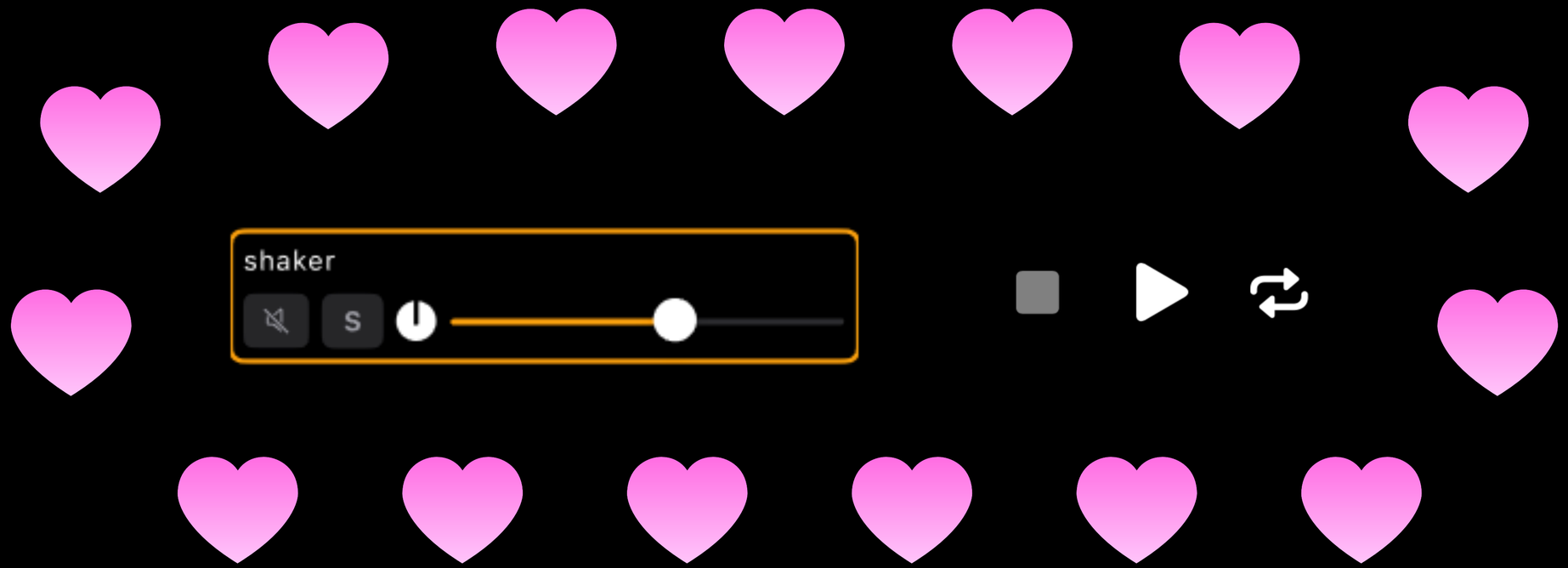

```
var body: some View {
    switch self {
    case .mute(let isMuted): Button {•••} label: {•••}
    case .solo(let soloed): Button {•••} label: {•••}
    case .pan(let boundValue):
        Circle()
            .frame(width: 25)
            .overlay(alignment: .top) {
                Rectangle()
                    .fill(.black)
                    .frame(width: 3, height: 15)
                    .rotationEffect(.degrees(boundValue.wrappedValue), anchor:
                        .bottom)
                    .gesture(
                        DragGesture()
                            .onChanged { value in
                                guard (-135...135).contains(value.location.x)
                                    else { return }
                                boundValue.wrappedValue = value.location.x +
                                    value.location.y
                            }
                    )
            }
    case .volume(let value):
        Slider(value: value, in: 0...1)
    }
}
```

Enums That Conform To View!!!!!

