

## 18. Access Control and Code Organization

Các kiểu Swift có thể được khai báo với các thuộc tính, phương thức, phương thức khởi tạo và thậm chí cả các kiểu lồng nhau khác. Những phần tử này có thể được nghĩ như giao diện - interface - cho code của bạn.

Khi code phát triển phức tạp, việc kiểm soát giao diện này trở thành một phần quan trọng của thiết kế phần mềm. Bạn có thể muốn tạo ra các phương thức mà phục vụ như "các trợ giúp - helpers" cho code của bạn, hoặc các thuộc tính mà được thiết kế để theo dõi các trạng thái bên trong mà bạn không muốn là một phần của giao diện (interface) code của bạn.

Swift giải quyết những vấn đề này với một khu vực tính năng được gọi là điều khiển truy cập - access control, mà cho phép bạn điều khiển giao diện có thể nhìn thấy được (viewable interface) code của bạn. Điều khiển truy cập cho phép bạn, tác giả thư viện, ẩn sự thực thi phức tạp tách khỏi người dùng. Trạng thái nội bộ ẩn này đôi khi tham chiếu tới là bất biến (the invariant), mà giao diện phổ biến của bạn sẽ luôn duy trì. Việc ngăn chặn truy cập trực tiếp trạng thái nội bộ của một mô hình và duy trì sự bất biến là một khái niệm thiết kế phần mềm cơ bản được gọi là đóng gói - encapsulation. Trong chương này, bạn sẽ học truy cập điều khiển là gì, vấn đề nó giải quyết, và cách áp dụng nó.

### Problems introduced by lack of access control

Tưởng tượng một chút bạn đang viết một thư viện ngân hàng. Thư viện này sẽ giúp phục vụ là nền tảng cho các khách hàng của bạn (ngân hàng khác) để viết phần mềm ngân hàng của họ.

Trong playground, bắt đầu với giao thức sau:

```
// A protocol describing core functionality for an account
protocol Account {
    associatedtype Currency
    var balance: Currency { get }
    func deposit(amount: Currency)
    func withdraw(amount: Currency)
}
```

Code này chứa Account, một giao thức mà mô tả bất cứ điều gì một tài khoản có - khả năng gửi (deposit), rút ra (withdraw) và kiểm tra số dư của quỹ. Bây giờ bổ sung một kiểu tuân theo với code bên dưới:

```

typealias Dollars = Double
// A U.S. Dollar based "basic" account.
class BasicAccount: Account {
    var balance: Dollars = 0.0
    func deposit(amount: Dollars) {
        balance += amount
    }
    func withdraw(amount: Dollars) {
        if amount <= balance {
            balance -= amount
        } else {
            balance = 0
        }
    }
}

```

Lớp tuân theo này, BasicAccount, thực thi deposit(amount:) và withdraw(amount:) bằng cách đơn giản cộng hoặc trừ từ số dư - balance. ( được nhập là Dollars , và tên hiệu - alias - với Double ).

Mặc dù code này là rất đơn giản, bạn có thể nhận thấy một vấn đề nhỏ. Thuộc tính balance trong giao thức Account được thiết kế chỉ đọc - nói cách khác, nó chỉ có một get được định nghĩa. Tuy nhiên , sự thực thi của BasicAccount yêu cầu số dư được khai báo là một biến để mà giá trị có thể được cập nhật khi Quỹ được gửi hoặc bị rút.

Không có gì có thể ngăn chặn code khác từ việc gán trực tiếp các giá trị mới cho số Dư :

```

// Create a new account
let account = BasicAccount()
// Deposit and withdraw some money
account.deposit(amount: 10.00)
account.withdraw(amount: 5.00)
// ... or do evil things!
account.balance = 1000000.00

```

Ồ không ! Mặc dù bạn đã thiết kế cẩn thận giao thức Account để chỉ có thể được gửi hoặc rút các Quỹ, các chi tiết thực thi của BasicAccount mà cho phép nó cập nhật Số dư riêng của nó có thể được sử dụng bất kỳ đâu trong code.

