MentorMate LLC

Как да представим правилно типа Пари в Swift

Често забравяни правила

Александър Петров
2015

Съдържание

Контрол на версиите

Версия	Дата	Коментар
1.0	Септември, 15 2015	Начална версия

Въведение

Всяка книга, чиято цел е да представи даден програмен език започва с основните типове данни, които се поддържат в него. Типа данни се характеризира с възможните стойности, които могат да се съхраняват в него; операциите, които могат да се извършват върху тях; смисъла им и начина по който се съхраняват. Примитивните типове, които се поддържат от всички програмни езици представят имплементация на широко разпространени данни като например целите и дробните числа, символните низове и т.н.. Има един тип данни, който е основен и изключително разпространен, но който липсва в списъка с примитивни типове идващи с програмните езици, това са Парите. Това, разбира се, води до известни проблеми, които до голяма степен са еднотипни. За щастие, програмните езици ни дават възможност да използваме вече съществуващите примитивни типове за да представим нашите данни или да създадем собствени типове. *Как обаче е най-добре да представим Парите?* Този документ се опитва да покрие всички методи за представяне на Парите, техните положителни и отрицателни страни. Въпреки че за примерите е използван Swift[1], принципите и подходите са валидни за почти всички съвременни програмни езици.

[1] Примерите показани тук са актуализирани да използват Swift 2.0 синтаксис.

Използване на примитивни типове за представяне на Парите

Представяне чрез тип с плаваща запетая

Най-естественото представяне на Парите от реалния свят е чрез реално число:

Swift

```swift
let test: Double = 0.1
```

Както се вижда обаче ние запазихме числото в паметта не точно като реално, а като число с плаваща запетая, а те не са съвсем еквивалентни.

Double и **Float** са неподходящи типове, защото не могат да представят точно числа като 0.1 (или кое да е друго число по десет на отрицателна степен) и това важи за всички програмни езици, които използват техническия стандарт IEEE-754 за изчисления с числа с плаваща запетая. Да направим следния тест в Playground[2]:

Swift

```swift
let price: Double = 0.1
var sum: Double   = 0.0

for var i = 0; i < 100000; i++ {
    sum += price
}

print("Result is: \(sum)")
// Result is: 10000.0000000188
```

Резултата е изведен с подразбиращото се закръгляне на функцията print[3] за извеждане на числа с плаваща запетая на екрана, както се вижда то не съвпада с нашите очаквания, в действителност числото не съвпада с правилната му стойност като реално число още след стотната итерация на цикъла.

Независимо дали правите закръгляне след всяка операция или в последния момент, когато представяте числото е възможно да получите резултат, който не отговаря на очакванията ви. Когато правите изчисления с пари просто не можете да си позволите да губите пари, но истината е, че точно това се случва когато работите с плаваща запетая.

Едно от решенията е да използваме цяло число за представяне на парите.

---

[2] Изпълнява се продължително време.
[3] Println преди Swift 2.0

Представяне чрез тип цяло число

Integer типовете по дефиниция не могат да представят дроби и са *“точни”* (липсват грешки при закръгляне, когато се представят десетични числа). При това решение обикновено парите се представят в техните под-единици (центове, стотинки, пенита и т.н.). Обикновено се избира неявна фиксирана десетична точка, т.е. последните няколко цифри в дясно са заделени за дробната част, така че най-малката сума, която може да бъде представена е равна на една под-единица.

Swift

```
typealias Money  = Int
let price: Money = 1032 // $10.32 in cents
```

Практически това означава, че за да работите по този начин с пари е нужно да съхранявате и да изчислявате винаги в под-единици и да форматирате, към текст, когато представяте числото на екрана.

Този подход има своите негативи:

- Изисква се доста работа за да се реализира както трябва, което може да доведе до грешки в крайния продукт.
- Стойностите, който могат да се съхраняват в Integer типовете са ограничени и когато настъпи препълване, в някои програмни езици, като например Objective C, това обикновено бива *“заметено под килима”*, например ако към най-голямото число, което може да се съхрани в Integer добавим 1, резултата е равен 0 (ако типа е без знак) или е равен на най-голямото отрицателно число, което е възможно да се съхрани (ако типа е със знак). В Swift обаче препълването води до грешка по време на изпълнение.
- Неявната фиксирана десетична точка е най-голямата слабост на този подход. Тя е трудно да бъде променена и й липсва гъвкавост. Просто е невъзможно приложението ви да поддържа повече от една валута (например: долари и лева). За да решите проблема ще трябва да запазвате и позицията на десетичната запетая заедно със сумата, за щастие някои програмни езици обаче предлагат точно такъв тип данни, както ще видим по-нататък.

