Intermediate Swift

Session 403

Brian Lanier
Developer Publications Engineer

Joe Groff
Swift Compiler Engineer

© 2014 Apple Inc. All rights reserved. Redistribution or public display not permitted without written permission from Apple.

What You Will Learn

What You Will Learn

Optionals

What You Will Learn

Optionals

Memory management

What You Will Learn

- Optionals
- Memory management
- Initialization

What You Will Learn

- Optionals
- Memory management
- Initialization
- Closures

What You Will Learn

Optionals

Memory management

Initialization

Closures

Pattern matching

Optionals

Brian Lanier
Developer Publications Engineer

String to Integer

```
> Enter your age:
> _
```

String to Integer

```
> Enter your age:
> _

let age = response.toInt()
```

String to Integer

```
> Enter your age:
>

let age = response.toInt()
```

String to Integer

```
> Enter your age:
> 34

let age = response.toInt()
```

String to Integer

```
> Enter your age:
>

let age = response.toInt()
```

String to Integer

```
> Enter your age:
> thirty

let age = response.toInt()
```

String to Integer

```
> Enter your age:
>

let age = response.toInt()
```

String to Integer

```
> Enter your age:
> wouldn't you like to know

let age = response.toInt()
```

String to Integer

```
> Enter your age:
>

let age = response.toInt()
```

String to Integer

```
> Enter your age:
> how rude!

let age = response.toInt()
```

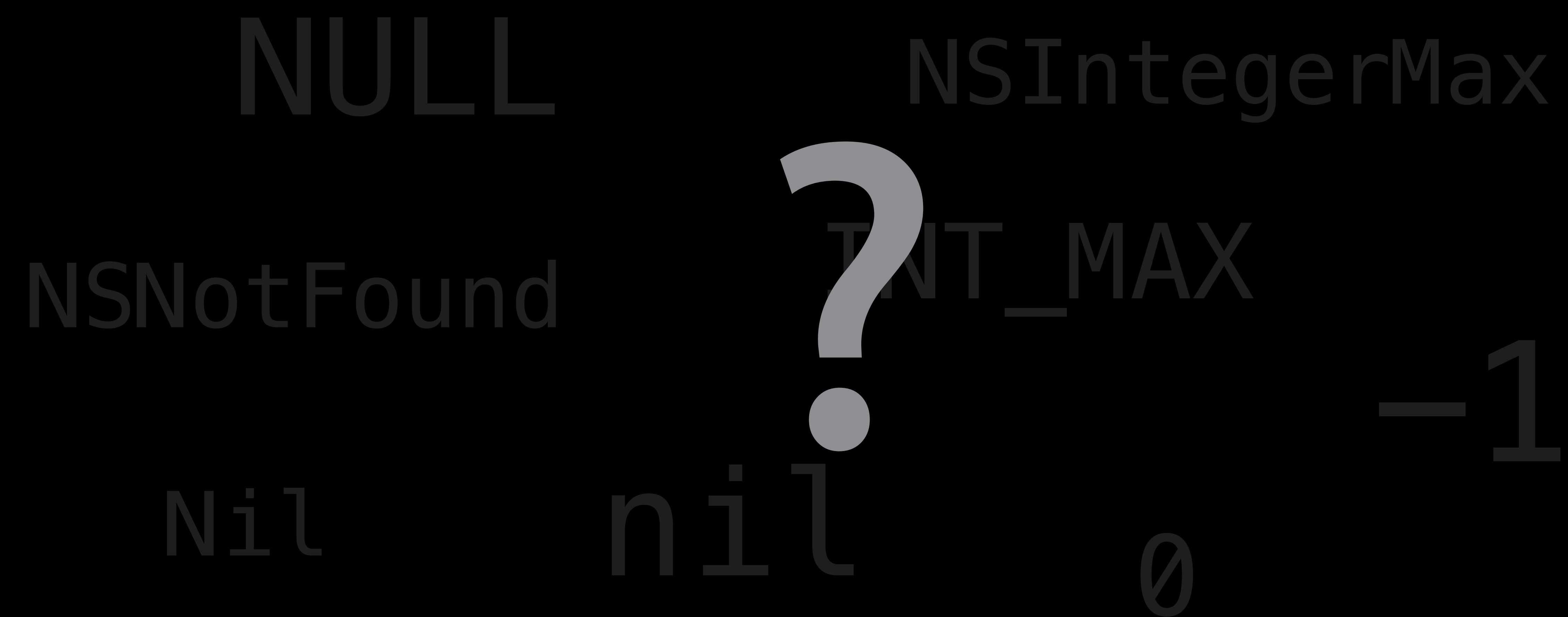
String to Integer

```
> Enter your age:
> how rude!

let age = response.toInt()
```

NULL
NSIntegerMax
NSNotFound
INT_MAX
-1
Nil
nil
0

String to Integer

```
> Enter your age:
> how rude!

let age = response.toInt()
```



Optional Type

SAFE

Optional Type

SAFE

Represents possibly missing values

Optional Type

SAFE

Represents possibly missing values

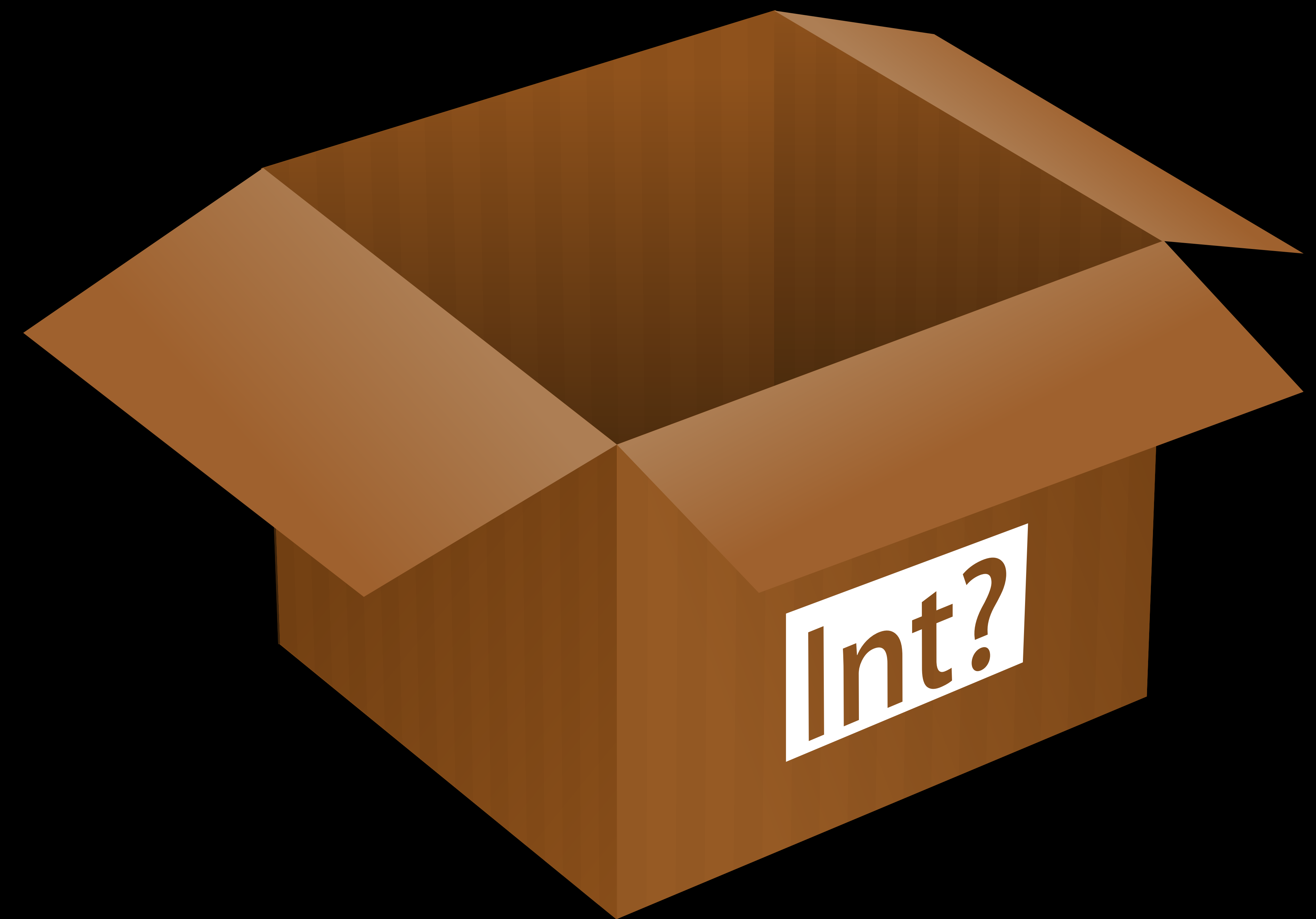
```
var optionalNumber: Int?
// default initialized to nil
```

Optional Type

SAFE

Represents possibly missing values

```
var optionalNumber: Int?
// default initialized to nil
```



Optional Type

SAFE

Represents possibly missing values

```
var optionalNumber: Int?
// default initialized to nil

optionalNumber = 6
```

Optional Type

SAFE

Represents possibly missing values

```
var optionalNumber: Int?
// default initialized to nil

optionalNumber = 6
```

One Sentinel to Rule Them All

```
> Enter your age:
> thirty

let age = response.toInt()
// age: Int? = nil
```

Non-Optional Types

```
var myString: String = "Look, a real string!"

var myObject: MyClass = MyClass()
```

Non-Optional Types

Non-optional types **can't** be `nil`

```
var myString: String = nil
// compile-time error

var myObject: MyClass = nil
// compile-time error
```

Optional Return Types

```
func findIndexOfString(string: String, array: String[]) -> Int {




}
```

Optional Return Types

```
func findIndexOfString(string: String, array: String[]) -> Int? {



}
```

Optional Return Types

```
func findIndexOfString(string: String, array: String[]) -> Int? {
    for (index, value) in enumerate(array) {



    }

}
```

Optional Return Types

```
func findIndexOfString(string: String, array: String[]) -> Int? {
    for (index, value) in enumerate(array) {
        if value == string {
            return index
        }
    }

}
```

Optional Return Types

```
func findIndexOfString(string: String, array: String[]) -> Int? {
    for (index, value) in enumerate(array) {
        if value == string {
            return index
        }
    }
    return nil
}
```

Unwrapping Optionals

```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]
let index = findIndexOfString("Madison", neighbors)
```

Unwrapping Optionals

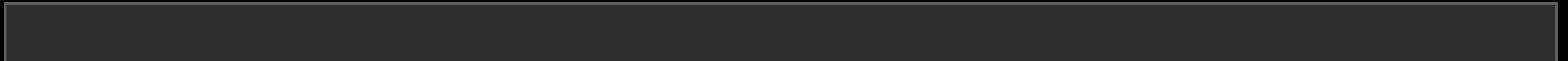
```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]
let index = findIndexOfString("Madison", neighbors)

if index {
    println("Hello, \(neighbors[index])")
} else {
    println("Must've moved away")
}
```

Unwrapping Optionals

```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]
let index: Int? = findIndexOfString("Madison", neighbors)

if index {
    println("Hello, \(neighbors[index])")
} else {
    println("Must've moved away")
}
// error: value of optional type 'Int?' not unwrapped
```

Forced Unwrapping

```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]
let index = findIndexOfString("Madison", neighbors)

if index {
    println("Hello, \(neighbors[index!])")
} else {
    println("Must've moved away")
}
```



Forced Unwrapping

```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]
let index = findIndexOfString("Madison", neighbors)

if index {
    println("Hello, \(neighbors[index!])")
} else {
    println("Must've moved away")
}
```

```
Hello, Madison
```

Forced Unwrapping

```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]
let index = findIndexOfString("Reagan", neighbors)


    println("Hello, \(neighbors[index!])")
    // runtime error!
```

Optional Binding

Test and unwrap at the **same** time

```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]
let index = findIndexOfString("Anna", neighbors)
```

Optional Binding

Test and unwrap at the **same** time

```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]
let index = findIndexOfString("Anna", neighbors)

if let indexValue = index {
    println("Hello, \(neighbors[indexValue])")
} else {
    println("Must've moved away")
}
```

Optional Binding

Test and unwrap at the **same** time

```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]
let index = findIndexOfString("Anna", neighbors)

if let indexValue = index {  // index is of type Int?
    println("Hello, \(neighbors[indexValue])")
} else {
    println("Must've moved away")
}
```

Optional Binding

Test and unwrap at the **same** time

```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]
let index = findIndexOfString("Anna", neighbors)

if let indexValue = index {  // indexValue is of type Int
    println("Hello, \(neighbors[indexValue])")
} else {
    println("Must've moved away")
}
```

Optional Binding

Test and unwrap at the **same** time

```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]
let index = findIndexOfString("Anna", neighbors)

if let indexValue = index {  // indexValue is of type Int
    println("Hello, \(neighbors[indexValue])")
} else {
    println("Must've moved away")
}
```

Optional Binding

Test and unwrap at the **same** time

```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]

if let index = findIndexOfString("Anna", neighbors) {
    println("Hello, \(neighbors[index])") // index is of type Int
} else {
    println("Must've moved away")
}
```

Optional Binding

Test and unwrap at the **same** time

```
var neighbors = ["Alex", "Anna", "Madison", "Dave"]

if let index = findIndexOfString("Anna", neighbors) {
    println("Hello, \(neighbors[index])") // index is of type Int
} else {
    println("Must've moved away")
}
```

```
Hello, Anna
```

Optional Binding

```
class Person {
    var residence: Residence?
}
```

Optional Binding

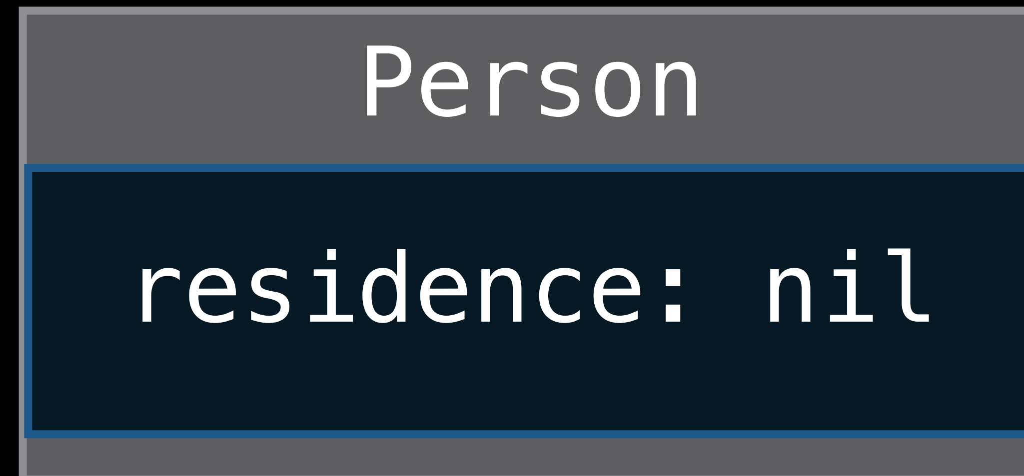
```
class Person {
    var residence: Residence?
}

class Residence {
    var address: Address?
}
```

Optional Binding

```
class Person {
    var residence: Residence?
}

class Residence {
    var address: Address?
}

class Address {
    var buildingNumber: String?
    var streetName: String?
    var apartmentNumber: String?
}
```

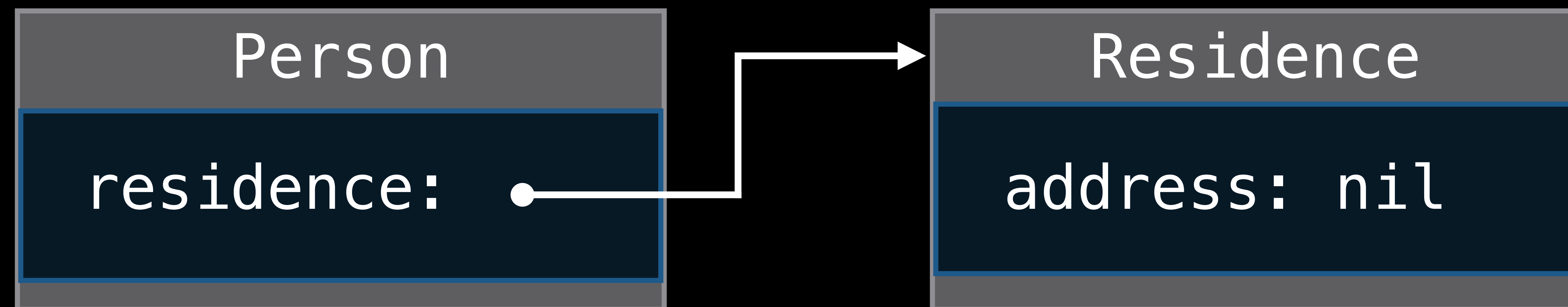
Optional Binding

```
let paul = Person()
```



Optional Binding

```
let paul = Person()
```



Optional Binding

Optional Binding

Optional Binding

```
var addressNumber: Int?

if let home = paul.residence {




}
```

Optional Binding

```
var addressNumber: Int?

if let home = paul.residence {
    if let postalAddress = home.address {




    }
}
```

Optional Binding

```
var addressNumber: Int?

if let home = paul.residence {
    if let postalAddress = home.address {
        if let building = postalAddress.buildingNumber {



        }
    }
}
```

Optional Binding

```
var addressNumber: Int?

if let home = paul.residence {
    if let postalAddress = home.address {
        if let building = postalAddress.buildingNumber {
            if let convertedNumber = building.toInt() {

            }
        }
    }
}
```

Optional Binding

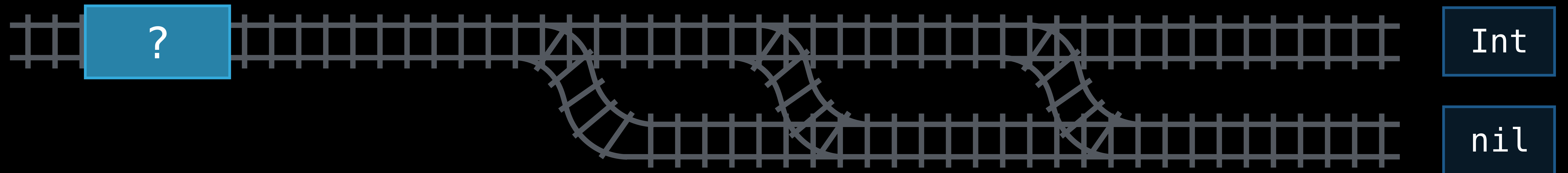
```
var addressNumber: Int?

if let home = paul.residence {
    if let postalAddress = home.address {
        if let building = postalAddress.buildingNumber {
            if let convertedNumber = building.toInt() {
                addressNumber = convertedNumber
            }
        }
    }
}
```

Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```

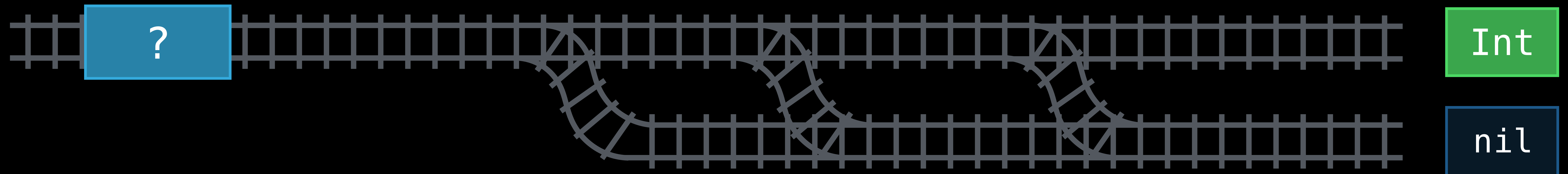
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



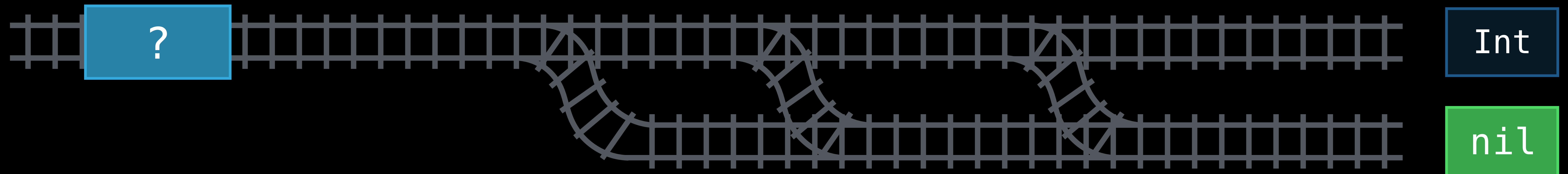
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



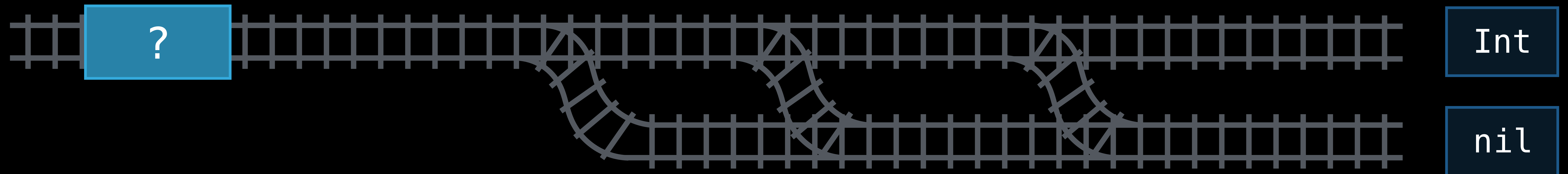
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



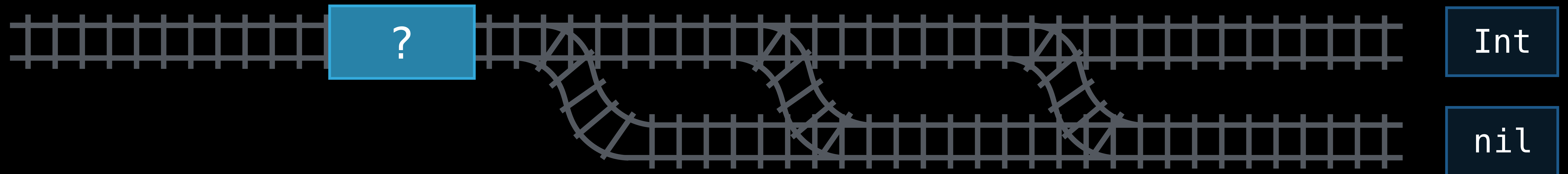
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



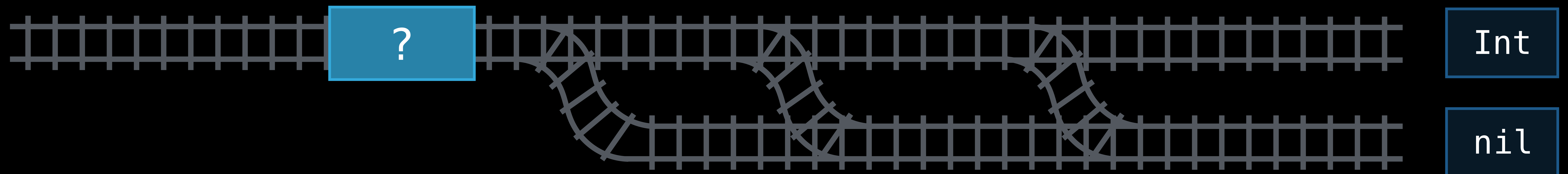
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



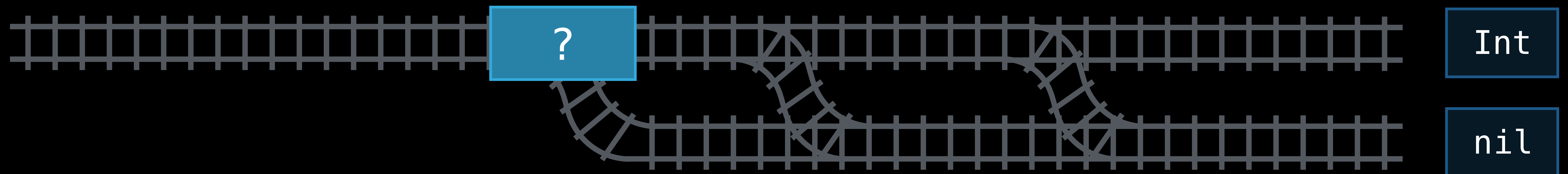
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```

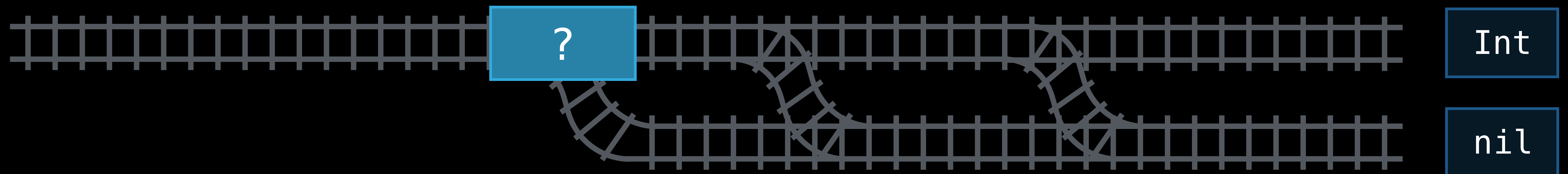


Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```

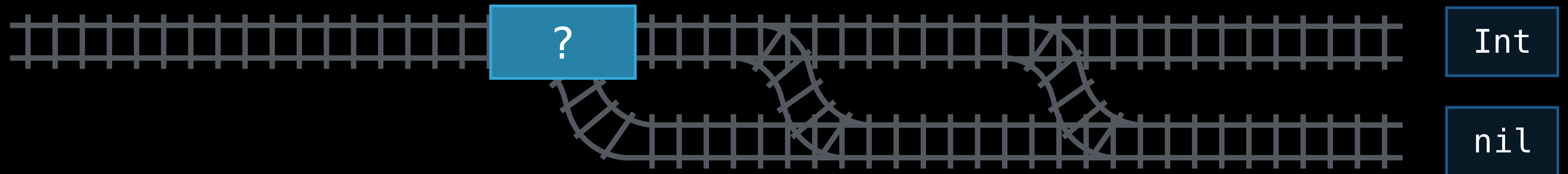
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



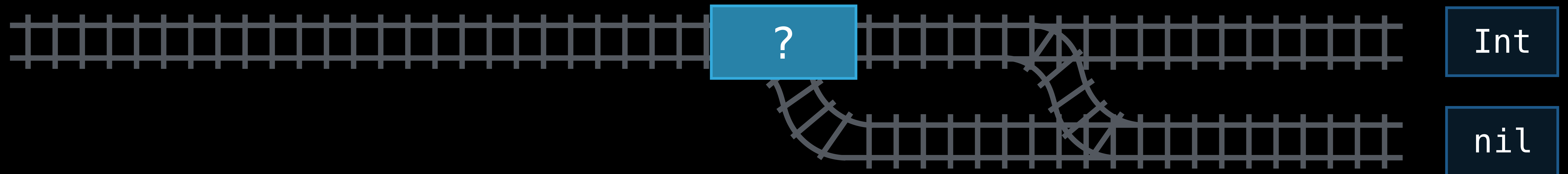
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



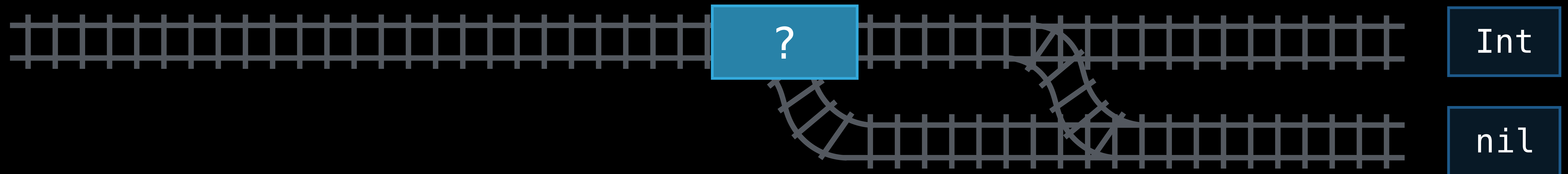
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



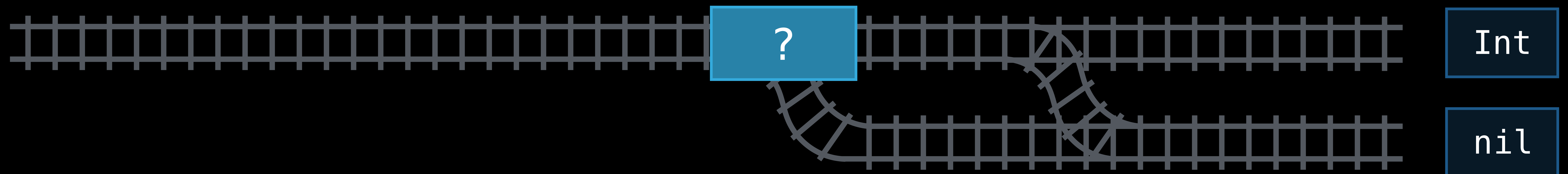
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



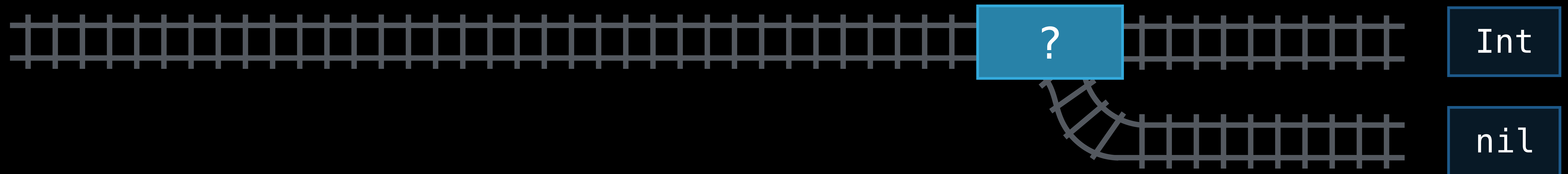
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