May thay, bạn có thể sử dụng điều khiển truy cập để giới hạn phạm vi mà mã của bạn có thể nhìn thấy được đối với các kiểu, các tập tin hoặc thậm trí cả các module khác.

Ghi chú:

Truy cập điều khiển không phải là tính năng bảo mật mà bảo vệ code của bạn từ các hackers độc hại. Thay vào đó, nó cho phép bạn thể hiện ý định bằng các tạo ra các lỗi biên dịch hữu ích nếu người dùng cố gắng truy cập trực tiếp chi tiết thực thi mà có thể làm ảnh hưởng tới sự bất biến (invariant)

## Introducing access control

Bạn có thể bổ sung các sửa đổi bằng cách đặt một từ khoá chỉnh sửa lên trước một thuộc tính, phương thức hoặc kiểu khai báo.

Bổ sung chỉnh sửa điều khiển truy cập `private(set)` tới định nghĩa của `balance` trong `BasicAccount` :

```
private(set) var balance: Dollars = 0.0
```

Sửa đổi truy cập bên trên được đặt trước khai báo thuộc tính , và có một chỉnh sửa optional `get/set` trong ngoặc đơn. Trong ví dụ này, setter của `balance` được tạo ra là `private`.

Bạn sẽ chuyển đổi các chi tiết của `private` ngắn hơn, nhưng bạn có thể thấy nó trong thực tế đã sẵn sàng: code của bạn không còn được biên dịch nữa !

```
// Create a new account
let account = BasicAccount()
// Deposit and withdraw some money
account.deposit(amount: 10.00)
account.withdraw(amount: 5.00)
// ... or do evil things!
account.balance = 1000000.00
```

Cannot assign to property: 'balance' setter is inaccessible

Bằng việc bổ sung `private` tới thuộc tính setter , thuộc tính đã tạo ra không thể truy cập được tới code đang sử dụng.

Điều này minh họa lợi ích cơ bản của các chỉnh sửa truy cập: truy cập bị hạn chế tới code mà cần hoặc phải có quyền truy cập , và bị hạn chế từ code mà không có . Hiệu quả , truy cập điều khiển cho phép bạn điều khiển khả năng truy cập giao diện của code trong khi định nghĩa bất kỳ thuộc tính , phương thức hoặc kiểu nào mà bạn cần phải thực thi hành vi bạn muốn.

Chỉnh sửa `private` được sử dụng trong ví dụ ngắn ở trên là một trong vài chỉnh sửa truy cập có sẵn cho bạn trong Swift :

- `private` : Chỉ có thể truy cập tới kiểu đang định nghĩa, tất cả các kiểu lồng và các extension trên kiểu đó trong cùng tập tin nguồn.
- `fileprivate` : Có thể truy cập từ bất cứ nơi nào trong tập tin nguồn mà nó được định nghĩa.
- `internal` : Có thể truy cập từ bất cứ nơi nào trong module mà nó định nghĩa. Điều này là cấp độ truy cập mặc định.
- `public` : Có thể truy cập từ bất cứ nơi nào trong module mà nó định

nghĩa, miễn là các module phần mềm khác mà import module này

- open : Giống như public, với khả năng bổ sung có thể được ghi đè bởi code trong các module khác.

Tiếp theo, bạn sẽ học thêm về các chỉnh sửa này, khi nào sử dụng chúng và cách áp dụng chúng cho code của bạn :

- **Private**

Sửa đổi truy cập private hạn chế truy cập cho thực thể nó được định nghĩa bên trong, cũng như bất kỳ kiểu lồng nhau nào trong nó - cũng được gọi là "lexical scope - phạm vi từ vựng ". Các extension trên kiểu trong cùng một tập tin nguồn cũng có thể truy cập vào thực thể.