Препоръчително е да използвате този подход само ако в програмния език на който програмирате липсва по-добра алтернатива. Ако обаче той поддържа Decimal тип, то изграждането на вашето решение върху него е препоръчително.

Представяне чрез десетичен тип (Decimal)

Decimal типа представя десетични дроби и аритметиката свързана с тях. Той е проектиран да извършва операции върху тях без загуба на точност и с предсказуемо поведение при закръгляне. Конкретната имплементация на Apple е NSDecimalNumber.

Swift

```swift
typealias Money  = NSDecimalNumber
let price: Money = Money(string: "10.32") // $10.32
```

Вътрешно NSDecimalNumber е реализиран със знак, мантиса и експонента. Знака показва дали числото с фиксирана запетая е положително или отрицателно, мантисата е цяло число без знак, което съдържа значещите цифри, а експонентата определя къде точно следва да бъде поставена десетичната запетая в мантисата.

Важно е да се отбележи, че стойността на NSDecimalNumber обекта не може да се промени след като той бъде създаден.

Всички аритметични методи в класа имат алтернативни варианти, които приемат като втори параметър NSDecimalNumberHandler, чрез който може да се укаже закръглянето, което следва да бъде приложено, ако в резултат на операцията стойността има повече знаци след десетичната запетая от нужните ни.

Swift

```swift
typealias Money = NSDecimalNumber
let price   = Money(string: "10.32") // $10.32
let percent = NSDecimalNumber(string: "0.70") // 70%
let roundingHandler = NSDecimalNumberHandler(roundingMode: .RoundBankers, scale: 2,
    raiseOnExactness: false, raiseOnOverflow: false, raiseOnUnderflow: false,
    raiseOnDivideByZero: false)

let discountedPrice = price.decimalNumberByMultiplyingBy(percent,
    withBehavior: roundingHandler)

print("Result is: \(discountedPrice)")
// Result is: 7.22
```

Въпреки че мантисата може да съдържа до 38 значещи цифри, а експонентата да е от минус 127-ма до 128-ма степен, което ни дава голям диапазон на възможните стойности, е нужно да внимаваме да не допуснем препълване.

Освен за Парите има много случай, когато искаме от компютрите да представят размерни количества: стойности като шест метра, тридесет килограма и т.н.. Обикновено те се представят от програмистите с числови типове. Това обаче е отзвук от миналото.

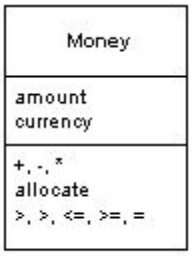
Благодарение на обектно ориентираните езици можем свободно да добавяме нови типове. При него, когато работим със стойности е добре да ги “опаковаме” в обект-стойност[4].

Съществува концепция за представянето на парите чрез обект-стойност. Тя е разработена от Мартин Фаулър (Martin Fowler) и Мат Фамъл (Matt Foemmel).

[4] В компютърната наука - обект-стойност е малък обект представящ прост субект. Тяхната еднаквост е базирана на тяхното състояние (стойността, която съдържат), а не на тяхната идентичност като обект. Примери за такива обекти са датите и низовете.

Представяне на Парите като обект-стойност

Основната идея на дизайн шаблона[5], който Мартин Фаулър предлага за представяне на Парите е да се използва клас или структура, който да може да се подава по стойност, да комбинира сумата с валутата, както и да гарантира определено поведение на обекта, като се предефинират операторите за сравнение, аритметичните оператори и като се добавят допълнителни помощни методи, които могат да са полезни при работата с него.



За сумата може да се използват цяло число или десетичен тип данни. В предишните точки бяха разгледани двата подхода, кои са техните предимства и недостатъци. Кой от тях ще изберем е въпрос на предпочитание.

Парите са тип обект-стойност, което предполага да бъдат сравнявани помежду си. За това е необходимо да се предефинират операциите за равенство и хеш стойност, за да са базирани на полетата за валута и сума. Два обекта Пари са равни ако валутите им съвпадат и сумите им са равни.

