

Table of Contents

I. Introdução	1
1. Introdução	3
Objetivos do livro	3
Como usar este livro	4
2. Primeiras instruções	5
Download das ferramentas para desenvolvedores	5
Introdução ao Xcode	5
Como usar um playground	7
Variação de variáveis e exibição no console	8
Continue assim!	11
Desafio	11
3. Tipos, constantes e variáveis	13
Tipos	13
Constantes vs. variáveis	14
Interpolação de strings	15
Desafio	16
II. Princípios básicos	17
4. Condicionais	19
if/else	19
Operador ternário	21
Ifs aninhados	22
else if	23
Desafio	23
5. Números	25
Inteiros	25
Criação de instâncias de inteiros	26
Operações com inteiros	27
Divisão de inteiros	27
Atalho de operadores	28
Operadores de excesso	28
Conversão entre tipos de inteiros	29
Números de ponto flutuante	30
Conclusão	31
6. Switch	33
O que é um switch?	33
Alterne as coisas	34
Alternância avançada	36
Correspondência de tuplas e padrão	38
Conclusão	40
Desafio	40
7. Loops	43
Loops for-in	43
Nota rápida sobre inferência de tipo	47
Loops for	47
Loops while	48
Loops do-while	48
Instruções de transferência de controle, Redux	49
Conclusão	51
Desafio	51
8. Strings	53
Trabalhando com strings	53

Unicode	54
Escalares de Unicode	54
Equivalência canônica e aplicativos	56
Elementos de contagem	56
Índices e faixas	57
Conclusão	58
Desafio	58
Desafio de bronze	58
Desafio de prata	58
9. Opcionais	59
Tipos de opcionais	59
Ligação opcional	60
Opcionais desempacotadas implicitamente	61
Encadeamento de opcionais	61
Operador de união nula	63
Conclusão	63
Desafio	63
III. Coleções e funções	65
10. Arrays	67
Criação de um array	67
Acesso e modificação de arrays	68
Igualdade e identidade de array	73
Arrays imutáveis	74
Conclusão	75
Desafio	75
11. Dicionários	77
Criação de um dicionário	77
Preenchimento de um dicionário	78
Acesso e modificação de um dicionário	78
Adição de um valor	80
Remoção de um valor	80
Looping	81
Dicionários imutáveis	81
Tradução de um dicionário em array	82
Conclusão	82
Desafio	82
12. Funções	83
Uma função básica	83
Parâmetros de função	83
Nomes de parâmetros	84
Parâmetros variadic	85
Valores padrão de parâmetros	86
Parâmetros in-out	87
Retorno de uma função	87
Escopo	88
Múltiplos retornos	88
Tipos de retorno de opcionais	90
Tipos de função	91
Conclusão	91
Desafio	91
13. Fechamentos	93
Sintaxe de fechamento	93
Sintaxe de expressão de fechamento	95
Funções como tipos de retorno	97

Funções como argumentos	98
Valores de captura de fechamentos	99
Fechamentos são tipos de referência	100
Conclusão	100
Desafio	101
IV. Enumerações, estruturas e classes	103
14. Enumerações	105
Enumerações básicas	105
Enumerações de valores brutos	107
Métodos	109
Valores associados	110
assert() e precondition()	112
Conclusão	113
Desafio	114
Para os mais curiosos: erro ao lidar com o Cocoa	114
15. Structs e classes	117
Um projeto novo	117
Estruturas	120
Métodos de instância	122
Métodos mutáveis	122
Classes	123
Uma classe de monstros	123
Herança	124
Nomes de parâmetros de métodos	126
Tipos de valor vs. tipos de referência	127
Identidade vs. igualdade	129
O que devemos usar?	129
Desafio	130
Desafio de bronze	130
Desafio de prata	130
Para os mais curiosos: métodos de tipo	130
Para os mais curiosos: Currying de função	131
16. Propriedades	137
Propriedades armazenadas básicas	137
Tipos aninhados	138
Propriedades armazenadas lentas	138
Propriedades computadas	141
Getter e setter	142
Observadores de propriedades	142
Propriedades de tipos	143
Controle de acesso	145
Conclusão	146
Desafios	146
Desafio de bronze	146
Desafio de prata	147
Desafio de ouro	147
17. Inicialização	149
Sintaxe do inicializador	149
Inicialização de structs	149
Inicializadores padrão	149
Inicializadores personalizados	150
Inicialização de classes	153
Inicializadores padrão	153
Inicialização e herança de classe	153

Inicializadores exigidos	158
Anulação da inicialização	159
Inicializadores suscetíveis a falhas	160
Um inicializador de cidade suscetível a falhas	160
Inicializadores suscetíveis a falhas em classes	162
Conclusão	163
Desafios	163
Desafio de prata	163
Desafio de ouro	163
Para os mais curiosos	164
Parâmetros de inicializadores	164
V. Conceitos avançados	165
18. Protocolos	167
Formatação de uma tabela de dados	167
Protocolos	170
Conformidade de protocolo	173
Herança de protocolos	174
Composição de protocolos	174
Métodos mutáveis	175
Conclusão	176
Desafios	176
Desafio de prata	176
Desafio de ouro	176
19. Extensões	177
Extensão de um tipo existente	177
Extensão de seu próprio tipo	178
Uso de extensões para adicionar conformidade de protocolos	179
Adição de um inicializador com uma extensão	179
Tipos aninhados e extensões	180
Extensões com funções	181
Conclusão	182
Desafios	182
Desafio de bronze	182
Desafio de prata	182
20. Genéricos	183
Estrutura de dados dos genéricos	183
Funções e métodos genéricos	185
Restrições de tipo	186
Protocolos de tipos associados	187
Cláusulas where de restrições de tipo	189
Conclusão	190
Desafios	190
Desafio de bronze	190
Desafio de prata	190
Desafio de ouro	191
Para os mais curiosos: Polimorfismo paramétrico	191
21. Gerenciamento de memória e ARC	193
Alocação de memória	193
Ciclos de referências fortes	194
Ciclos de referência em fechamentos	198
Conclusão	201
Desafios	201
Desafio de bronze	201
Desafio de prata	201

22. Igualável e comparável	203
Conformidade com igualável	203
Conformidade com comparável	205
Herança do comparável	207
Conclusão	207
Desafios	207
Desafio de bronze	207
Desafio de ouro	208
Para os mais curiosos: Operadores personalizados	208
Index	211

Part I

Introdução

1

Introdução

A WWDC (World Wide Developer's Conference, Conferência Mundial para Desenvolvedores) da Apple é um importante evento anual para a comunidade de desenvolvedores, mas o ano de 2014 foi particularmente especial. Nesse ano, a Apple introduziu uma linguagem totalmente nova, denominada Swift, para o desenvolvimento de aplicativos para iOS e OS X. O objetivo deste livro é apresentar a linguagem de programação Swift para aqueles interessados no desenvolvimento de aplicativos para hardware Apple, além de servir como recurso de referência futura.

E quanto à Objective-C, a *Língua Franca* anterior da Apple para suas plataformas? Ainda é necessário saber essa linguagem? Por enquanto, acreditamos que a resposta é um irrefutável “sim”. Por exemplo, a biblioteca Cocoa da Apple, a qual você utilizará extensivamente, é escrita em Objective-C e, portanto, será mais fácil realizar a depuração se você compreender essa linguagem. Além disso, a maior parte dos materiais de aprendizagem e dos aplicativos existentes para Mac e iOS são escritos em Objective-C. Como desenvolvedor para iOS ou Mac, você provavelmente se deparará com a Objective-C, logo é sensato estar familiarizado com essa linguagem. De fato, a Apple tornou mais fácil, e às vezes preferível, misturar e combinar a Objective-C com a Swift no mesmo projeto.

Mas é necessário saber a linguagem Objective-C para aprender a Swift? Não, porém mal não faz! De fato, a linguagem Swift parece e se comporta de forma muito semelhante à Objective-C. Se você souber a Objective-C, a Swift será uma linguagem muito familiar. Se você não souber a Objective-C, alguns dos novos recursos e padrões da Swift não parecerão tão transcendentais para você. De qualquer maneira, você tem total condição de aprender essa linguagem.

Objetivos do livro

Escrevemos este livro para todos os tipos de desenvolvedores para iOS e Mac OS X, tanto para especialistas nessas plataformas como para aqueles que estiverem lidando pela primeira vez com as tecnologias da Apple. Para aqueles que estão começando a desenvolver softwares agora, destacaremos e implementaremos de forma consciente as melhores práticas para a linguagem Swift e para a programação em geral. Nossa estratégia é ensiná-lo os fundamentos de programação durante a aprendizagem da linguagem Swift. Para os desenvolvedores mais experientes, acreditamos que este livro será uma introdução útil para a nova linguagem de sua plataforma.

Também escrevemos este livro com inúmeros exemplos para que você possa usá-lo como referência no futuro, sempre que precisar. Portanto, em vez de nos concentrarmos somente em conceitos abstratos e em questões teóricas, preferimos escrever favorecendo a prática. Isso não significa que não discutiremos alguns assuntos mais complicados. Pelo contrário, nossa abordagem favorece o uso de exemplos concretos para elucidar as ideias mais difíceis e também expor as melhores práticas que tornam o código mais legível, mais fácil de manter e mais divertido de escrever.

Esperamos mostrar a você como pode ser divertido criar aplicativos para o ecossistema da Apple. Escrever um código pode ser extremamente frustrante, mas também pode ser algo ainda mais gratificante. Há algo de mágico e emocionante na solução de um problema, sem falar na alegria que você sente ao criar um aplicativo que ajuda e traz felicidade aos outros. No fim das contas, nosso desejo é ensiná-lo a escrever um bom código e se divertir no processo.