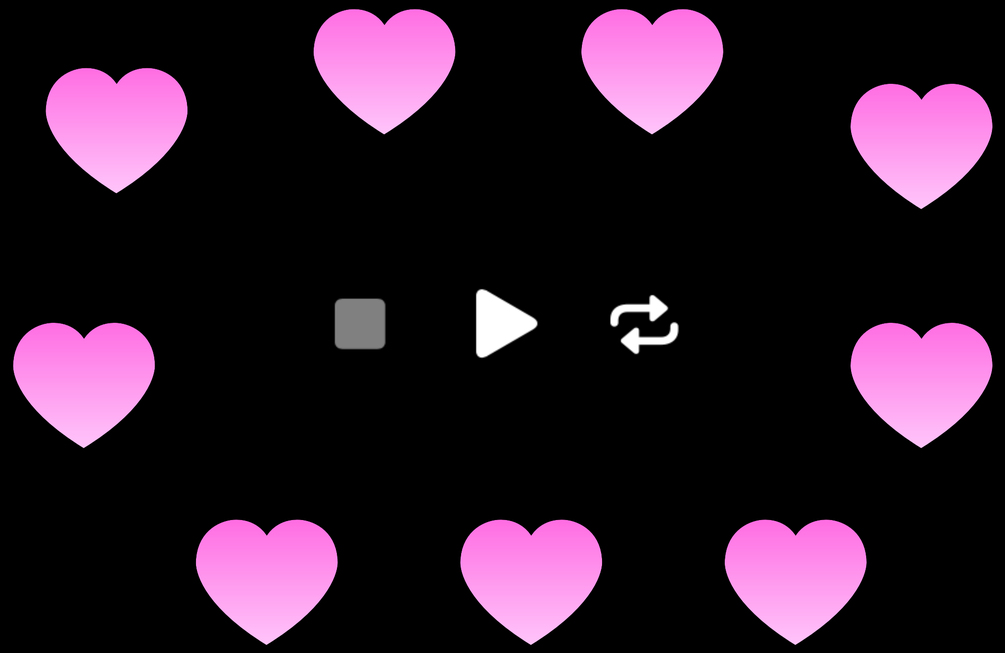
```
enum TrackControls: View {
    case mute(Binding<Bool>)
    case solo(Binding<Bool>)
    case pan(Binding<CGFloat>)
    case volume(Binding<Float>)
}
```


```
HStack {
    TrackControls.mute($track.isMuted)

    TrackControls.solo($soloed)

    TrackControls.pan($track.scaledPan)
        .opacity(track.isMuted ? 0.5 : 1)

    TrackControls.volume($track.sliderVolume)
        .opacity(track.isMuted ? 0.5 : 1)
        .accentColor(color)
}
```

Enums That Conform To View!!!!!