Để minh họa, chúng ta tiếp tục với thư viện ngân hàng của bạn bằng cách mở rộng hành vi của BasicAccount để tạo ra một CheckingAccount :

```
class CheckingAccount: BasicAccount {
  private let accountNumber = UUID().uuidString
  class Check {
    let account: String
    var amount: Dollars
    private(set) var cashed = false
    func cash() {
      cashed = true
    }
    init(amount: Dollars, from account: CheckingAccount) {
      self.amount = amount
      self.account = account.accountNumber
    }
  }
}
```

CheckingAccount có một accountNumber được khai báo là private. CheckingAccount cũng có một kiểu lồng nhau Check mà có thể đọc giá trị private của accountNumber trong phương thức khởi tạo của nó.

Ghi chú:

Trong ví dụ này lớp UUID được sử dụng để tạo ra số tài khoản duy nhất. Lớp này là một phần của Foundation, vì vậy đừng quên import nó !

Việc kiểm tra tài khoản sẽ có thể được viết và tiền mặt kiểm tra là tốt. Bổ các phương thức sau tới CheckingAccount :

```

func writeCheck(amount: Dollars) -> Check? {
    guard balance > amount else {
        return nil
    }
    let check = Check(amount: amount, from: self)
    withdraw(amount: check.amount)
    return check
}
func deposit(_ check: Check) {
    guard !check.cashed else {
        return
    }
    deposit(amount: check.amount)
    check.cash()
}

```

Trong khi CheckingAccount vẫn có thể tạo ra các khoản gửi và rút ra cơ bản, bây giờ nó cũng có thể viết và gửi séc ! phương thức writeCheck(amount:) kiểm tra Số dư đủ trước khi rút ra số tiền và tạo ra séc, và deposit(\_) sẽ không gửi séc (check) nếu nó đã được thanh toán. (cashed)

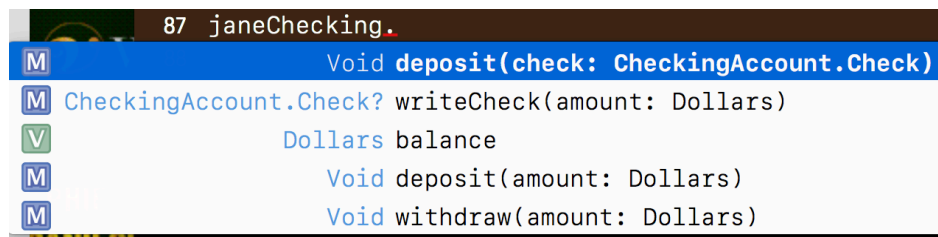
Cho code này thử trong playground bằng cách cho John viết một séc cho Jane:

```

// Create a checking account for John. Deposit $300.00
let johnChecking = CheckingAccount()
johnChecking.deposit(amount: 300.00)
// Write a check for $200.00
let check = johnChecking.writeCheck(amount: 200.0)!
// Create a checking account for Jane, and deposit the check.
let janeChecking = CheckingAccount()
janeChecking.deposit(check)
janeChecking.balance // 200.00
// Try to cash the check again. Of course, it had no effect on
// Jane's balance this time :]
janeChecking.deposit(check)
janeChecking.balance // 200.00

```

Tất nhiên, mã này làm việc tuyệt vời ; câu chuyện thực sự là những gì mà code này không thể làm được. Hãy nhớ rằng điều khiển truy cập cho phép bạn điều khiển giao diện cho code của bạn. hãy nhìn vào những gì cửa sổ tự động hoàn thành hiển thị là giao diện cho CheckingAccount :



```

87 janeChecking.
Void deposit(check: CheckingAccount.Check)
CheckingAccount.Check? writeCheck(amount: Dollars)
Dollars balance
Void deposit(amount: Dollars)
Void withdraw(amount: Dollars)

```

accountNumber được xử lý như một chi tiết thực thi của CheckingAccount, và nó không hiển thị tới code sử dụng.