Como usar este livro

A programação pode ser algo complicado e este livro foi criado para facilitá-la. Como podemos ajudá-lo nesse aspecto? Siga estes passos:

- Leia este livro! Sério! Não basta folheá-lo durante a noite antes de dormir.
- Digite os exemplos enquanto lê o livro. A memória muscular é parte do que você aprende neste livro. Se os seus dedos souberem aonde ir e o que digitar sem você pensar demais, você estará no caminho certo para se tornar um desenvolvedor mais eficiente.
- Cometa erros! De acordo com nossa experiência, o modo mais rápido de aprender como as coisas funcionam é antes entender o que faz com que elas não funcionem. Desmonte nossos exemplos de código e conserte-os para que funcionem novamente.
- Experimente até onde sua imaginação permitir. O quanto antes você começar a resolver seus próprios problemas com a linguagem Swift, seja mexendo no código encontrado no livro ou agindo por conta própria, mais rápido você se tornará um desenvolvedor melhor.
- Faça os desafios. Como mencionamos acima, é importante começar a resolver os problemas com a linguagem Swift o quanto antes. Isso o ajudará a começar a pensar como desenvolvedor.

O melhor modo de aprimorar alguma habilidade é com a prática. É claro que algumas pessoas têm um talento especial para uma coisa ou outra, mas o talento nunca ocupará o lugar do trabalho duro. Se você quer se tornar um desenvolvedor, vamos começar! Se você acabar achando que não é muito bom nisso, e daí?! Continue insistindo, pois temos certeza de que você vai se surpreender. De qualquer modo, seus próximos passos estão adiante. Siga em frente!

2

Primeiras instruções

Vamos começar a aprender a linguagem Swift! Neste capítulo, você configurará seu ambiente e terá uma visão geral de algumas ferramentas que utilizará todos os dias como desenvolvedor para iOS e Mac. Além disso, você vai colocar a mão na massa e trabalhar com um pouco de código para conhecer melhor a linguagem Swift e o Xcode. Para isso, os seguintes itens estão envolvidos:

- Download das ferramentas para desenvolvedores
- Introdução ao Xcode
- Criação de um playground para testar códigos

Download das ferramentas para desenvolvedores

Para trabalhar com a linguagem Swift, você deve baixar e instalar o *Integrated Development Environment* (IDE, ou Ambiente de Desenvolvimento Integrado) da Apple, denominado Xcode. O Xcode tem tudo o que você precisa para criar aplicativos para iOS e Mac. Lembre-se de baixar o Xcode 6 ou superior. O Xcode 6 requer o Mac OS X 10.9.3 ou superior. Ou seja, isso significa que você precisará de um Mac para trabalhar com este livro. Se você ainda não tiver o Xcode, é possível baixar o aplicativo na App Store para Mac gratuitamente.

Introdução ao Xcode

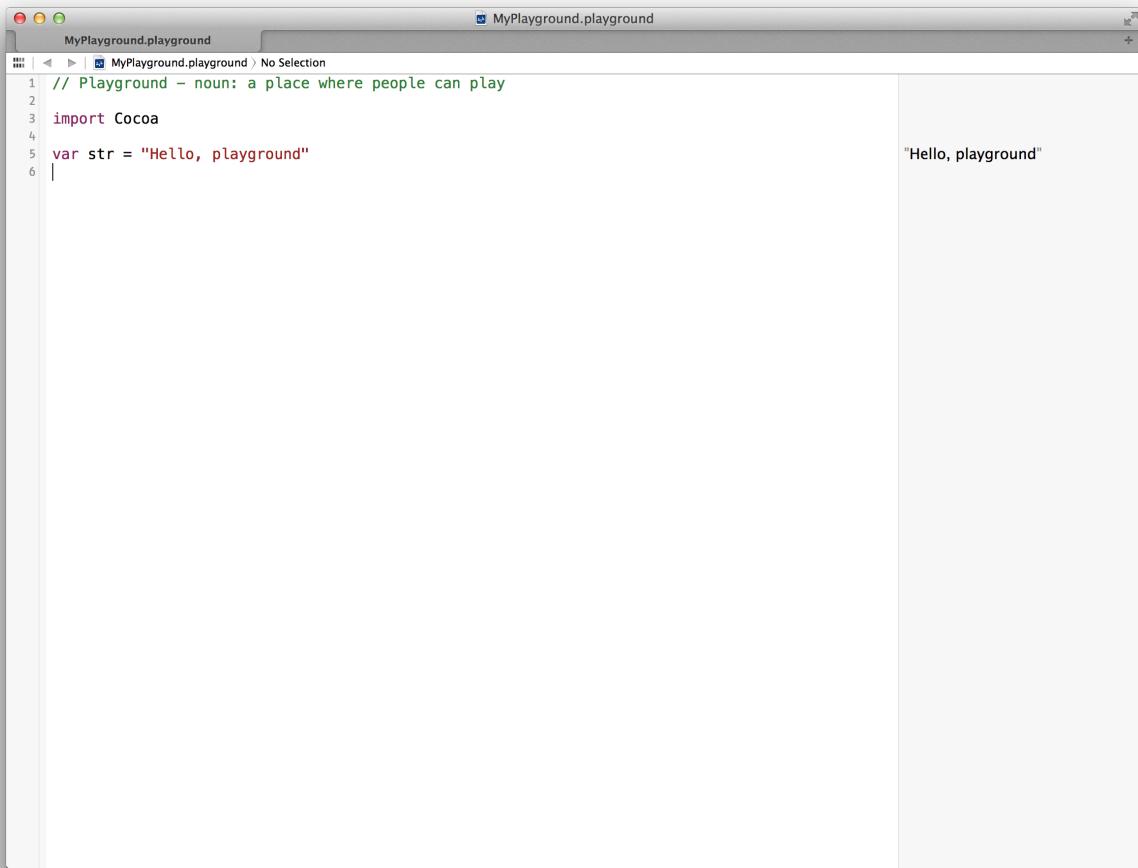
Agora que você já instalou o Xcode, abra-o e você verá uma janela que apresenta uma série de templates de projeto para ajudá-lo a começar. Cada template cria um projeto adequado para um tipo específico de programa. Por enquanto, selecione "Get started with a playground".

Figure 2.1 Seleção do template do playground



Depois de clicar no template do Playground, surgirá uma janela perguntando como você quer nomear o projeto e onde quer salvá-lo. Nomeie e salve o projeto como quiser. Para ajudá-lo a manter a organização, recomendamos criar uma pasta em um local conveniente e salvar todo o seu trabalho nela. Também será solicitado que você escolha uma plataforma (OS X ou iOS). Selecione OS X, mesmo se você for um desenvolvedor para iOS. Os recursos da linguagem Swift que serão abrangidos são comuns a ambas as plataformas.

Figure 2.2 Seu novo playground



Os playgrounds são um novo recurso lançado no Xcode 6 que oferecem um ambiente interativo para desenvolver e avaliar de forma rápida o código em Swift. Um playground não exige que você compile e execute um projeto completo. Além disso, os playgrounds avaliam seu código em Swift em tempo real. Portanto, eles são ideais para testar e experimentar a linguagem Swift em um ambiente leve. Você usará os playgrounds com frequência no decorrer deste livro quando quiser um feedback rápido sobre seu código em Swift.

Você utilizará tanto os playgrounds como as ferramentas de linha de comando nativas ao se familiarizar com a linguagem Swift. Por que não usar somente os playgrounds? Você deixaria de aproveitar muitos dos recursos do Xcode e não seria exposto ao IDE com a mesma intensidade. Confie em nós quando dizemos que você passará boa parte do tempo no Xcode e, por isso, é uma boa ideia se familiarizar com ele o quanto antes.

Como usar um playground

Como é possível ver na Figura 2.2, os playgrounds em Swift são divididos basicamente em duas seções. À esquerda, temos o editor de código em Swift e, à direita, a barra lateral de resultados. O código no editor é avaliado e executado, se possível, sempre que a fonte for alterada. Os resultados do código são exibidos à direita na barra lateral de resultados.

Logo na primeira linha, você provavelmente notará que o texto está na cor verde e é antecedido por duas barras comuns: //. As linhas antecedidas por duas barras comuns indicam ao compilador que elas não passam de meros

comentários de esclarecimento do desenvolvedor. Normalmente, os comentários são usados como documentação em linha ou notas do desenvolvedor. Se você remover essas barras, verá que a cor do texto mudará e o compilador emitirá avisos relacionados à incapacidade de analisar sintaticamente a expressão. Além disso, você pode adicionar e remover comentários de determinada seleção pressionando "command-barra comum" em seu teclado. Vá em frente, destaque o código em seu playground. Você deve notar que o editor alterna entre comentários ativados e desativados quando você usa a combinação de teclas acima.

Também é possível notar que o playground importa o framework Cocoa bem no topo do editor de código. Essa importação significa que nosso playground possui total acesso a todas as Interfaces de Programação de Aplicativos (APIs) disponibilizadas para os desenvolvedores para iOS e Mac pelo framework Cocoa. Caso você não esteja familiarizado com o termo, uma API é similar a uma prescrição, ou conjunto de definições, de como um programa pode ser escrito.

No editor de código abaixo da importação, há uma linha que diz: `var str = "Hello, playground"`. A barra lateral de resultados à direita deve exibir o seguinte texto: "Hello, playground". Vamos analisar essa linha de código para entender melhor o que está acontecendo. À esquerda do sinal de igual, haverá o texto `var str`. `var` é uma palavra-chave em Swift que significa que você está declarando uma variável, que é um conceito importante que você verá com mais detalhes no próximo capítulo. Por enquanto, digamos que uma variável representa algum valor que você espera que mude ou varie. À direita do sinal de igual, haverá o seguinte texto: "Hello, playground".