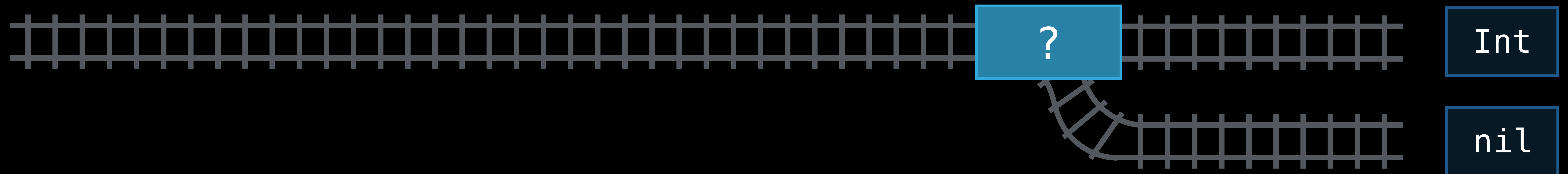
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



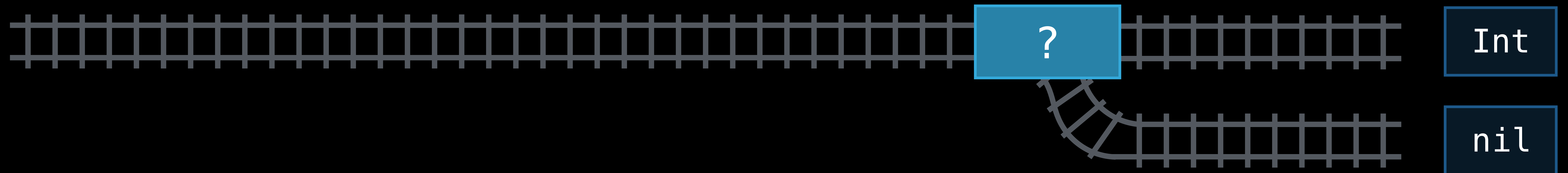
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



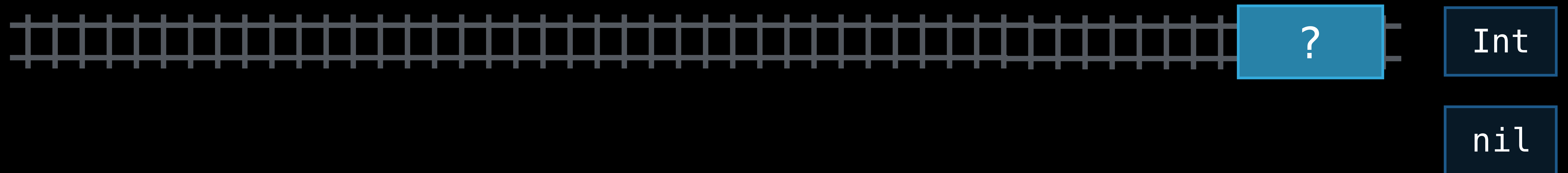
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



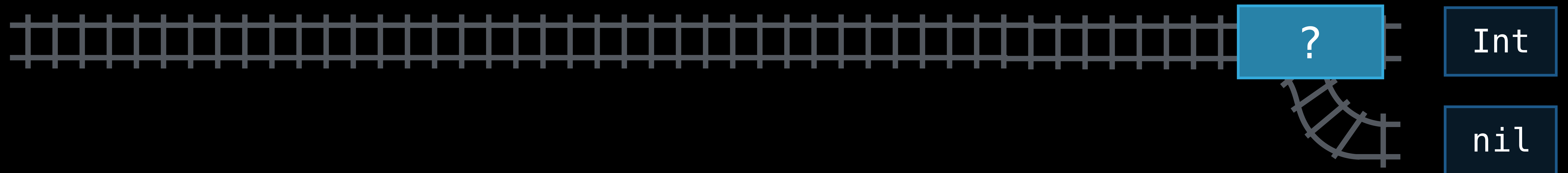
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



Optional Chaining

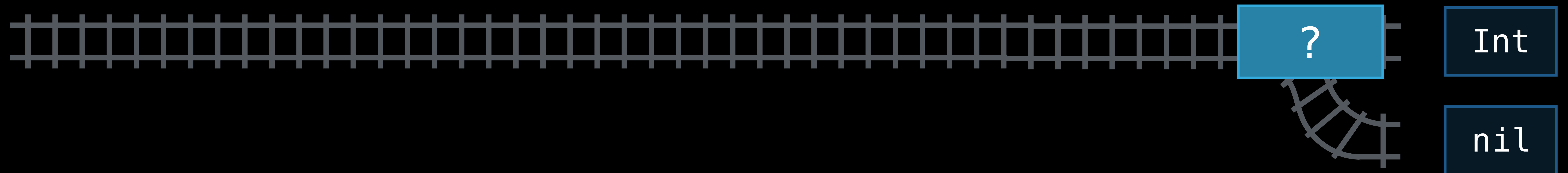
```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



Optional Chaining

"243"

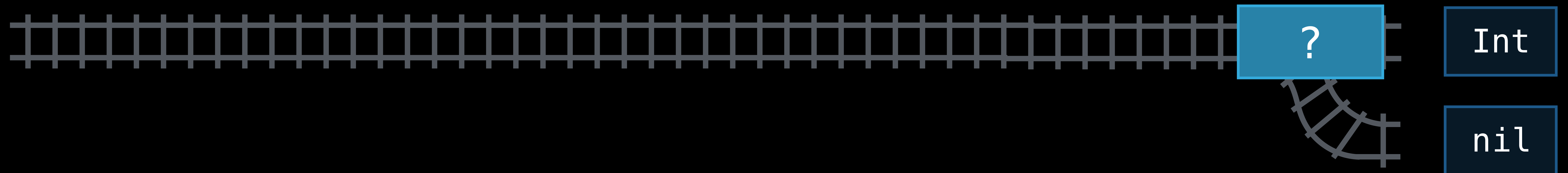
```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



Optional Chaining

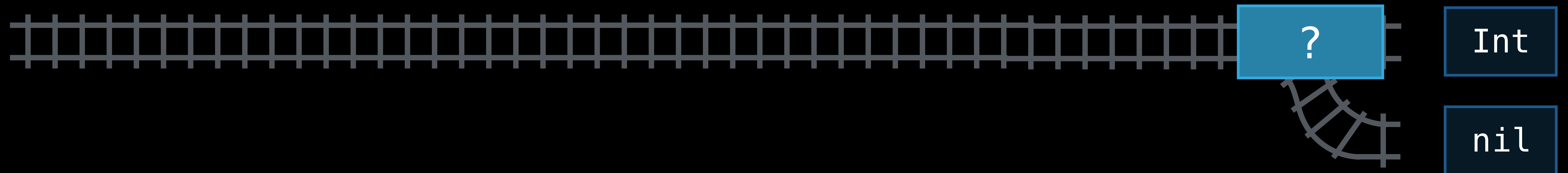
243

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



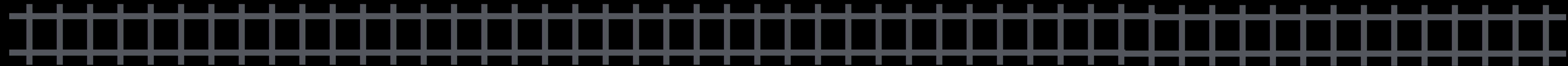
Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```

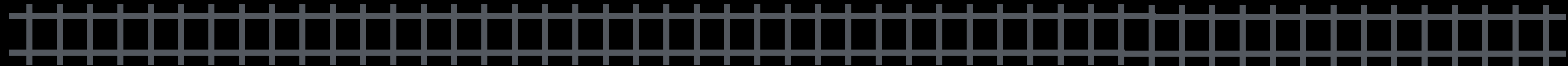


?

nil

Optional Chaining

```
addressNumber = paul.residence?.address?.buildingNumber?.toInt()
```



243

Optional Chaining

Combine with optional binding to unwrap

Optional Chaining

Combine with optional binding to unwrap

```
if let addressNumber = paul.residence?.address?.buildingNumber?.toInt() {
    addToDatabase("Paul", addressNumber)
}
```

Optionals Under the Hood

```
enum Optional<T> {
    case None
    case Some(T)
}
```

Optionals

SAFE

Optionals

SAFE

Use optionals to safely work with possibly missing values

- Missing values are nil
- Present values are wrapped in an optional

Optionals

SAFE

Use optionals to safely work with possibly missing values

- Missing values are nil
- Present values are wrapped in an optional

Unwrap an optional to access its underlying value

- Use the forced-unwrapping operator (`!`) only if you are sure
- Use `if let` optional binding to test and unwrap at the same time

Optionals

SAFE

Use optionals to safely work with possibly missing values

- Missing values are nil
- Present values are wrapped in an optional

Unwrap an optional to access its underlying value

- Use the forced-unwrapping operator (`!`) only if you are sure
- Use `if let` optional binding to test and unwrap at the same time

Optional chaining (`?`) is a concise way to work with chained optionals

Memory Management

Joe Groff
Swift Compiler Engineer

Automatic Reference Counting

Automatic Reference Counting

```
class BowlingPin {}

func juggle(count: Int) {
    var left = BowlingPin()
```

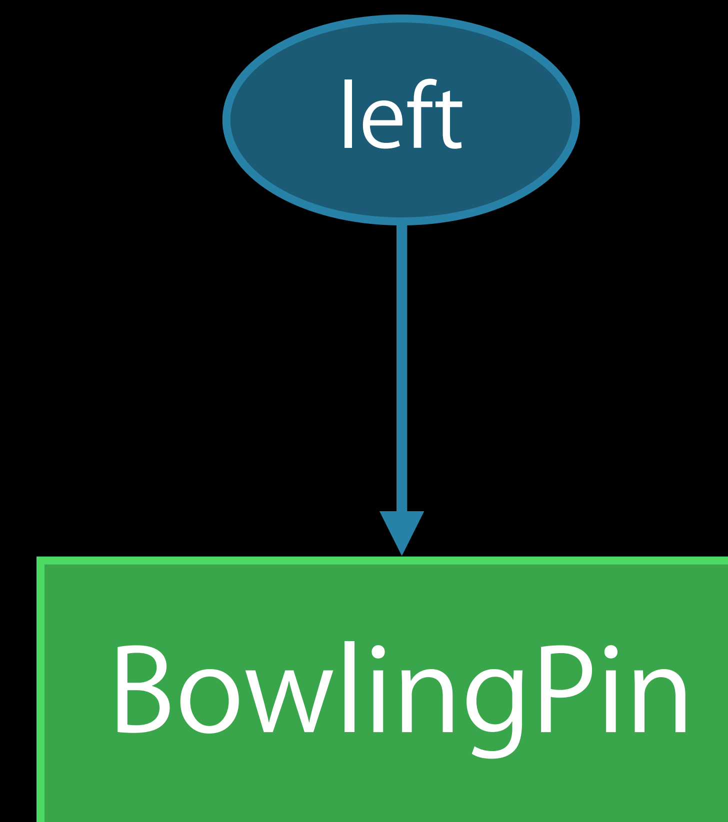
Automatic Reference Counting

```
class BowlingPin {}

func juggle(count: Int) {
    var left = BowlingPin()
```

BowlingPin

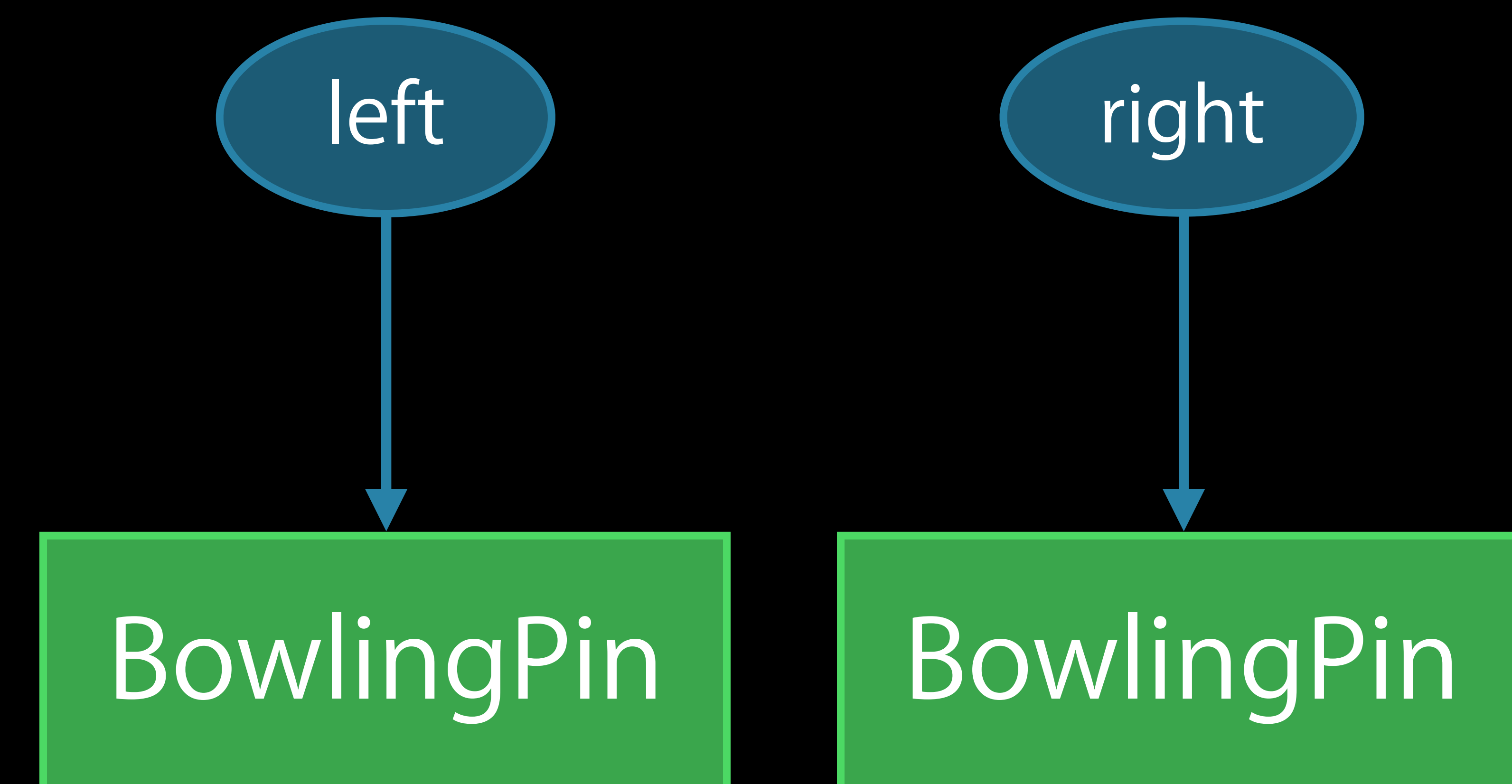
Automatic Reference Counting

```
class BowlingPin {}

func juggle(count: Int) {
    var left = BowlingPin()
```



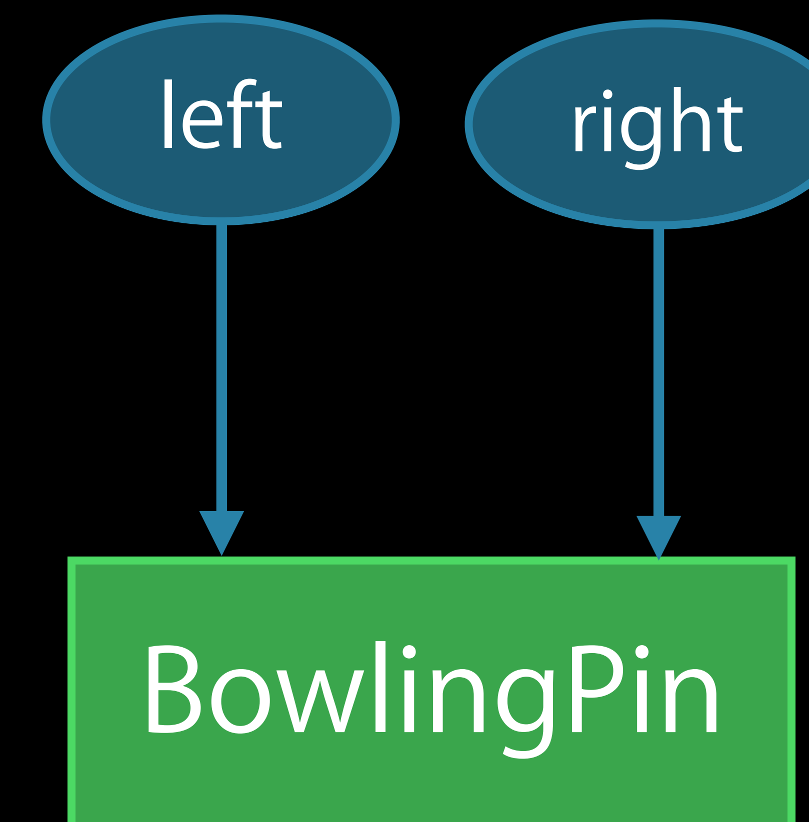
Automatic Reference Counting

```
class BowlingPin {}

func juggle(count: Int) {
    var left = BowlingPin()
    if count > 1 {
        var right = BowlingPin()
```



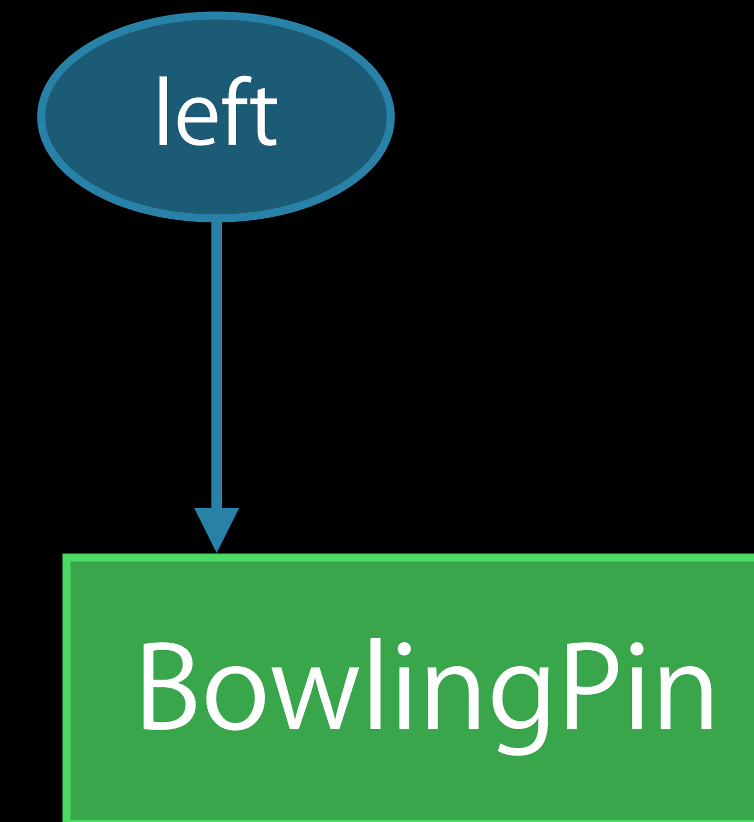
Automatic Reference Counting

```
class BowlingPin {}

func juggle(count: Int) {
    var left = BowlingPin()
    if count > 1 {
        var right = BowlingPin()
        right = left
```



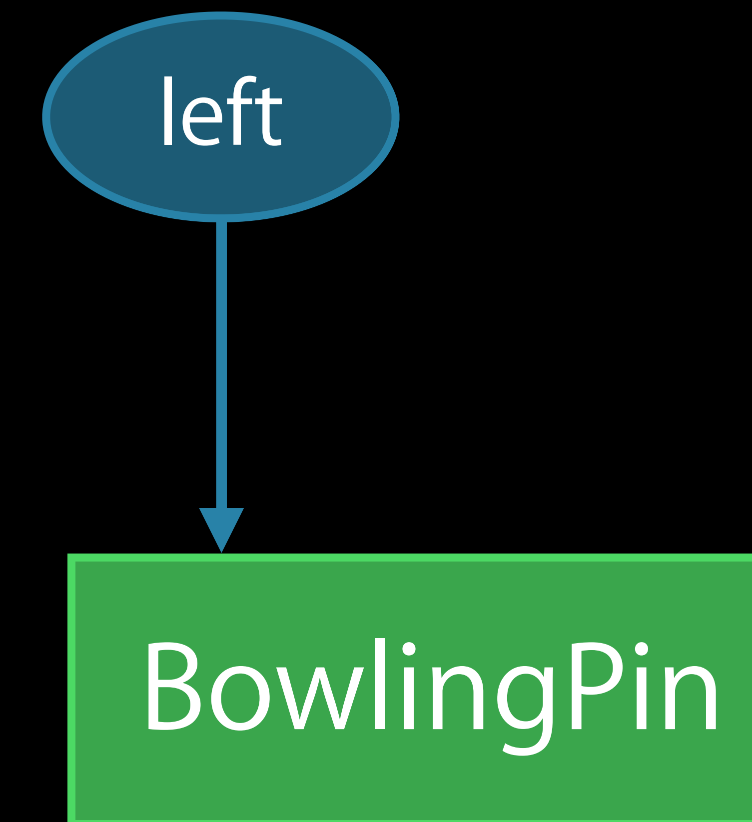
Automatic Reference Counting

```
class BowlingPin {}

func juggle(count: Int) {
    var left = BowlingPin()
    if count > 1 {
        var right = BowlingPin()
        right = left
    } // right goes out of scope
```



Automatic Reference Counting

```
class BowlingPin {}

func juggle(count: Int) {
    var left = BowlingPin()
    if count > 1 {
        var right = BowlingPin()
        right = left
    } // right goes out of scope
```



Automatic Reference Counting

```
class BowlingPin {}

func juggle(count: Int) {
    var left = BowlingPin()
    if count > 1 {
        var right = BowlingPin()
        right = left
    } // right goes out of scope
} // left goes out of scope
```

Ownership

Ownership

```
class Apartment {
    var tenant: Person?
}
```

Ownership

```
class Apartment {
    var tenant: Person?
}
class Person {
    var home: Apartment?
```

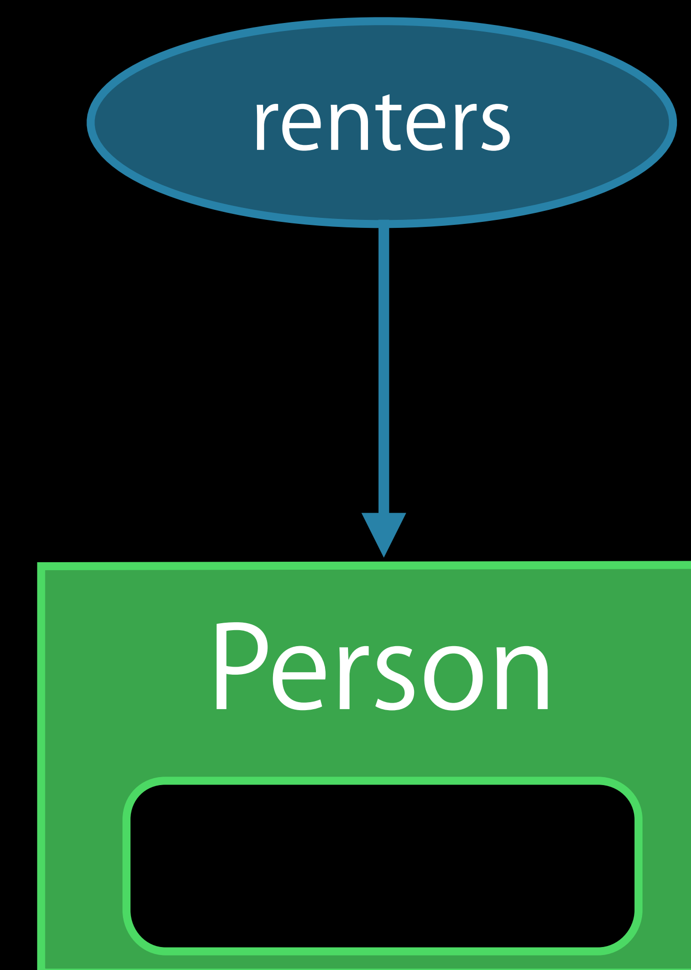
Ownership

```
class Apartment {
    var tenant: Person?
}
class Person {
    var home: Apartment?

    func moveIn(apt: Apartment) {
        self.home = apt
        apt.tenant = self
    }
}
```

Ownership

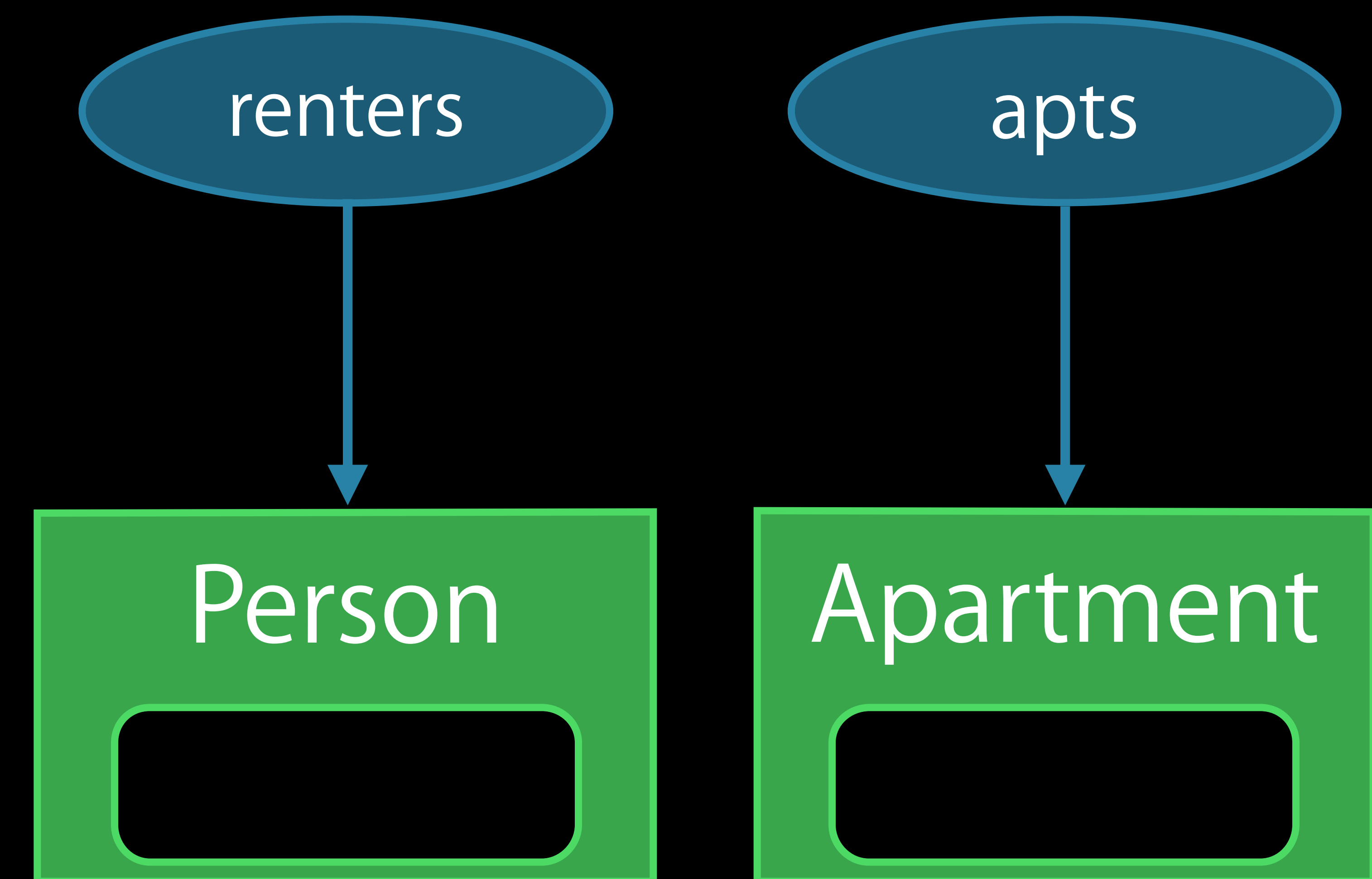
Ownership

```
var renters = ["Elsvette": Person()]
```



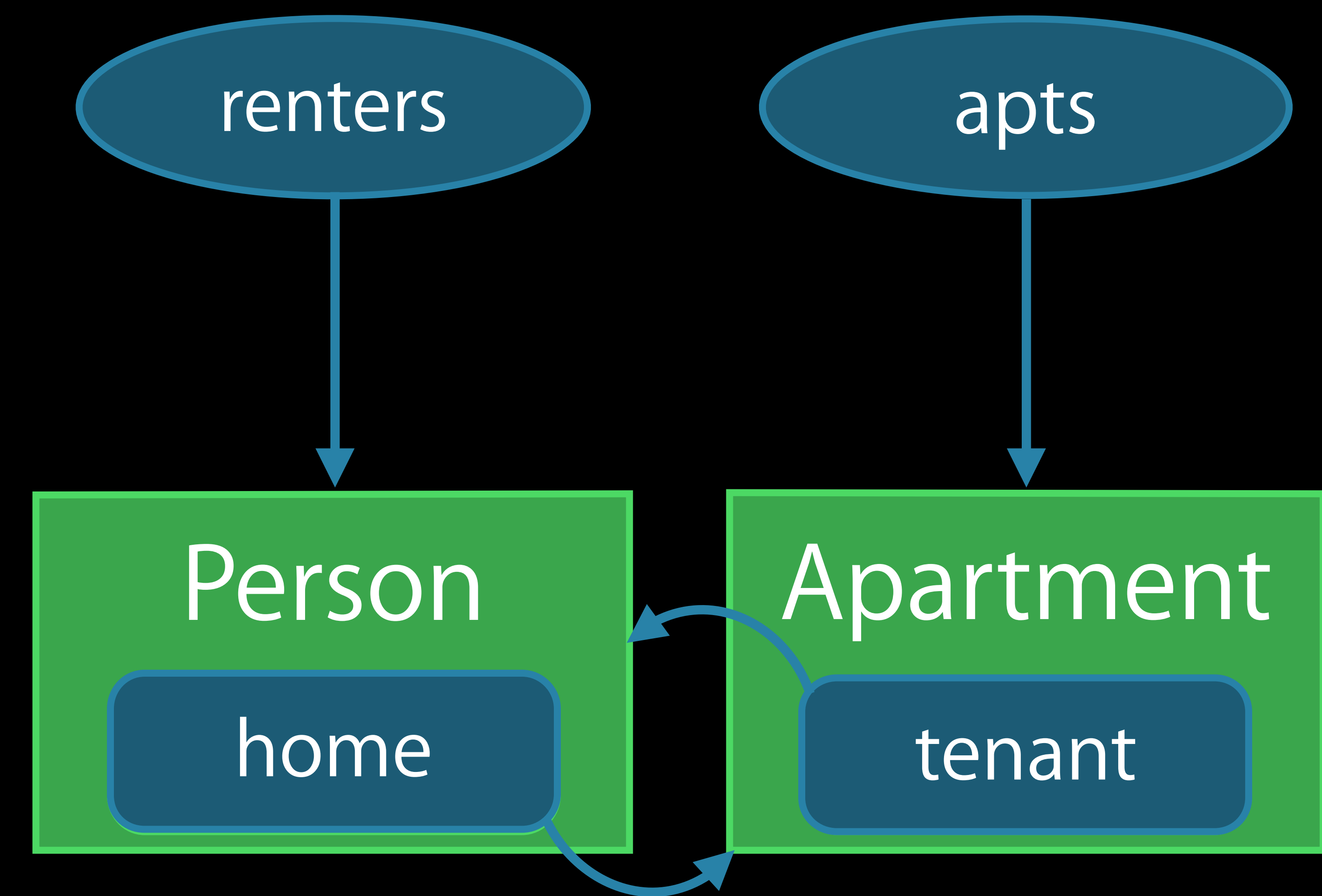
Ownership

```
var renters = ["Elsvette": Person()]
var apts = [507: Apartment()]
```