Tương tự như vậy, Check tạo ra setter cho cashed riêng tư và yêu cầu người tiêu dùng sử dụng cash() thay vào đó :

```
deposit(amount: check.amount)
check.cash()
check.cashed = true
}
```

Cannot assign to property: 'cached' setter is inaccessible

Giao diện này cho Check một cách để người tiêu dùng đánh dấu một séc là đã thanh toán (cached) , mà không phải là cách khác ! Nói cách khác, không thể không-thanh toán một séc.

```
check.|
String account
Dollars amount
Void cash()
Bool cached
```

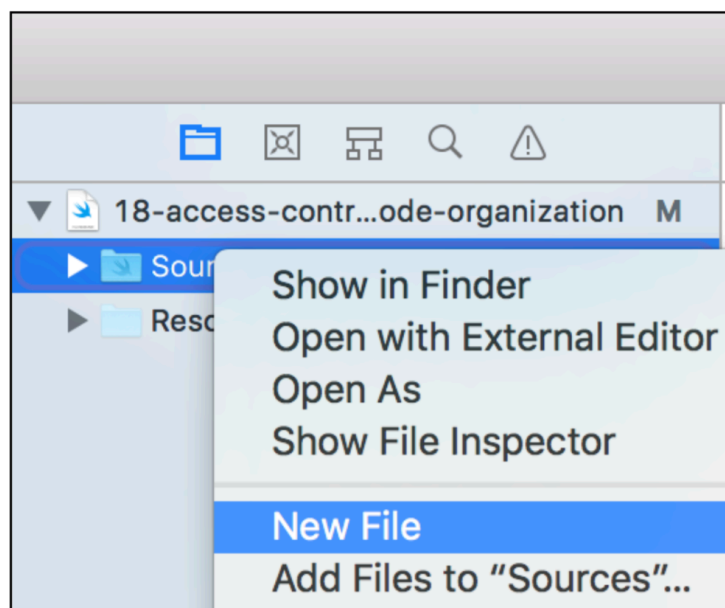
Expected member name following ' \$300.00

Trong khi thuộc tính account có giá trị của nó từ CheckingAccount, nhưng đó là chi tiết thực thi khác. Điều quan trọng đó là các chỉnh sửa truy cập cho phép code định dạng giao diện riêng của nó bất kể mã được sử dụng để thực thi nó.

- **Playground sources**

Trước khi tới với phần còn lại của chương này , bạn sẽ cần phải học một tính năng mới của playground Swift : source files

Trong Xcode, đảm bảo Project Navigator được hiển thị bằng cách tới View\Navigators\Show Project Navigator. Bên dưới cây thư mục playground bạn sẽ thấy một thư mục hơi mở có tên là Sources :



Nhấp chuột phải vào thư mục, select New File và đặt tên tập tin Account.swift. Di chuyển giao thức Account, lớp BasicAccount và Dollars typealias tất cả tới tập tin này.

Tạo ra một tập tin nguồn khác và đặt tên nó là Checking.swift. Chuyển CheckingAccount vào trong tập tin này.

Đó là nó! Nhưng điều quan trọng phải lưu ý về thư mục Source đó là code trong nó được coi là một module riêng biệt từ code trong playground của bạn.

Bạn có thể bình luận phần còn lại của code trong playground của bạn bây giờ. Nó sẽ không thể "thấy" code mà bạn vừa di chuyển cho đến phần sau trong chương này.

- **Fileprivate**

Liên quan mật thiết tới private là fileprivate, mà cho phép truy cập vào bất kỳ code nào được viết trong cùng một tập tin với thực thể, thay vì cùng phạm vi từ vựng (lexical) và các extension trong cùng tập tin, mà private có.

Bạn sẽ sử dụng hai tập tin mới bạn vừa tạo ra để thử sử dụng !

Ngay bây giờ , không có gì ngăn ngừa một coder lộn xộn người mà không đọc tài liệu bằng việc tạo ra một Check riêng của chúng. Trong code an toàn của bạn, bạn muốn một Check chỉ có nguồn gốc từ CheckingAccount để mà nó có thể theo dõi số dư.