Na linguagem Swift, o texto entre aspas significa que você está criando uma `String`. As `Strings` são usadas para representar uma coleção ordenada de caracteres. O template nomeou essa nova variável de `str`, mas você poderia ter nomeado a variável praticamente com o que você quisesse (Há limitações quanto a isso, é claro. Tente mudar o nome `str` para `var`. O que acontece? Por que você acha que não consegue nomear sua variável com `var`?). Portanto, agora você está em condições de compreender o texto exibido à direita, na barra lateral de resultados; ele é o valor da `string` que você acabou de atribuir à variável `str`.

Variação de variáveis e exibição no console

Na seção acima, você criou uma variável chamada de `str` e atribuiu a ela um texto representando uma instância do tipo `String`. Os tipos descrevem uma estrutura específica para representar os dados e definem o que o tipo pode e não pode fazer com esses dados. Por exemplo, o tipo `String` é projetado para trabalhar com uma coleção ordenada de caracteres e define uma série de funções para trabalhar com essa coleção ordenada de caracteres.

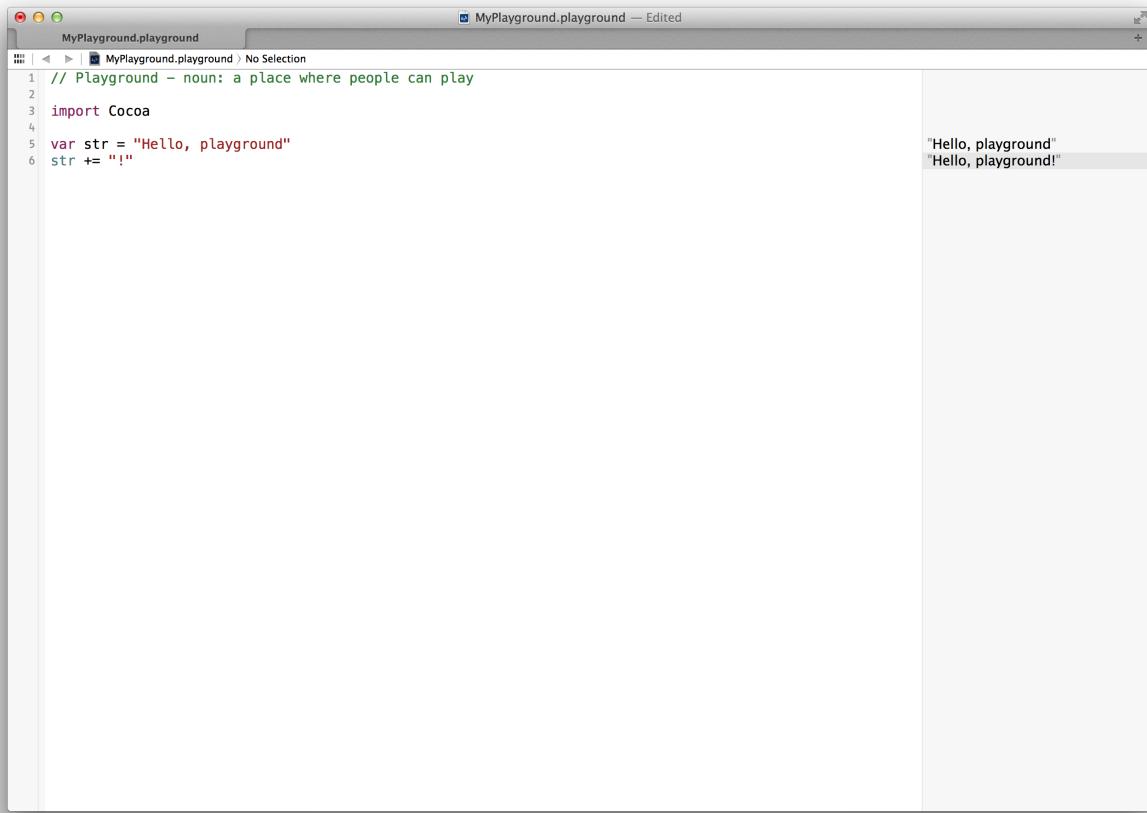
Para deixar isso mais claro, lembre-se de que você criou o `str` acima como uma *variável*. Isso significa que você pode alterar a variável; ou seja, você pode variar o valor de `str`. Adicione pontuação ao final da string já criada para torná-la mais parecida com uma frase normal. Observe que, sempre que nos referirmos a uma listagem de código anterior e adicionarmos um código novo, listaremos o código anterior com o código novo destacado em negrito.

```
import Cocoa  
  
var str = "Hello, playground"  
  
str += "!"
```

Ótimo! Você acabou de adicionar um ponto de exclamação em nossa instância de `string`. Para isso, você usou o *operador de atribuição de composto* `+=` para adicionar e atribuir. O *operador de atribuição de adição* permite que combinemos as operações de adição e atribuição em um único operador. (Esses operadores serão tratados com mais detalhes no capítulo 4.) Logo, o resultado foi a adição de um ponto de exclamação, que é uma instância de `string`, ao final da variável `str`.

Você viu alguma coisa na barra lateral de resultados à direita? Se você tiver digitado tudo da maneira apresentada acima, você deve ver uma nova linha de resultados representando o novo valor de `str`. Esse novo valor de `str` terá um ponto de exclamação adicionado ao final de sua string. Consulte a figura 2.3 para garantir que seu editor esteja igual ao nosso.

Figure 2.3 Variação de str

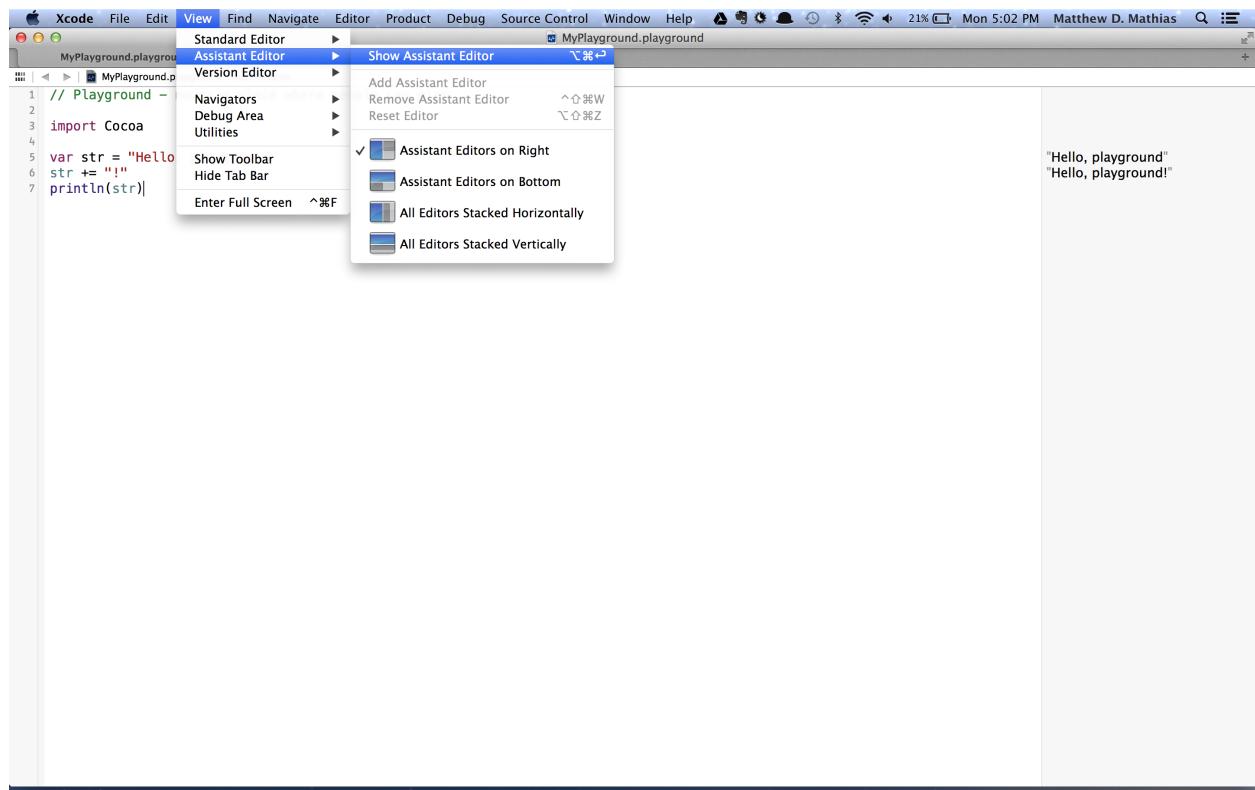


Em seguida, adicione algum código para exibir no console o valor contido na variável `str`. Para isso, você precisará utilizar uma função chamada de `println()`, que significa “exibir linha”. As funções são agrupamentos de código relacionado que enviam instruções ao computador para completar uma tarefa específica. `println()` é uma função usada para exibir um valor no console seguido de uma quebra de linha.

```
import Cocoa
var str = "Hello, playground"
str += "!"
println(str)
```

Normalmente, você entra no console para verificar o valor atual de alguma variável de interesse. Você usará a função `println()` com frequência depois que o seu fluxo de trabalho de desenvolvimento diário mudar de playgrounds em Swift para aplicativos completos em Xcode. No momento, a barra lateral de resultados do playground está cobrindo o console. Você precisa abrir o *editor assistente* para vê-lo. Conforme exibido na figura 2.4, clique em "View", "Assistant Editor" e, em seguida, em "Show Assistant Editor". Está vendo o atalho de teclado ao lado dessa última etapa? Como opção, você pode pressionar as teclas "command-option-enter" ao mesmo tempo em seu teclado para exibir o *editor assistente*.