Най-важното поведение, което трябва да се имплементира е аритметичното. Трябва да може да използваме Парите, точно толкова лесно, колкото и всяко друго число. Нужно е обаче да въведем някои ограничения. При добавяне и изваждане е нужно да проверим дали двата обекта имат една и съща валута, като е най-лесно да върнем грешка за отговор ако това не е факт. Съществуват и по-сложни подходи, при които обекта съдържа “кошница” с валути и може да осъществява аритметични операции между тях, като конвертира валутите една към друга.

Умножението и делението са значително по-сложни, заради проблеми възникващи при закръглянето. Това е най-видимо ако трябва да се разделят парите на части. Например ако искаме да разделим 5 стотинки на две части, съответно 70% и 30%. Математически получаваме 3.5 стотинки за едната част и 1.5 за другата. Как обаче да ги закръглим без

[5] Шаблоните за дизайн (англ.: Software design pattern) представляват концепция предназначена за разрешаване на често срещани проблеми в обектно-ориентираното програмиране. Тази концепция предлага стандартни решения за архитектурни и концептуални проблеми в компютърното програмиране.

да изгубим или спечелим стотинка? Има различни подходи, но Мартин Фаулър препоръчва да се използва алгоритъм, който следи за остатъка и ако има такъв го разпределя стотинка по стотинка, по отделните части, докато не го изчерпи.

Един от очевидните допълнителни помощни методи е този за конвертиране между различни валути. Директния подход е да умножим по курса и да закръглим резултата към новата валута. Този подход работи в повечето случай, но понякога трябва да бъдат приложени специални правила за закръгляне, за това е хубаво да използваме Converter обект[6], който да реализира алгоритъмът за конвертиране.

Операторите за сравнение ни помагат при операции като сортиране. При тях отново трябва да се следи дали валутите съвпадат и отново е въпрос на избор дали да се върне грешка (да се хвърли изключение) или да се пробва конвертиране на едната валута към другата.

Обекта Пари може също така да предоставя помощен метод за извеждане на екран. Също така той може да има възможност да анализира низ от символи, като дава възможност да се инициализира по-лесно, чрез входни данни от потребителския интерфейс. Тук библиотеките на платформата, върху която разработвате, могат да влязат в употреба ако предлагат локализация със специфичните формати на валутите.

Също е много важно обекта Пари да не може да се променя. Ако например искате да промените сумата трябва да замените обекта с нов. Ако обекта променя полетата си, това може да доведе до проблеми с псевдонимите.

Подаването по стойност по дефиниция, в някой съвременни езици, води до подобряване на бързодействието. След като не се броят референциите на обектите в хийпа, натоварването върху ARC[7]/GC[8] е по-малко и от там се постига по-високата производителност.

[6] виж Стратегия (шаблон)
[7] Automatic Reference Counting
[8] Garbage Collector

Структурата Money

Основното предназначение на структурата Money е да “опакова” сумата и валутата. След това тя дефинира всички математически операции върху сумата. Добавени са специални функции за разпределяне на суми между получатели, както и някои помощни метода.

Неизменяемост

Парите са представени като обект-стойност. Важен аспект е да не се дава възможност полетата на структурата да се променят, с което да се избегнат проблеми с псевдонимите. Същевременно полетата трябва да са видими извън структурата, за да може нейните клиенти да виждат техните стойности.

Swift

```
private(set) var currency: Currency
private(set) var amount: NSDecimalNumber
```

Сравняване

Swift подхода за сравняване на обекти е да се имплементира протокола Comparable.

Swift

```
extension Money: Comparable {
    func compareTo(other: Money) -> NSComparisonResult {
        self.assertSameCurrencyAs(other)
        return self.amount.compare(other.amount)
    }
}

...

private func assertSameCurrencyAs(other: Money) {
    precondition(self.currency == other.currency,
        "Both Money Objects must be of same currency")
}
```

Освен това трябва да предефинираме операторите за сравнение:

Swift

```
func <=(lhs: Money, rhs: Money) -> Bool {
    let result = lhs.compareTo(rhs)
    return (result == .OrderedDescending || result == .OrderedSame)
}

func >=(lhs: Money, rhs: Money) -> Bool {
    let result = lhs.compareTo(rhs)
    return (result == .OrderedAscending || result == .OrderedSame)
}

func >(lhs: Money, rhs: Money) -> Bool {
    let result = lhs.compareTo(rhs)
    return (result == .OrderedAscending)
}

func <(lhs: Money, rhs: Money) -> Bool {
    let result = lhs.compareTo(rhs)
    return (result == .OrderedDescending)
}
```

Comparable включва в себе си още един протокол Equatable.