Trong lớp Check of Checking.swift, thử bổ sung chỉnh sửa private tới phương thức khởi tạo.

```
private init(amount: Dollars, from account: CheckingAccount) {
```

Trong khi điều này ngăn chặn code xấu từ việc tạo ra một Check, bạn sẽ lưu ý rằng nó cũng ngăn chặn CheckingAccount khỏi việc tạo ra một cái như vậy. Các thực thể private có thể được truy cập từ bất cứ điều gì trong cùng phạm vi chuyển đổi (lexical), nhưng trong trường hợp này CheckingAccount là một bước ngoài phạm vi của Check. May thay , đây là nơi fileprivate rất có ích.

Thay thế phương thức khởi tạo với fileprivate :

```
fileprivate init(amount: Dollars, from account: CheckingAccount) {
```

Tuyệt ! Bây giờ CheckingAccount vẫn có thể viết Séc, nhưng bạn không thể tạo ra chúng từ bất cứ nơi nào khác.

Chỉnh sửa fileprivate là lý tưởng cho code mà là "dính liền" với một tập tin

nguồn, có nghĩa là code đó liên quan chặt chẽ hoặc đủ phục vụ với những mục đích phổ biến để có được sự chia sẻ nhưng truy cập được bảo vệ. Check và CheckingAccount là những ví dụ tuyệt vời của hai kiểu dính liền (cohesive).

- **Internal, public , and open**

Với private và file-private bạn đã có thể bảo vệ mã từ việc truy cập bởi các kiểu và các tập tin khác. Những truy cập hiệu chỉnh truy cập được chỉnh sửa từ các cấp độ truy cập mặc định là internal.

Cấp độ truy cập internal nghĩa là một thực thể có thể được truy cập từ bất kỳ nơi nào trong module phần mềm trong đó nó được định nghĩa. Tạo thời điểm này trong cuốn sách, bạn đã viết tất cả code của bạn trong một tập tin playground duy nhất, nghĩa là nó tất cả trong cùng modul.

Khi bạn bổ sung code tới thư mục chủ Sources trong playground của bạn, bạn đã tạo ra hiệu quả một module mà playground của bạn được sử dụng. Cách các playground được định nghĩa trong Xcode, tất cả các tập tin trong thư mục Sources là một phần của một module và mọi thứ trong playground là module khác mà tiêu thụ (consumes) module trong thư mục Sources.

- Internal

Trở lại playground của bạn, bỏ bình luận code mà xử lý John gửi séc tới Jane :

```
//Create a checking account for John. Deposit $300.00
let johnChecking = CheckingAccount()
johnChecking.deposit(amount: 300.00)
```

! Use of unresolved identifier 'CheckingAccount'

Bởi vì CheckingAccount không có truy cập chỉnh sửa được chỉ định , nó được coi là internal và do đó không thể truy cập tới playground mà sử dụng (consume) module tồn đó nó được định nghĩa.

Kết quả đó là Swift hiển thị một lỗi build cho việc cố gắng sử dụng kiểu CheckingAccount.

Để sửa chữa điều này, bạn sẽ cần phải học về các chỉnh sửa truy cập public và open.

Ghi chú:

Bởi vì internal là cấp độ truy cập mặc định, bạn không bao giờ cần phải khai báo rõ ràng code của bạn là internal. Dù bạn sử dụng từ khoá trong các định nghĩa là một vấn đề về phong cách và sở thích hay không

- Public

Để làm cho CheckingAccount hiển thị trong playground của bạn, bạn sẽ cần



phải thay đổi cấp độ truy cập từ internal thành public. Một Thực thể mà là public có thể thấy và được sử dụng bởi code bên ngoài module trong nơi nó được định nghĩa.