Figure 2.4 Exibição do editor assistente



Agora que o *editor assistente* está visível, você deve ver algo semelhante à figura 2.5. Em particular, observe a nova caixa branca no *editor assistente* chamada de "Console Output". O texto “Hello, playground!” deve estar visível no console.

Figure 2.5 Seu primeiro código em Swift

The screenshot shows the Xcode interface with a playground window titled "MyPlayground.playground". The left pane contains the following Swift code:

```

1 // Playground - noun: a place where
2 // people can play
3 import Cocoa
4
5 var str = "Hello, playground"
6 str += "!"
7 println(str)

```

The right pane shows the "Console Output" with the result:

```

Hello, playground!
Hello, playground!

```

Continue assim!

Vamos rever rapidamente o que você realizou até agora. Você:

- Instalou o Xcode
- Criou e se familiarizou com um playground
- Usou uma variável e a modificou
- Aprendeu sobre o tipo `String`
- Usou uma função para exibir no console

Esse é um progresso e tanto e nós nem mencionamos todos os bugs, erros de digitação e problemas de instalação que você pode ter encontrado! Logo, logo você criará seus próprios aplicativos, mas, antes disso, lembre-se de ter perseverança. Aprender algo novo é trabalhoso e frustrante, mas também é extremamente gratificante. Você chegou até aqui e, conforme for prosseguindo, perceberá que quase tudo que veremos em seguida é uma mera variação dos temas que vimos até aqui.

Desafio

Você verá que muitos dos capítulos deste livro terminam com uma seção que apresenta alguns desafios. O objetivo dessas seções é fazer com que você trabalhe por conta própria para que possa aprofundar seu entendimento da

linguagem Swift e também ganhar um pouco mais de experiência. Este capítulo não será diferente! Vamos começar. Mas, antes, você provavelmente precisará criar um novo playground para trabalhar. Vá em frente e faça isso.

Além de diversos outros detalhes interessantes, você aprendeu sobre o tipo `String` e como exibir no console usando `println()`. Use seu novo playground para criar uma nova instância do tipo `String`. Ajuste o valor dessa instância para ser igual ao seu sobrenome. Exiba o valor no console

3

Tipos, constantes e variáveis

Este capítulo o apresentará a constantes, variáveis e tipos de dados básicos da linguagem Swift. Esses elementos são os blocos de construção fundamentais de qualquer programa. Você usará constantes e variáveis para armazenar valores e transferir dados entre seus aplicativos. Os tipos descrevem a natureza dos dados mantidos pela constante ou variável. Existem diferenças importantes entre as constantes e as variáveis, além de cada um dos tipos de dados, que definem seus respectivos casos de uso.

Tipos

As variáveis e as constantes possuem um tipo de dados. Os tipos descrevem a natureza dos dados e fornecem informações para o compilador sobre como lidar com os dados armazenados pela variável ou constante. Com base no tipo da constante ou variável, o compilador será capaz de saber a quantidade de memória que deve ser reservada e também poderá ajudar na *verificação do tipo*, um recurso da linguagem Swift que ajuda a impedir que você atribua o tipo de dados errado a uma variável, por exemplo.

Vamos imaginar que você esteja modelando uma pequena cidade em seu código. Pode ser que haja uma variável para o número de semáforos da cidade. Suponhamos que você escreva o seguinte código para atribuir um valor a essa variável. Vá em frente e crie um novo playground de OS X e digite o código abaixo.

```
import Cocoa  
var number0fStopLights = "Four"
```

A primeira coisa que podemos notar é que você atribuiu uma instância do tipo `String` à variável chamada de `number0fStopLights`. A linguagem Swift usa *inferência de tipo* para determinar o tipo de dados de sua variável. Isso significa que o compilador sabe que a variável `number0fStopLights` é do tipo `String` porque o valor à direita do sinal de igual, usado para atribuir um valor para o que está à esquerda do sinal, é uma instância de `String`. Nesse caso, o compilador sabe que "Four" é do tipo `String` porque ele é uma literal de `String`, conforme indicado pelas aspas.

Se você tentar adicionar 2 a essa variável, o compilador emitirá um erro indicando que essa operação não faz sentido. Você recebe esse erro porque está tentando adicionar um número a uma variável que é uma instância do tipo `String`. O que significa adicionar um número a uma string? Essa adição dobra a string e resulta em "FourFour"? Coloca "2" no final e resulta em "Four2"? Ninguém sabe - é por isso que não faz sentido adicionar um número a uma instância de `String`.

```
import Cocoa  
  
var number0fStopLights = "Four"  
number0fStopLights + 2
```

Na verdade, não faz muito sentido `number0fStopLights` ser do tipo `String` em primeiro lugar. Em vez disso, como essa variável representa explicitamente o número de semáforos em nossa cidade hipotética, use um tipo numérico. A

linguagem Swift oferece um tipo `Int` perfeito para sua variável, que será um inteiro pequeno. Altere seu código para corresponder ao seguinte:

```
import Cocoa  
var numberOfStopLights = "Four"  
var numberOfStopLights: Int = 4  
numberOfStopLights + 2
```

Há alguns itens a serem observados aqui. Primeiramente, observe que você está excluindo sua atribuição original de `"Four"` para a variável `numberOfStopLights`. Neste livro, seguiremos a convenção de colocar em negrito e tachar o código antigo que deve ser removido.

Além disso, observe como o código é diferente da nossa primeira tentativa de atribuição de um valor a `numberOfStopLights`. Enquanto antes você contava com a capacidade da linguagem Swift de inferir um tipo de variável com base em seu valor, você agora está declarando explicitamente que a variável `numberOfStopLights` será do tipo `Int` usando a sintaxe de *anotação de tipo* da linguagem Swift. Os dois pontos no código acima representam `"...do tipo..."`. Sendo assim, o código pode ser lido mais ou menos desta forma: “declarar uma variável chamada de `numberOfStopLights` do tipo `Int` que inicia com o valor de 4.” Se você reatribuísse posteriormente sua instância de `String` anterior de `"Four"`, o compilador emitiria um aviso indicando que ele não pode converter uma string em um inteiro.

Finalmente, observe também que seu erro desapareceu. Não há nenhum problema em adicionar 2 a sua variável de inteiro que representa o número de semáforos de sua cidade. Na verdade, como você declarou que essa instância é uma variável, essa operação é perfeitamente natural. Voltaremos a essa questão posteriormente neste capítulo.

A linguagem Swift tem uma série de outros tipos de dados usados com frequência, incluindo números (tema do Chapter 5) e strings, que contêm dados textuais (tema do Chapter 8). Existem outros tipos usados normalmente, como os *tipos de coleção*, os quais você verá em breve.

Constantes vs. variáveis

O título deste capítulo inclui os termos “constantes” e “variáveis”. O que *são* eles exatamente? Até agora, você viu apenas variáveis. Os valores das variáveis podem variar, o que significa que você pode atribuir um novo valor a uma variável. No código apresentado, contudo, os valores nunca variaram e tudo funcionaria perfeitamente se fossem declarados como constantes. Você não pode alterar o valor de uma constante.

Na verdade, criar uma constante de `numberOfStopLights` seria sem dúvida melhor considerando que você não solicitou que o valor variasse. Como regra geral: use variáveis para instâncias que variam e constantes para instâncias invariáveis. Depois de definir um valor para uma constante, não é possível alterá-lo. Como você pode imaginar, essa limitação não se aplica a uma variável; você pode alterar o valor de uma variável o quanto quiser.

A linguagem Swift tem uma sintaxe diferente para declarar tanto constantes como variáveis. Use a palavra-chave `let` para declarar que uma instância é uma constante e a palavra-chave `var` para declarar uma variável. Adicione algumas constantes e variáveis ao playground atual para expandir a pequena cidade.

```
import Cocoa  
var numberOfStopLights: Int = 4  
numberOfStopLights + 2  
let numberOfStopLights: Int = 4
```

Observe que sua variável `numberOfStopLights` agora está declarada como constante por meio da palavra-chave `let`. Essa alteração faz algum sentido considerando que a cidade sendo modelada é pequena e não deve receber nenhum semáforo novo no futuro próximo. Já que estamos falando do tamanho da cidade, adicione um `Int` para representar a população.

```
import Cocoa

let numberOfWorklights: Int = 4
var population: Int
```

A população da sua cidade, embora não deva ser grande, provavelmente variará um pouco para cima ou para baixo. Sendo assim, você declarou a `population` com a palavra-chave `var` para transformar essa instância em variável. Observe que você declarou que a `population` é uma instância do tipo `Int`. Você fez isso também porque a população de uma cidade é contada por pessoa inteira. Também é importante salientar que a `population` não é inicializada com nenhum valor e, portanto, é um `Int` vazio. Você também pode usar o sinal de igual, conhecido também como *operador de atribuição* para atribuir à `population` seu valor inicial.

```
import Cocoa

let numberOfWorklights: Int = 4
var population: Int
population = 5422
```

Interpolação de strings

Toda cidade precisa de um nome. Seria bom ter uma breve descrição da cidade para ser usada pelo Conselho de Turismo. Sua cidade está bastante estável e, portanto, não mudará o nome tão cedo. Transforme o nome da cidade em uma constante do tipo `String`.