Swift

```
extension Money {
    // value objects are equal if all their fields are equal
    func equals(other: Money) -> Bool {
        return ((self.currency == other.currency) &&
            self.amount.compare(other.amount) == .OrderedSame)
    }
}

func ==(lhs: Money, rhs: Money) -> Bool {
    return lhs.equals(rhs)
}
```

Разбира се след като сме предефинирали оператора за равенство не трябва да забравяме направим същото и с хеш стойността. За целта трябва да се имплементира нов протокол - Hashable.

Swift

```
extension Money: Hashable {
    var hashValue : Int {
        get {
            return self.amount.hashValue ^ self.currency.hashValue
        }
    }
}
```

Добавяне, изваждане и умножение

За да е по-лесно да се извършват аритметични операции с всякакъв тип числови типове се налага да се предефинират многократно операторите +, - и *. Добра практика е да изнесем общата логика в отделни методи, за да спазим DRY[9] принципа:

Swift

```
func add(money: Money) -> Money {
    self.assertSameCurrencyAs(money)
    return self.add(money.amount)
}

func add(amount: NSDecimalNumber) -> Money {
    let newAmount =
        self.amount.decimalNumberByAdding(amount, withBehavior: self.roundingHandler)

    return Money(amount: newAmount, currency: self.currency)
}

func subtract(money: Money) -> Money {
    self.assertSameCurrencyAs(money)
    return self.subtract(money.amount)
}

func subtract(amount: NSDecimalNumber) -> Money {
    let newAmount =
        self.amount.decimalNumberBySubtracting(amount,
            withBehavior: self.roundingHandler)

    return Money(amount: newAmount, currency: self.currency)
}

...
```

[9] Don't repeat yourself

```swift
...

func multiply(multiplier: NSDecimalNumber) -> Money {
    let newAmount   =
        self.amount.decimalNumberByMultiplyingBy(multiplier,
            withBehavior: self.roundingHandler)

    return Money(amount: newAmount, currency: self.currency)
}
```

Методите са видими за другите класове. Те могат да се използват в случаите в които трябва да се имплементира клас с различна от подразбиращата се стратегия за конвертиране или ако предпочитате директната работа с тях вместо с предефинираните оператори.

Конвертиране между валути

Добавени са два метода за конвертиране. Първият използва подразбиращата се стратегия за конвертиране, която трябва да е полезна в болшинството от случаите. Вторият дава възможност да се зададе нова стратегия за конвертирането.

```swift
func convertTo(currency: Currency, usingExchangeRate multiplier: NSDecimalNumber) -> Money {
    return self.convertTo(currency, usingExchangeRate: multiplier,
                          strategy: NormalConvertStrategy())
}

func convertTo(currency: Currency, usingExchangeRate
  multiplier: NSDecimalNumber, strategy: ConvertCurrenciesStrategy) -> Money {
    return strategy.convertTo(self, toCurrency: currency, usingExchangeRate: multiplier)
}
```

Стратегия дизайн шаблона е реализиран с протокол.

```swift
protocol ConvertCurrenciesStrategy {
    func convertTo(money: Money, toCurrency currency: Currency,
                   usingExchangeRate multiplier: NSDecimalNumber) -> Money
}
```

Swift

```
class NormalConvertStrategy: ConvertCurrenciesStrategy {
    func convertTo(money: Money, toCurrency currency: Currency,
      usingExchangeRate multiplier: NSDecimalNumber) -> Money
    {
        precondition(money.currency != currency, "Cannot convert to the same currency")
        precondition(multiplier.isPositive(),
                     "Cannot convert using a negative conversion multiplier")

        // Uses target currency scale for rounding
        return money.multiply(multiplier, currency: currency)
    }
}
```

Разпределяне между получатели

В тези методи е логиката за разделяне на пари между сметки, която няма как да бъде реализирана точно, поради проблеми с остатъка, които вече споменахме.