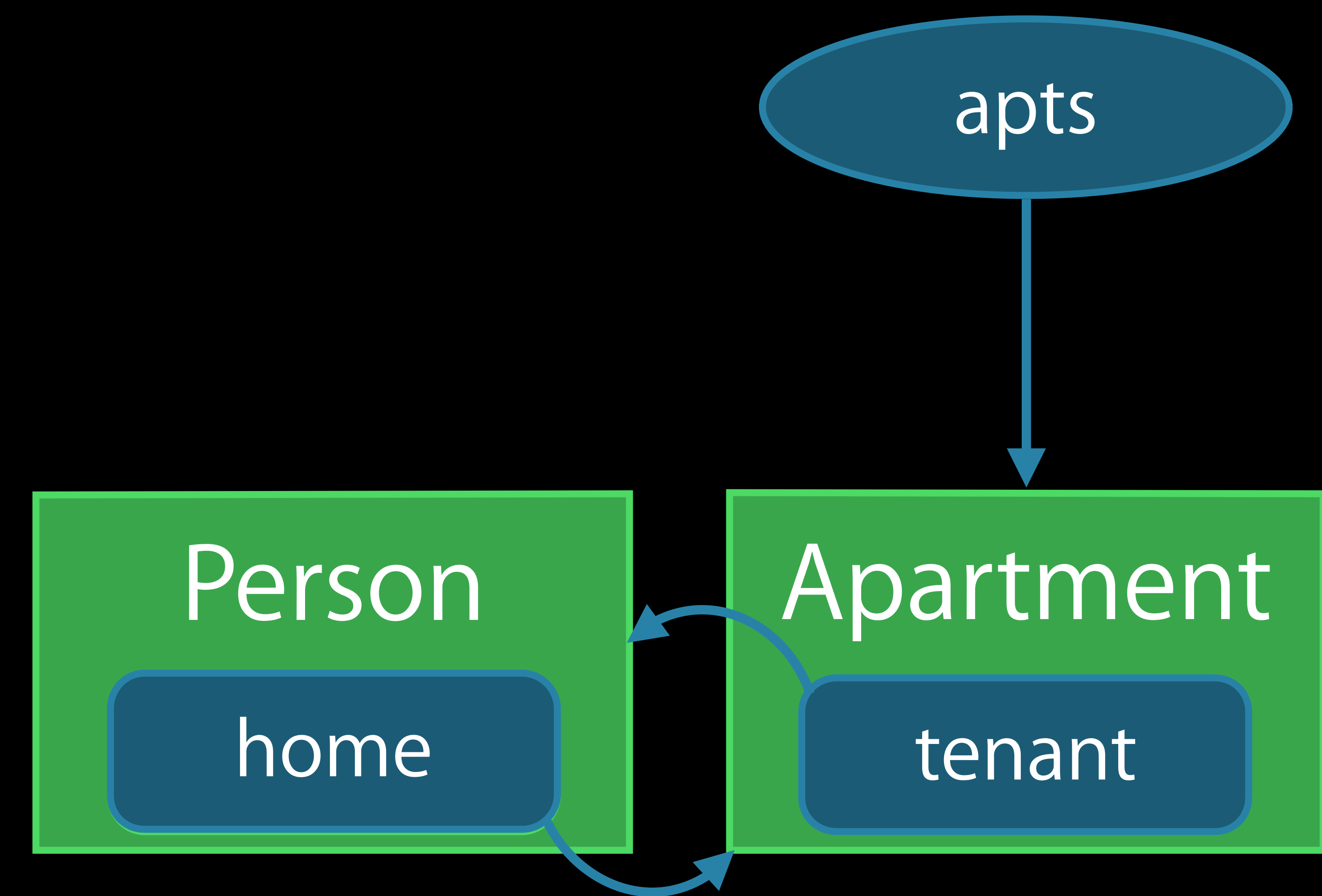
Ownership

```
var renters = ["Elsvette": Person()]
var apts = [507: Apartment()]
renters["Elsvette"]!.moveIn(apts[507]!)
```



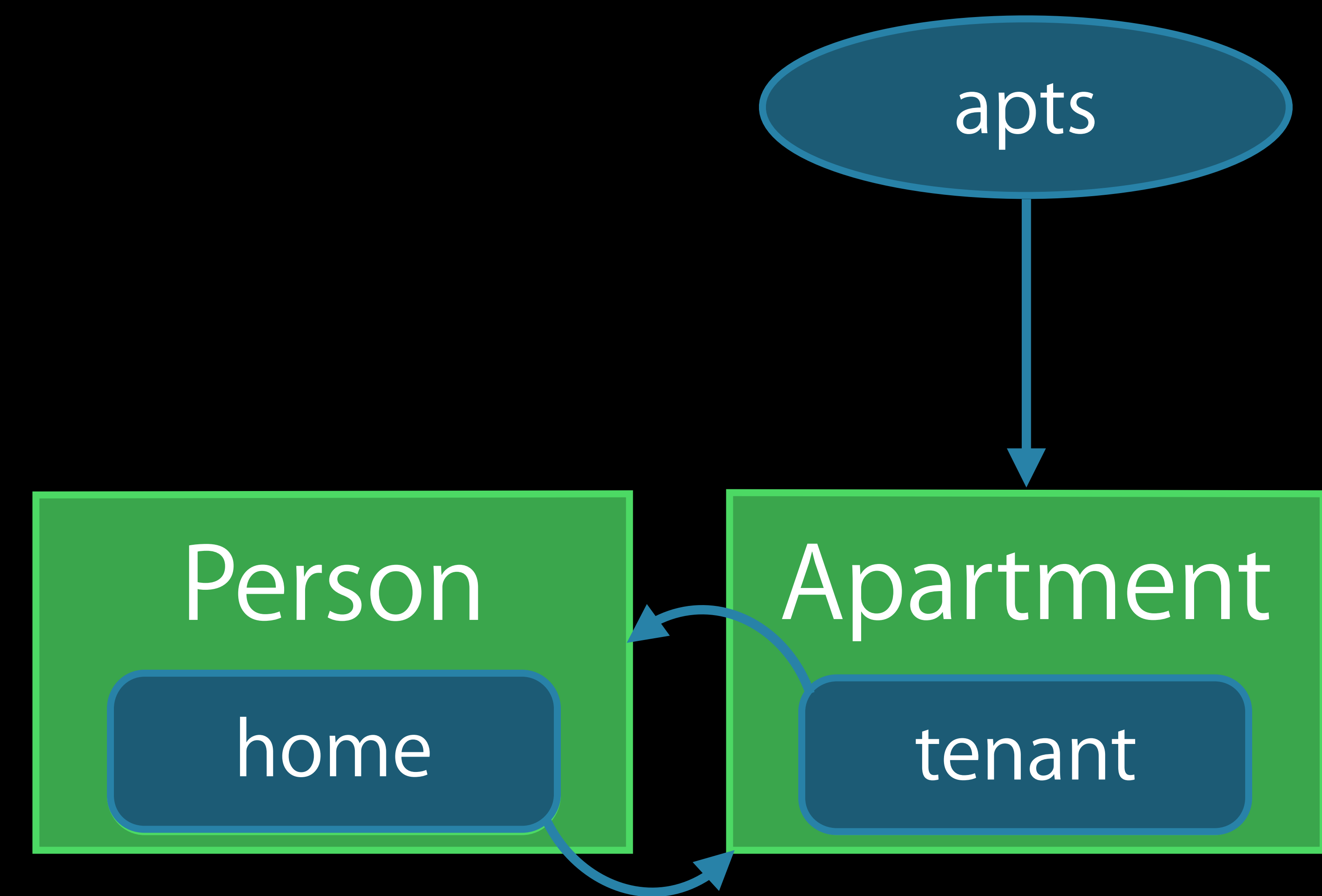
Ownership

```
var renters = ["Elsvette": Person()]
var apts = [507: Apartment()]
renters["Elsvette"]!.moveIn(apts[507]!)

renters["Elsvette"] = nil
```



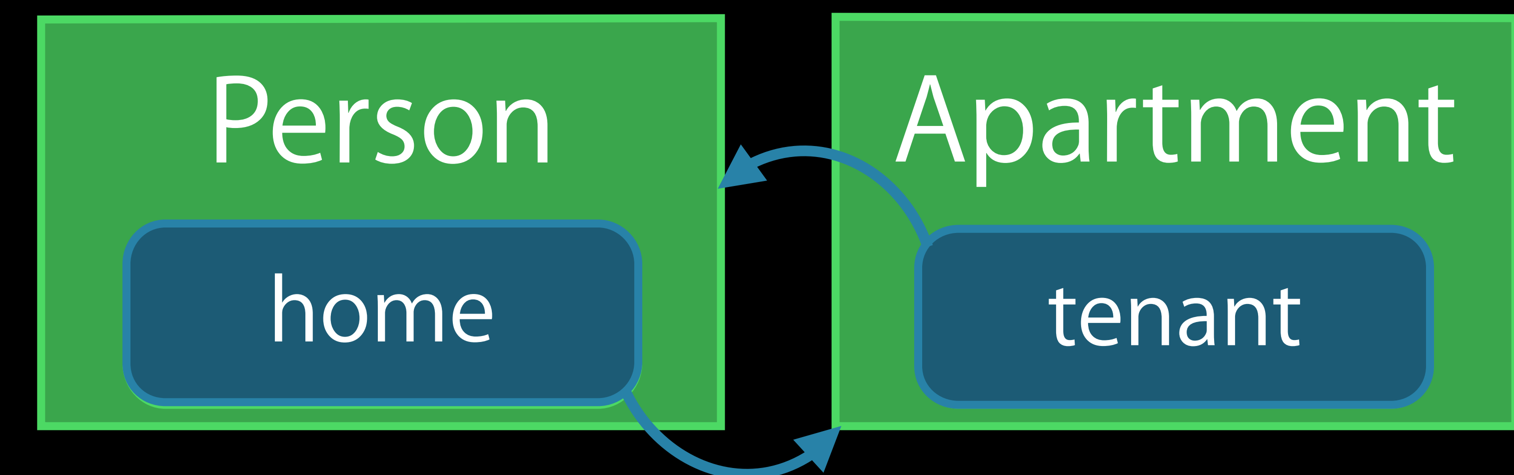
Ownership

```
var renters = ["Elsvette": Person()]
var apts = [507: Apartment()]
renters["Elsvette"]!.moveIn(apts[507]!)

renters["Elsvette"] = nil
```



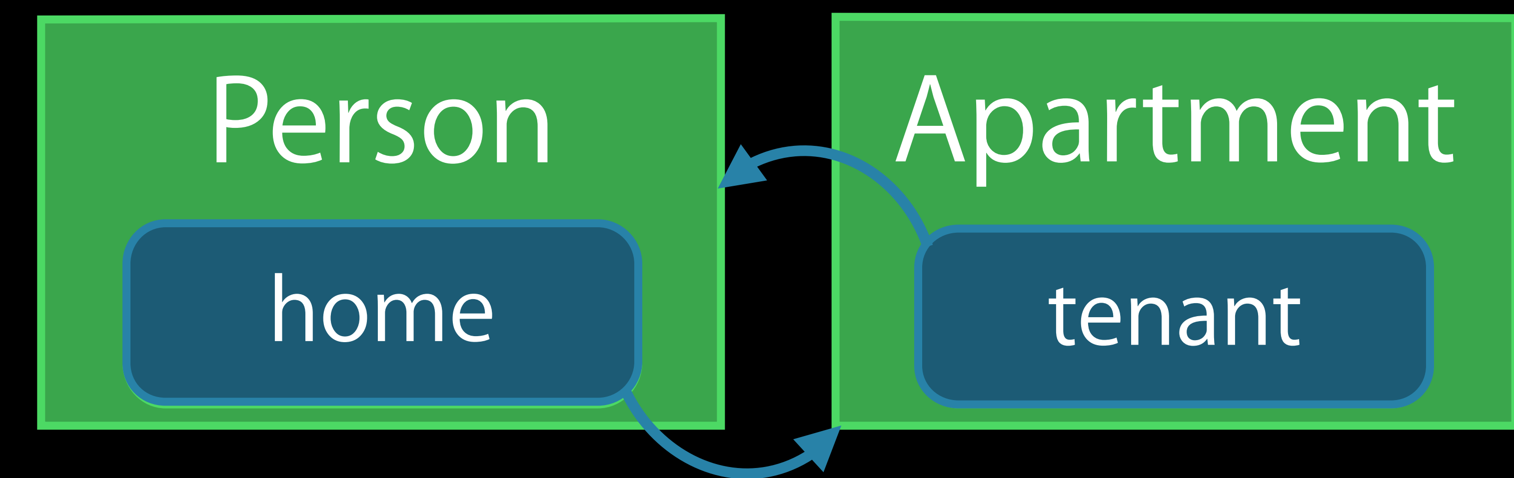
Ownership

```
var renters = ["Elsvette": Person()]
var apts = [507: Apartment()]
renters["Elsvette"]!.moveIn(apts[507]!)

renters["Elsvette"] = nil
apts[507] = nil
```



Ownership

```
var renters = ["Elsvette": Person()]
var apts = [507: Apartment()]
renters["Elsvette"]!.moveIn(apts[507]!)

renters["Elsvette"] = nil
apts[507] = nil
```



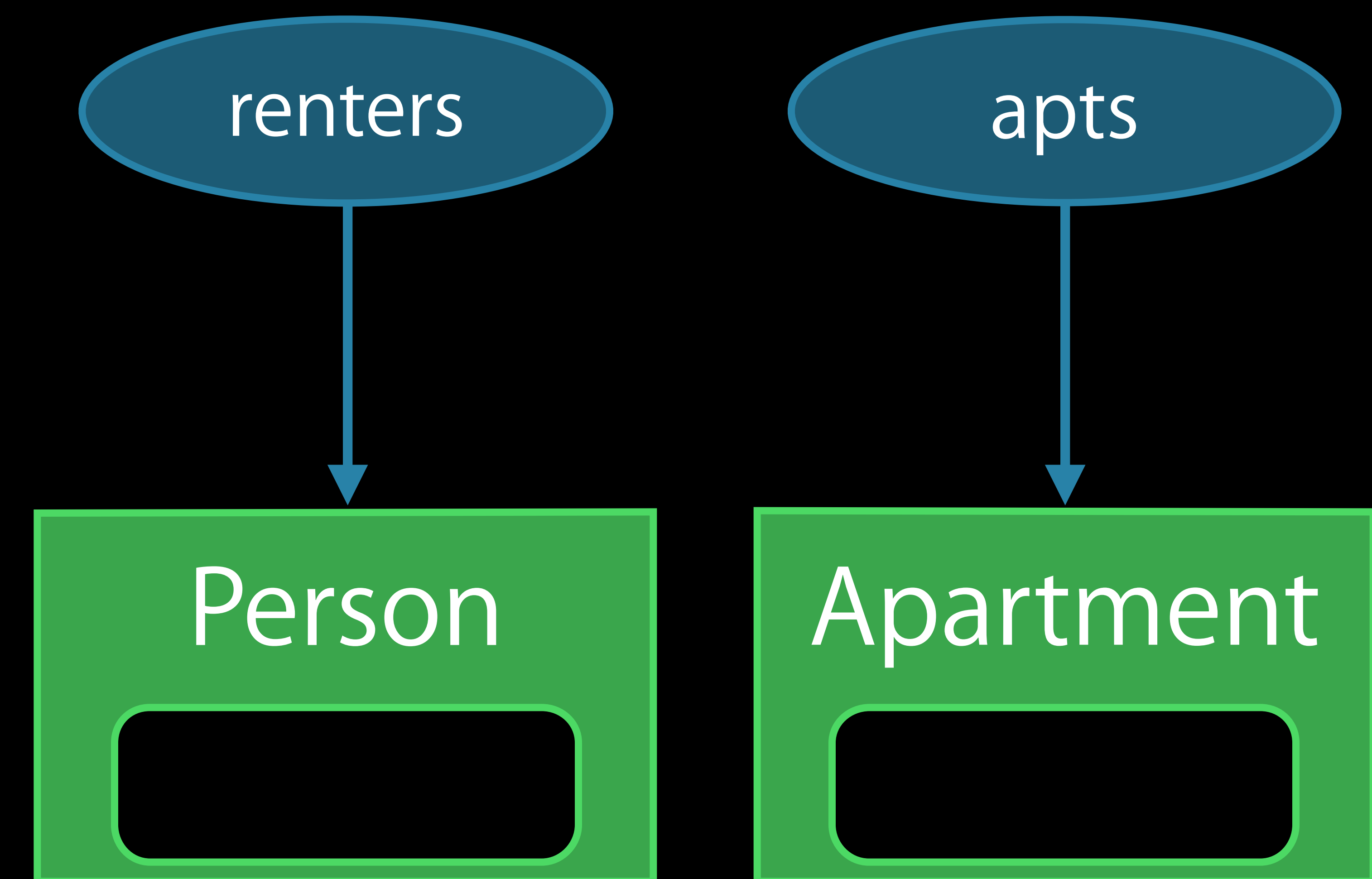
Weak References

```
class Apartment {
    var tenant: Person?
}
class Person {
    var home: Apartment?

    func moveIn(apt: Apartment) {
        self.home = apt
        apt.tenant = self
    }
}
```

Weak References

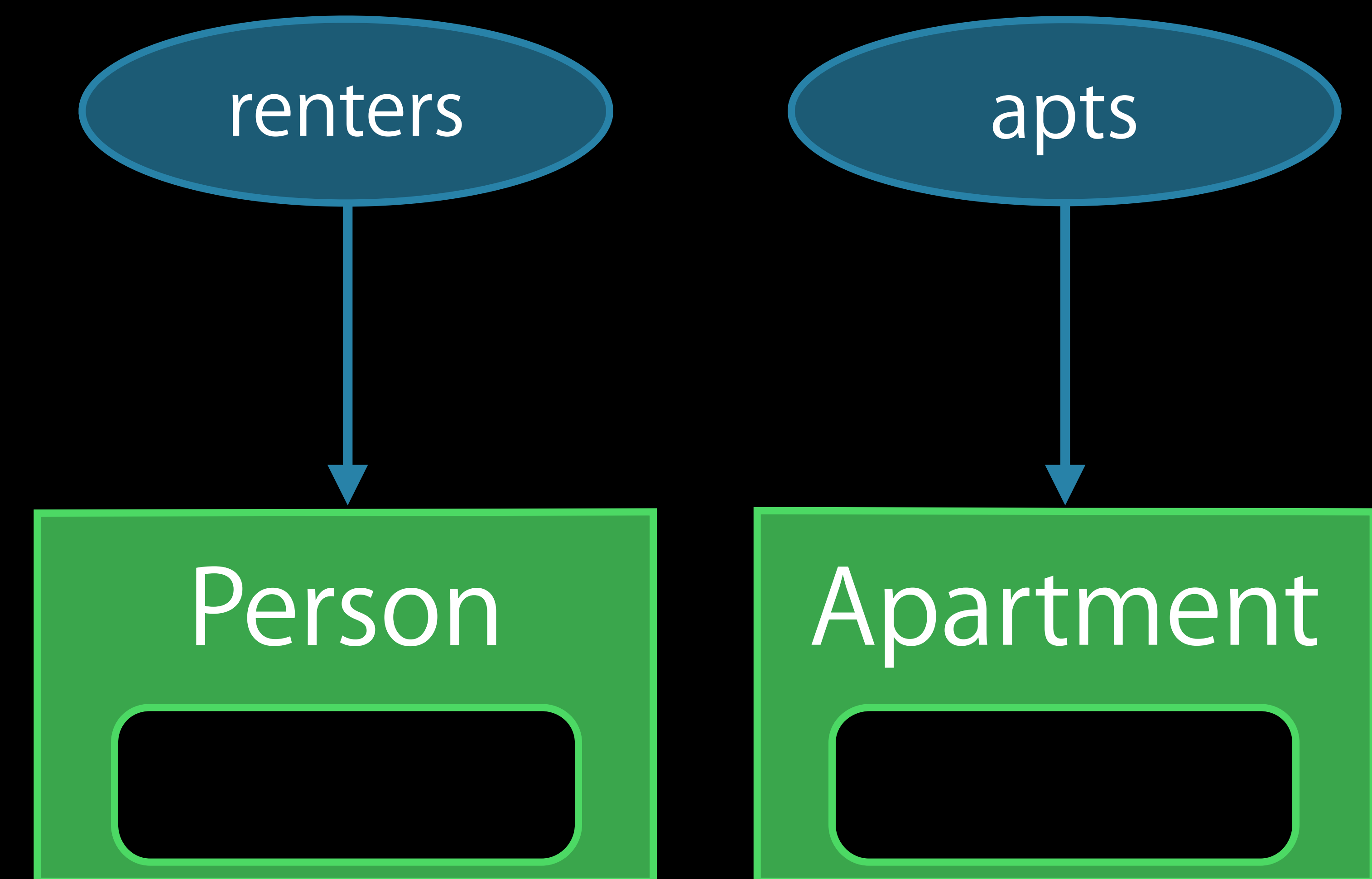
```
class Apartment {
    weak var tenant: Person?
}
class Person {
    weak var home: Apartment?

    func moveIn(apt: Apartment) {
        self.home = apt
        apt.tenant = self
    }
}
```

Weak References



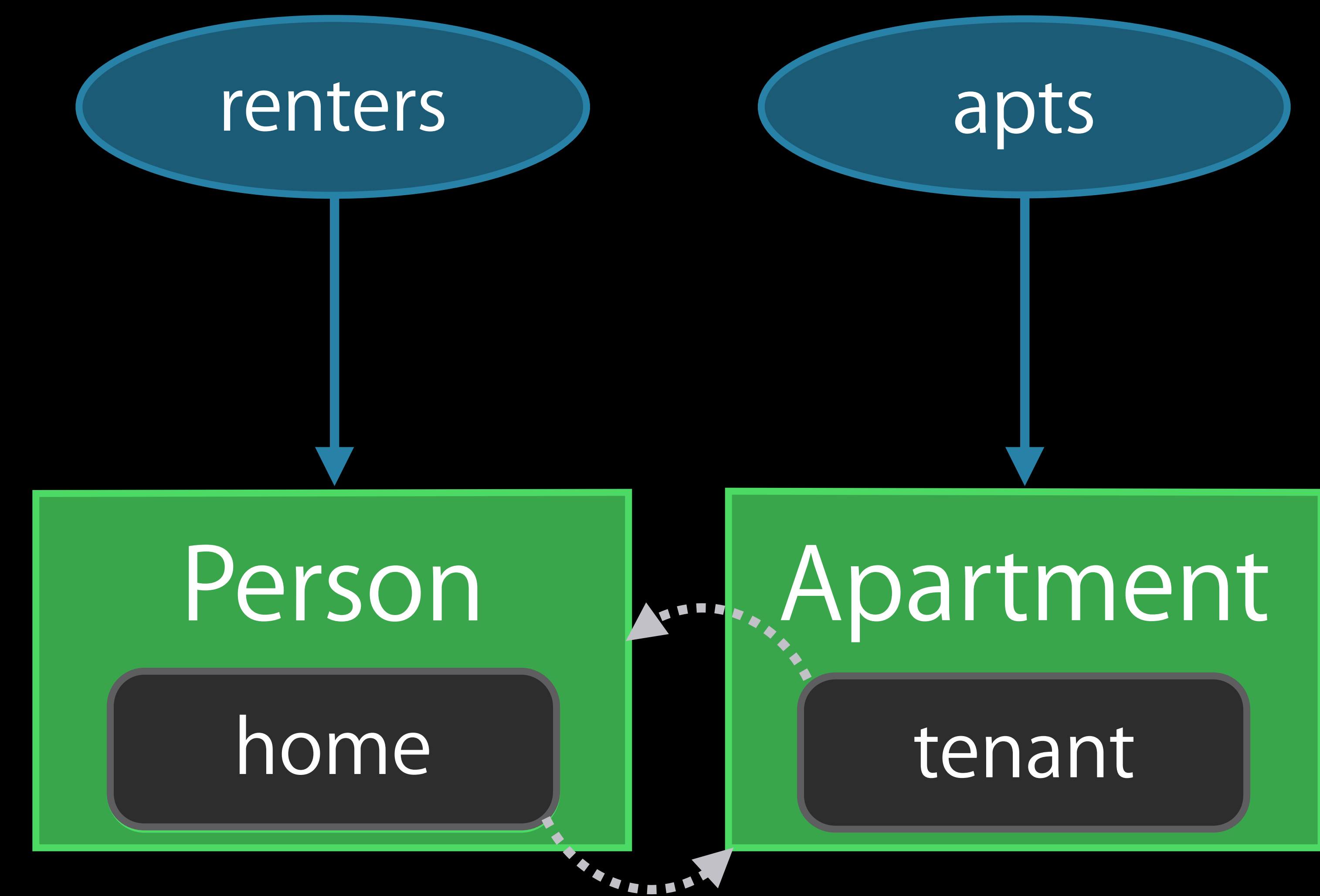
Weak References

```
var renters = ["Elsvette": Person()]
var apts = [507: Apartment()]
```



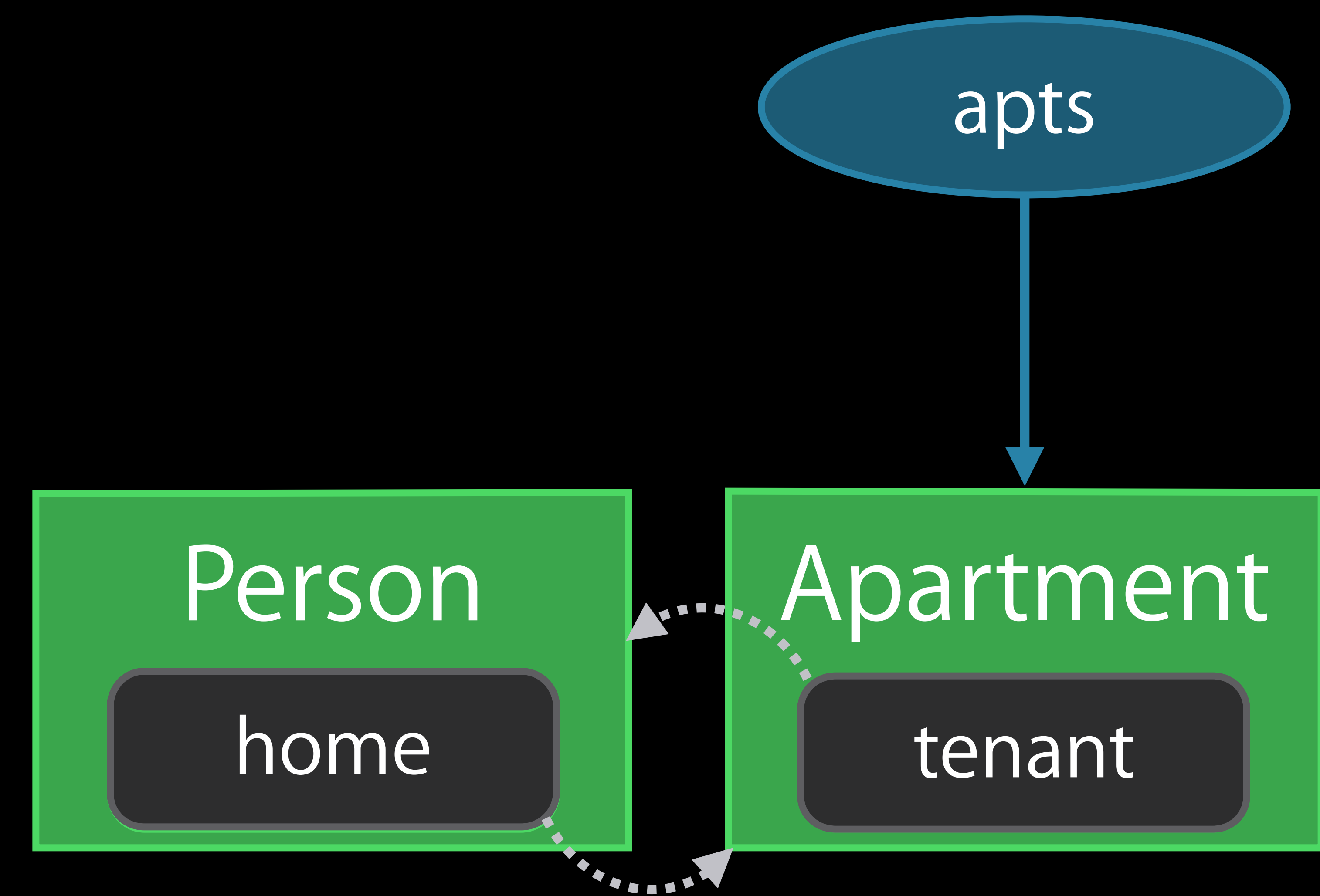
Weak References

```
var renters = ["Elsvette": Person()]
var apts = [507: Apartment()]
renters["Elsvette"]!.moveIn(apts[507]!)
```