A descrição será uma constante do tipo `String`, embora você vá criá-la de forma um pouco diferente das constantes e variáveis já criadas. A descrição incluirá todos os dados que você viu até aqui e você vai criá-la usando um recurso da linguagem Swift chamado de *interpolação de strings* que permite colocar valores de constante e de variável diretamente em uma nova string. Após criar a descrição, você vai exibi-la no console:

```
import Cocoa

let numberOfWorklights: Int = 4
var population: Int
population = 5422
let townName: String = "Knowhere"
let townDescription = "\((townName) has a population of \(population)
and \(numberOfWorklights) stop lights."
println(townDescription)
```

A interpolação de strings é um método que você pode usar para criar um novo valor de `String` combinando variáveis e constantes. Você pode, então, atribuir essa string a uma nova variável ou constante, ou apenas exibi-la no console. A sintaxe `\()` representa um placeholder na literal da `String` que acessa um valor da instância e o coloca dentro de uma nova `String`. Por exemplo, `\(townName)` acessará o valor da constante `townName` e o colocará dentro da nova instância de `String`.

O resultado do novo código incluído acima é exibido na imagem abaixo. Como podemos ver, o texto “`Knowhere` has a population of 5422 and 4 stop lights.” é exibido no console no editor assistente à direita.

Figure 3.1 Breve descrição de Knowhere

The screenshot shows an Xcode playground window titled "VariablesAndConstants.playground". The left pane contains Swift code:

```
1 // Playground - noun: a place where people can play
2
3 import Cocoa
4
5 //var number0fStopLights = "Four"
6 //var number0fStopLights: Int = 4
7 //number0fStopLights + 2
8 let number0fStopLights: Int = 4
9 var population: Int
10 population = 5422
11 let townName: String = "Knowhere"
12 let townDescription = "\((townName) has a population of \(population)
13 and \(number0fStopLights) stop lights."
14 println(townDescription)
```

The right pane shows the "Console Output" with the following text:

```
4
5,422
'Knowhere'
'Knowhere has a population o
- 30 SEC +
```

Desafio

Adicione uma nova variável a seu playground para representar a taxa de desemprego de Knowhere. Qual tipo de dados você deve usar? Atribua um valor a essa variável e atualize sua instrução de `println()` para registrar essa nova informação no console.

Part II

Princípios básicos

4

Condicionais

Nos capítulos anteriores, seu código progrediu de forma relativamente simples: você declarou algumas constantes e variáveis relativamente simples e, então, atribuiu valores a elas. Mas, obviamente, um aplicativo só ganha vida de verdade, e a programação se torna um pouco mais desafiadora, quando ele passa a tomar decisões com base no conteúdo de suas variáveis. Por exemplo, um jogo pode permitir que o jogador pule de um arranha-céu se ele tiver coletado determinado poder. Para tomar esse tipo de decisão, precisamos usar instruções de condicionais.

if/else

Usamos as instruções de `if/else` para executar o código de maneira condicional com base em uma condição lógica específica. O que isso significa? Você tem que tomar uma decisão relativamente simples entre duas alternativas e, dependendo dessa decisão, uma ramificação de código ou outra é executada (mas não ambas). Sua pequena cidade do capítulo anterior possui uma agência dos correios ou não possui. Se sua pequena cidade possuir uma agência dos correios, você poderá comprar selos. Se sua pequena cidade não possuir uma agência dos correios, você precisará viajar até a cidade vizinha para comprar selos. A existência da agência dos correios é sua condição lógica específica. Os comportamentos diferentes são "comprar selos na cidade" ou "comprar selos em outra cidade".

Algumas situações são mais complexas do que uma decisão do tipo sim/não. Você verá um mecanismo mais flexível chamado de `switch` no Chapter 6. Mas, por enquanto, não vamos complicar as coisas.

Vá em frente e crie um novo playground de OS X. Nomeie-o Condicionais e salve onde quiser. Quando o playground estiver pronto, deixe-o igual ao código abaixo. O código exibe a sintaxe básica para uma instrução de `if/else`:

```
import Cocoa

var population: Int = 5422
var message: String

if population < 10000 {
    message = "\(population) is a small town!"
} else {
    message = "\(population) is pretty big!"
}

println(message)
```

Primeiro você declarou `population` como uma instância do `Int` e, então, atribuiu a ele um valor de 5.422. Em seguida, você declarou uma variável chamada `message` que é da `String`. Dessa vez, no entanto, deixe essa variável não inicializada, o que significa que ainda não há nenhum valor associado a `message`. Em vez disso, escolha atribuir um valor a `message` depois de ter avaliado a sua instrução de `if/else` condicional. Observe que você está usando a *interpolação de strings* para colocar a população na string.

A figura 4.1 mostra como o seu playground deve ficar. Observe que o console e a barra lateral de resultados indicam que `message` foi definida para ser igual à literal da string quando a condicional avaliar como verdadeira. Como isso aconteceu?

Figure 4.1 Descrição condicional da população da cidade

The screenshot shows the Xcode playground interface. On the left, the code editor displays the following Swift code:

```

1 // Playground - noun: a place where people can
2 play
3
4 import Cocoa
5
6 var population: Int = 5422
7 var message: String
8
9 if population < 10000 {
10     message = "\(population) is a small town!"
11 } else {
12     message = "\(population) is pretty big!"
13 }
14
15 println(message)

```

The right side shows the console output window titled "Console Output" which contains the text "5422 is a small town!". Below the output window is a timeline bar with a slider set to "30 SEC".

A condição na instrução de `if/else` testa se a população da sua cidade é ou não inferior a 10.000 por meio do operador de comparação `<`. Se a condição avaliar como sendo verdadeira, você deve definir `message` para ser igual à primeira literal da string ("X é uma cidade pequena!"). Se a condição avaliar como sendo falsa, a população é igual ou superior a 10.000, logo `message` será igual à segunda literal da string ("X é bem grande!"). No nosso caso, a população da cidade é de fato inferior a 10.000 e a `message` foi definida como: "5422 é uma cidade pequena!". A Table 4.1 a seguir lista os operadores de comparação da linguagem Swift.

Table 4.1 Operadores de comparação

Operador	Descrição
<code><</code>	Avalia se o número à esquerda é inferior ao número à direita.
<code><=</code>	Avalia se o número à esquerda é inferior ou igual ao número à direita.
<code>></code>	Avalia se o número à esquerda é superior ao número à direita.
<code>>=</code>	Avalia se o número à esquerda é superior ou igual ao número à direita.
<code>==</code>	Avalia se o número à esquerda é igual ao número à direita.
<code>!=</code>	Avalia se o número à esquerda não é igual ao número à direita.
<code>===</code>	Avalia se as duas instâncias apontam para a mesma referência.
<code>!==</code>	Avalia se as duas instâncias não apontam para a mesma referência.

Comentário à parte: em algum momento você precisará saber a diferença entre `==` e `===` da tabela acima. Por ora, provavelmente não faz muito sentido, mas depois que você vir structs e classes no Chapter 15, isso deve ficar mais claro. Marque esta descrição para ver depois:

`==` retorna um Boolean que indica se duas instâncias são iguais ou não. Em outras palavras, `==` verifica se há igualdade ou *equivalência*. `===` retorna um Boolean que indica se os dois objetos apontam ou não para o mesmo

local na memória, se os dois objetos são *idênticos*. Se ambos residirem no mesmo local na memória, então `==` retornará `true`. De modo similar, `!=` verifica se duas referências *não* apontam para o mesmo local na memória. Se não apontarem, esse operador retornará `true`.

Esses dois últimos operadores verificam a *identidade* e, portanto, são chamados de *operadores de identidade*. Igualdade e identidade são conceitos distintos. Um objeto pode ser igual a outro objeto quanto ao valor, mas pode não apontar exatamente para o mesmo local na memória. Sendo assim, os desenvolvedores têm definições distintas para *igualdade* e *identidade*. As instâncias podem ser iguais (possuir o mesmo valor), mas não idênticas (ocupar o mesmo local na memória). É importante separar essas duas definições na sua cabeça.

Às vezes, você só está preocupado com um aspecto da condição sendo avaliada. Ou seja, você só quer executar o código se determinada condição for atendida. Por exemplo, considere o código abaixo.

```
import Cocoa

var population: Int = 5422
var message: String
var hasPostOffice: Bool = true

if population < 10000 {
    message = "\u2022(population) is a small town!"
} else {
    message = "\u2022(population) is pretty big!"
}

println(message)

if !hasPostOffice {
    println("Where do we buy stamps?")
}
```

O código acima possui um novo tipo de operação chamado de *operador lógico*. O `!` testa se `hasPostOffice` é falso ou não. Você pode pensar no `!` como inversor de um valor Boolean. Verdadeiro se torna falso e falso se torna verdadeiro. Se `hasPostOffice` for falso, você não saberá onde comprar selos e precisará perguntar. Esse operador é conhecido como "não lógico". Como sua cidade possui uma agência dos correios (o `hasPostOffice` foi inicializado como verdadeiro), você sabe onde comprar selos e não precisa perguntar nada. Assim, nossa condição (verdadeira, mas invertida para falsa), no fim das contas, avalia como falso e a função `println()` nunca é chamada. Consulte a tabela Table 4.2 abaixo para obter uma listagem e definição dos operadores lógicos da linguagem Swift.

Table 4.2 Operadores lógicos

Operador	Descrição
<code>&&</code>	E lógico; verdade se e somente se ambos forem verdadeiros (caso contrário, falso)
<code> </code>	OU lógico; verdadeiro se qualquer um for verdadeiro (falso somente se ambos forem falsos)
<code>!</code>	NÃO lógico; verdadeiro vira falso, falso vira verdadeiro

Operador ternário

O *operador ternário* é muito similar às instruções de `if/else`, mas é muito mais conciso quanto à sintaxe. A sintaxe é semelhante a: `a ? b : c`. Em português, o operador ternário pode ser lido como: "se a for verdadeiro, então faça b. Caso contrário, faça c." A verificação da população da cidade que usou `if/else` poderia ser reescrita usando o operador ternário.

```
if population < 10000 {
    message = "\u2022(population) is a small town!"
} else {
    message = "\u2022(population) is pretty big!"
}

message = population < 10000 ? "\u2022(population) is a small town!" :
                                "\u2022(population) is pretty big!"
```

O operador ternário pode ser uma fonte de controvérsia. Alguns programadores o amam. Alguns programadores o abominam. Nós estamos no meio entre os dois. Esse uso particular não é muito elegante. Sua atribuição a `message` requer que você digite mais do que um simples `a ? b : c`. Essa sintaxe é boa para instruções concisas, mas se sua instrução de operador ternário começar a usar a próxima linha, é preferível continuar usando `if/else`.