Swift

```
func allocate(n: Int) -> [Money] {
    precondition(n > 0,
        "Recepients count should be presented as positive number, greater than zero.")

    let low         = self.devide(NSDecimalNumber(integer: n))
    let high        = low.add(self.oneSubunit())

    // create array for all results and init it with nil
    var results     = [Money?](count: n, repeatedValue: nil)
    let reminder    = self.amountInSubunits() % n

    for (index, _) in results.enumerate() {
        if index < reminder {
            results[index] = high
        } else {
            results[index] = low
        }
    }

    return results.map { $0! }
}
```

Swift

```
func allocate(ratios: [Int]) -> [Money] {
    let total       = ratios.reduce(0, combine: +)
    var reminder    = self.amountInSubunits()
    var results     = [Money?](count: ratios.count, repeatedValue: nil)

    precondition(total > 0,
        "Ratios total should be presented as positive number, greater than zero.")

    for (index, item) in ratios.enumerate() {
        let cents: Int  = self.amountInSubunits() * item / total
        results[index]  =
            Money(amount: self.amountFromSubunits(cents), currency: self.currency)
        reminder        -= cents
    }

    for var index = 0; index < reminder; index++ {
        results[index]!.add(self.oneSubunit())
    }

    return results.map { $0! }
}
```

Използвани са следните помощни функции:

Swift

```
// Subunits is a fraction of the base (ex. cents, stotinka, etc.)

func amountInSubunits() -> Int {
    let power = Int16(self.currency.maximumFractionDigits)
    return self.amount.decimalNumberByMultiplyingByPowerOf10(power).integerValue
}

func oneSubunit() -> NSDecimalNumber {
    let exp = Int16(-self.currency.maximumFractionDigits)
    return NSDecimalNumber(mantissa: 1, exponent: exp, isNegative: false)
}

...

private func amountFromSubunits(cents: Int) -> NSDecimalNumber {
    let exp         = Int16(-self.currency.maximumFractionDigits)
    let mantissa    = UInt64(abs(cents))
    let isNegative  = (cents < 0)
    return NSDecimalNumber(mantissa: mantissa, exponent: exp, isNegative: isNegative)
}
```

Конструктори

Подразбиращ се конструктор приемащ като параметри сумата, представена в основният числов тип, с който работим и валутата.

Swift

```
init(amount: NSDecimalNumber = NSDecimalNumber.zero(),
  currency: Currency = Money.defaultCurrency())
{
    self.currency           = currency

    // Since there is no error handling in Swift 1.0 all NSDecimalNumber exceptions
    // are suppressed instead in case of error operations will return NaN
    self.roundingHandler    = NSDecimalNumberHandler(roundingMode: .RoundBankers,
        scale: Int16(self.currency.maximumFractionDigits), raiseOnExactness: false,
        raiseOnOverflow: false, raiseOnUnderflow: false, raiseOnDivideByZero: false)

    // set amount - insure valid value is stored
    let actualAmount        =
        (amount != NSDecimalNumber.notANumber()) ? amount : NSDecimalNumber.zero()
    self.amount             =
        actualAmount.decimalNumberByRoundingAccordingToBehavior(self.roundingHandler)
}
```

Втори, помощен конструктор, който приема като параметри символен низ представящ сумата (може да е в локализиран формат или само число) и валутата.

Swift

```
init(amount: String, currency: Currency = Money.defaultCurrency()) {
    let actualAmount = Money.decimalNumberFromAmountString(amount, currency: currency)
    self.init(amount: actualAmount, currency: currency)
}

...

private static func decimalNumberFromAmountString(amount: String,
  currency: Currency) -> NSDecimalNumber
{
    // asume that sting contains formatted value
    if let amountFromString = currency.formatter.numberFromString(amount) {
        return NSDecimalNumber(decimal: amountFromString.decimalValue)
    }

    // expect that string contains number value
    return NSDecimalNumber(string: amount)
}
```

Други помощни функции

Методи за проверка на стойността.

```swift
func isZero() -> Bool {
    return self.amount.isZero()
}

func isNegative() -> Bool {
    return self.amount.isNegative()
}

func isPositive() -> Bool {
    return self.amount.isPositive()
}
```

Метод, който връща сумата като абсолютна стойност.

```swift
func absoluteAmount() -> NSDecimalNumber {
    return self.isPositive() ? self.amount : self.amount.inverted()
}
```