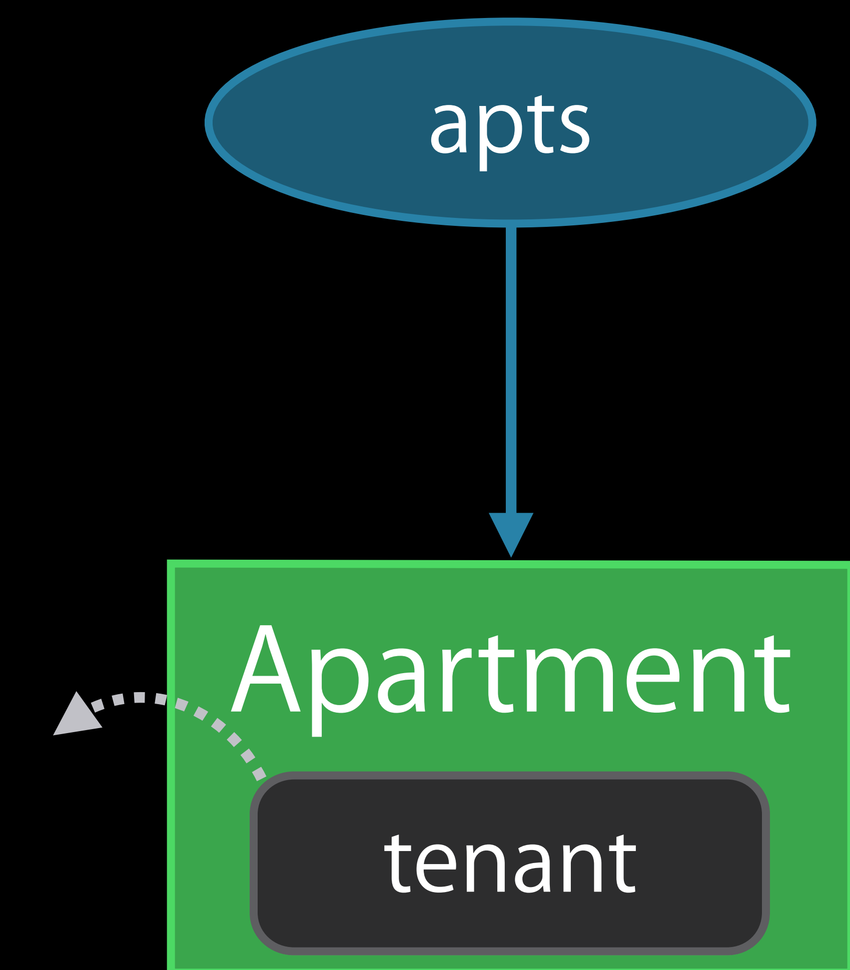
Weak References

```
var renters = ["Elsvette": Person()]
var apts = [507: Apartment()]
renters["Elsvette"]!.moveIn(apts[507]!)

renters["Elsvette"] = nil
```



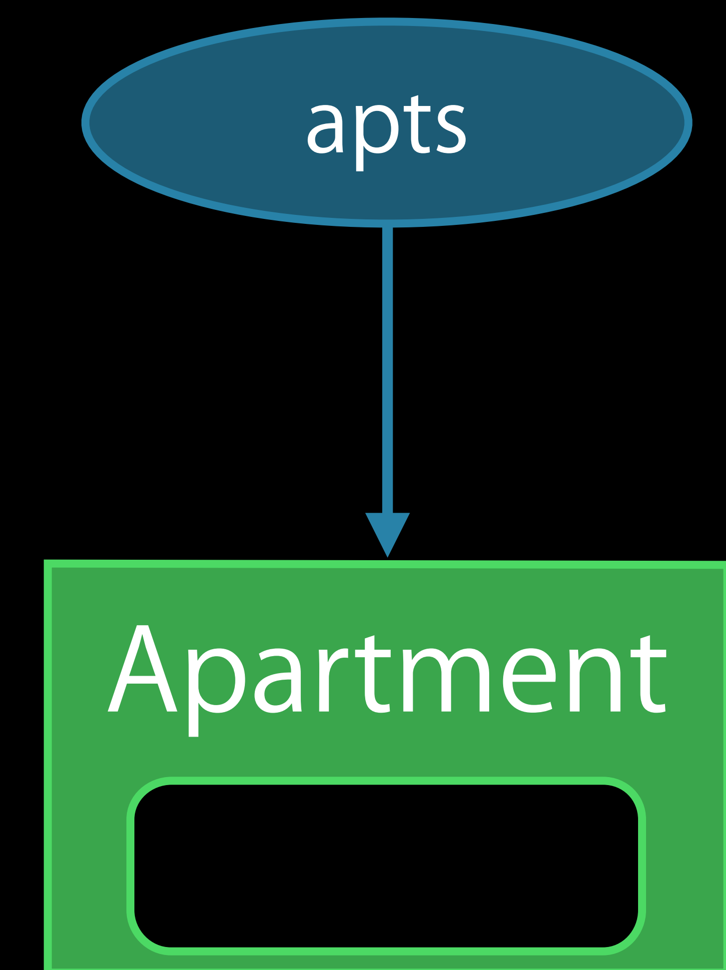
Weak References

```
var renters = ["Elsvette": Person()]
var apts = [507: Apartment()]
renters["Elsvette"]!.moveIn(apts[507]!)

renters["Elsvette"] = nil
```



Weak References

```
var renters = ["Elsvette": Person()]
var apts = [507: Apartment()]
renters["Elsvette"]!.moveIn(apts[507]!)

renters["Elsvette"] = nil
```



Weak References

```
var renters = ["Elsvette": Person()]
var apts = [507: Apartment()]
renters["Elsvette"]!.moveIn(apts[507]!)

renters["Elsvette"] = nil
apts[507] = nil
```

Using Weak References

Using Weak References

Weak references are optional values

Using Weak References

Weak references are optional values

Binding the optional produces a strong reference

```
if let tenant = apt.tenant {
    tenant.buzzIn()
}
```

Using Weak References

Weak references are optional values

Binding the optional produces a strong reference

```
if let tenant = apt.tenant {
    tenant.buzzIn()
}

apt.tenant?.buzzIn()
```

Using Weak References

Using Weak References

Testing a weak reference alone does not produce a strong reference

```
if apt.tenant {
    apt.tenant!.cashRentCheck()
    apt.tenant!.greet()
}
```

Using Weak References

Testing a weak reference alone does not produce a strong reference

Chaining doesn't preserve a strong reference between method invocations

```
if apt.tenant {
    apt.tenant!.cashRentCheck()
    apt.tenant!.greet()
}

apt.tenant?.cashRentCheck()
apt.tenant?.greet()
```

Same-Lifetime Relationships

Same-Lifetime Relationships

```
class Person {
    var card: CreditCard?
}
```

Same-Lifetime Relationships

```
class Person {
    var card: CreditCard?
}
class CreditCard {
    let holder: Person
```

Same-Lifetime Relationships

```
class Person {
    var card: CreditCard?
}
class CreditCard {
    let holder: Person
```

Same-Lifetime Relationships

```
class Person {
    var card: CreditCard?
}
class CreditCard {
    let holder: Person

    init(holder: Person) {
        self.holder = holder
    }
}
```

Same-Lifetime Relationships

```
class Person {
    var card: CreditCard?
}
class CreditCard {
    let holder: Person

    init(holder: Person) {
        self.holder = holder
    }
}
```

Same-Lifetime Relationships

```
class Person {
    var card: CreditCard?
}
class CreditCard {
    weak let holder: Person

    init(holder: Person) {
        self.holder = holder
    }
}
```

Same-Lifetime Relationships

```
class Person {
    var card: CreditCard?
}
class CreditCard {
    weak let holder: Person?

    init(holder: Person) {
        self.holder = holder
    }
}
```

Same-Lifetime Relationships

```
class Person {
    var card: CreditCard?
}
class CreditCard {
    weak var holder: Person?

    init(holder: Person) {
        self.holder = holder
    }
}
```

Unowned References

```
class Person {
    var card: CreditCard?
}
class CreditCard {
    let holder: Person

    init(holder: Person) {
        self.holder = holder
    }
}
```

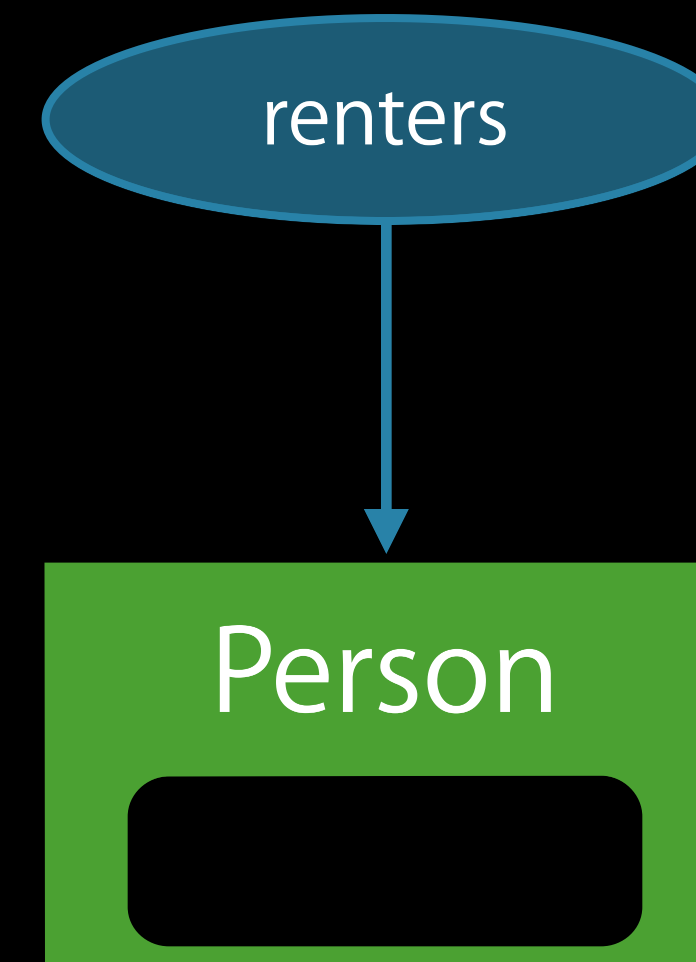
Unowned References

```
class Person {
    var card: CreditCard?
}
class CreditCard {
    unowned let holder: Person

    init(holder: Person) {
        self.holder = holder
    }
}
```

Unowned References

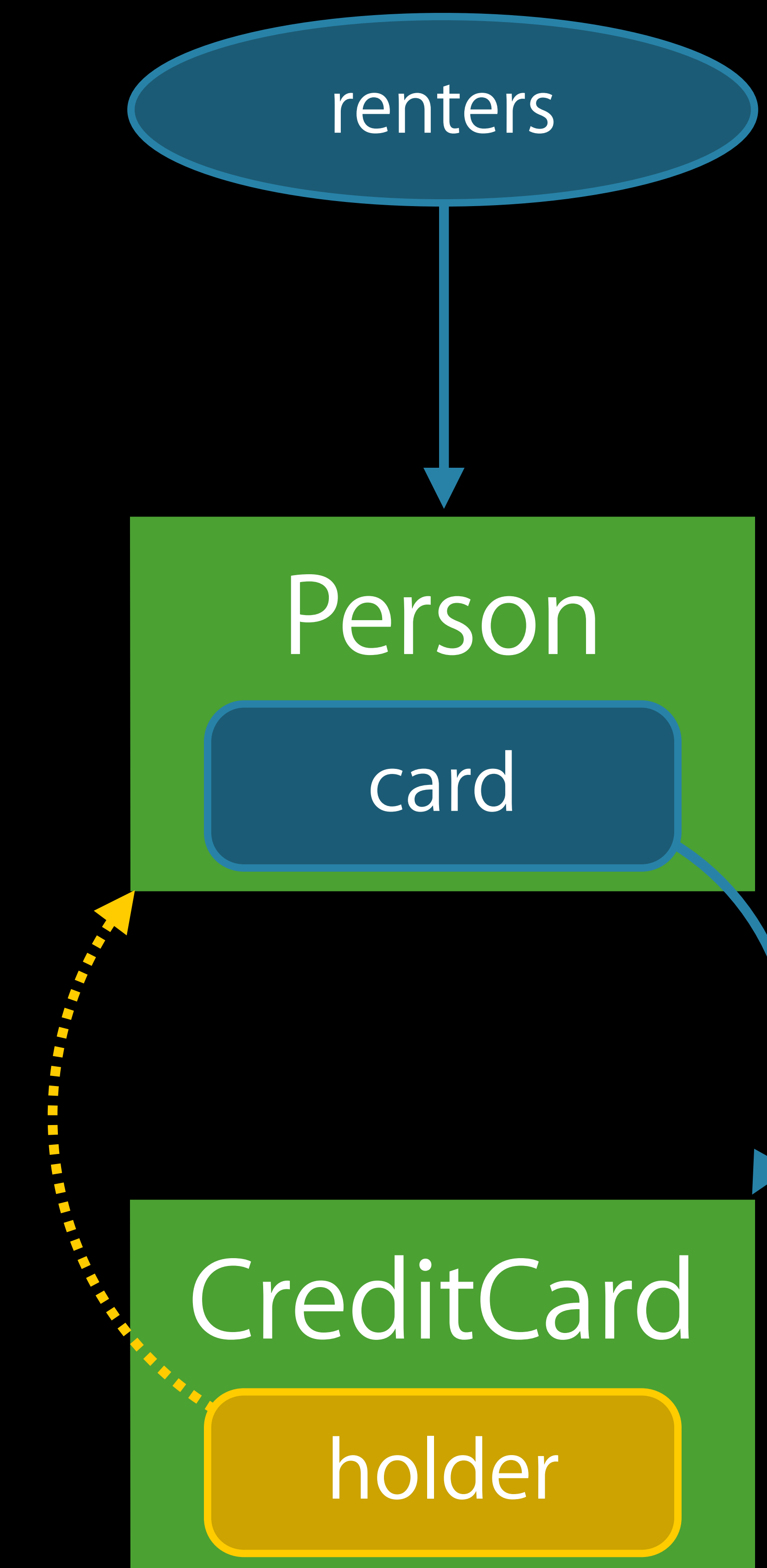
Unowned References

```
var renters = ["Elsvette": Person()]
```



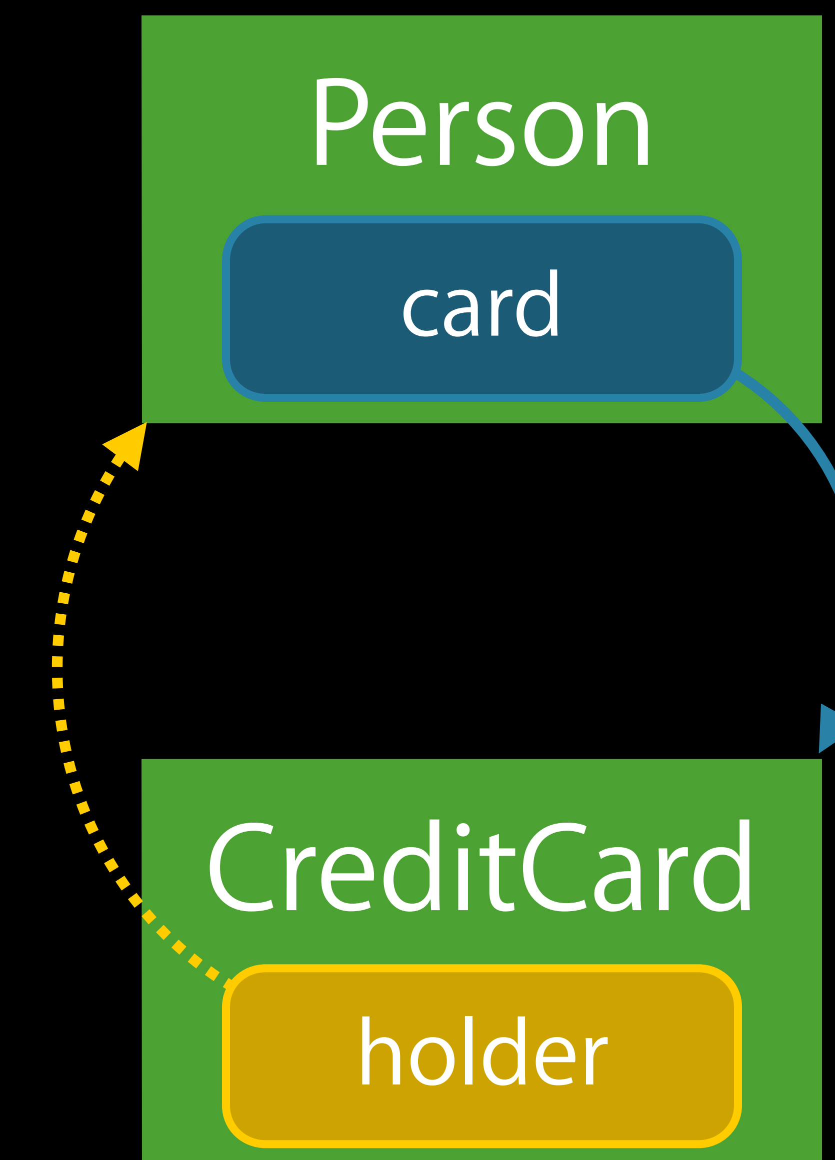
Unowned References

```
var renters = ["Elsvette": Person()]
customers["Elsvette"]!.saveCard()
```



Unowned References

```
var renters = ["Elsvette": Person()]
customers["Elsvette"]!.saveCard()
customers["Elsvette"] = nil
```



Unowned References

```
var renters = ["Elsvette": Person()]
customers["Elsvette"]!.saveCard()
customers["Elsvette"] = nil
```

Using Unowned References

Using Unowned References

```
let holder = card.holder
```

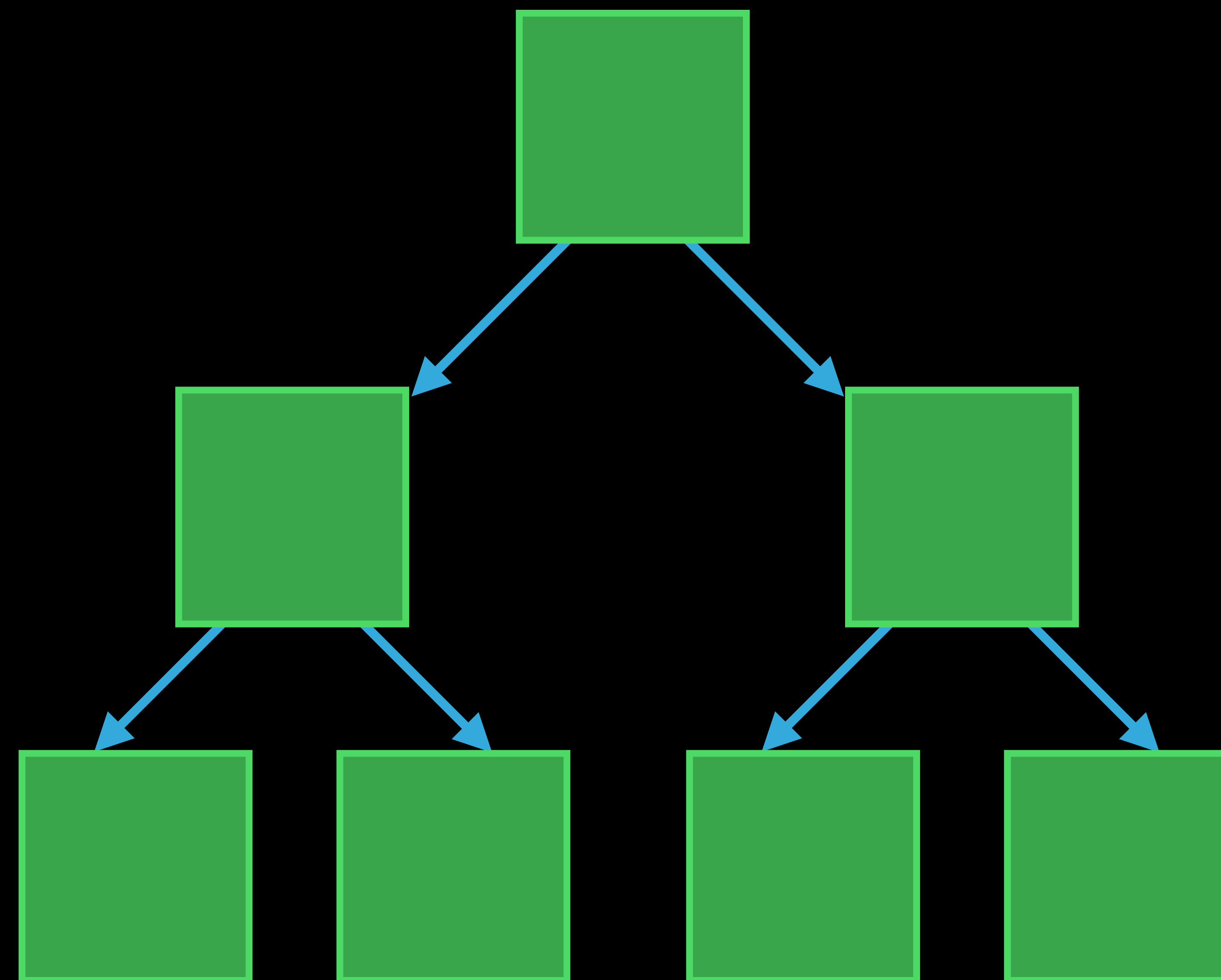
Using Unowned References

```
let holder = card.holder
card.holder.charge(2.19)
```

Strong, Weak, and Unowned References

Strong, Weak, and Unowned References

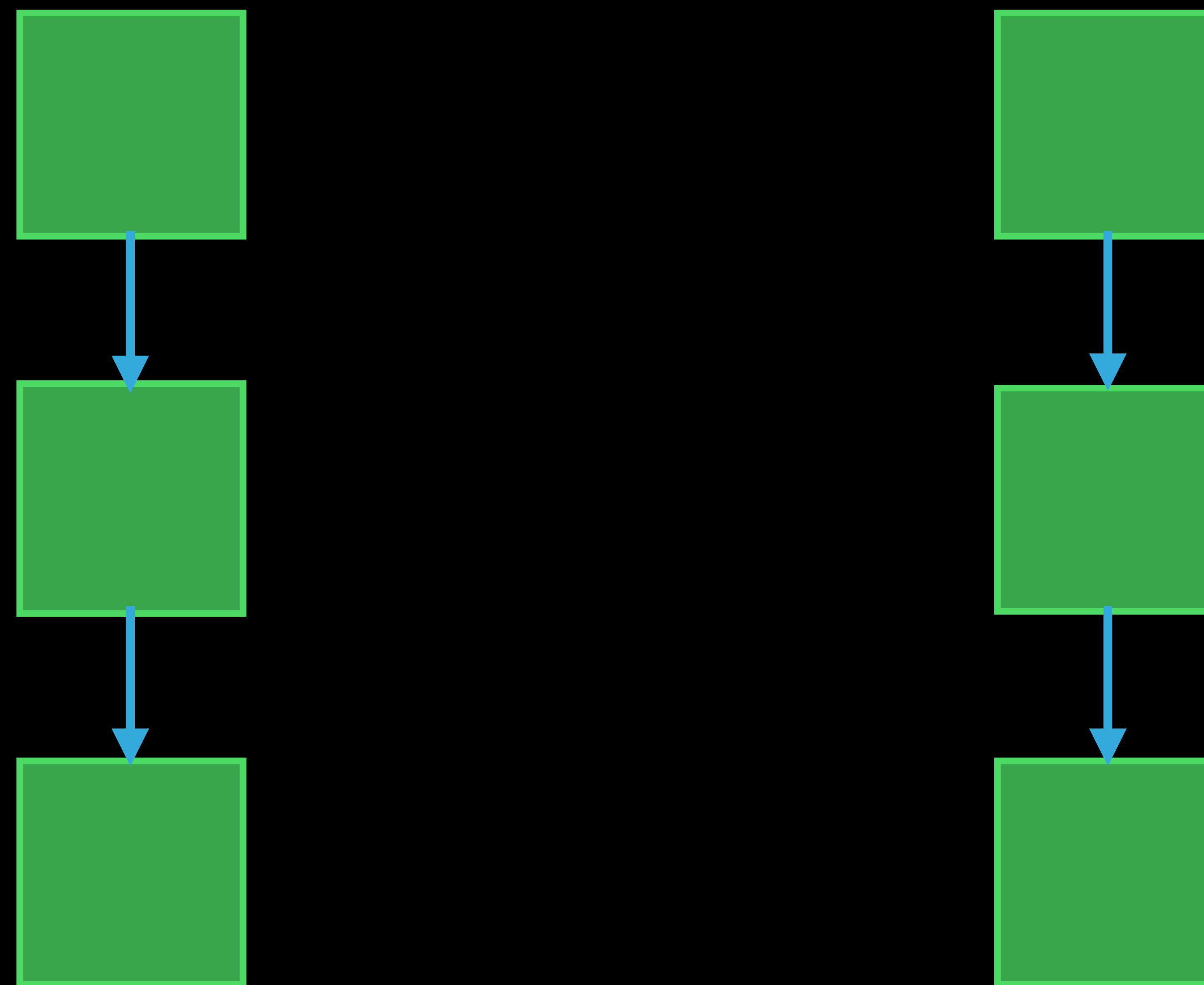
Use strong references from owners to the objects they own



Strong, Weak, and Unowned References

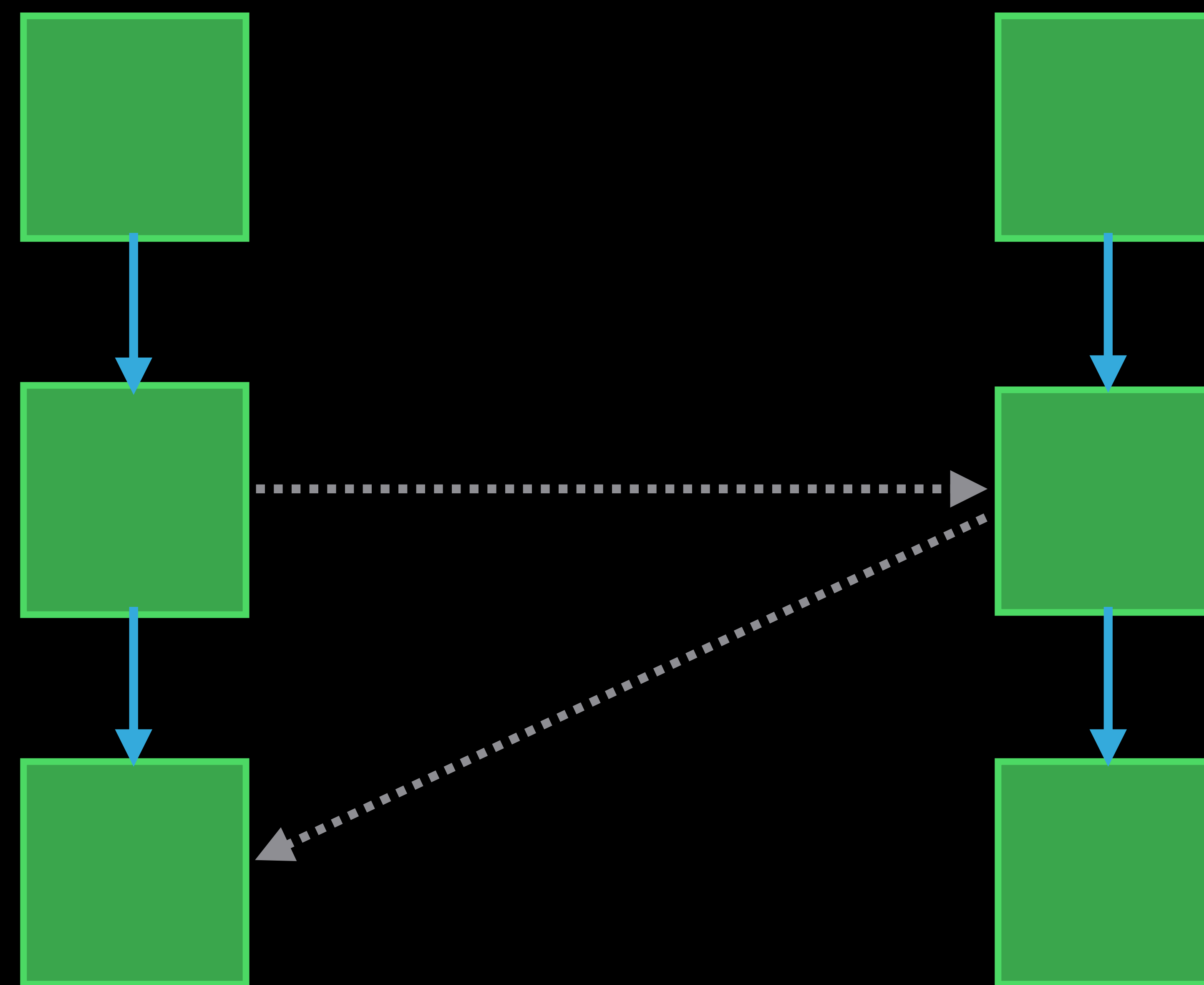
Strong, Weak, and Unowned References

Use `weak` references among objects with independent lifetimes



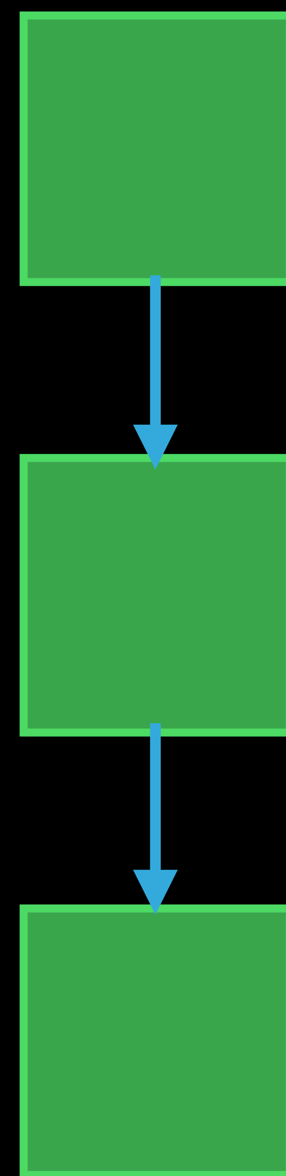
Strong, Weak, and Unowned References

Use `weak` references among objects with independent lifetimes



Strong, Weak, and Unowned References

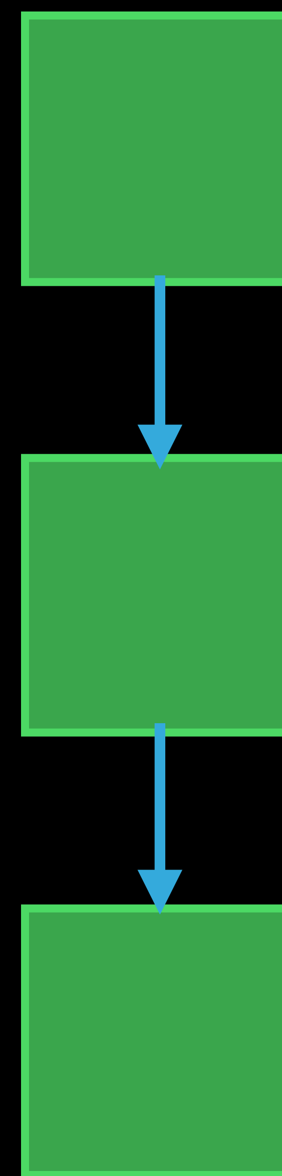
Use `weak` references among objects with independent lifetimes