Use o desfazer para remover o operador ternário e restaurar sua instrução de `if/else`:

```
if population < 10000 {  
    message = "\u2028(population) is a small town!"  
} else {  
    message = "\u2028(population) is pretty big!"  
}  
  
message = population < 10000 ? "\u2028(population) is a small town!" +  
"\u2028(population) is pretty big!"
```

Ifs aninhados

Você pode aninhar suas instruções de `if` para executar o código de forma condicional após uma condição anterior avaliada como `true`. Para isso, escreva uma instrução nova de `if/else` dentro das chaves de outra instrução de `if/else`. No código abaixo, aninhe uma instrução de `if/else` dentro do bloco `else`:

```
import Cocoa  
  
var population: Int = 5422  
var message: String  
var hasPostOffice: Bool = true  
  
if population < 10000 {  
    message = "\u2028(population) is a small town!"  
} else {  
    message = "\u2028(population) is pretty big!"  
    if population >= 10000 && population < 50000 {  
        message = "\u2028(population) is a medium town!"  
    } else {  
        message = "\u2028(population) is pretty big!"  
    }  
}  
  
println(message)  
  
if !hasPostOffice {  
    println("Where do we buy stamps?")  
}
```

Sua cláusula de nested `if` utiliza o comparador `>=` e o operador lógico `&&` para verificar se a `population` está dentro da faixa entre 10.000 e 50.000. Você está usando essa faixa para definir uma cidade de médio porte de forma arbitrária. Como a `population` da sua cidade não está dentro dessa faixa, nossa `message` é definida como “5422 é uma cidade pequena!”, como antes. Tente aumentar a população para exercitar as outras ramificações.

As instruções de `if/else` aninhado são comuns na programação; você vai encontrá-las por aí e também vai escrevê-las. Você pode aninhar essas instruções até onde quiser, pois não há limites. Contudo, ao aninhá-las de maneira muito profunda, você corre o risco de tornar o código mais difícil de ler, forçando o leitor de seu código a manter em mente grande parte do programa. Um ou dois níveis funcionam bem, mas além disso o seu código fica cada vez menos legível e com manutenção cada vez mais difícil. Existem algumas formas de expressar o código que não exigem instruções aninhadas. Vamos refatorar o código que acabamos de escrever para deixá-lo mais fácil de acompanhar.

Na verdade, antes de refatorar o código, pode ser que você queira saber o que essa palavra significa. *Refatoração* é um termo que significa alterar o código para que ele faça o mesmo trabalho que antes, mas de maneira diferente. Talvez seja mais eficiente, ou talvez apenas tenha uma aparência melhor ou seja mais fácil de ler. Quando você refatora algo, você não está mudando o comportamento, apenas a aparência.

else if

A condicional `else if` permite que você encadeie múltiplas instruções condicionais juntas. Para facilitar um pouco a leitura do seu código, extraia a instrução de `if/else` aninhada para transformá-la em uma cláusula independente que avalie se sua cidade é de porte médio ou não. `else if` permite que você compare múltiplos casos e executa o código de forma condicional dependendo da cláusula avaliada como verdadeira. Você pode ter quantas cláusulas de `if/else/if` quiser, mas apenas uma será executada.

```
import Cocoa

var population: Int = 5422
var message: String
var hasPostOffice: Bool = true

if population < 10000 {
    message = "\u2022(population) is a small town!"
} else if population >= 10000 && population < 50000 {
    message = "\u2022(population) is a medium town!"
} else {
    if population >= 10000 && population < 50000 {
        message = "\u2022(population) is a medium town!"
    else {
        message = "\u2022(population) is pretty big!"
    }
    message = "\u2022(population) is pretty big!"
}

println(message)

if !hasPostOffice {
    println("Where do we buy stamps?")
}
```

O código acima adiciona uma cláusula `else if`, mas você poderia ter encadeado muitas outras. Esse bloco de código é uma melhoria em comparação com o `if/else` aninhado descrito acima. Se você acabar com um monte de instruções de `if/else`, talvez seja melhor usar outro mecanismo, como o `switch` descrito no Chapter 6. Fique ligado.

Desafio

Adicione outra instrução de `else if` ao código de dimensionamento da cidade para verificar se a `population` da sua cidade é muito grande. Sinta-se livre para escolher seus próprios pontos de corte da população. Defina a variável `message` de forma correspondente.

5

Números

No nível mais básico, os computadores trabalham fundamentalmente com números. Os números também são o elemento principal no desenvolvimento de softwares; você pode usá-los para monitorar a temperatura, determinar quantas letras há em uma frase e até mesmo quantos zumbis estão infestando uma cidade. Existem dois tipos básicos de números: números inteiros e números de ponto flutuante.

Inteiros

Um inteiro é um número que não possui casa decimal ou componente fracionário - um número inteiro. Inteiros são usados frequentemente para representar uma contagem de "coisas", como o número de páginas em um livro. Uma diferença entre os inteiros usados por computadores e os números que você conhece é que um tipo de inteiro em um computador ocupa uma quantidade fixa de memória. Portanto, eles não podem representar todos os números inteiros possíveis - possuem um valor mínimo e máximo que podem representar. Para ver como isso funciona, crie um novo playground e nomeie-o `Numbers.playground` e insira o código abaixo:

```
println("The maximum Int value is \(Int.max).")
println("The minimum Int value is \(Int.min).")
```

Abra a visão da linha do tempo do seu playground. Você deve ver o seguinte resultado:

```
The maximum Int value is 9223372036854775807.
The minimum Int value is -9223372036854775808.
```

Qual é o raciocínio por trás desses valores? Lembre-se de que tipos de inteiro usam uma quantidade fixa de memória. A memória utilizada por um inteiro normalmente é descrita como o número de bits que ela requer; um bit é um 0 ou 1 simples. No OS X, `Int` é um inteiro de 64 bits, o que significa que ele tem 2^{64} valores possíveis. Além disso, `Int` é um inteiro *com sinal*, o que significa que pode representar números inteiros tanto positivos como negativos. O valor de `Int` mínimo é igual a $-2^{63} = -9.223.372.036.854.775.808$ e o valor de `Int` máximo é igual a $2^{63} - 1 = 9.223.372.036.854.775.807$ (o "- 1" oferece uma representação para número 0).

No iOS, `Int` é um pouco mais complicado. A Apple introduziu dispositivos de 64 bits com os aparelhos iPhone 5S, iPad Air e iPad mini com tela Retina. Se você escrever um aplicativo para iOS para esses dispositivos, o que é chamado de direcionamento à arquitetura de 64 bits, `Int` será um inteiro de 64 bits como no OS X. Por outro lado, se você direcionar à arquitetura de 32 bits, como iPhone 5 ou iPad 2, `Int` será, em vez disso, um inteiro de 32 bits. O compilador determina o tamanho adequado para `Int` quando ele compila seu programa. Se for necessário saber o tamanho exato de um inteiro, você pode usar um dos tipos de inteiros explicitamente dimensionados da linguagem Swift. Use `Int32` para ver os valores mínimo e máximo para um inteiro de 32 bits:

```
println("The maximum Int value is \(Int.max).")
println("The minimum Int value is \(Int.min).")
println("The maximum value for a 32-bit integer is \(Int32.max).")
println("The minimum value for a 32-bit integer is \(Int32.min).")
```

`Int32` é o tipo de inteiro com sinal de 32 bits da linguagem Swift. `Int8`, `Int16` e `Int64` também são disponibilizados para tipos de inteiro com sinal de 8 bits, 16 bits e 64 bits. Os tipos de inteiro dimensionado existem quando você realmente precisa saber o tamanho de um inteiro subjacente, como para alguns algoritmos (comuns em criptografia)

ou se você precisar transferir inteiros entre computadores (como no envio de dados pela internet). Você não vai usar muito esses tipos; um bom estilo em Swift é usar `Int` para a maioria dos casos de uso.

Todos os tipos de inteiro vistos até aqui têm sinal, o que significa que podem representar números positivos e negativos. A linguagem Swift também tem tipos de inteiro sem sinal para representar números inteiros superiores ou iguais a 0. Experimente usar alguns deles:

```
println("The maximum Int value is \(Int.max).")
println("The minimum Int value is \(Int.min).")
println("The maximum value for a 32-bit integer is \(Int32.max).")
println("The minimum value for a 32-bit integer is \(Int32.min).")

println("The maximum UInt value is \(UInt.max).")
println("The minimum UInt value is \(UInt.min).")
println("The maximum value for a 32-bit unsigned integer is \(UInt32.max).")
println("The minimum value for a 32-bit unsigned integer is \(UInt32.min).")
```

Todo tipo de inteiro com sinal (`Int8`, `Int16`, etc.) tem um tipo de inteiro sem sinal correspondente; cada um deles é nomeado com o prefixo `U` na frente do tipo de inteiro (`UInt8`, `UInt16`, etc.). O valor mínimo para todo tipo sem sinal é igual a 0. O valor máximo para um tipo sem sinal de N bits é igual a $2^N - 1$; por exemplo, o valor máximo para um tipo sem sinal de 64 bits é igual a $2^{64} - 1 = 18.446.744.073.709.551.615$. É possível entender a relação entre os valores mínimo e máximo dos tipos com sinal e sem sinal da seguinte maneira: o valor máximo de `UInt64` é igual ao valor máximo de `Int64` mais o valor absoluto do valor mínimo de `Int64`; ambos possuem 2^{64} valores possíveis, mas a versão com sinal precisa dedicar metade deles para números negativos.