Bổ sung chỉnh sửa public tới lớp CheckingAccount :

```
public class CheckingAccount: BasicAccount {
```

Bạn sẽ cũng cần phải bổ sung public tới BasicAccount khi CheckingAccount là lớp con của nó :

```
public class BasicAccount: Account {
```

Playground bây giờ sẽ nhận dạng CheckingAccount , nhưng bạn vẫn không thể khởi tạo nó .

```
//Create a checking account for John. Deposit $300.00
let johnChecking = CheckingAccount()
johnChecking.deposit(300)
! 'CheckingAccount' initializer is inaccessible due to 'internal' protection level
```

Trong khi kiểu bản thân bây giờ là public, những thành viên của nó vẫn là internal và vì vậy không hiển thị bên ngoài module . Bạn sẽ cần phải bổ sung chỉnh sửa public tới tất cả các thực thể mà bạn muốn là một phần của giao diện module của bạn .

Bắt đầu bằng việc bổ sung một khởi tạo public tới BasicAccount và CheckingAccount:

```
public init() {}
```

```
public override init() {}
```

Tiếp theo, trong BasicAccount , bổ sung public tới balance, deposit(amount:) và withdraw(amount:). Bạn cũng sẽ cần phải làm cho typealias Dollars là public , vì typealias này bây giờ được sử dụng theo các phương thức public.

Cuối cùng, trong CheckingAccount , bổ sung public tới writeCheck(amount:), deposit(\_:) và lớp Check.

Lưu tất cả các tập tin ; bạn sẽ thấy rằng mọi thứ build và run !

Ghi chú:

Mặc dù BasicAccount chấp nhận Account, bạn có thể nhận thấy rằng playground không thể thấy Account, mà nó cũng không biết rằng BasicAccount tuân theo nó. Để sử dụng các module, sự tuân theo giao thức sẽ được ẩn đi nếu

bản thân giao thức không thể truy cập được.

- Open

Bây giờ CheckingAccount và các thành viên public của nó được hiển thị trong playground, bạn có thể sử dụng giao diện ngân hàng của bạn như được thiết kế.

Vâng- gần như ! Thư viện ngân hàng sẽ cung cấp một tập hợp các tài khoản phổ biến chẳng hạn như kiểm tra tài khoản, nhưng cũng được mở để mở rộng khả năng cho bất kỳ các kiểu tài khoản đặc biệt mà ngân hàng có thể có.

Trong playground của bạn, tạo ra một SavingsAccount tích lũy lãi suất mà là lớp con của BasicAccount :

```
class SavingsAccount : BasicAccount {  
    var interestRate : Double  
  
    init(interestRate : Double) {  
        self.interestRate = interestRate  
    }  
    func processInterest() {  
        let interest = balance*interestRate  
        deposit(amount: interest)  
    }  
}
```

Trong khi BasicAccount đã khai báo public và có thể truy cập tới playground, Swift sẽ cho thấy một lỗi build khi cố gắng tạo ra lớp con của BasicAccount :

```
class SavingsAccount : BasicAccount {  
    var interestRate : Double  
}
```

! Cannot inherit from non-open class 'BasicAccount' outside of its defining module

Đối với một lớp, phương thức hoặc thuộc tính bị ghi đè bởi code trong module khác, nó yêu cầu phải được khai báo là open. Mở Account.swift và thay thế chỉnh sửa truy cập public cho lớp BasicAccount với open :

```
open class BasicAccount : Account {
```

Bạn có thấy nó đi cùng với nhau không? Các giao diện mà bạn đã tạo ra bằng cách sử dụng cho phép lớp con public và open của BasicAccount để cung cấp các kiểu tài khoản mới. withdraw(amount:) và deposit(amount:), bởi vì chúng là public, có thể được sử dụng bởi các lớp con đó. Sự thực thi của withdraw(amount:) và deposit(amount:) là an toàn từ việc bị ghi đè bởi vì chúng chỉ là public, mà không phải là open!

Hãy tưởng tượng nếu bạn có thể ghi đè lên với withdraw(amount:) và deposit(amount:)

```
override func deposit(amount: Dollars) {  
    //LOL  
    super.deposit(amount: 1_000_000.00)  
}
```

! Overriding non-open instance method outside of its defining module x

Ồ không !