Strong, Weak, and Unowned References

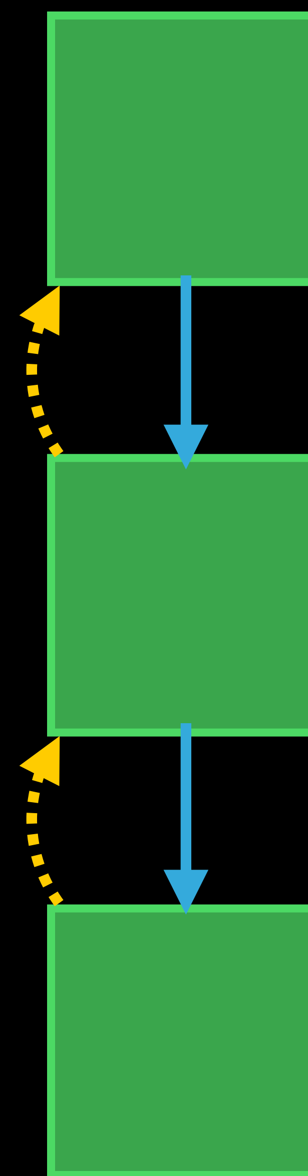
Strong, Weak, and Unowned References

Use `unowned` references from owned objects with the same lifetime



Strong, Weak, and Unowned References

Use unowned references from owned objects with the same lifetime



Strong, Weak, and Unowned References

Use `unowned` references from owned objects with the same lifetime

Memory Management

Automatic, safe, predictable

Think about relationships between objects

Model ownership using strong, weak, and unowned references

Initialization

Brian Lanier
Developer Publications Engineer

Every value **must** be initialized before it is used

SAFE

Every value **must** be initialized before it is used

Initialization

```
var message: String
```

Initialization

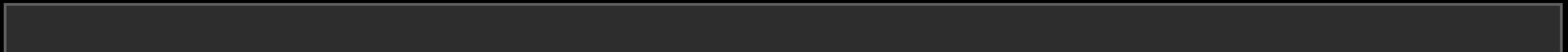
```
var message: String

if sessionStarted {
    message = "Welcome to Intermediate Swift!"
}

println(message)
```

Initialization

```
var message: String

if sessionStarted {
    message = "Welcome to Intermediate Swift!"
}

println(message)
// error: variable 'message' used before being initialized
```

Initialization

```
var message: String

if sessionStarted {
    message = "Welcome to Intermediate Swift!"
} else {
    message = "See you next year!"
}

println(message)
```



Initialization

```
var message: String

if sessionStarted {
    message = "Welcome to Intermediate Swift!"
} else {
    message = "See you next year!"
}

println(message)
```

```
Welcome to Intermediate Swift!
```

Initializers

Initializers

Initializers handle the responsibility of fully initializing an instance

Initializers

Initializers handle the responsibility of fully initializing an instance

```
init() {…}
```

Initializers

Initializers handle the responsibility of fully initializing an instance

```
init() {…}

let instance = MyClass()
```

Initializers

Initializers handle the responsibility of fully initializing an instance

```
init() {…}

let instance = MyClass()
```

Initializers

Initializers handle the responsibility of fully initializing an instance

```
init(luckyNumber: Int, message: String) {…}

let instance = MyClass()
```

Initializers

Initializers handle the responsibility of fully initializing an instance

```
init(luckyNumber: Int, message: String) {…}

let instance = MyClass(luckyNumber: 42, message: "Not today")
```

Structure Initialization

Structure Initialization

```
struct Color {
    let red, green, blue: Double
}
```

Structure Initialization

```
struct Color {
    let red, green, blue: Double
    init(grayScale: Double) {
        red = grayScale
        green = grayScale
        blue = grayScale
    }
}
```

Structure Initialization

```
struct Color {
    let red, green, blue: Double
    init(grayScale: Double) {
        red = grayScale
        green = grayScale
        blue = grayScale
    }
}
```

Structure Initialization

```
struct Color {
    let red, green, blue: Double
    init(grayScale: Double) {

        green = grayScale
        blue = grayScale
    }
}
// error: variable 'self.red' used before being initialized
```

Structure Initialization

```
struct Color {
    let red, green, blue: Double
    mutating func validateColor() { ... }
    init(grayScale: Double) {
        red = grayScale
        green = grayScale
        validateColor()
        blue = grayScale
    }
}
```

Structure Initialization

```
struct Color {
    let red, green, blue: Double
    mutating func validateColor() { ... }
    init(grayScale: Double) {
        red = grayScale
        green = grayScale
        validateColor()
        blue = grayScale
    }
}
```

Structure Initialization

```
struct Color {
    let red, green, blue: Double
    mutating func validateColor() { ... }
    init(grayScale: Double) {
        red = grayScale
        green = grayScale
        validateColor()
        blue = grayScale
    }
}
// error: 'self' used before being initialized
```

Structure Initialization

```
struct Color {
    let red, green, blue: Double
    mutating func validateColor() { ... }
    init(grayScale: Double) {
        red = grayScale
        green = grayScale
        self.validateColor()
        blue = grayScale
    }
}
// error: 'self' used before being initialized
```

Structure Initialization

```
struct Color {
    let red, green, blue: Double
    mutating func validateColor() { ... }
    init(grayScale: Double) {
        red = grayScale
        green = grayScale
        blue = grayScale
        validateColor()
   }
}
```

Memberwise Initializers

```
struct Color {
    let red, green, blue: Double
}
```

Memberwise Initializers

```
struct Color {
    let red, green, blue: Double
}

let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
```

Default Values

```
struct Color {
    let red = 0.0, green = 0.0, blue = 0.0
}
```

Default Initializer

```
struct Color {
    let red = 0.0, green = 0.0, blue = 0.0
}

let black = Color()
```

Class Initialization

Class Initialization

```
class Car {
    var paintColor: Color
    init(color: Color) {
        paintColor = color
    }
}
```

Class Initialization

```
class Car {
    var paintColor: Color
    init(color: Color) {
        paintColor = color
    }
}
```

Class Initialization

```
class Car {
    var paintColor: Color
    init(color: Color) {
        paintColor = color
    }
}

class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }
}
```

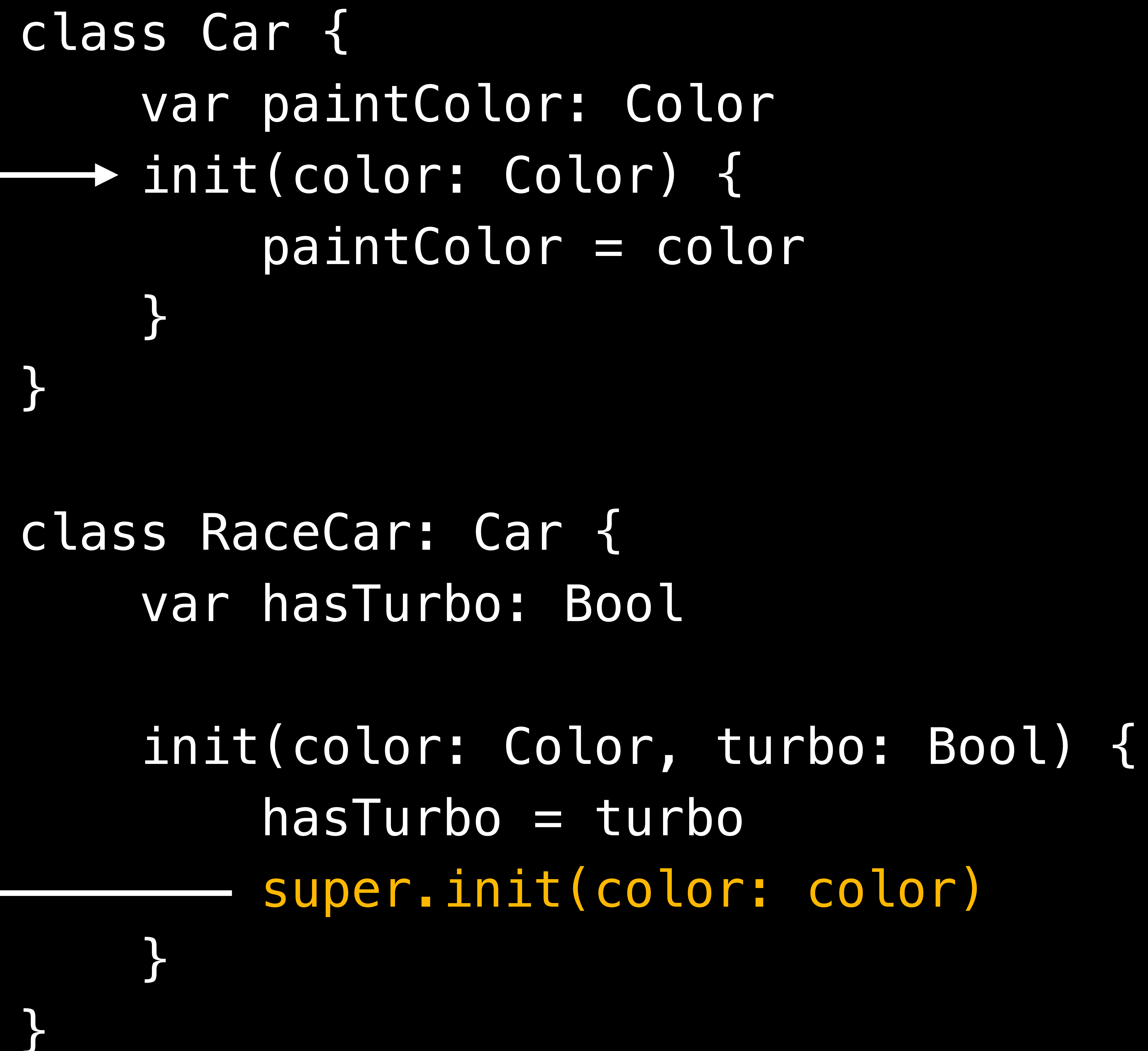
Class Initialization

```
class Car {
    var paintColor: Color
    init(color: Color) {
        paintColor = color
    }
}

class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }
}
```

Class Initialization

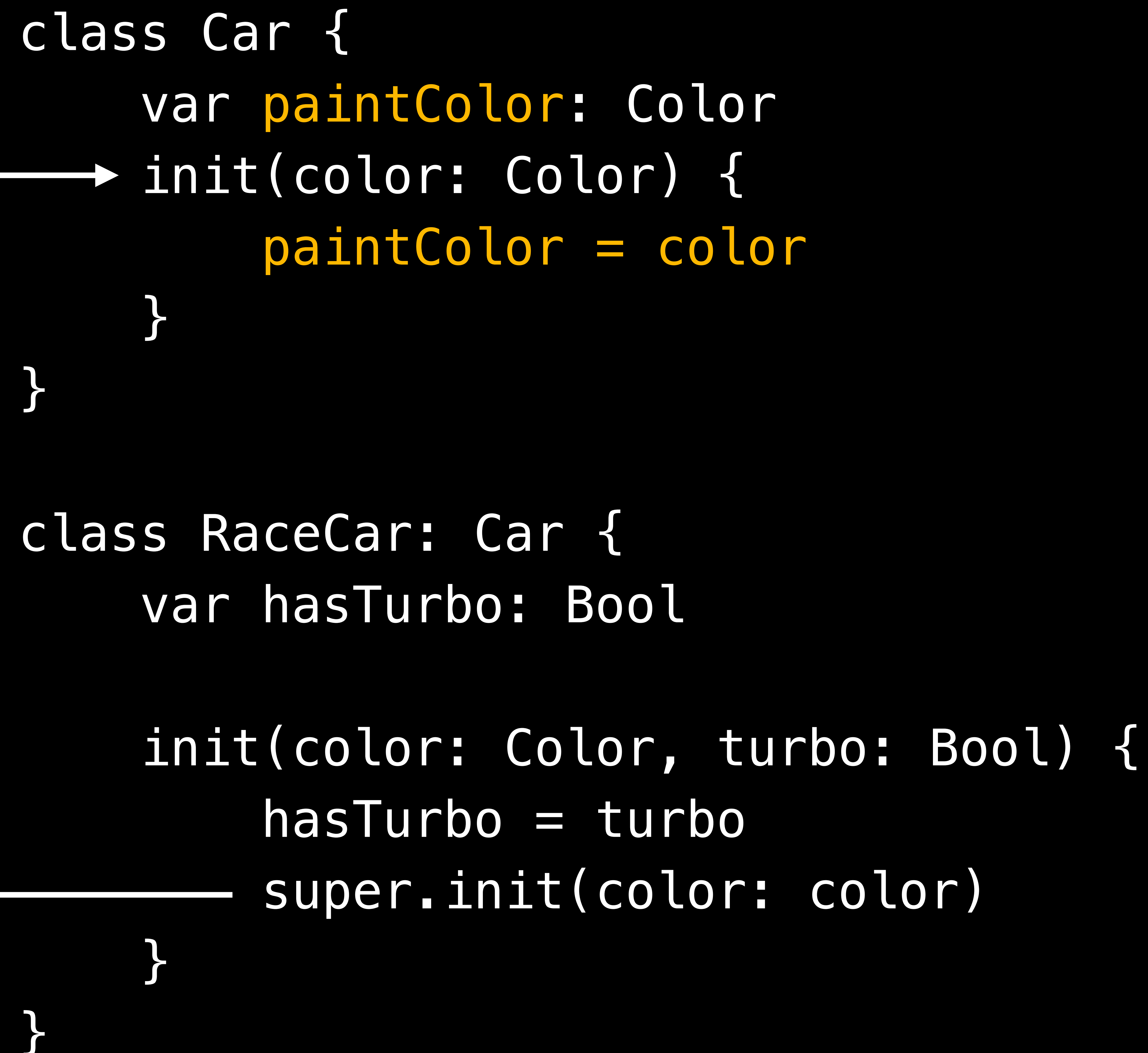
```
class Car {
    var paintColor: Color
    init(color: Color) {
        paintColor = color
    }
}

class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }
}
```

Class Initialization

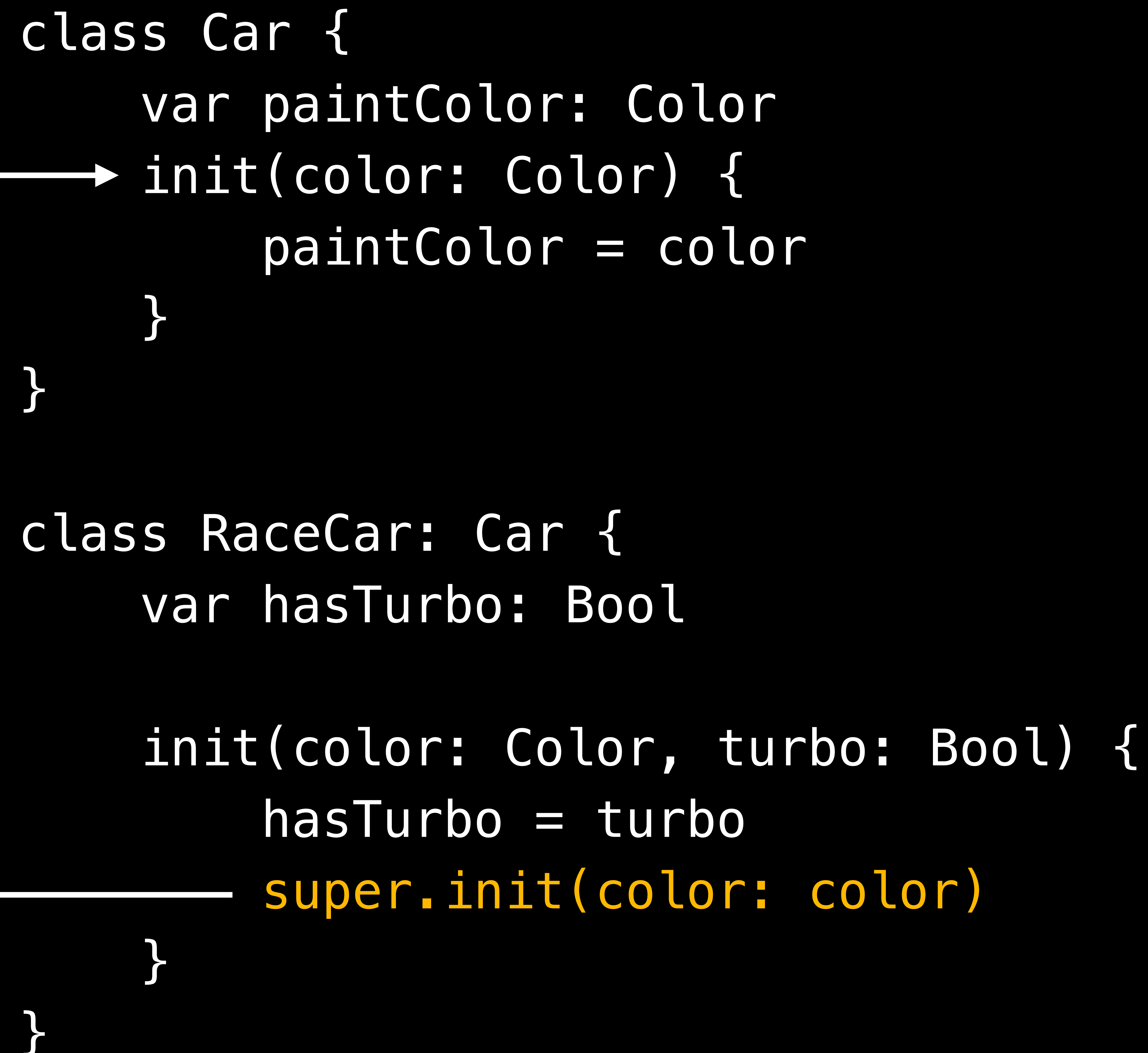


```
class Car {
    var paintColor: Color
    init(color: Color) {
        paintColor = color
    }
}

class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }
}
```

Class Initialization



Class Initialization



Class Initialization

```
class Car {
    var paintColor: Color
    init(color: Color) {
        paintColor = color
    }
}

class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }
}
```

Class Initialization

```
class Car {
    var paintColor: Color
    init(color: Color) {
        paintColor = color
    }
}

class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        super.init(color: color)
        hasTurbo = turbo
    }
}
```

Class Initialization

```
class Car {
    var paintColor: Color
    init(color: Color) {
        paintColor = color
    }
}

class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        super.init(color: color)
        hasTurbo = turbo
    }
}
```

Class Initialization

```
class Car {
    var paintColor: Color
    func fillGasTank() {...}
    init(color: Color) {
        paintColor = color
        fillGasTank()
    }
}

class RaceCar: Car {
    var hasTurbo: Bool
    override func fillGasTank() { ... }
    init(color: Color, turbo: Bool) {
        super.init(color: color)
        hasTurbo = turbo
    }
}
```

Class Initialization

```
class Car {
    var paintColor: Color
    func fillGasTank() {...}
    init(color: Color) {
        paintColor = color
        fillGasTank()
    }
}

class RaceCar: Car {
    var hasTurbo: Bool
    override func fillGasTank() { ... }
    init(color: Color, turbo: Bool) {
        super.init(color: color)
        hasTurbo = turbo
    }
}
```

Class Initialization

```
class Car {
    var paintColor: Color
    func fillGasTank() {...}
    init(color: Color) {
        paintColor = color
        fillGasTank()
    }
}

class RaceCar: Car {
    var hasTurbo: Bool
    override func fillGasTank() { ... }
    init(color: Color, turbo: Bool) {
        super.init(color: color)
        hasTurbo = turbo
    }
}
```

Class Initialization

```
class Car {
    var paintColor: Color
    func fillGasTank() {...}
    init(color: Color) {
        paintColor = color
        fillGasTank()
    }
}

class RaceCar: Car {
    var hasTurbo: Bool
    override func fillGasTank() { ... }
    init(color: Color, turbo: Bool) {
        super.init(color: color)
        hasTurbo = turbo
    }
}
```

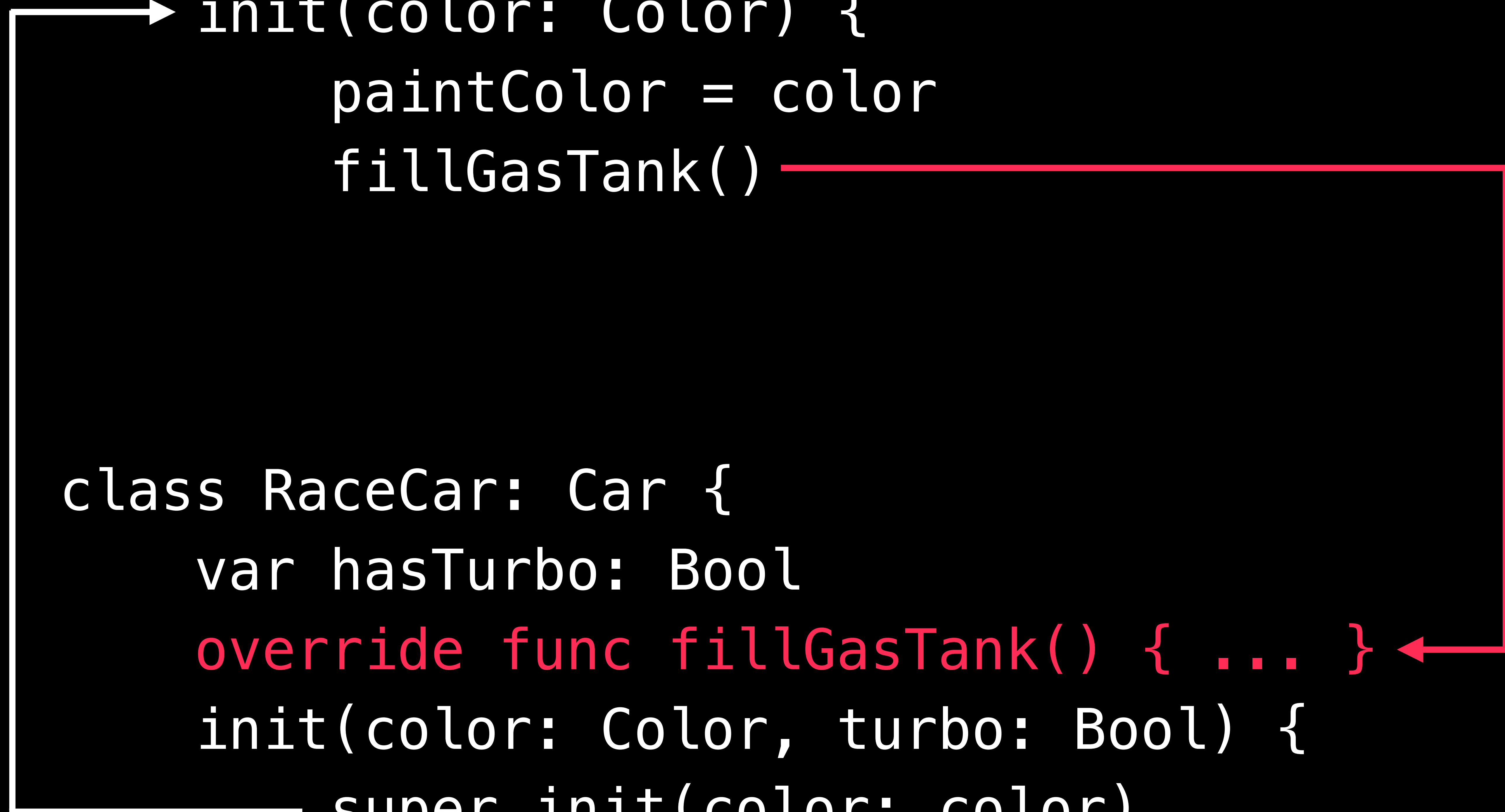
Class Initialization

```
class Car {
    var paintColor: Color
    func fillGasTank() {...}
    init(color: Color) {
        paintColor = color
        fillGasTank()
    }
}

class RaceCar: Car {
    var hasTurbo: Bool
    override func fillGasTank() { ... }
    init(color: Color, turbo: Bool) {
        super.init(color: color)
        hasTurbo = turbo
    }
}
```

Class Initialization

```
class Car {
    var paintColor: Color
    func fillGasTank() {...}
    init(color: Color) {
        paintColor = color
        fillGasTank()
    }
}

class RaceCar: Car {
    var hasTurbo: Bool
    override func fillGasTank() { ... }
    init(color: Color, turbo: Bool) {
        super.init(color: color)
        hasTurbo = turbo
    }
}
```

Class Initialization



```
class Car {
    var paintColor: Color
    func fillGasTank() {...}
    init(color: Color) {
        paintColor = color
        fillGasTank()



class RaceCar: Car {
    var hasTurbo: Bool
    override func fillGasTank() { ... }
    init(color: Color, turbo: Bool) {
        super.init(color: color)
        hasTurbo = turbo
```

Class Initialization

```
class Car {
    var paintColor: Color
    func fillGasTank() {...}
    init(color: Color) {
        paintColor = color
        fillGasTank()
    }
}

class RaceCar: Car {
    var hasTurbo: Bool
    override func fillGasTank() { ... }
    init(color: Color, turbo: Bool) {
        super.init(color: color)
        hasTurbo = turbo
    }
} // error: property 'hasTurbo' not initialized at super.init call
```

Class Initialization

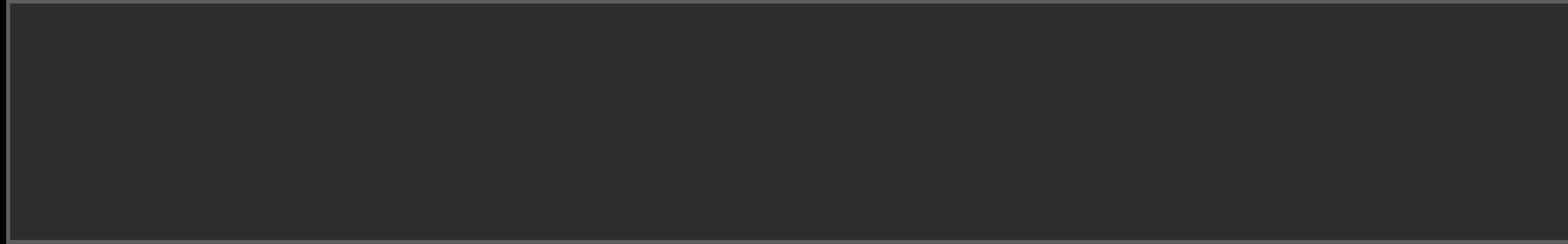
```
class Car {
    var paintColor: Color
    func fillGasTank() {...}
    init(color: Color) {
        paintColor = color
        fillGasTank()
    }
}

class RaceCar: Car {
    var hasTurbo: Bool
    override func fillGasTank() { ... }
    init(color: Color, turbo: Bool) {
        super.init(color: color)
        hasTurbo = turbo
    }
}
```

Class Initialization

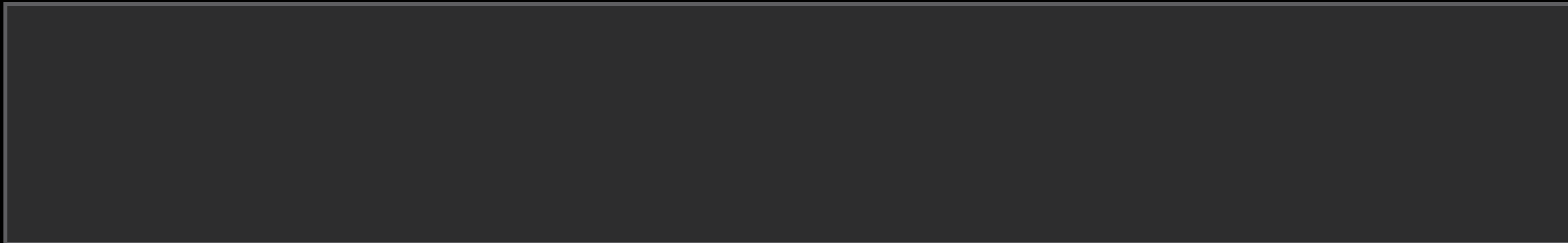
```
class Car {
    var paintColor: Color
    func fillGasTank() {...}
    init(color: Color) {
        paintColor = color
        fillGasTank()
    }
}

class RaceCar: Car {
    var hasTurbo: Bool
    override func fillGasTank() { ... }
    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }
}
```