Como você provavelmente notou no código que digitou em seu playground, `UInt` é o tipo de inteiro sem sinal correspondente a `Int` e, da mesma maneira que `Int`, `UInt` é um inteiro de 64 bits no OS X e pode ser de 32 ou 64 bits no iOS dependendo do destino. Algumas quantidades parecem ser representadas de forma mais adequada por um inteiro sem sinal. Por exemplo, não faz nenhum sentido uma contagem de uma série de objetos resultar em um valor negativo. No entanto, o estilo da linguagem Swift dá preferência ao uso de `Int` para todos os inteiros (incluindo contagens) a menos que um inteiro sem sinal seja exigido pelo algoritmo ou código sendo escrito. O porquê disso envolve assuntos que iremos cobrir mais tarde neste capítulo, então voltaremos em breve aos motivos por trás da preferência consistente por `Int`.

Criação de instâncias de inteiros

Você criou instâncias de `Int` no Chapter 3, onde aprendeu que pode declarar um tipo explícita ou implicitamente:

```
println("The maximum value for a 32-bit unsigned integer is \(UInt32.max).")
println("The minimum value for a 32-bit unsigned integer is \(UInt32.min).")

let myInt: Int = 10 // declare type explicitly
let myOtherInt = 3 // also of type Int, inferred by the compiler
```

Você pode criar instâncias dos outros tipos de inteiro usando declarações de tipo explícitas:

```
let myInt: Int = 10 // declare type explicitly
let myOtherInt = 3 // also of type Int

let myUInt: UInt = 40
let myInt32: Int32 = -1000
```

O que acontece se você tentar criar uma instância com um valor inválido? Por exemplo, você pode tentar criar um `UInt` com valor negativo ou um `Int8` com valor superior a 127. Experimente e descubra:

```
let myUInt: UInt = 40
let myInt32: Int32 = -1000

// Trouble ahead!
let myBadUInt: UInt = -1
let myBadInt8: Int8 = 200
```

Você deve ver um ponto de exclamação na cor vermelha no Xcode. Clique no ponto de exclamação para visualizar o erro. O compilador informa: "Integer literal overflow when stored into...". Um "integer literal" é o número inteiro digitado por você; -1 e 200 são literais de inteiro. "Overflow when stored into..." ocorre quando o compilador tenta armazenar o seu número no tipo especificado por você, ele não cabe na faixa de valores permitida do tipo. Um Int8 pode conter valores de -128 a 127; 200 está fora dessa faixa, tentar armazenar 200 em um Int8 resulta em excesso. Remova o código problemático para que possamos continuar.

```
// Trouble ahead!
let myBadUInt: UInt = 1
let myBadInt8: Int8 = 200
```

Operações com inteiros

A linguagem Swift permite que você realize operações matemáticas básicas com os inteiros. As operações aritméticas + (adição), - (subtração) e * (multiplicação) existem e permitem que você faça contas simples com números. Tente exibir o resultado de alguma operação aritmética:

```
let myUInt: UInt = 40
let myInt32: Int32 = -1000

println(10 + 20)
println(30 - 5)
println(5 * 6)
```

Os operadores também apresentam os conceitos de *precedência* e *associatividade*, os quais definem a ordem das operações quando há múltiplos operadores em uma única expressão. Por exemplo:

```
println(10 + 20)
println(30 - 5)
println(5 * 6)

println(10 + 2 * 5) // 20, because 2 * 5 is evaluated before 10 + 2
println(30 - 5 - 5) // 20, because 30 - 5 is evaluated before 5 - 5
```

Você poderia memorizar as regras que regem a precedência e a associatividade, mas, em vez disso, recomendamos usar parêntesis para deixar suas intenções explícitas:

```
println(10 + 2 * 5) // 20, because 2 * 5 is evaluated before 10 + 2
println(30 - 5 - 5) // 20, because 30 - 5 is evaluated before 5 - 5
println(10 + (2 * 5))
println((30 - 5) - 5)
```

Divisão de inteiros

Qual é o valor da expressão 11 / 3? Seria de se esperar (e com razão) que fosse 3,66666666667, mas experimente fazer:

```
println(10 + (2 * 5))
println((30 - 5) - 5)

println(11 / 3) // prints 3
```

O resultado de qualquer operação entre dois inteiros sempre será outro inteiro do mesmo tipo. 3,66666666667 não é um número inteiro e não pode ser representado como inteiro, logo a linguagem Swift trunca a parte fracionária, restando apenas o 3. Se o resultado for negativo, como -11 / 3, a parte fracionária ainda será truncada, resultando em -3. A divisão de inteiros, portanto, sempre é arredondada para 0.

Às vezes, também é útil obter o resto de uma operação de divisão. O operador de resto, %, retorna exatamente isso. Observe que usar o operador de resto em um inteiro negativo pode não retornar o que se espera:

```
println(11 / 3) // prints 3
println(11 % 3) // prints 2
println(-11 % 3) // prints -2 (surprise!)
```

Atalho de operadores

Todos os operadores que você viu até agora retornam um novo valor. Também existem versões de todos esses operadores que modificam uma variável no lugar. Uma operação extremamente comum na programação é a de *incremento* de um inteiro (adicionar 1 a ele) ou a de *decremento* de um inteiro (subtrair 1 dele). Você pode usar o operador `++` e o operador `--` para fazer essas operações:

```
println(-11 % 3) // prints -2 (surprise!)
```

```
var x = 10
x++
println("x has been incremented to \(x)")
x--
println("x has been decremented to \(x)")
```

E se você quiser aumentar `x` em um número diferente de 1? Você pode usar o operador `+=`, que combina adição e atribuição:

```
x--
println("\(x) has been decremented")

x += 10 // equivalent: x = x + 10
println("x has had 10 added to it and is now \(x)")
```

Também existem atalhos de operadores de combinação de operação e atribuição para as outras operações matemáticas básicas: `-=`, `*=`, `/=` e `%=`.

Operadores de excesso

Qual será o valor de `z` no seguinte código? (Pense nisso por um minuto antes de digitá-lo e descobrir se tem razão.)

```
let y: Int8 = 120
let z = y + 10
```

Se você achou que o valor de `z` seria igual a 130, você não está sozinho. Em vez disso, você provavelmente não consegue ver nenhum valor. Abra a visão da linha do tempo do seu playground. Abaixo de todo resultado das partes anteriores deste capítulo, você deve ver diversas linhas de números e nomes aparentemente sem sentido, começando por algo similar a "Playground execution failed: Execution was interrupted." Vamos analisar o que está acontecendo:

1. O compilador infere o tipo de `z` como sendo `Int8`, porque `y` é um `Int8`, logo `y + 10` também deve ser.
2. Quando seu playground é executado, a linguagem Swift adiciona 10 a `y`, resultando em 130.
3. Antes de armazenar o resultado de volta em `z`, a linguagem Swift verifica se 130 é um valor válido para um `Int8`.

Mas `Int8` só pode conter valores de -128 a 127; 130 é grande demais! Seu playground, portanto, atinge uma *armadilha*, que interrompe o programa. Discutiremos armadilhas em mais detalhes no Chapter 14. Por enquanto, basta saber que uma armadilha resulta na interrupção imediata e turbulenta do seu programa, o que indica um problema grave que precisa ser examinado.

A linguagem Swift fornece *operadores de excesso* que têm um comportamento diferente quando o valor é grande demais (ou pequeno demais). Em vez de prender o programa na armadilha, eles o "circundam". O operador de adição em excesso é `&+`; tente usá-lo agora:

```
let y: Int8 = 120
let z = y + 10
let z = y &+ 10
println("120 &+ 10 is \(z)")
```

O resultado de uma adição em excesso de $120 + 10$ e do armazenamento do resultado em um `Int8` é igual a -126 . Pense em aumentar em incrementos de um número por vez. Após chegar em 127 , aumentar em mais um realiza o circundamento para -128 . Então, $120 + 8 = -128$, $120 + 9 = -127$ e $120 + 10 = -126$.

Existem versões correspondentes de excesso de outros operadores aritméticos: `&-`, `&*`, `&/` e `&%`. O porquê de existirem as versões de excesso de subtração e multiplicação deve estar claro, mas o que significa divisão ou resto "em excesso"? Se você tentar dividir por 0 ou obter o resto da divisão por 0 , provocará uma armadilha. Se você usar o operador de divisão em excesso para dividir por 0 (ou usar o operador de resto em excesso para obter o resto da divisão por 0), receberá 0 de volta em vez de uma armadilha.

Operações de inteiros inesperadamente em excesso ou insuficiência podem ser uma fonte de bugs graves e difíceis de encontrar. A linguagem Swift é projetada para priorizar a segurança e minimizar esses erros. O comportamento padrão da linguagem Swift de usar uma armadilha quando houver cálculos em excesso pode ser uma surpresa para você se já tiver programado em outras linguagens; a maioria das outras linguagens tem como padrão o comportamento de "circundamento" fornecido pelos operadores de excesso da Swift. A filosofia da linguagem Swift diz que é muito melhor prender em uma armadilha (o que pode resultar em uma "falha" do programa, pelo menos na perspectiva do usuário) do que ter uma brecha de segurança em potencial. Existem alguns casos de uso da aritmética de circundamento, portanto, esses operadores especiais estão disponíveis sempre que você precisar deles.