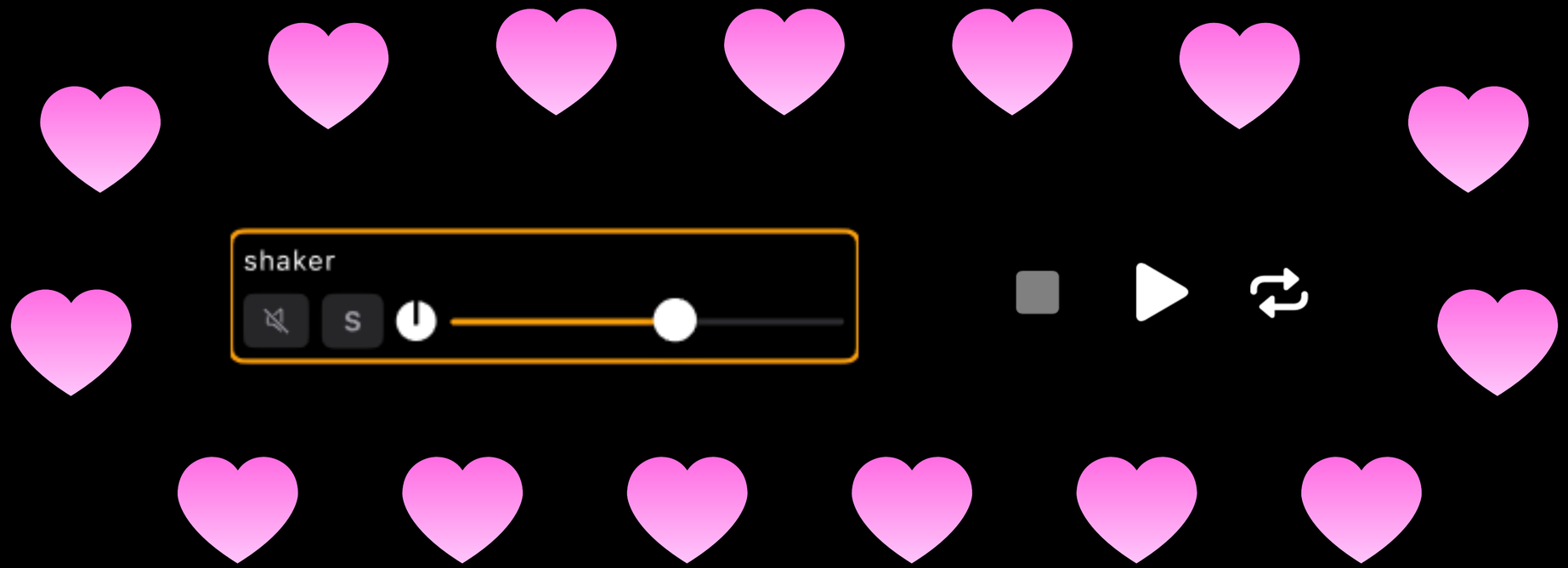


Enums That Conform To View!!!!!



```
enum PlaybackControls: View {
    case playPause(isPlaying: Bool, () -> Void)
    case stop(() -> Void)
    case repeating(repeatMode: RepeatMode, () -> Void)

    var systemName: String { ••• }
    private var isPlayPause: Bool { ••• }
    private var repeatMode: RepeatMode? { ••• }

    var action: () -> Void {
        switch self {
        case .playPause(_, let action): action
        case .stop(let action): action
        case .repeating(_, let action): action
        }
    }

    var body: some View {
        Button {
            action()
        } label: {
            Image(systemName: systemName)
                .imageScale(isPlayPause ? .large : .medium)
                .symbolVariant(.fill)
                .fontWeight(.bold)
        }
    }
}
```

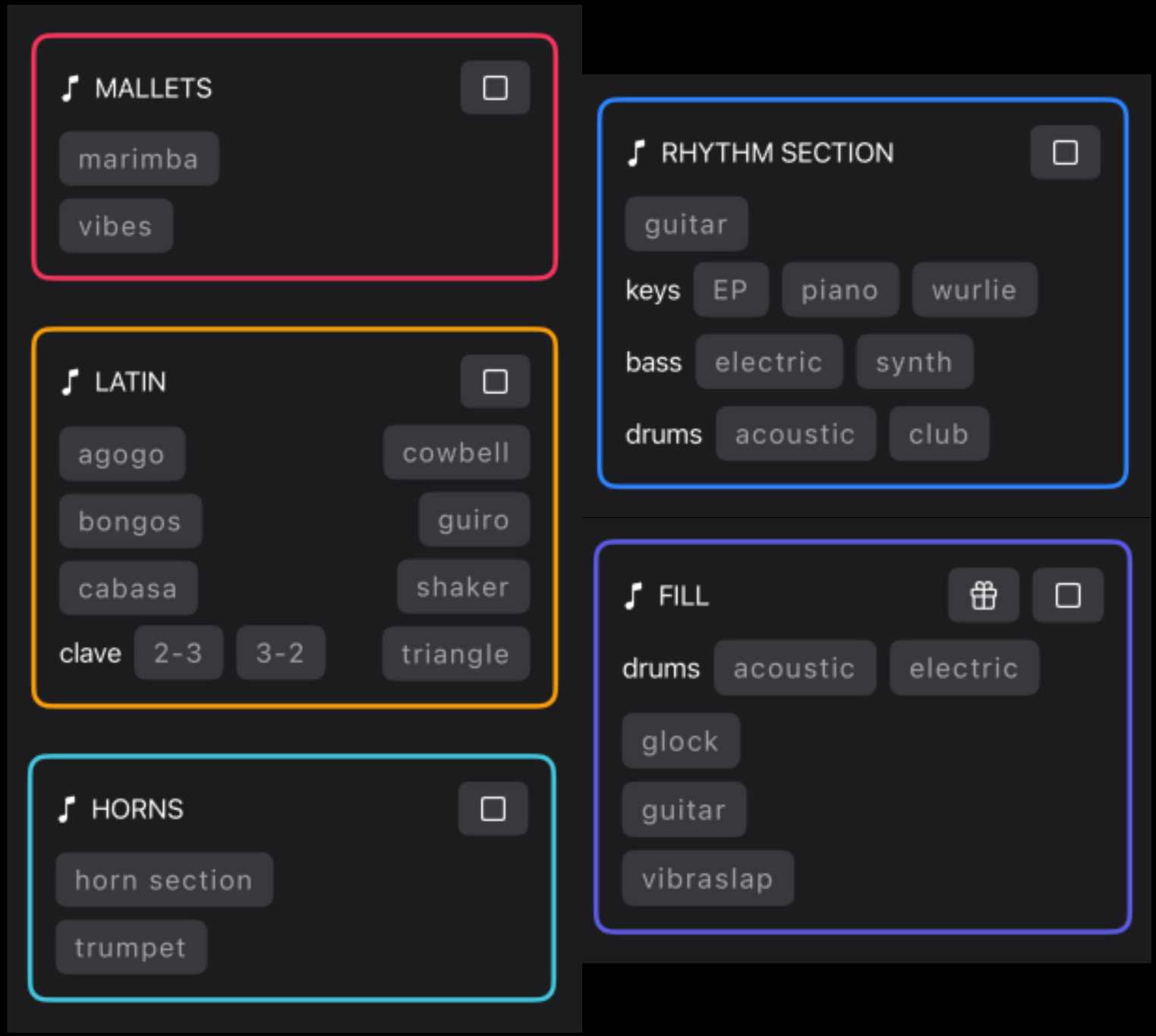
Enums That Conform To View!!!!!



Why enums?

Why enums?

Enums. Are. The. Sh*t.



Questions?

Github

code
deck
resources