Initializer Delegation

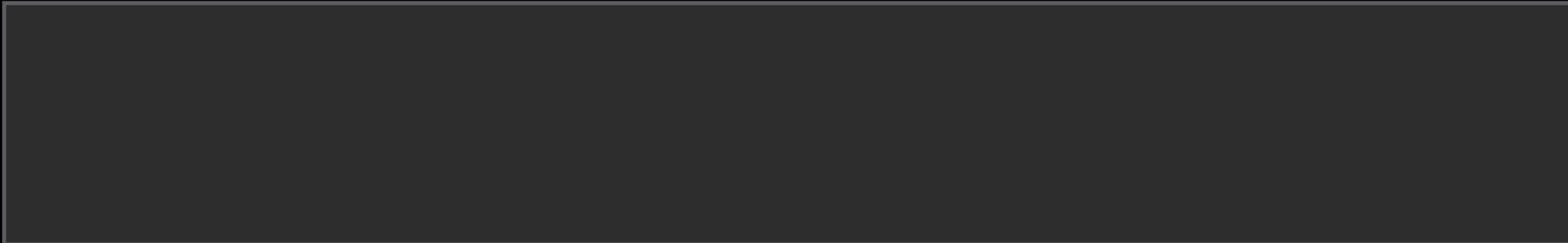
BaseClass



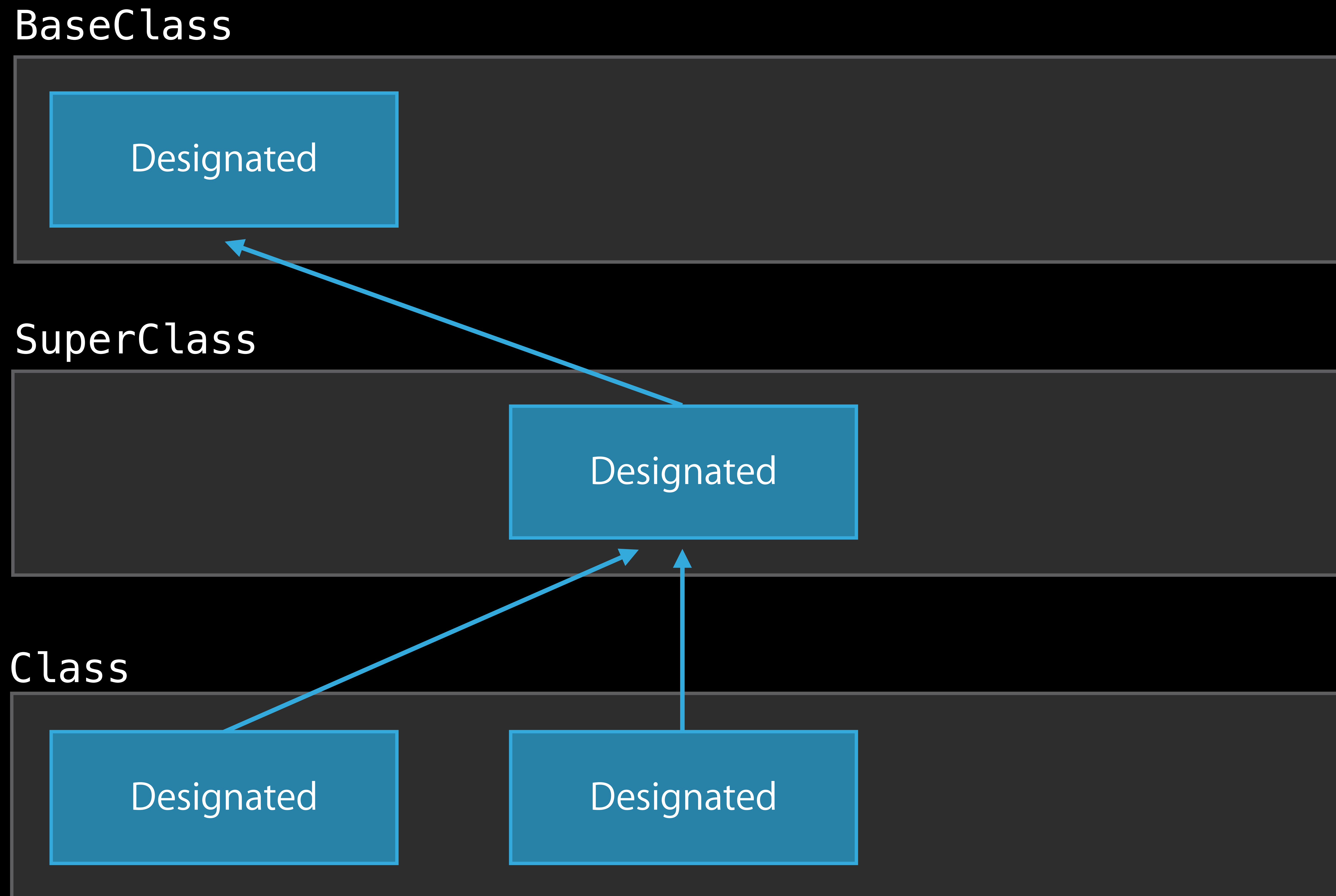
SuperClass



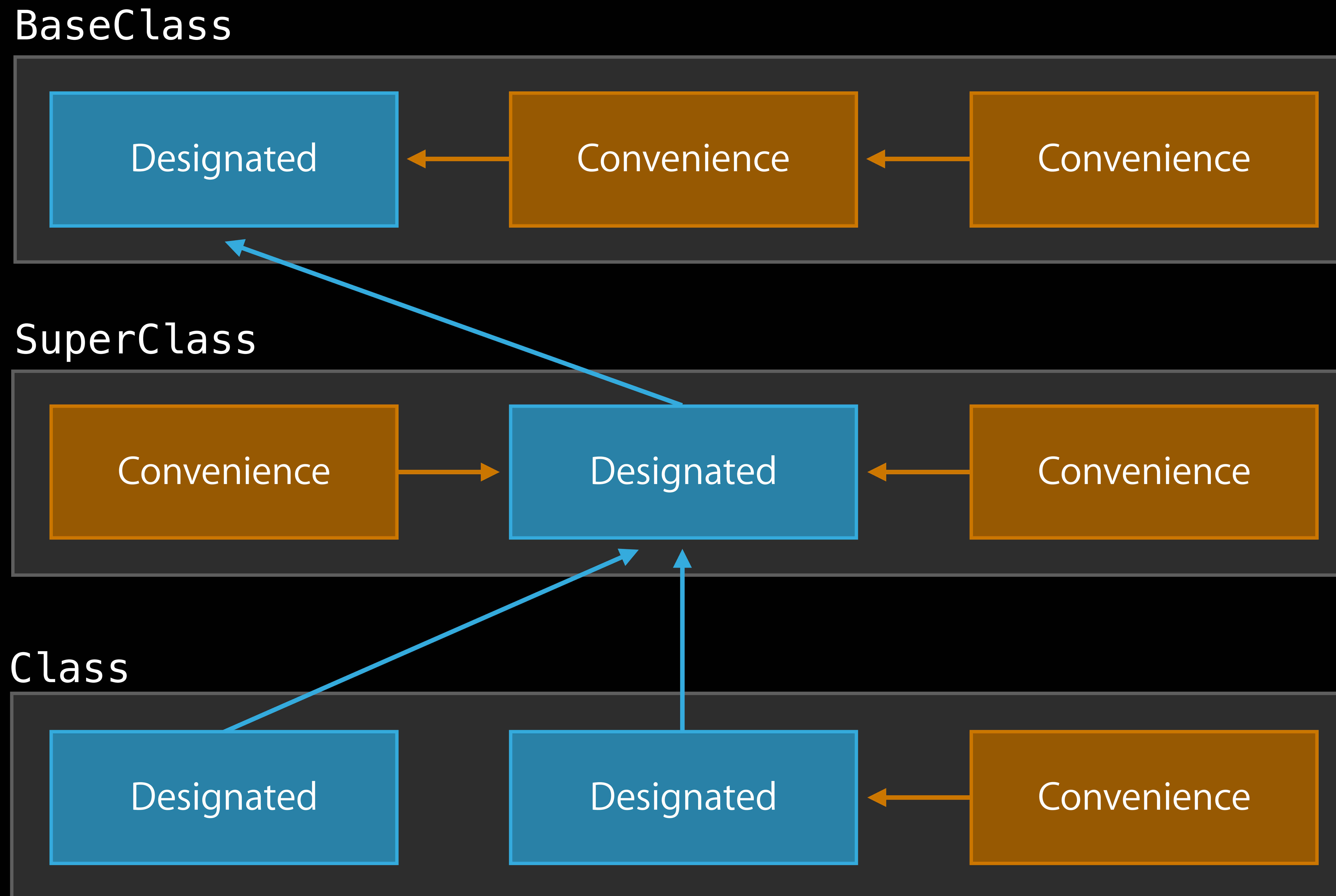
Class



Initializer Delegation



Initializer Delegation



Convenience Initializers

```
class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }

}
```

Convenience Initializers

```
class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }

    init(color: Color) {
        self.init(color: color, turbo: true)
    }




}
```

Convenience Initializers

```
class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }

    init(color: Color) {
        self.init(color: color, turbo: true)
    }




}
```

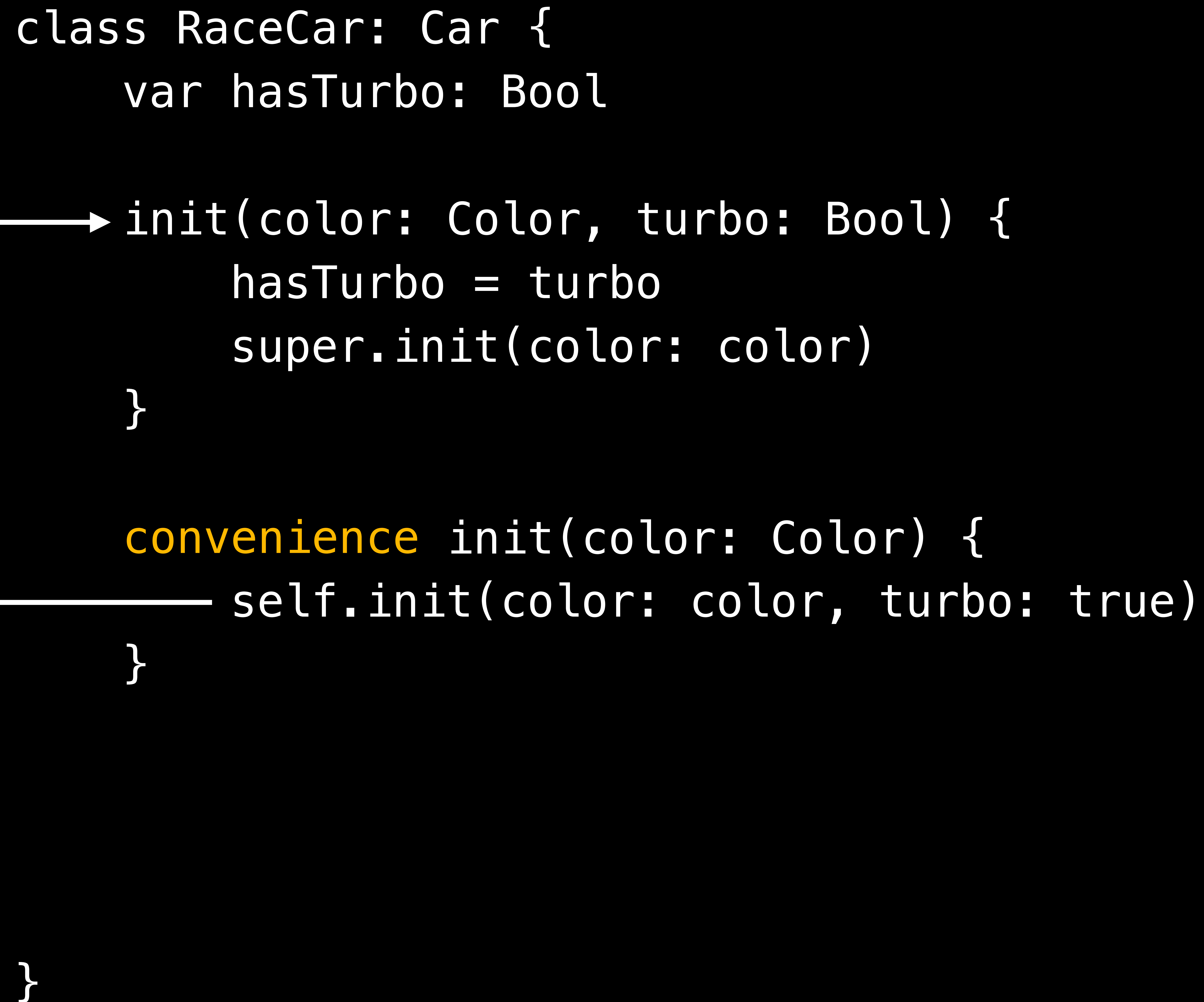
Convenience Initializers

```
class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }

    init(color: Color) {
        self.init(color: color, turbo: true)
    }



}
```

Convenience Initializers

```
class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }

    init(color: Color) {
        self.init(color: color, turbo: true)
    }



}
```

Convenience Initializers



Convenience Initializers

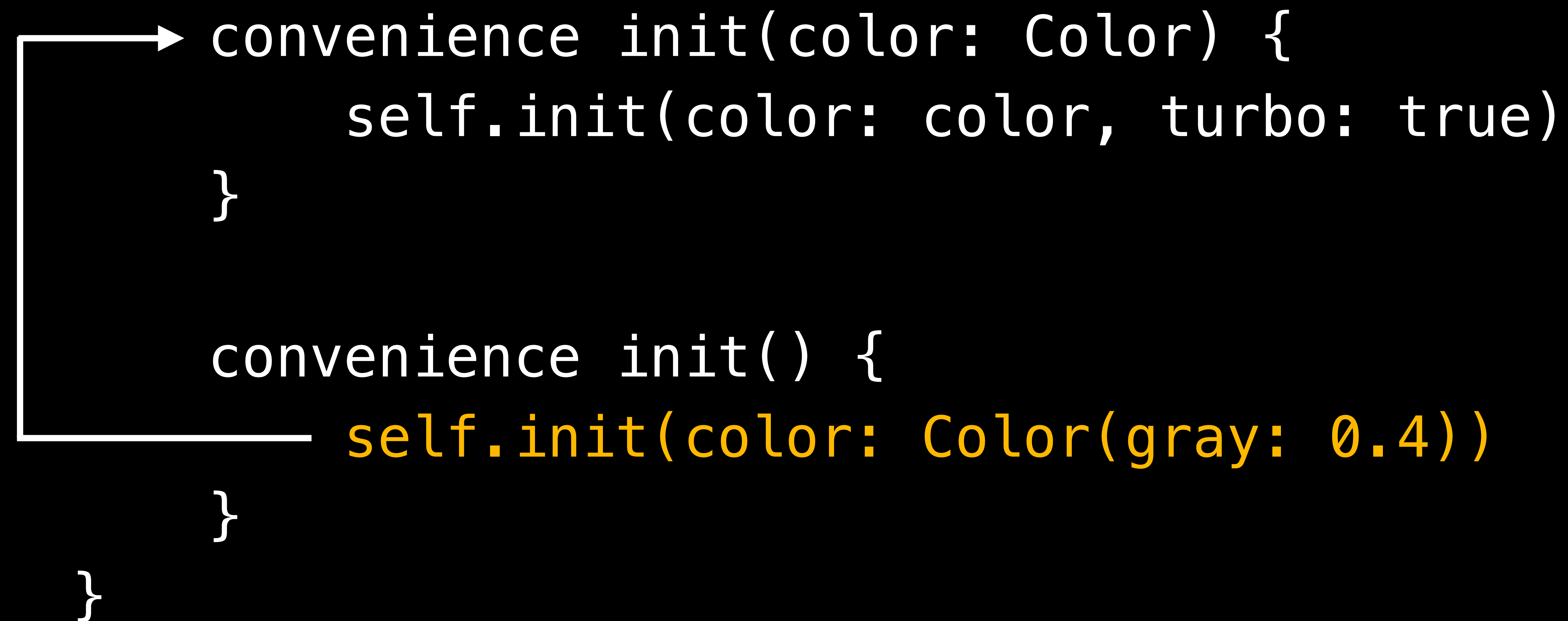
```
class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }

    convenience init(color: Color) {
        self.init(color: color, turbo: true)
    }

    convenience init() {
        self.init(color: Color(gray: 0.4))
    }
}
```

Convenience Initializers

```
class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }

    convenience init(color: Color) {
        self.init(color: color, turbo: true)
    }

    convenience init() {
        self.init(color: Color(gray: 0.4))
    }
}
```

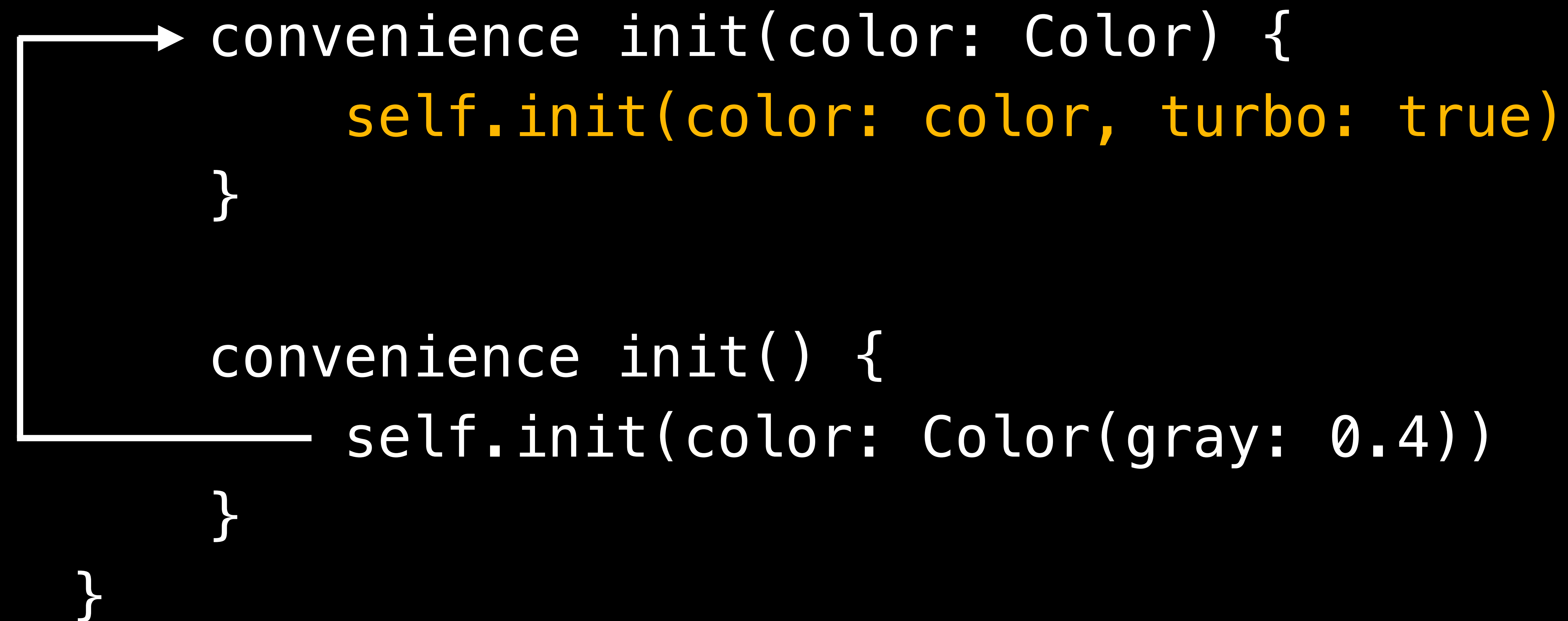
Convenience Initializers

```swift
class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }
```



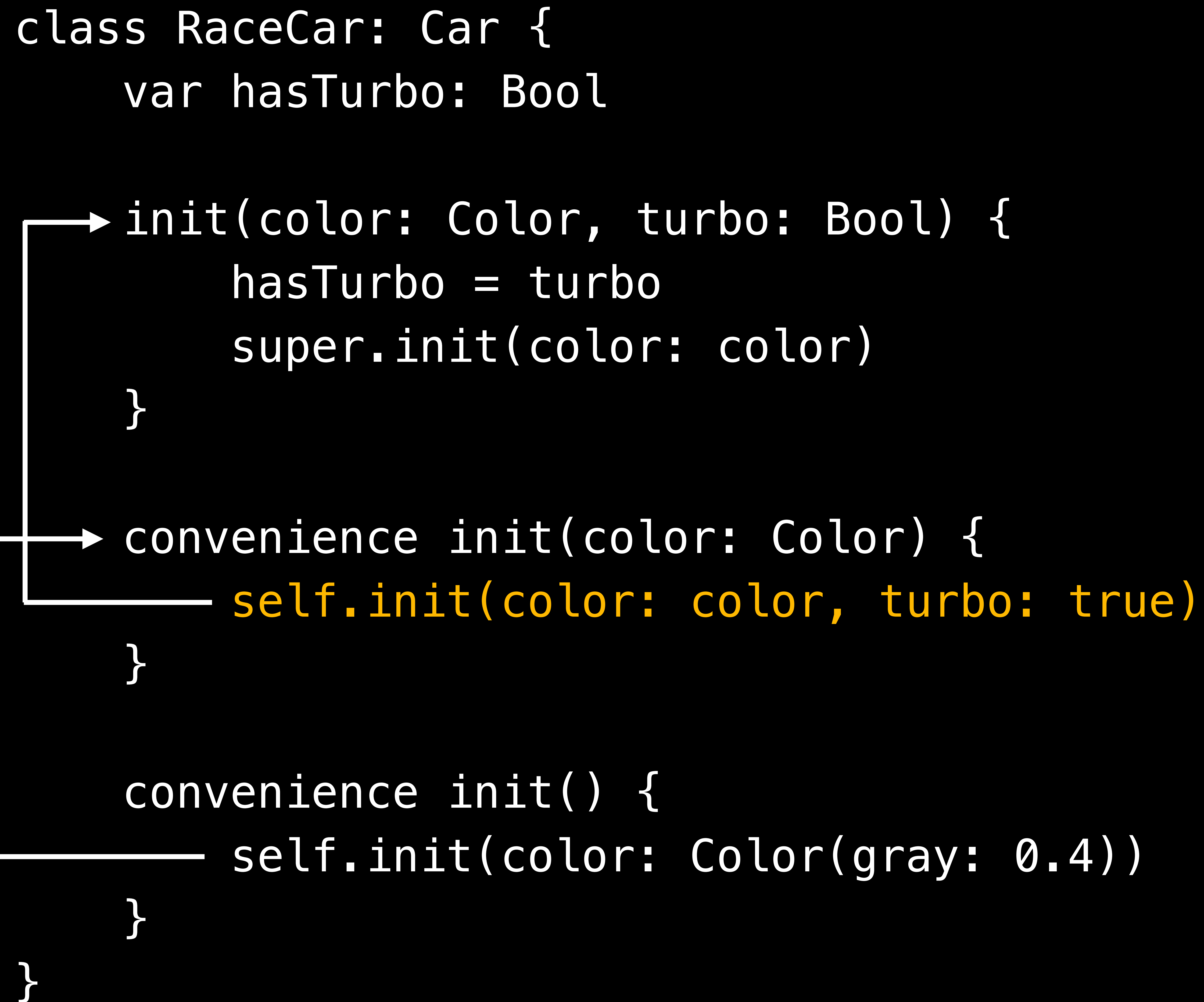
Convenience Initializers

```
class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }
```



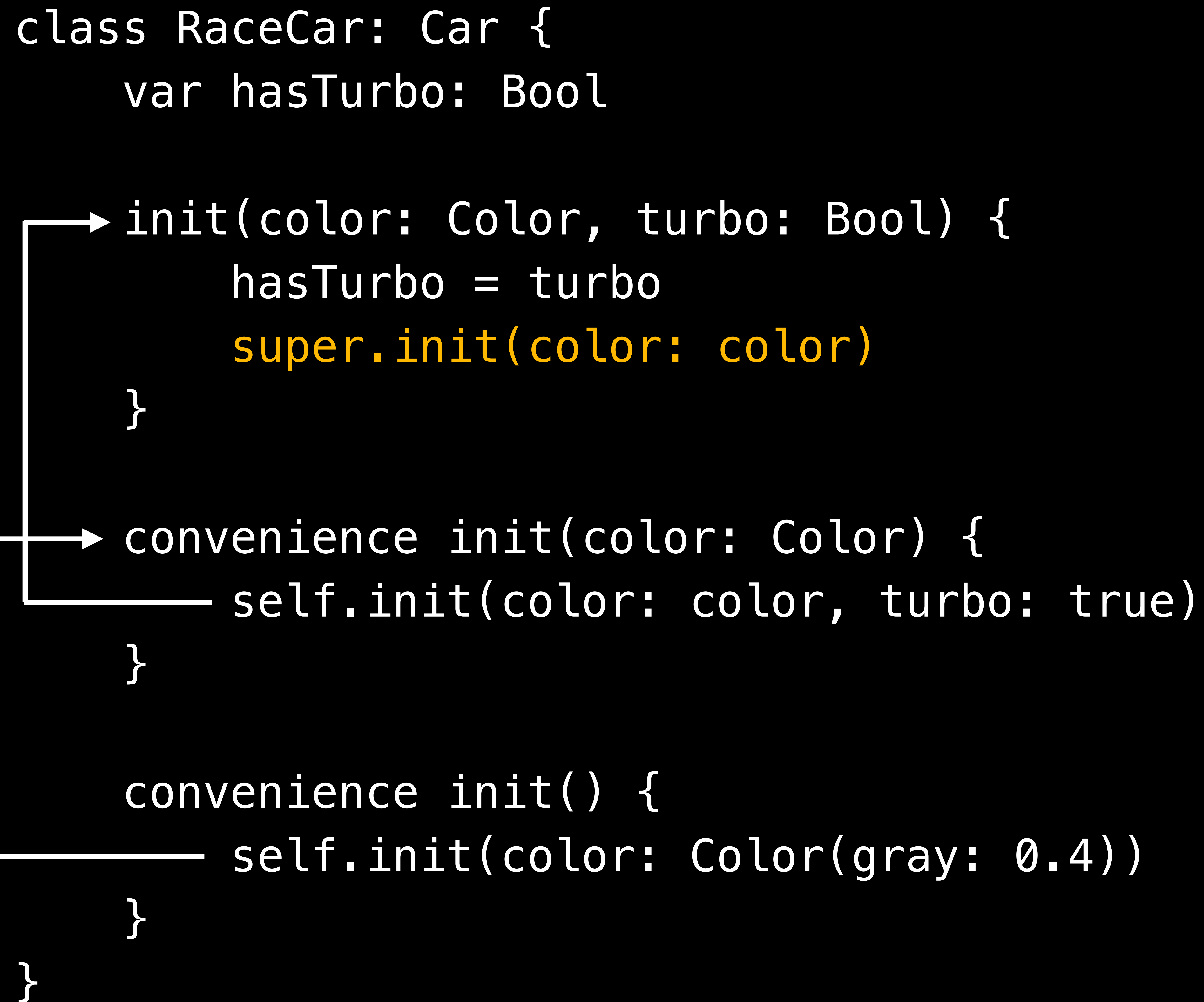
```
    convenience init(color: Color) {
        self.init(color: color, turbo: true)
    }

    convenience init() {
        self.init(color: Color(gray: 0.4))
    }
}
```

Convenience Initializers



```
class RaceCar: Car {
    var hasTurbo: Bool

    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }

    convenience init(color: Color) {
        self.init(color: color, turbo: true)
    }

    convenience init() {
        self.init(color: Color(gray: 0.4))
    }
}
```

Convenience Initializers



Initializer Inheritance

```
class FormulaOne: RaceCar {
    let minimumWeight = 642
```

Initializer Inheritance

```
class FormulaOne: RaceCar {
    let minimumWeight = 642

    // inherited from RaceCar
    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }
    convenience init(color: Color) {
        self.init(color: color, turbo: true)
    }
    convenience init() {
        self.init(color: Color(gray: 0.4))
    }
}
```

Initializer Inheritance

```
class FormulaOne: RaceCar {
    let minimumWeight = 642

    // inherited from RaceCar
    init(color: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(color: color)
    }
    convenience init(color: Color) {
        self.init(color: color, turbo: true)
    }
    convenience init() {
        self.init(color: Color(gray: 0.4))
    }
}
```

Initializer Inheritance

```
class FormulaOne: RaceCar {
    let minimumWeight = 642

    init(color: Color) {
        super.init(color: color, turbo: false)
    }




}
```

Initializer Inheritance

```
class FormulaOne: RaceCar {
    let minimumWeight = 642

    init(color: Color) {
        super.init(color: color, turbo: false)
    }

    // not inherited from RaceCar
    init(color: Color, turbo: Bool)
    convenience init()
}
```

Lazy Properties

Lazy Properties

```
class Game {
    var multiplayerManager = MultiplayerManager()
    var singlePlayer: Player?
    func beginGameWithPlayers(players: Player...) {







    }
}
```

Lazy Properties

```
class Game {
    var multiplayerManager = MultiplayerManager()
    var singlePlayer: Player?
    func beginGameWithPlayers(players: Player...) {
        if players.count == 1 {
            singlePlayer = players[0]
        }



    }
}
```

Lazy Properties

```
class Game {
    var multiplayerManager = MultiplayerManager()
    var singlePlayer: Player?
    func beginGameWithPlayers(players: Player...) {
        if players.count == 1 {
            singlePlayer = players[0]
        } else {
            for player in players {
                multiplayerManager.addPlayer(player)
            }
        }
    }
}
```

Lazy Properties

```
class Game {
    var multiplayerManager = MultiplayerManager()
    var singlePlayer: Player?
    func beginGameWithPlayers(players: Player...) {
        if players.count == 1 {
            singlePlayer = players[0]
        } else {
            for player in players {
                multiplayerManager.addPlayer(player)
            }
        }
    }
}
```

Lazy Properties

```
class Game {
    @lazy var multiplayerManager = MultiplayerManager()
    var singlePlayer: Player?
    func beginGameWithPlayers(players: Player...) {
        if players.count == 1 {
            singlePlayer = players[0]
        } else {
            for player in players {
                multiplayerManager.addPlayer(player)
            }
        }
    }
}
```

Deinitialization

Deinitialization

```
class FileHandle {
    let fileDescriptor: FileDescriptor
    init(path: String) {
        fileDescriptor = openFile(path)
    }



}
```

Deinitialization

```
class FileHandle {
    let fileDescriptor: FileDescriptor
    init(path: String) {
        fileDescriptor = openFile(path)
    }
    deinit {
        closeFile(fileDescriptor)
    }
}
```

Initialization

SAFE

Initialization

SAFE

Initialize all values before you use them

Set all stored properties **first**, then call super.init

Initialization

SAFE

Initialize all values before you use them

Set all stored properties **first**, then call super.init

Designated initializers only delegate **up**

Convenience initializers only delegate **across**

Initialization

SAFE

Initialize all values before you use them

Set all stored properties **first**, then call super.init

Designated initializers only delegate **up**

Convenience initializers only delegate **across**

Deinitializers are there… if you need them

Closures

Joe Groff
Swift Compiler Engineer

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort(



println(clients)
```

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({

})

println(clients)
```

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool


})

println(clients)
```

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool in


})

println(clients)
```

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool in
    return a < b
})

println(clients)
```

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool in
     return a < b
})

println(clients)
```

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool in
     return a < b
})

println(clients)
```

```
[Babbage, Buenaventura, Pestov, Sreeram]
```

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool in


})

println(clients)
```