Nếu bạn đang tạo ra một thư viện, bạn thường muốn hạn chế khả năng các phương thức và các thuộc tính ghi đè vì bạn có thể tránh những hành vi khác bất ngờ. Chỉnh sửa truy cập open cho phép bạn điều khiển rõ ràng những gì các module khác thực hiện tới mã của bạn.

## Organizing code into extensions - Tổ chức code vào trong các Extension.

Một chủ đề của điều khiển truy cập là ý tưởng mà code của bạn nên được kết hợp lỏng lẻo (loosely couple) và độ kết dính cao (highly conhesive). Giới hạn code kết hợp lỏng lẻo một thực thể biết về bao nhiêu thực thể khác, mà lần lượt tạo ra các phần khác nhau code của bạn ít phụ thuộc vào các phần khác. Code kết dính cao ,như bạn đã học cho từ trước, giúp code liên quan chặt chẽ làm việc cùng nhau để hoàn thành nhiệm vụ,

Các tính năng Swift chẳng hạn như các chỉnh sửa truy cập , khi được sử dụng với các extension , có thể giúp bạn tổ chức code của mình cũng như khuyến khích để thiết kế phần mềm tốt.

## Extensions by behavior - các extension theo hành vi

Một chiến lược hiệu quả trong Swift là tổ chức code của bạn vào trong các extension theo hành vi. Bạn thậm trí có thể áp dụng các chỉnh sửa truy cập vào bản thân các extension, mà sẽ giúp bạn phân loại toàn bộ các phần của code là pulic, internal hoặc private.

Bắt đầu bằng cách bổ sung một số âm mưu (fraud) bảo vệ cơ bản tới CheckingAccount. Bổ sung các thuộc tính sau tới CheckingAccount :

```
private var issuedChecks : [Int] = []  
private var currentCheck = 1
```

Những thuộc tính này theo dõi séc mà được viết bởi việc kiểm tra tài khoản. Tiếp theo, Bổ sung private extension sau :

```
private extension CheckingAccount {
    func inspectForFraud(with checkNumber : Int) -> Bool {
        return issuedChecks.contains(checkNumber)
    }
    func nextNumber() -> Int {
        let next = currentCheck
        currentCheck += 1
        return next
    }
}
```

Hai phương thức này có thể được sử dụng bởi CheckingAccount để định rõ số của một Séc mà đã được viết , cũng như kiểm tra để đảm bảo nó thực tế đã được đưa ra bởi tài khoản.

Đáng chú ý , phần extension này được đánh dấu là private. Một private extension đánh dấu ngầm tất cả các thành viên của nó là private. Những công cụ bảo vệ chống gian lận này có ý nghĩa được sử dụng bởi duy nhất CheckingAccount - bạn chắc chắn không muốn code khác gia tăng số currentCheck.

Việc đặt hai phương thức này cùng với nhau cũng ràng buộc cùng hai phương thức liên quan và gắn kết. Nó rõ ràng cho bạn và ai khác cũng duy trì code mà hai điều này gắn kết và giúp giải quyết một nhiệm vụ chung.

## Extensions by protocol conformance - Các extension bởi sự tuân theo giao thức

Một kỹ thuật hiệu quả khác để tổ chức các extension của bạn dựa trên sự tuân theo giao thức. Bạn đã thấy kỹ thuật này trong Chương 16 "Protocols". Ví dụ, hãy làm cho CheckingAccount tuân theo CustomStringConvertible bằng cách bổ sung extension sau :

```
extension CheckingAccount : CustomStringConvertible {
    public var description : String {
        return "Checking Balance: ${balance}"
    }
}
```

Extension này thực thi CustomStringConvertible và điều quan trọng hơn :

- Làm cho nó mô tả rõ ràng là một phần của CustomStringConvertible
- Không giúp tuân theo các giao thức khác
- Có thể dễ dàng bị xóa mà không gây thiệt hại cho các phần còn lại của CheckingAccount
- là dễ để hiểu hơn !