Клас метод, който връща най-често използваната валута, като подразбираща се.

```swift
static func defaultCurrency() -> Currency {
    return Currency.currencyForLocaleIdentifier(localeIdentifier: "en_US")!
}
```

Помощен метод за извеждане на валутите като текст в локализиран формат.[10]

```swift
var description: String {
    let formatter     = currency.formatter
    let string        = formatter.stringFromNumber(self.amount)
    return string ?? ""
}
```

[10] Swift 1.x използва протокола Printable, който е заменен в 2.0 с CustomStringConvertible

Възможни оптимизации

Една логична оптимизация е да се отдели функционалността за разпределяне на парите на части в отделен клас. Освен че ще подобри четимостта на кода, това ще позволи по-лесно предефиниране на алгоритъма, чрез просто наследяване, което за съжаление е невъзможно в момента, тъй като Парите са представени като структура.

Би било от полза ако се добавят допълнително конструктори за да е по-лесно да се създават обекти Money от основните числови типове.

Класът Currency

Класът Currency е фасада на NSNumberFormatter, който има възможност да представя валути. При инициализацията му използваме NSLocale, откъдето той взема предварително дефинираните стойности за необходимите ни свойства.

Swift

```
lazy var formatter: NSNumberFormatter = {
    var formatter = NSNumberFormatter()
    formatter.numberStyle = .CurrencyStyle
    formatter.locale = self.locale
    return formatter
}()
```

Конструктори

NSLocale се инжектира през конструктора, тъй като зависим от него.

Swift

```
// Set locale as constructor DI
init?(locale: NSLocale) {
    // always set locale to avoid compiler errors
    self.locale = locale

    // fail if we can't get currency code from locale
    if !Currency.isLocaleAssociatedWithCurrencyCode(locale) {
        return nil
    }
}

// init with current locale
convenience init?() {
    self.init(locale: NSLocale.currentLocale())
}

private let locale: NSLocale

...

// Check if Locale object is Associated With Currency Code
class func isLocaleAssociatedWithCurrencyCode(locale: NSLocale) -> Bool {
    let string  = locale.objectForKey(NSLocaleCurrencyCode) as? String
    return (string != nil && !string!.isEmpty)
}
```

Свойства

Всички свойства могат да се променят, като се правят нужните проверки за валидност.

Swift

```
static let  availableDecimalSeparators  = Set<String>([",", ",", "."])
static let  availableGroupingSeparators = Set<String>([",", "،", " ", "’", "\'", "."])

var code: String {
    get {
        return self.formatter.currencyCode
    }
    set {
        if (NSLocale.ISOCurrencyCodes() ).contains(newValue) == true {
            self.formatter.currencyCode = newValue
        }
    }
}

var symbol: String {
    get {
        return self.formatter.currencySymbol ?? ""
    }
    set {
        self.formatter.currencySymbol = newValue
    }
}

var maximumFractionDigits: Int {
    get {
        return self.formatter.maximumFractionDigits
    }
    set {
        if newValue > 0 && newValue <= 3 {
            self.formatter.maximumFractionDigits = newValue
        }
    }
}

var decimalSeparator: String {
    get {
        return self.formatter.currencyDecimalSeparator ?? ""
    }
    set {
        if Currency.availableDecimalSeparators.contains(newValue) {
            self.formatter.currencyDecimalSeparator = newValue
        }
    }
}
...
```

Swift

```
...

var groupingSeparator: String? {
    get {
        return self.formatter.currencyGroupingSeparator
    }
    set {
        if newValue == nil || Currency.availableGroupingSeparators.contains((newValue!)) {
            self.formatter.currencyDecimalSeparator = newValue
            self.formatter.usesGroupingSeparator    = (newValue != nil)
        }
    }
}
```

Методи

Класът дава възможност да се сравняват негови инстанции на базата на техните полета, което както споменахме води и до промяна на хеш стойността. Това е нужно за съответните функции в структурата Money.

