Swift Quiz

Vincent Pradeilles (@v_pradeilles) – PhotoRoom

Question #01

Overridden methods and default value of arguments

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}
```

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}
```

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
```

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()
```

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()

// Will this print: "SubClass implementation – value: 100"?
```

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()

// Will this print: "SubClass implementation – value: 100"?
```

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()

// Will this print: "SubClass implementation – value: 100"?
```

Yes! 

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()

// Will this print: "SubClass implementation – value: 100"?
```

No! 

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10
```

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
    }
}

let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10
```

```
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10
```

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10
```

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10
```

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10
```

This code actually mixes two different mechanisms:

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10
```

This code actually mixes two different mechanisms:

- The implementation of the method is **dynamically resolved** (i.e. it depends on the type of the instance at runtime)

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10
```

This code actually mixes two different mechanisms:

- The implementation of the method is **dynamically resolved** (i.e. it depends on the type of the instance at runtime)
- The default value of the argument is **statically resolved** (i.e. it depends on the type of the variable at compile time)

```swift
class BaseClass {
    func display(value: Int = 10) {
        print("BaseClass implementation – value: \(value)")
    }
}

class SubClass: BaseClass {
    override func display(value: Int = 100) {
        print("SubClass implementation – value: \(value)")
   }
}

let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10
```

```
let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10
```

```swift
let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10


let instance: SubClass = SubClass()
instance.display()

> SubClass implementation – value: 100
```

```
let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10
```

```
let instance: SubClass = SubClass()
instance.display()

> SubClass implementation – value: 100
```

```
let instance = SubClass()
instance.display()

> SubClass implementation – value: 100
```

```swift
let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10

let instance: SubClass = SubClass()
instance.display()

> SubClass implementation – value: 100

let instance = SubClass()
instance.display()

> SubClass implementation – value: 100
```

```
let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10

let instance: SubClass = SubClass()
instance.display()

> SubClass implementation – value: 100

let instance = SubClass()
instance.display()

> SubClass implementation – value: 100
```

Be weary of providing a default value to an argument when you’re working with a class!

```
let instance: BaseClass = SubClass()
instance.display()

> SubClass implementation – value: 10

let instance: SubClass = SubClass()
instance.display()

> SubClass implementation – value: 100

let instance = SubClass()
instance.display()

> SubClass implementation – value: 100
```

Be weary of providing a default value to an argument when you’re working with a class!

(Unless the method is either **`private`** or **`final`**)

Question #02

Constant Property and Mutating Method

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}
```

Yes! 

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}
```

No! 

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
        self.age += 1 // ❌
    }
}
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}

Person(age: 20)
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}

var person = Person(age: 20)
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}

var person = Person(age: 20)

person.incrementAge()
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}

var person = Person(age: 20)

person =
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}

var person = Person(age: 20)

person = Person(age: )
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
    }
}

var person = Person(age: 20)

person = Person(age: person.age + 1)
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
        person = Person(age: person.age + 1)
    }
}

var person = Person(age: 20)
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
        person = Person(age: self.age + 1)
    }
}

var person = Person(age: 20)
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
        self = Person(age: self.age + 1)
    }
}

var person = Person(age: 20)
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
        self = Person(age: self.age + 1)
    }
}

var person = Person(age: 20)

person.incrementAge()
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
        self = Person(age: self.age + 1)
    }
}

var person = Person(age: 20)

person.incrementAge() 🥳
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
        self = Person(age: self.age + 1)
    }
}
```

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
        self = Person(age: self.age + 1)
    }
}
```

- This technique is known as “re-assigning self”

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
        self = Person(age: self.age + 1)
    }
}
```

- This technique is known as “re-assigning self”

```
struct Person {
    let age: Int

    mutating func incrementAge() {
        // Can we increment `age` here?
        self = Person(age: self.age + 1)
    }
}
```

- This technique is known as “re-assigning self”
- It can result in some tricky code, as it allows assigning a new value without having a “=” operator at the call site

Question #03

Empty Enum

```swift
enum Empty { }

func magic<T>(_ empty: Empty) -> T { }

// Do you think this code builds?
```

```
enum Empty { }

func magic<T>(_ empty: Empty) -> T { }

// Do you think this code builds?
```

```swift
enum Empty { }

func magic<T>(_ empty: Empty) -> T { }

// Do you think this code builds?
```

```swift
enum Empty { }

func magic<T>(_ empty: Empty) -> T { }

// Do you think this code builds?
```

```swift
enum Empty { }

func magic<T>(_ empty: Empty) -> T { }

// Do you think this code builds?
```

```swift
enum Empty { }

func magic<T>(_ empty: Empty) -> T { }

// Do you think this code builds?
```

Yes! 

```
enum Empty { }

func magic<T>(_ empty: Empty) -> T { }

// Do you think this code builds?
```

No! 

```swift
enum Empty { }

func magic<T>(_ empty: Empty) -> T { }

// Do you think this code builds? ✅
```

```
struct Person {
    let age: Int
}

([] as [Person]).allSatisfy { $0.age > 18 }
```

```
struct Person {
    let age: Int
}

// this evaluates to `true`
([] as [Person]).allSatisfy { $0.age > 18 }
```

```swift
enum Empty { }

func magic<T>(_ empty: Empty) -> T { }

// Do you think this code builds? ✅
```

```
enum Empty { }
```

```
enum Never { }
```

```
enum Never { }

func handle<Value>(result: Result<Value, Never>) {
    switch result {
    case .success(let value):
        print(value)
//    case .failure(let error):
//        no need to implement an
//        impossible code path
    }
}
```

```
fatalError()
```

Summary

Unconditionally prints a given message and stops execution.

Declaration

```
func fatalError(_ message: @autoclosure () -> String = String(),
file: StaticString = #file, line: UInt = #line) -> Never
```

Parameters

`message`	The string to print. The default is an empty string.
`file`	The file name to print with `message`. The default is the file where `fatalError(_:file:line:)` is called.
`line`	The line number to print along with `message`. The default is the line number where `fatalError(_:file:line:)` is called.

Open in Developer Documentation

`fatalError()`

Summary

Unconditionally prints a given message and stops execution.

Declaration

```
func fatalError(_ message: @autoclosure () -> String = String(),
file: StaticString = #file, line: UInt = #line) -> Never
```

Parameters

`message`	The string to print. The default is an empty string.
`file`	The file name to print with `message`. The default is the file where `fatalError(_:file:line:)` is called.
`line`	The line number to print along with `message`. The default is the line number where `fatalError(_:file:line:)` is called.

Open in Developer Documentation

`fatalError()`

```
enum Never { }
```

```
enum Never { }
```

- Never is a type that lets us represent impossible codepaths

`enum Never { }`

- Never is a type that lets us represent impossible codepaths
- The Swift compiler is smart enough to understand it and adapt the errors and warnings it will emit in consequence

`enum Never { }`

- Never is a type that lets us represent impossible codepaths
- The Swift compiler is smart enough to understand it and adapt the errors and warnings it will emit in consequence
- Never can also be helpful when prototyping and trying to make new code build successfully 👌

`enum Never { }`

- Never is a type that lets us represent impossible codepaths
- The Swift compiler is smart enough to understand it and adapt the errors and warnings it will emit in consequence
- Never can also be helpful when prototyping and trying to make new code build successfully 👌
- (If you're interested in the mathematical foundation of this, check out what's the Curry-Howard correspondence)


That's all Folks!

Thank You! 

Twitter

YouTube

https://www.photoroom.com/company/