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool in
     return a > b
})

println(clients)
```

```
[Sreeram, Pestov, Buenaventura, Babbage]
```

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool in


})

println(clients)
```

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool in
     return countElements(a) < countElements(b)
})

println(clients)
```

```
[Pestov, Sreeram, Babbage, Buenaventura]
```

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool in


})

println(clients)
```

Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool in
     return a < b
})

println(clients)
```

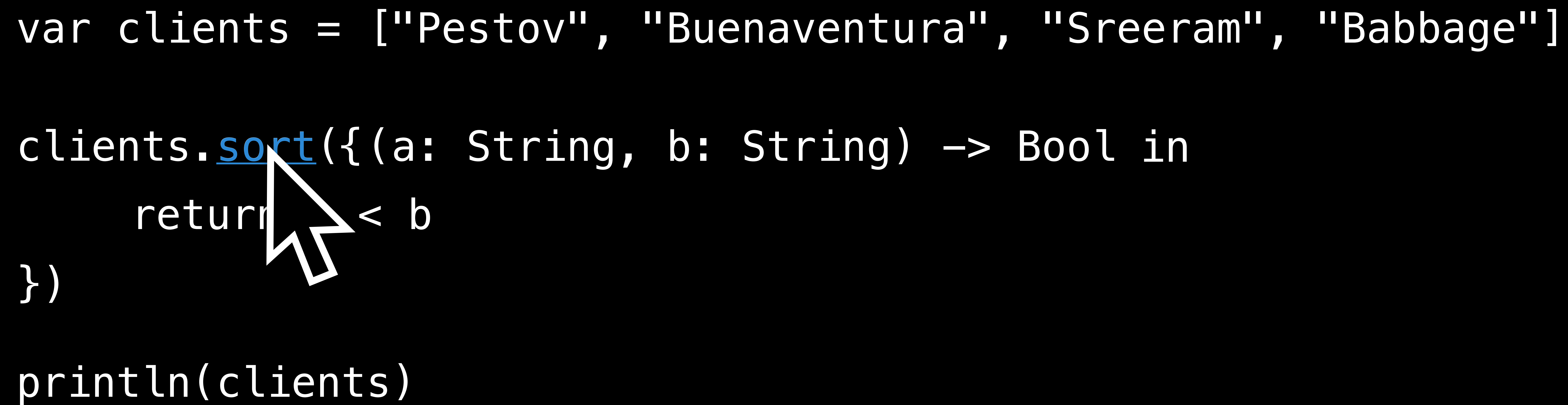
Closures

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool in
     return a < b
})

println(clients)
```

Type Inference

```
var clients = ["Pestov", "Buenaventura", "Sreeram", "Babbage"]

clients.sort({(a: String, b: String) -> Bool in
     return a < b
})

println(clients)
```

Type Inference



Type Inference

```
struct Array<T> {
    func sort(order: (T, T) -> Bool)
}
```

Type Inference

```
struct Array<String> {
    func sort(order: (String, String) -> Bool)
}
```

Type Inference

```
struct Array<String> {
    func sort(order: (String, String) -> Bool)
}
```

```
clients.sort({(a: String, b: String) -> Bool in
    return a < b
})
```

Type Inference

```
struct Array<String> {
    func sort(order: (String, String) -> Bool)
}
```

```
clients.sort({ a, b in
    return a < b
})
```

Implicit Return

```
clients.sort({ a, b in
    return a < b
})
```

Implicit Return

```
clients.sort({ a, b in a < b })
```

Implicit Arguments

```
clients.sort({ a, b in a < b })
```

Implicit Arguments

```
clients.sort({ $0 < $1 })
```

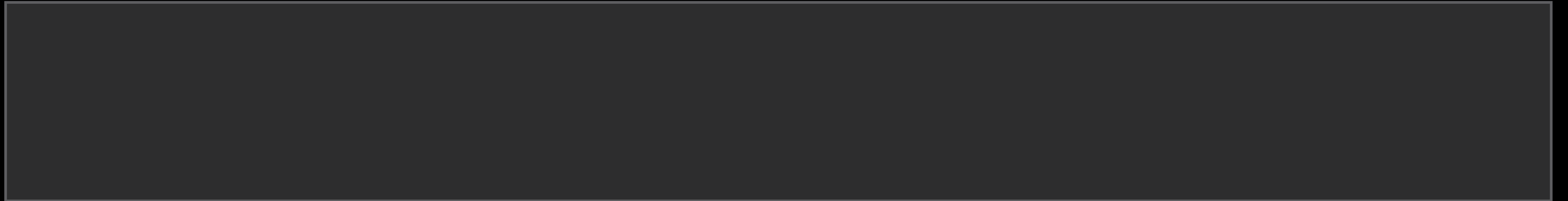
Trailing Closures

```
clients.sort({ $0 < $1 })
```

Trailing Closures

```
clients.sort { $0 < $1 }
```

Functional Programming



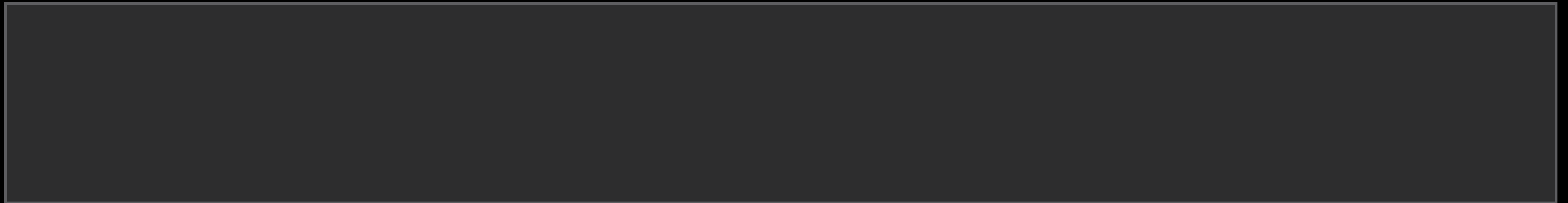
Functional Programming

```
println(words)
```

```
a
aardvark
aardwolf…
```

Functional Programming

```
println(words.filter { $0.hasSuffix("gry") })
```



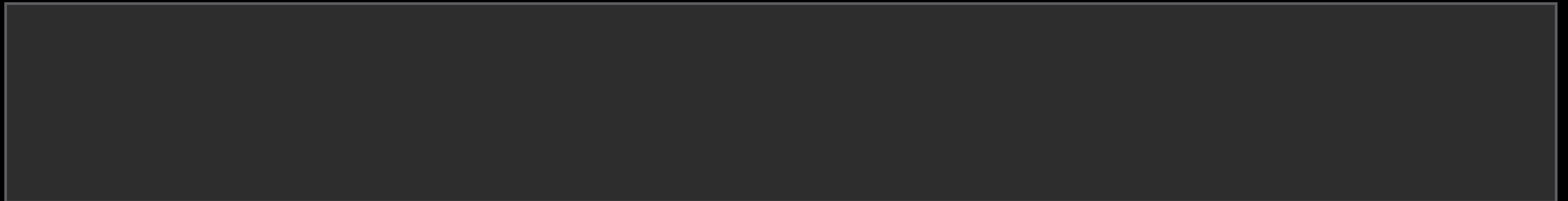
Functional Programming

```
println(words.filter { $0.hasSuffix("gry") })
```

```
angry
hungry
```

Functional Programming

```
println(words.filter { $0.hasSuffix("gry") }
             .map { $0.uppercaseString })
```



Functional Programming

```
println(words.filter { $0.hasSuffix("gry") }
             .map { $0.uppercaseString })
```

```
ANGRY
HUNGRY
```

Functional Programming

```
println(words.filter { $0.hasSuffix("gry") }
             .map { $0.uppercaseString }
             .reduce("HULK") { "\($0) \($1)" })
```

Functional Programming

```
println(words.filter { $0.hasSuffix("gry") }
             .map { $0.uppercaseString }
             .reduce("HULK") { "\($0) \($1)" })
```

```
HULK ANGRY HUNGRY
```

Functional Programming

```
println(words.filter { $0.hasSuffix("gry") }
             .map { $0.uppercaseString }
             .reduce("HULK") { "\($0) \($1)" })
```

```
HULK ANGRY HUNGRY
```

Functional Programming

```
println(words.filter { $0.hasSuffix("gry") }
             .map { $0.uppercaseString }
             .reduce("HULK") { "\($0) \($1)" })
```

Functional Programming

```
println("GRR!! " + words.filter { $0.hasSuffix("gry") }
                        .map { $0.uppercaseString }
                        .reduce("HULK") { "\($0) \($1)" } + "!!!")
```

```
GRR!! HULK ANGRY HUNGRY!!!
```

Captures

Captures

```
func sum(numbers: Int[]) {
    var sum = 0

    numbers.map {
```

Captures

```
func sum(numbers: Int[]) {
    var sum = 0

    numbers.map {
        sum += $0
    }
```

Captures

```
func sum(numbers: Int[]) {
    var sum = 0

    numbers.map {
        sum += $0
    }
```

Captures

```
func sum(numbers: Int[]) {
    var sum = 0

    numbers.map {
        sum += $0
    }
```

Captures

```
func sum(numbers: Int[]) {
    var sum = 0

    numbers.map {
        sum += $0
    }

    return sum
}
```

Function Values—Closures

```
numbers.map {
    sum += $0
}
```

Function Values—Functions

```
numbers.map {
    println($0)
}
```

Function Values—Functions

```
numbers.map (println)
```

Function Values—Methods

```
var indexes = NSMutableIndexSet()
numbers.map {
    indexes.addIndex($0)
}
```

Function Values—Methods

```
var indexes = NSMutableIndexSet()
numbers.map (indexes.addIndex)
```

Closures Are ARC Objects

Closures Are ARC Objects

```
var onTemperatureChange: (Int) -> Void = {}
```

onTempChange

Closures Are ARC Objects

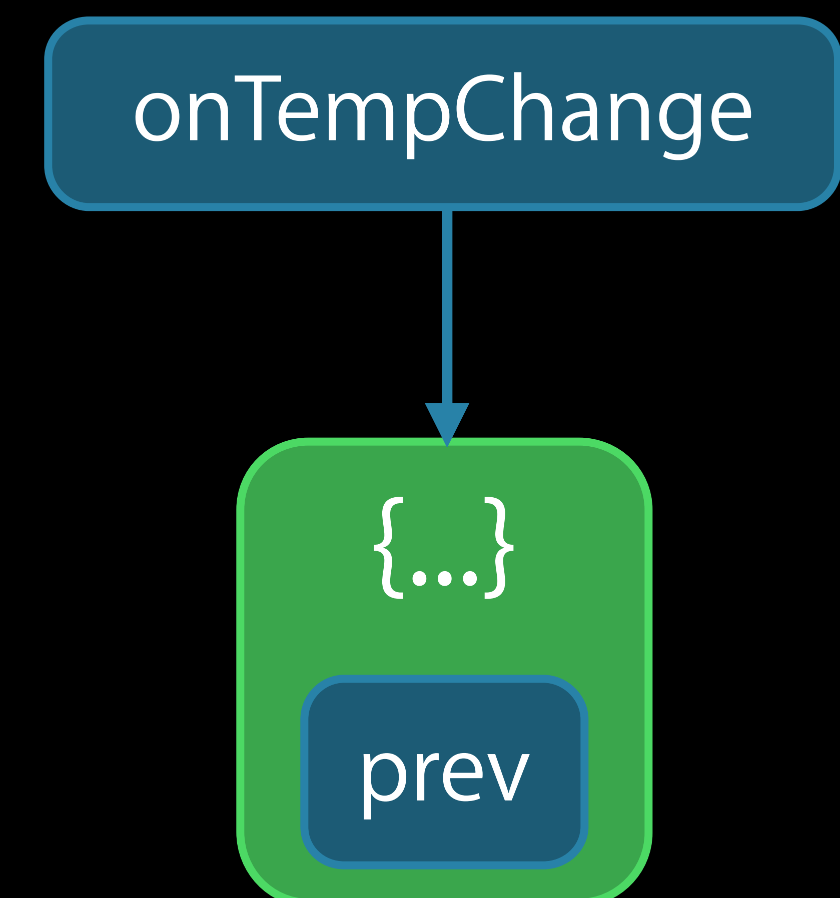
```
var onTemperatureChange: (Int) -> Void = {}

func logTemperatureDifferences(initial: Int) {
    var prev = initial
```

onTempChange

prev

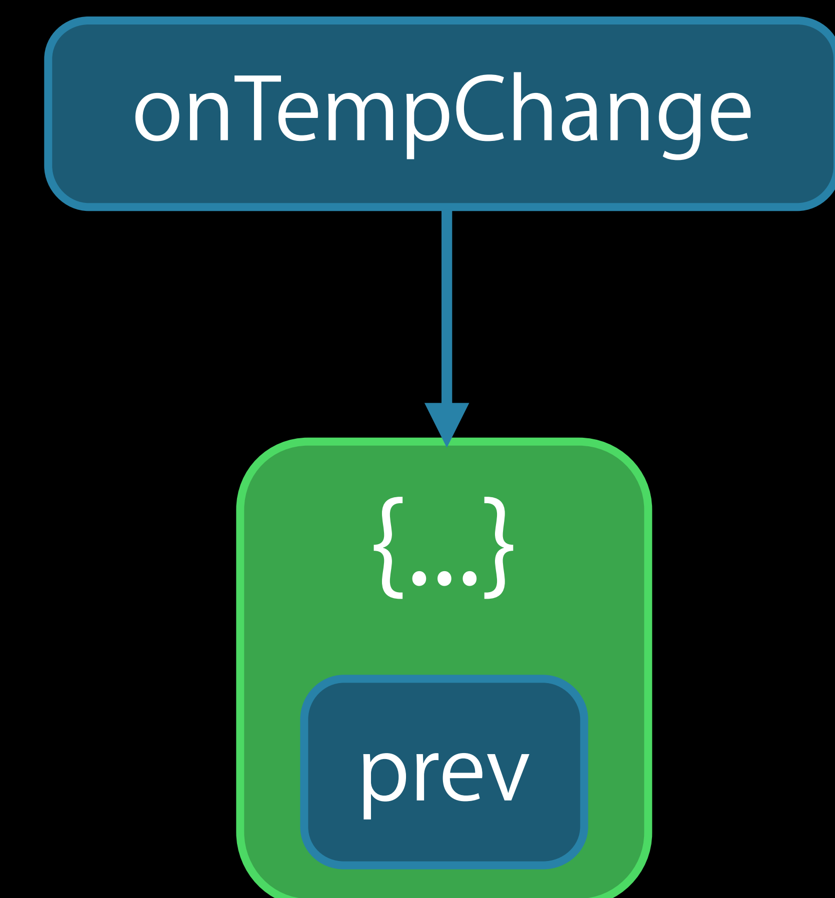
Closures Are ARC Objects

```
var onTemperatureChange: (Int) -> Void = {}

func logTemperatureDifferences(initial: Int) {
    var prev = initial
    onTemperatureChange = { next in
        println("Changed \(next - prev)°F")
        prev = next
    }
```



Closures Are ARC Objects

```
var onTemperatureChange: (Int) -> Void = {}

func logTemperatureDifferences(initial: Int) {
    var prev = initial
    onTemperatureChange = { next in
        println("Changed \(next - prev)°F")
        prev = next
    }
} // scope ends
```



Functions Are ARC Objects

```
var onTemperatureChange: (Int) -> Void = {}

func logTemperatureDifferences(initial: Int) {
    var prev = initial
    func log(next: Int) {
        println("Changed \(next - prev)°F")
        prev = next
    }
```

onTempChange

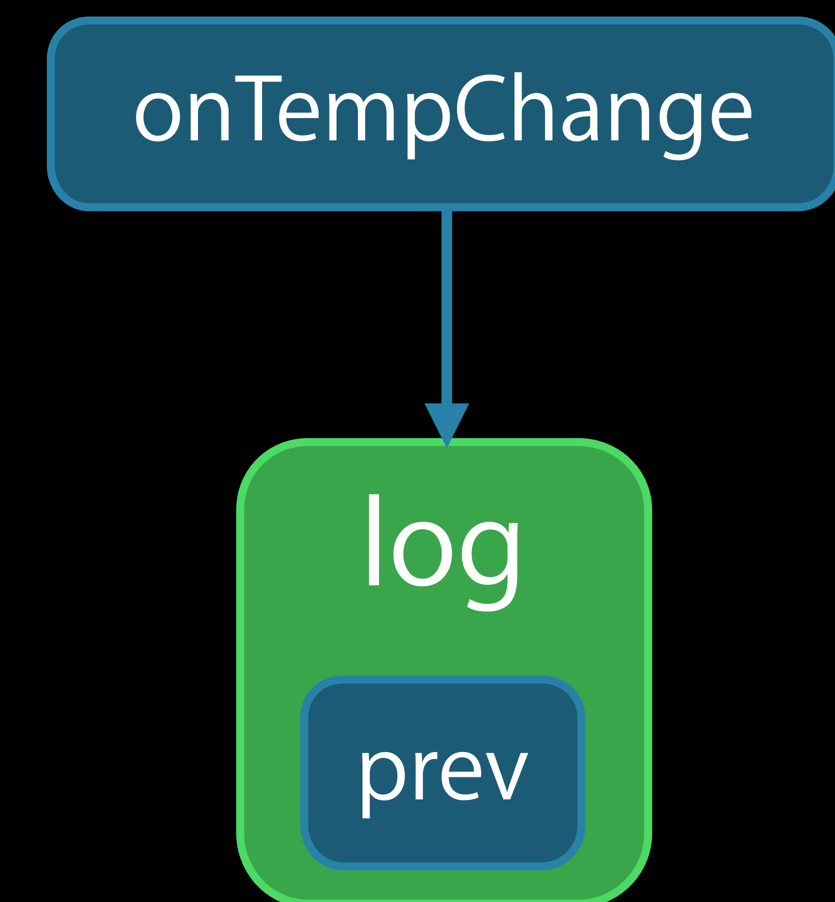
Functions Are ARC Objects

```
var onTemperatureChange: (Int) -> Void = {}

func logTemperatureDifferences(initial: Int) {
    var prev = initial
    func log(next: Int) {
        println("Changed \(next - prev)°F")
        prev = next
    }
```

log

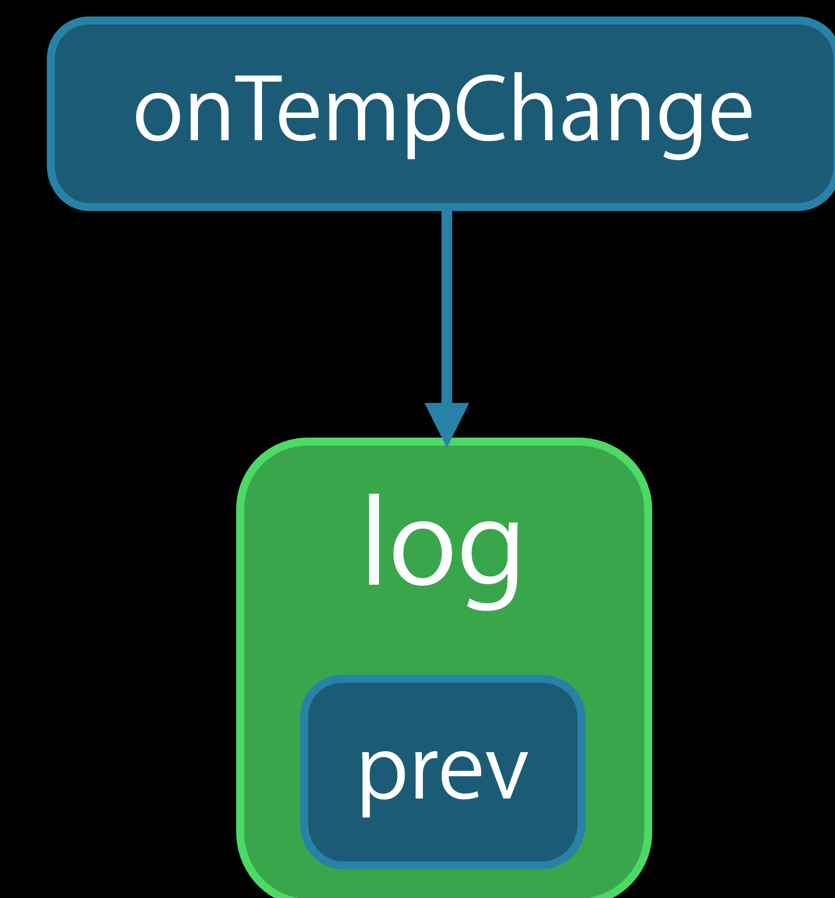
Functions Are ARC Objects

```
var onTemperatureChange: (Int) -> Void = {}

func logTemperatureDifferences(initial: Int) {
    var prev = initial
    func log(next: Int) {
        println("Changed \(next - prev)°F")
        prev = next
    }

    onTemperatureChange = log
```



Functions Are ARC Objects

```
var onTemperatureChange: (Int) -> Void = {}

func logTemperatureDifferences(initial: Int) {
    var prev = initial
    func log(next: Int) {
        println("Changed \(next - prev)°F")
        prev = next
    }

    onTemperatureChange = log
} // scope ends
```



Ownership of Captures

Ownership of Captures

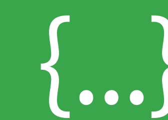
```
class TemperatureNotifier {
    var onChange: (Int) -> Void = {}
```

Ownership of Captures

```
class TemperatureNotifier {
    var onChange: (Int) -> Void = {}
    var currentTemp = 72

    init() {
        onChange = { temp in
            currentTemp = temp
        }
    }
}
```

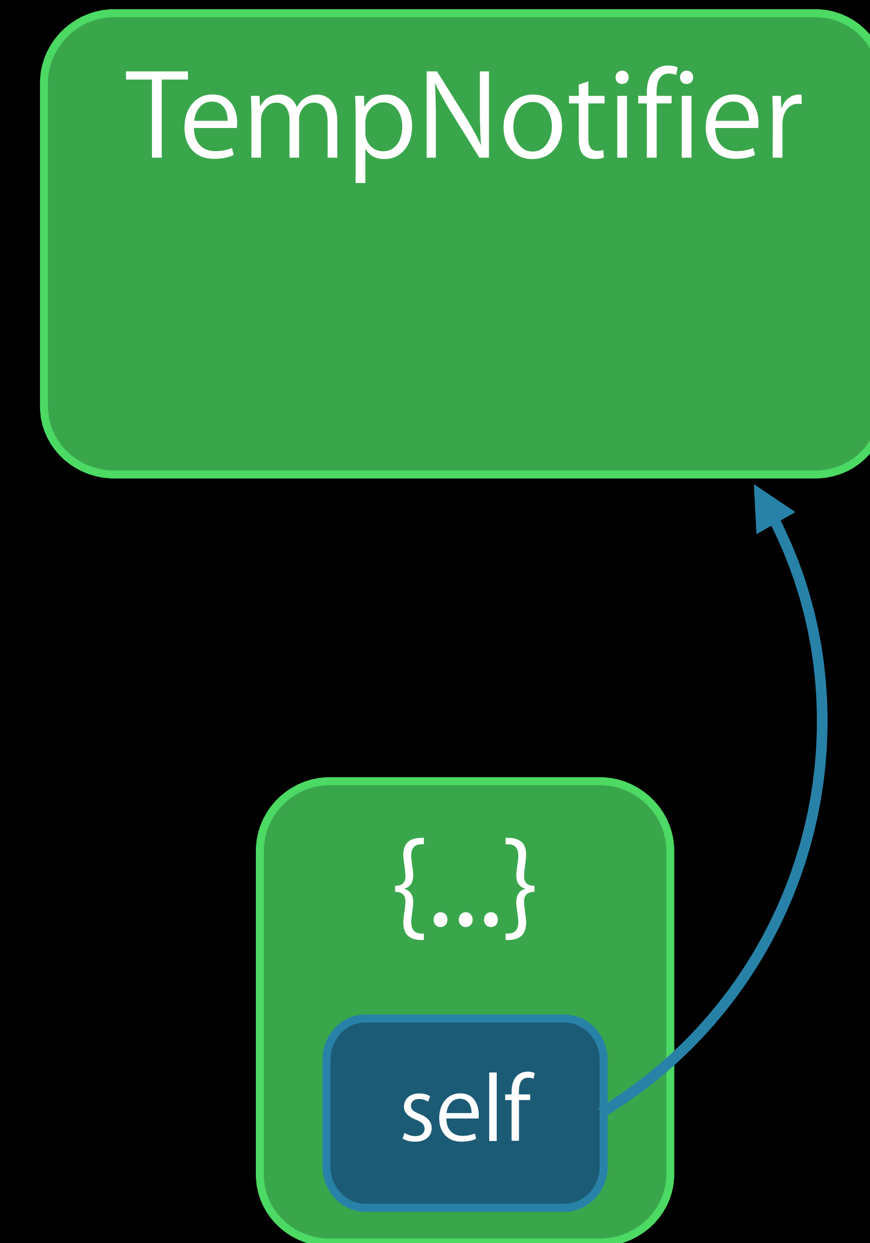
Ownership of Captures

```
class TemperatureNotifier {
    var onChange: (Int) -> Void = {}
    var currentTemp = 72

    init() {
        onChange = { temp in
          currentTemp = temp
        }
    }
}
```



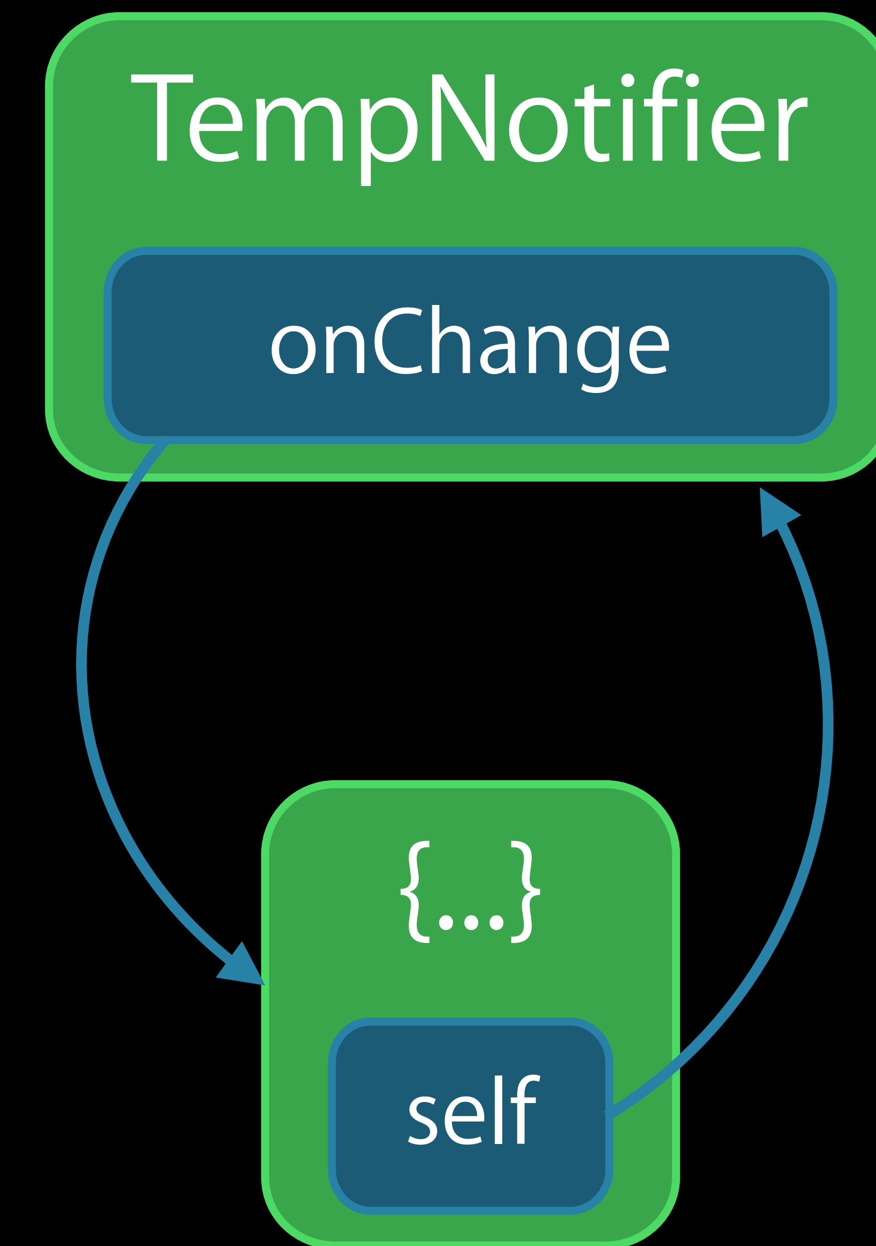
Ownership of Captures

```
class TemperatureNotifier {
    var onChange: (Int) -> Void = {}
    var currentTemp = 72

    init() {
        onChange = { temp in
          self.currentTemp = temp
        }
    }
}
```