Swift

```
extension Currency: Hashable {
    var hashValue : Int {
        get {
            // decimal and grouping separator are only for presentation
            // and should not be a part of equality check
            return self.locale.hashValue ^ self.code.hashValue ^
                self.symbol.hashValue ^ self.maximumFractionDigits
        }
    }
}

extension Currency {
    func equals(other: Currency) -> Bool {
        return self.locale == other.locale &&
            self.code == other.code && self.symbol == other.symbol &&
            self.maximumFractionDigits == other.maximumFractionDigits
    }
}

func ==(lhs: Currency, rhs: Currency) -> Bool {
    return lhs.equals(rhs)
}
```

В класът са включени и помощни методи, които ни помагат да създадем инстанция от идентификатор за локализация или ISO код на валутата.

Swift

```swift
class func currencyForLocaleIdentifier(localeIdentifier string: String) -> Currency? {
    // Check for valid identifier
    if (NSLocale.availableLocaleIdentifiers() ).contains(string) == false {
        return nil
    }

    let locale = NSLocale(localeIdentifier: string)
    let currency = Currency(locale: locale)
    return currency
}

class func currencyWithCurrencyCode(currencyCode string: String) -> Currency? {
    let availableLocaleIdentifiers: [String]    =
        NSLocale.availableLocaleIdentifiers()

    for localeIdentifier in availableLocaleIdentifiers {
        let locale = NSLocale(localeIdentifier: localeIdentifier)

        // check if locale currency code is equal to filter
        if let localeCurrencyCode = locale.objectForKey(NSLocaleCurrencyCode) as? String
            where localeCurrencyCode == string {
                return Currency(locale: locale)
        }
    }

    return nil
}
```

Кога да го използваме

Във всяко приложение, в което трябва да извършваме операции с пари.

Когато работим в обектно-ориентирана среда няма особена причина да не се ползва. Въздействието върху бързодействието е минимално, благодарение на приложените оптимизации. Въпреки това програмистите често избягват да използват подобни малки обекти главно защото не ги познават.

Заключение

Представянето на Парите, дори и с допълнителните помощни методи е доста прост шаблон, който представя обект-стойност във финансов контекст. Освен това капсулира логиката за пресмятанията и закръгленията, което дава възможност на клиентите (класове/структури), които го ползват да представят по-високо концептуално ниво.

Полезни препратки

1. Статия за представяне на парите от серията *“често забравяни правила”* - http://everything2.com/title/never+store+currency+in+a+float
2. Описание на световните валути: http://chartsbin.com/view/1378
3. Статия за техническия стандарт IEEE-754 за изчисления с числа с плаваща запетая - http://en.wikipedia.org/wiki/IEEE_floating_point
4. Статия относно използването на Integer тип за представяне на пари - http://floating-point-gui.de/formats/integer/
5. Описание на Decimal типа - https://en.wikipedia.org/wiki/Decimal_data_type
6. Статия относно имплементиране на функциите за равенство и хеш стойност в обект - https://www.mikeash.com/pyblog/friday-qa-2010-06-18-implementing-equality-and-hashing.html
7. Полезни примери на Java - http://www.javapractices.com/topic/TopicAction.do?Id=13
8. Статия за NSDecimalNumber - http://rypress.com/tutorials/objective-c/data-types/nsdecimalnumber
9. Статия за NSDecimalNumber - http://www.cimgf.com/2008/04/23/cocoa-tutorial-dont-be-lazy-with-nsdecimalnumber-like-me/
10. Статия за структурите в Swift - http://www.codingexplorer.com/structures-swift/
11. Описание на обект-стойност (Value Object) - https://en.wikipedia.org/wiki/Value_object
12. Описание на Шаблоните за дизайн (Design Patterns) - https://bg.wikipedia.org/wiki/Шаблони_за_дизайн
13. Доказателство за алгоритъма на Мартин Фаулър за разпределяне на парите на части - http://stackoverflow.com/questions/1679292/proof-that-fowlers-money-allocation-algorithm-is-correct
14. Стратегия (шаблон) - https://en.wikipedia.org/wiki/Strategy_pattern
15. Обект-стойност - http://martinfowler.com/bliki/ValueObject.html
16. PHP имплементация на дизайн шаблона за Пари на Мартин Фаулър - http://code.tutsplus.com/tutorials/money-pattern-the-right-way-to-represent-value-unit-pairs--net-35509
17. PHP имплементация на дизайн шаблона за Пари на Мартин Фаулър - http://verraes.net/2011/04/fowler-money-pattern-in-php/
18. Don't repeat yourself - https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

Литература

1. Patterns of Enterprise Application Architecture - Fowler, Martin