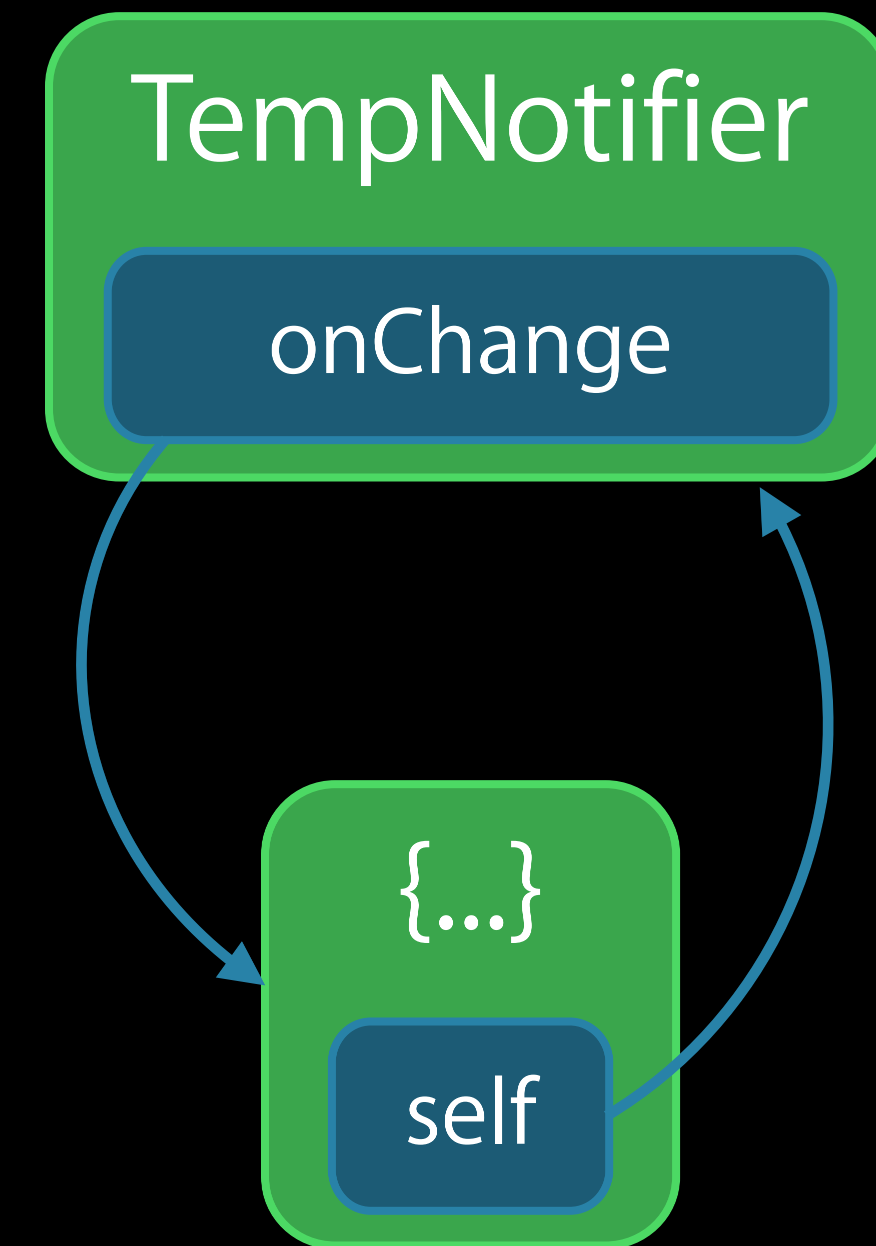
Ownership of Captures

```
class TemperatureNotifier {
    var onChange: (Int) -> Void = {}
    var currentTemp = 72

    init() {
        onChange = { temp in
            self.currentTemp = temp
        }
    }
}
```



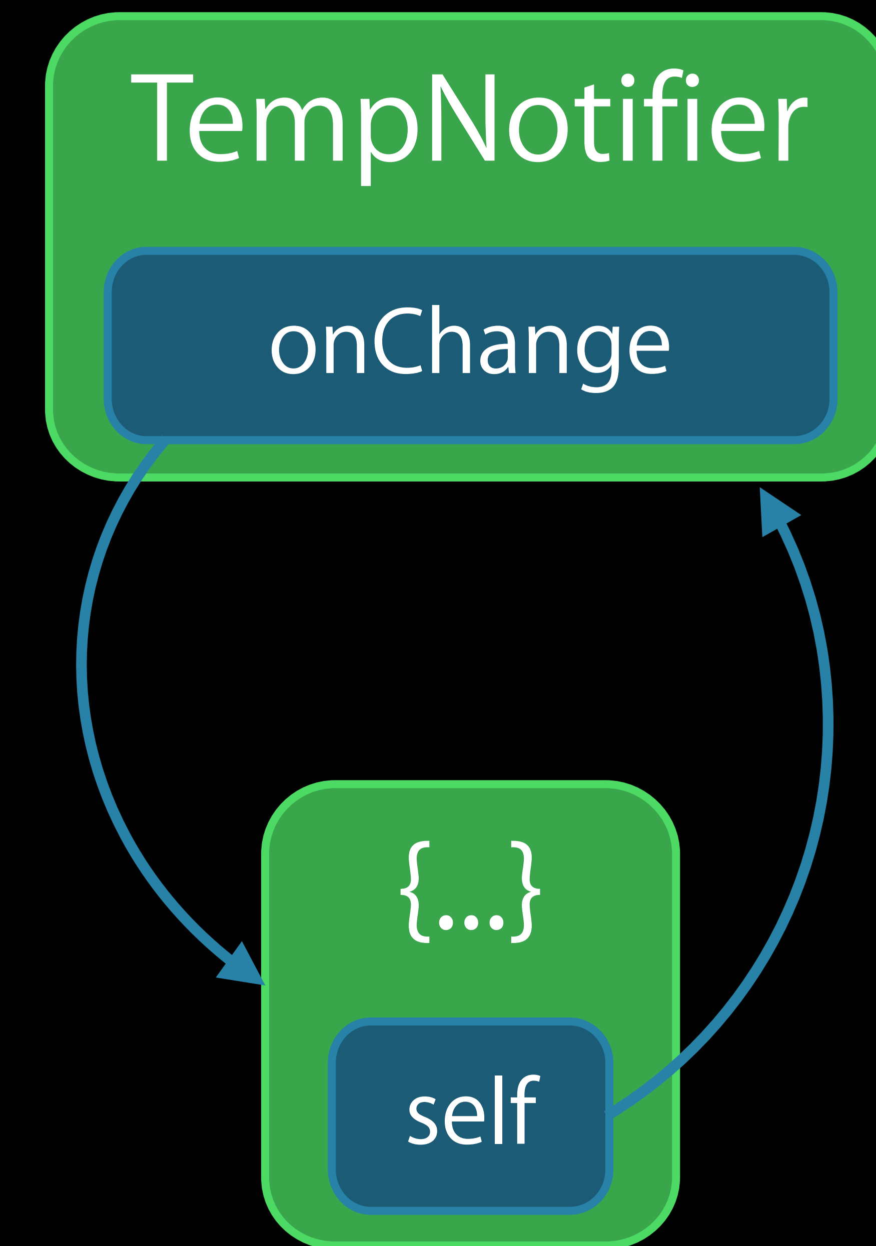
Ownership of Captures

```
class TemperatureNotifier {
    var onChange: (Int) -> Void = {}
    var currentTemp = 72

    init() {
        onChange = { temp in
            self.currentTemp = temp
        }
    }
}
```



Ownership of Captures

```
class TemperatureNotifier {
    var onChange: (Int) -> Void = {}
    var currentTemp = 72

    init() {
        onChange = { temp in
            currentTemp = temp
        } // error: requires explicit 'self'
    }
}
```



Ownership of Captures

```
class TempNotifier {
    var onChange: (Int) -> Void = {}
    var currentTemp = 72

    init() {
        onChange = { temp in
          self.currentTemp = temp
        }
    }
}
```

Ownership of Captures

```
class TempNotifier {
    var onChange: (Int) -> Void = {}
    var currentTemp = 72

    init() {
        unowned let uSelf = self
        onChange = { temp in
          self.currentTemp = temp
        }
    }
}
```

Ownership of Captures

```
class TempNotifier {
    var onChange: (Int) -> Void = {}
    var currentTemp = 72

    init() {
        unowned let uSelf = self
        onChange = { temp in
          uSelf.currentTemp = temp
        }
    }
}
```

Capture Lists

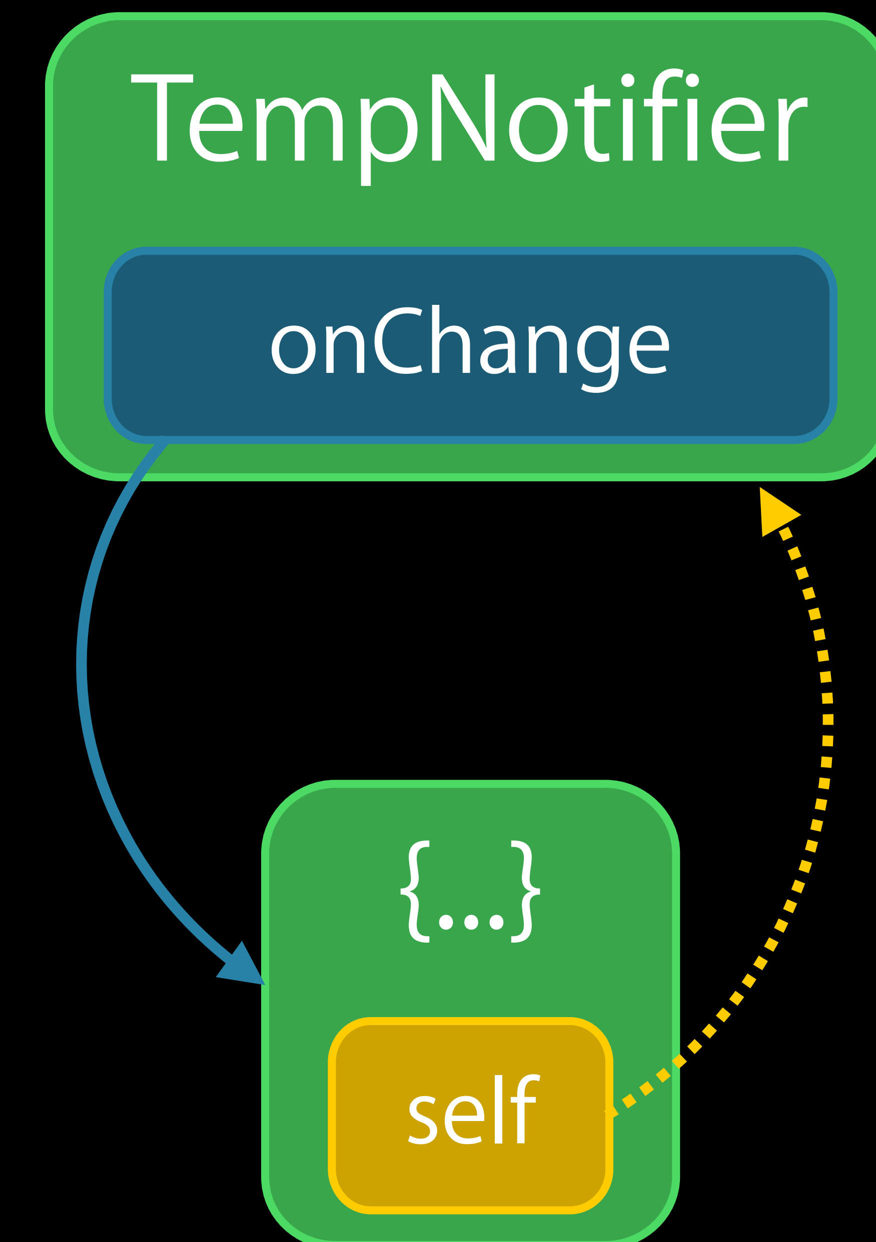
```
class TempNotifier {
    var onChange: (Int) -> Void = {}
    var currentTemp = 72

    init() {
        onChange = { temp in
          self.currentTemp = temp
        }
    }
}
```

Capture Lists

```
class TempNotifier {
    var onChange: (Int) -> Void = {}
    var currentTemp = 72

    init() {
        onChange = {[unowned self] temp in
          self.currentTemp = temp
        }
    }
}
```

Capture Lists

```
class TempNotifier {
    var onChange: (Int) -> Void = {}
    var currentTemp = 72

    init() {
        onChange = {[unowned self] temp in
          self.currentTemp = temp
        }
    }
}
```



Closures

Closures

Concise, expressive syntax

Closures

Concise, expressive syntax

Simple memory model

Closures

Concise, expressive syntax

Simple memory model

Supports functional programming idioms

Pattern Matching

Joe Groff
Swift Compiler Engineer

Switch

```
func describe(value: Int) {
    switch value {
      case 1:
        println("the loneliest number that you'll ever do")
      case 2:
        println("can be as bad as one")
      default:
        println("just some number")
    }
}
```

Switch

```
func describe(value: String) {
    switch value {
      case "one":
        println("the loneliest number that you'll ever do")
      case "two":
        println("can be as bad as one")
      default:
        println("just some number")
    }
}
```

Switch

```
func describe(value: String) {
    switch value {
      case "one":
        println("a few")
      case "two":
        println("a lot")
      default:
        println("a ton")
    }
}
```

Switch

```
func describe(value: Int) {
    switch value {
      case 0...4:
        println("a few")
      case 5...12:
        println("a lot")
      default:
        println("a ton")
    }
}
```

Enumerations

```
enum TrainStatus {
    case OnTime
    case Delayed
}
```

Enumerations

```
enum TrainStatus {
    case OnTime
    case Delayed(Int)
}
```

Enumerations

```
switch trainStatus {
  case .OnTime:
    println("on time")
  case .Delayed:
    println("delayed")
}
```

Enumerations

```
switch trainStatus {
  case .OnTime:
    println("on time")
  case .Delayed(let minutes):
    println("delayed")
}
```

Enumerations

```
switch trainStatus {
  case .OnTime:
    println("on time")
  case .Delayed(let minutes):
    println("delayed by \(minutes) minutes")
}
```

Enumerations

```
switch trainStatus {
  case .OnTime:
    println("on time")
  case .Delayed(let minutes):
    println("delayed by \(minutes) minutes")
}

println("let's take \(minutes) and come back")
// error: use of unresolved identifier 'minutes'
```

Patterns

```
switch trainStatus {
  case .OnTime:
    println("on time")
  case .Delayed(let minutes):
    println("delayed by \(minutes) minutes")
}
```

Patterns

```
switch trainStatus {
  case .OnTime:
    println("on time")
  case .Delayed(let minutes):
    println("delayed by \(minutes) minutes")
}
```

Patterns

```
switch trainStatus {
  case .OnTime:
    println("on time")
  case .Delayed(let minutes):
    println("delayed by \(minutes) minutes")
}
```

Patterns Compose

```
switch trainStatus {
  case .OnTime:
    println("on time")
```

Patterns Compose

```
switch trainStatus {
  case .OnTime:
    println("on time")
  case .Delayed(1):
    println("nearly on time")
```

Patterns Compose

```
switch trainStatus {
  case .OnTime:
    println("on time")
  case .Delayed(1):
    println("nearly on time")
  case .Delayed(2...10):
    println("almost on time, I swear")
```

Patterns Compose

```
switch trainStatus {
  case .OnTime:
    println("on time")
  case .Delayed(1):
    println("nearly on time")
  case .Delayed(2...10):
    println("almost on time, I swear")
  case .Delayed(_):
    println("it'll get here when it's ready")
```

Patterns Compose

```
enum VacationStatus {
    case Traveling(TrainStatus)
    case Relaxing(daysLeft: Int)
}
```

Patterns Compose

Patterns Compose

```
switch vacationStatus {
```

Patterns Compose

```
switch vacationStatus {
  case .Traveling(.OnTime):
    tweet("Train's on time! Can't wait to get there!")
```

Patterns Compose

```
switch vacationStatus {
  case .Traveling(.OnTime):
    tweet("Train's on time! Can't wait to get there!")
  case .Traveling(.Delayed(1...15)):
    tweet("Train is delayed.")
```

Patterns Compose

```
switch vacationStatus {
  case .Traveling(.OnTime):
    tweet("Train's on time! Can't wait to get there!")
  case .Traveling(.Delayed(1...15)):
    tweet("Train is delayed.")
  case .Traveling(.Delayed(_)):
    tweet("OMG when will this train ride end #railfail")
```

Type Patterns

Type Patterns

```
func tuneUp(car: Car) {
    switch car {
```

Type Patterns

```
func tuneUp(car: Car) {
    switch car {
      case let formulaOne as FormulaOne:
```

Type Patterns

```
func tuneUp(car: Car) {
    switch car {
      case let formulaOne as FormulaOne:
```

Type Patterns

```
func tuneUp(car: Car) {
    switch car {
      case let formulaOne as FormulaOne:
```

Type Patterns

```
func tuneUp(car: Car) {
    switch car {
      case let formulaOne as FormulaOne:
        formulaOne.enterPit()
```

Type Patterns

```
func tuneUp(car: Car) {
    switch car {
      case let formulaOne as FormulaOne:
        formulaOne.enterPit()
      case let raceCar as RaceCar:
        if raceCar.hasTurbo { raceCar.tuneTurbo() }
```

Type Patterns

```
func tuneUp(car: Car) {
    switch car {
      case let formulaOne as FormulaOne:
        formulaOne.enterPit()
      case let raceCar as RaceCar:
        if raceCar.hasTurbo { raceCar.tuneTurbo() }
        fallthrough
```

Type Patterns

```
func tuneUp(car: Car) {
    switch car {
      case let formulaOne as FormulaOne:
        formulaOne.enterPit()
      case let raceCar as RaceCar:
        if raceCar.hasTurbo { raceCar.tuneTurbo() }
        fallthrough
      default:
        car.checkOil()
        car.pumpTires()
    }
}
```

Tuple Patterns

Tuple Patterns

```
let color = (1.0, 1.0, 1.0, 1.0)
switch color {
  case (0.0, 0.5...1.0, let blue, _):
    println("Green and \(blue * 100)% blue")
```

Tuple Patterns

```
let color = (1.0, 1.0, 1.0, 1.0)
switch color {
  case (0.0, 0.5...1.0, let blue, _):
    println("Green and \(blue * 100)% blue")
```

Tuple Patterns

```
let color = (1.0, 1.0, 1.0, 1.0)
switch color {
  case (0.0, 0.5...1.0, let blue, _):
    println("Green and \(blue * 100)% blue")
```

Tuple Patterns

```
let color = (1.0, 1.0, 1.0, 1.0)
switch color {
  case (0.0, 0.5...1.0, let blue, _):
    println("Green and \(blue * 100)% blue")
```

Tuple Patterns

```
let color = (1.0, 1.0, 1.0, 1.0)
switch color {
  case (0.0, 0.5...1.0, let blue, _):
    println("Green and \(blue * 100)% blue")
```

Tuple Patterns

```
let color = (1.0, 1.0, 1.0, 1.0)
switch color {
  case (0.0, 0.5...1.0, let blue, _):
    println("Green and \(blue * 100)% blue")
```

Tuple Patterns

```
let color = (1.0, 1.0, 1.0, 1.0)
switch color {
  case (0.0, 0.5...1.0, let blue, _):
    println("Green and \(blue * 100)% blue")

  case let (r, g, b, 1.0) where r == g && g == b:
    println("Opaque grey \(r * 100)%")
```

Validating a Property List

Validating a Property List

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
```

Validating a Property List

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State?
```

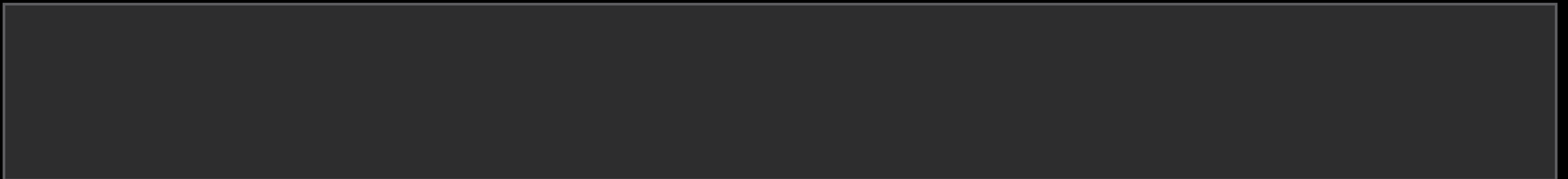
Validating a Property List

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State?

stateFromPlist(["name": "California",
                "population": 38_040_000,
                "abbr": "CA"])
```

```
State(name: California, population: 38040000, abbr: CA)
```

Validating a Property List

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State?

stateFromPlist(["name": "California",
                "population": "hella peeps",
                "abbr": "CA"])
```



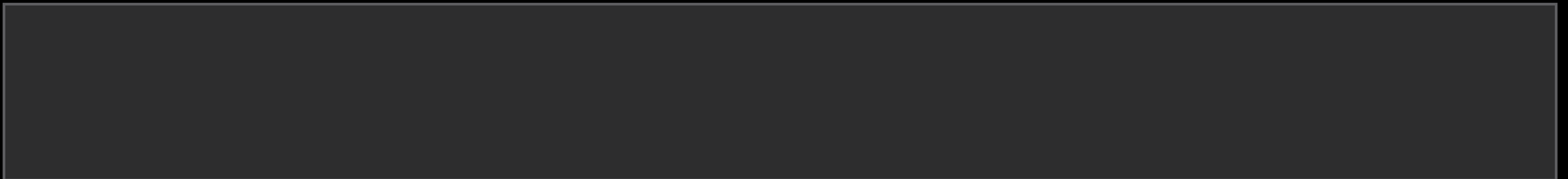
Validating a Property List

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State?

stateFromPlist(["name": "California",
                "population": "hella peeps",
                "abbr": "CA"])
```

```
nil
```

Validating a Property List

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State?

stateFromPlist(["name": "California",
                "population": 38_040_000,
                "abbr": "Cali"])
```



Validating a Property List

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State?

stateFromPlist(["name": "California",
                "population": 38_040_000,
                "abbr": "Cali"])
```

```
nil
```

Putting Patterns Together

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {
    var name: NSString?
    switch list["name"] {
      case
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {
    var name: NSString?
    switch list["name"] {
      case .Some(                        ):
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {
    var name: NSString?
    switch list["name"] {
      case .Some(             as NSString):
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {
    var name: NSString?
    switch list["name"] {
      case .Some(let listName as NSString):
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {
    var name: NSString?
    switch list["name"] {
      case .Some(let listName as NSString):
        name = listName
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {
    var name: NSString?
    switch list["name"] {
      case .Some(let listName as NSString):
        name = listName
      default:
        name = nil
    }
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {
    var name: NSString?
    switch list["name"] {
      case .Some(let listName as NSString):
        name = listName
      default:
        name = nil
    }
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {

    switch (list["name"], list["population"], list["abbr"]) {
      case .Some(let listName as NSString):
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {

    switch (list["name"], list["population"], list["abbr"]) {
      case (.Some(let listName as NSString),
            .Some(let pop as NSNumber),
            .Some(let abbr as NSString))
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {

    switch (list["name"], list["population"], list["abbr"]) {
      case (.Some(let listName as NSString),
            .Some(let pop as NSNumber),
            .Some(let abbr as NSString))
      where abbr.length == 2:
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {

    switch (list["name"], list["population"], list["abbr"]) {
      case (.Some(let listName as NSString),
            .Some(let pop as NSNumber),
            .Some(let abbr as NSString))
      where abbr.length == 2:
        return State(name: listName, population: pop, abbr: abbr)
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {

    switch (list["name"], list["population"], list["abbr"]) {
      case (.Some(let listName as NSString),
            .Some(let pop as NSNumber),
            .Some(let abbr as NSString))
      where abbr.length == 2:
        return State(name: listName, population: pop, abbr: abbr)
      default:
        return nil
```

Putting Patterns Together

```
func stateFromPlist(list: Dictionary<String, AnyObject>)
  -> State? {

    switch (list["name"], list["population"], list["abbr"]) {
      case (.Some(let listName as NSString),
            .Some(let pop as NSNumber),
            .Some(let abbr as NSString))
      where abbr.length == 2:
        return State(name: listName, population: pop, abbr: abbr)
      default:
        return nil
    }
}
```

Pattern Matching

Pattern Matching

Tests the structure of values

Pattern Matching

Tests the structure of values

Improves readability—Patterns are concise and composable

Pattern Matching

Tests the structure of values

Improves readability—Patterns are concise and composable

Improves safety—Variable bindings are tied to the conditions they depend on

Swift

Swift

Optionals

Swift

Optionals

Memory management

Swift

Optionals

Memory management

Initialization

Swift

Optionals

Memory management

Initialization

Closures

Swift

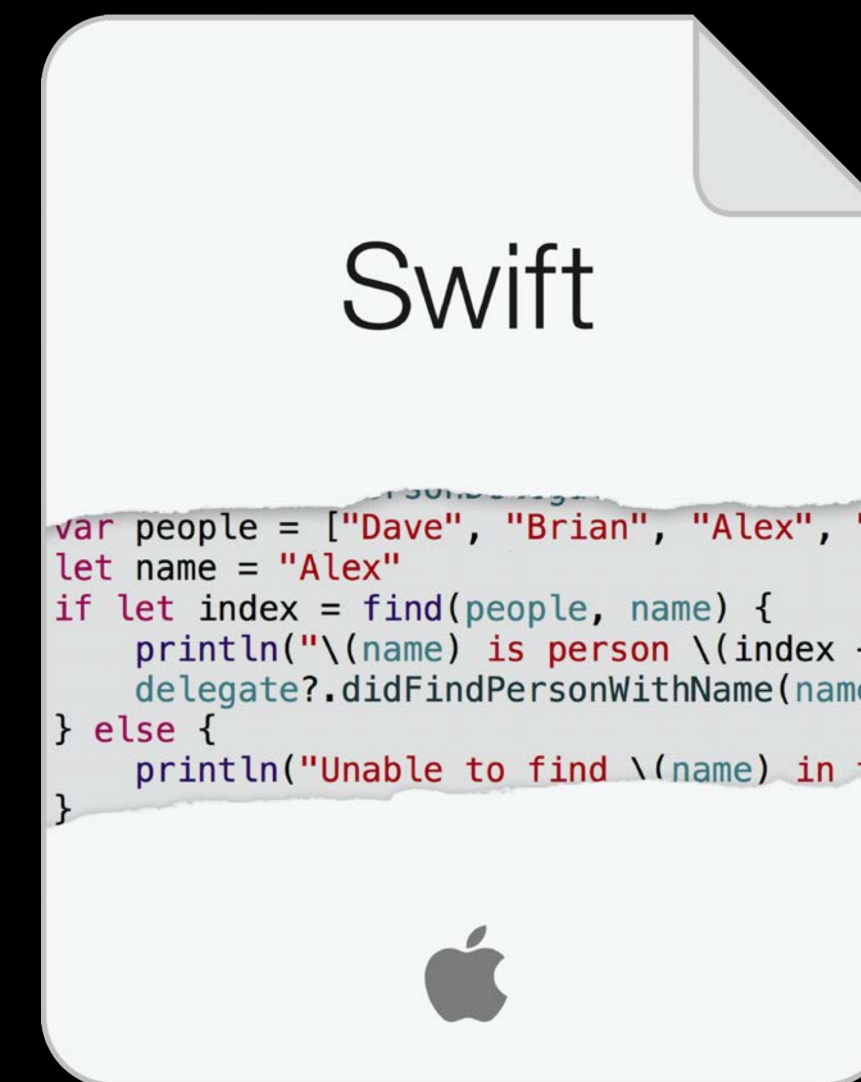
- Optionals
- Memory management
- Initialization
- Closures
- Pattern matching

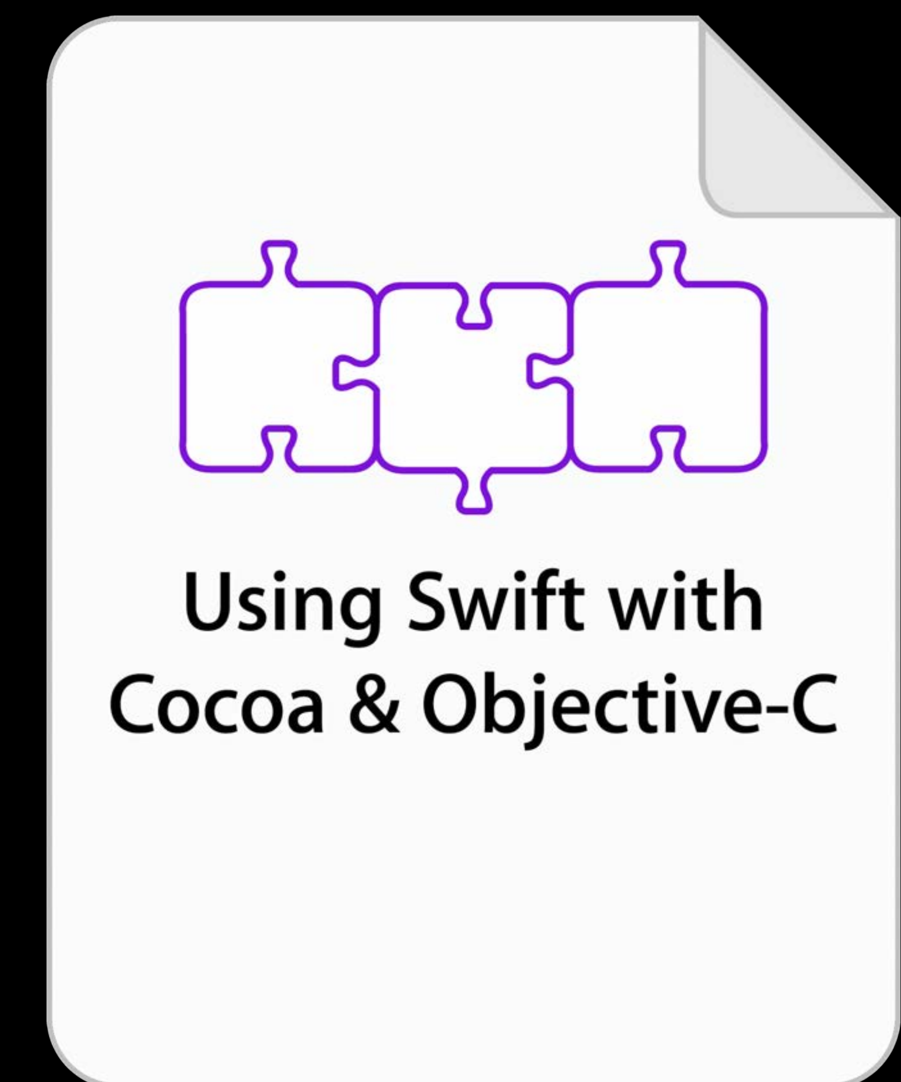
More Information

Dave DeLong
App Frameworks and Developer Tools Evangelist
delong@apple.com

Documentation
The Swift Programming Language
Using Swift with Cocoa and Objective-C
http://developer.apple.com

Apple Developer Forums
http://devforums.apple.com





Related Sessions

• Introduction to Swift	Presidio	Tuesday 2:00PM
• Advanced Swift	Presidio	Thusday 11:30AM
• Integrating Swift with Objective-C	Presidio	Wednesday 9:00AM
• Swift Interoperability in Depth	Presidio	Wednesday 3:15PM
• Swift Playgrounds	Presidio	Wednesday 11:30AM
• Introduction to LLDB and the Swift REPL	Mission	Thursday 10:15AM
• Advanced Swift Debugging in LLDB	Mission	Friday 9:00AM

Labs

Swift	Tools Lab A	Wednesday 2:00PM
Swift	Tools Lab A	Thursday 9:00AM
Swift	Tools Lab A	Thursday 2:00PM
Swift	Tools Lab A	Friday 9:00AM
Swift	Tools Lab A	Friday 2:00PM

WWDC14