Conversão entre tipos de inteiros

Até agora, todas as operações que vimos foram entre dois valores com exatamente o mesmo tipo. O que acontece se você tentar operar em números com tipos diferentes?

```
let a: Int = 200
let b: Int8 = 50
let c = a + b // Uh-oh!
```

Esse é um erro de tempo de compilação. Você não pode somar `a` e `b` porque eles não são do mesmo tipo. Algumas linguagens convertem automaticamente os tipos para que você possa realizar operações como essa; a Swift não faz isso. Em vez disso, você precisa converter os tipos manualmente para que correspondam. Nesse caso, você poderia converter `a` em um `Int8` ou converter `b` em um `Int`. Apenas um desses terá êxito (Por quê? Releia a seção anterior!), então faça isso agora:

```
let a: Int = 200
let b: Int8 = 50
let c = a + b // Uh-oh!
let c = a + Int(b)
```

Agora podemos retornar à recomendação de manter o `Int` para quase todas as necessidades de inteiro na linguagem Swift, mesmo para valores que normalmente só fariam sentido como valores positivos (como a contagem de "coisas"). A inferência de tipo padrão da Swift para literais é `Int` e você pode realizar operações entre um `Int` e qualquer outro tipo de inteiro sem converter um deles. Usar `Int` consistentemente ao longo do seu código reduzirá significativamente a necessidade de converter tipos e permitirá que você use inferência de tipo para inteiros de maneira bastante livre.

Exigir que você, o programador, decida como converter variáveis para fazer contas entre diferentes tipos é outro fator que distingue a linguagem Swift das outras. Novamente, essa exigência preza a segurança e a exatidão. A linguagem C, por exemplo, converte números de tipos diferentes para fazer contas entre eles, mas as conversões que ela realiza pode, às vezes, resultar em perdas - você pode perder informações quando essa linguagem fizer a conversão! O código em Swift que exigir contas entre números de tipos diferentes será mais verborrágico, mas ele também será mais claro quanto a quais conversões estão sendo realizadas. O aumento na verbosidade facilitará seu raciocínio e a manutenção do código que está fazendo as contas.

Números de ponto flutuante

Para representar um número com casa decimal, como 3,2, usamos um número de ponto flutuante. Em computadores, números de ponto flutuante são armazenados como *mantissa* e *expoente*, de maneira similar a como você escreve um número em notação científica: 123,45 também poderia ser escrito como $1,2345 \times 10^2$ ou $12,345 \times 10^1$. Além disso, números de ponto flutuante muitas vezes são imprecisos: existem muitos números que não podem ser armazenados com precisão perfeita em um número de ponto flutuante, portanto, o computador muitas vezes armazenará algo muito aproximado ao número esperado.

A linguagem Swift tem dois tipos básicos de número de ponto flutuante: `Float`, que é um número de ponto flutuante de 32 bits, e `Double`, que é um número de ponto flutuante de 64 bits. Os tamanhos de bit diferentes de `Float` e `Double` não determinam uma faixa básica de valores mínimo e máximo como fazem para os inteiros. Em vez disso, os tamanhos de bit determinam o nível de precisão que os números possuem; `Double` tem mais precisão que `Float`, o que significa que ele pode armazenar aproximações mais precisas. O tipo inferido padrão para literais de ponto flutuante em Swift é o `Double`. Você pode usar a mesma sintaxe que usou para inteiros para declarar tipos explícitos:

```
let d1 = 1.1 // implicitly Double
let d2: Double = 1.1
let f1: Float = 100.3
```

Todos os mesmos operadores numéricos também funcionam em números de ponto flutuante, incluindo o operador de resto:

```
let d1 = 1.1 // implicitly Double
let d2: Double = 1.1
let f1: Float = 100.3

println(10.0 + 11.4)
println(11.0 / 3.0)
println(12.4 % 5.0)
```

O fato de números de ponto flutuante serem inherentemente imprecisos é uma diferença importante em relação aos números inteiros que você deve ter sempre em mente. Lembre-se do operador `==` do Chapter 4, que determina se dois valores são iguais um em relação ao outro. Você pode comparar números de ponto flutuante:

```
println(10.0 + 11.4)
println(11.0 / 3.0)
println(12.4 % 5.0)

if d1 == d2 {
    println("d1 and d2 are the same!")
}
```

Até agora, tudo bem. O valor de `d1` é igual a 1,1, então poderíamos esperar que adicionar 0,1 a esse valor resultaria em 1,2. No entanto, experimente fazer isso:

```
if d1 == d2 {
    println("d1 and d2 are the same!")
}

println("d1 + 0.1 is \(d1 + 0.1)")
if d1 + 0.1 == 1.2 {
    println("d1 + 0.1 is equal to 1.2")
}
```

Os resultados podem ser muito surpreendentes! Você deve ver o resultado "d1 + 0.1 is 1.2" em seu primeiro `println()`, mas o `println()` dentro da instrução `if` não é executado. Por que não? Acontece que diversos números, incluindo 1,2, não podem ser representados exatamente como número de ponto flutuante; em vez disso, o computador armazena uma aproximação muito próxima a 1,2. Ao adicionar 1,1 e 0,1, o resultado acaba sendo, na verdade, algo como 1,2000000000000001 e o valor armazenado quando você digitou a literal 1,2 acaba sendo, na verdade, algo como 1,1999999999999999. A linguagem Swift vai arredondar ambos para 1,2 quando você exibi-

los, mas eles não são tecnicamente iguais, o que explica o porquê de o `println()` dentro da instrução `if` não ser executado.

Todos os detalhes sórdidos por trás da aritmética de ponto flutuante estão muito além do escopo deste livro. A moral da história é estar ciente de que existem algumas possíveis ciladas ao trabalhar com números de ponto flutuante. Uma consequência disso é que você nunca deve usar números de ponto flutuante para valores que devem ser exatos (como cálculos relacionados a dinheiro) - existem outras ferramentas disponíveis para essas finalidades.

Conclusão

Você acabou de completar a visão geral dos tipos de número básicos contidos na linguagem Swift. Agora você conhece os diversos tipos de inteiro com e sem sinal e sabe por que deve tentar continuar usando `Int` o máximo possível. Você aprendeu sobre operadores aritméticos básicos, incluindo as formas de atalho e versões em excesso. Você também possui conhecimento básico sobre tipos de ponto flutuante e sabe que podem ser usados para representar números que possuem casa decimal, mas também sabe que os computadores não conseguem representar números de ponto flutuante com exatidão. Finalmente, você aprendeu como fazer a conversão entre números de tipos diferentes e que a linguagem Swift exige essa conversão manual.

6

Switch

Em um dos capítulos anteriores, vimos um tipo de instrução de condicional: `if/else`. Ao longo do caminho, discutimos que `if/else` pode ser um tanto inadequado para lidar com situações que tenham mais do que algumas condições. Agora vamos alternar o foco para a instrução de `switch`. Diferentemente de `if/else`, `switch` é usado idealmente para lidar com múltiplos casos. Como veremos abaixo, a instrução de `switch` da linguagem Swift é um recurso incrivelmente flexível e poderoso.

O que é um `switch`?

As instruções de `if/else` executam o código dependendo de a condição em consideração avaliar ou não como verdadeiro. As instruções de `switch`, pelo contrário, consideram um valor particular e tentam correspondê-lo a uma série de casos. Se houver correspondência, o `switch` executa o código associado ao corpo do caso em questão. Veja a sintaxe geral de uma instrução de `switch`.

```
switch aValue {  
    case someValueToCompare:  
        // Do something to respond  
  
    case anotherValueToCompare:  
        // Do something to respond  
  
    default:  
        // Do something to handle no matches  
}
```

Acima, compararmos somente com dois casos, mas poderíamos ter incluído um número arbitrário de outros casos para fazer a comparação. Se `aValue` corresponder a qualquer um dos casos de comparação abaixo, o corpo do caso será executado. Está implícito nisso o requisito de que o tipo em cada um dos casos deve corresponder ao tipo sendo comparado. De maneira mais concreta, o tipo de `aValue` deve ser correspondido pelo tipo de `someValueToCompare` e `anotherValueToCompare`. Por último, se `aValue` corresponder a `someValueToCompare`, o código embaixo do caso será executado. Nesse caso, há apenas um comentário: `// do something to respond`.

É importante mencionar a esta altura que a instrução de `switch` acima provocará um erro de tempo de compilação. Por quê? Se estiver curioso, vá em frente e digite a instrução acima em um Playground. Se `aValue` tiver um valor, e todos os casos também, você poderá ver um erro para cada um dos casos dizendo: "'case' label in a 'switch' should at least one executable statement". Esse erro significa que cada caso deve ter pelo menos uma linha de código executável associado a ele. Como os comentários são intrinsecamente não executáveis, nossa instrução de `switch` não atende a esse requisito. Assim, o erro sublinha o ponto em que cada caso nas instruções de `switch` representa uma ramificação de execução separada.

Observe o uso do caso `default`: O caso `default` não é obrigatório, mas ajuda a cumprir um importante requisito da linguagem Swift em relação às instruções de `switch`: elas devem abranger todos os casos. Esse requisito significa que se deve associar um caso a todos os valores do tipo verificado. Aqui é onde entra o caso `default`: o caso `default` é executado se não ocorrer nenhuma correspondência nos casos listados acima dele. Normalmente usamos um caso `default` quando não faz sentido ter um caso específico para cada valor no tipo a ser correspondido.

Portanto, se aValue não corresponder a someValueToCompare ou anotherValueToCompare, o código abaixo do caso default será executado.

Alterne as coisas

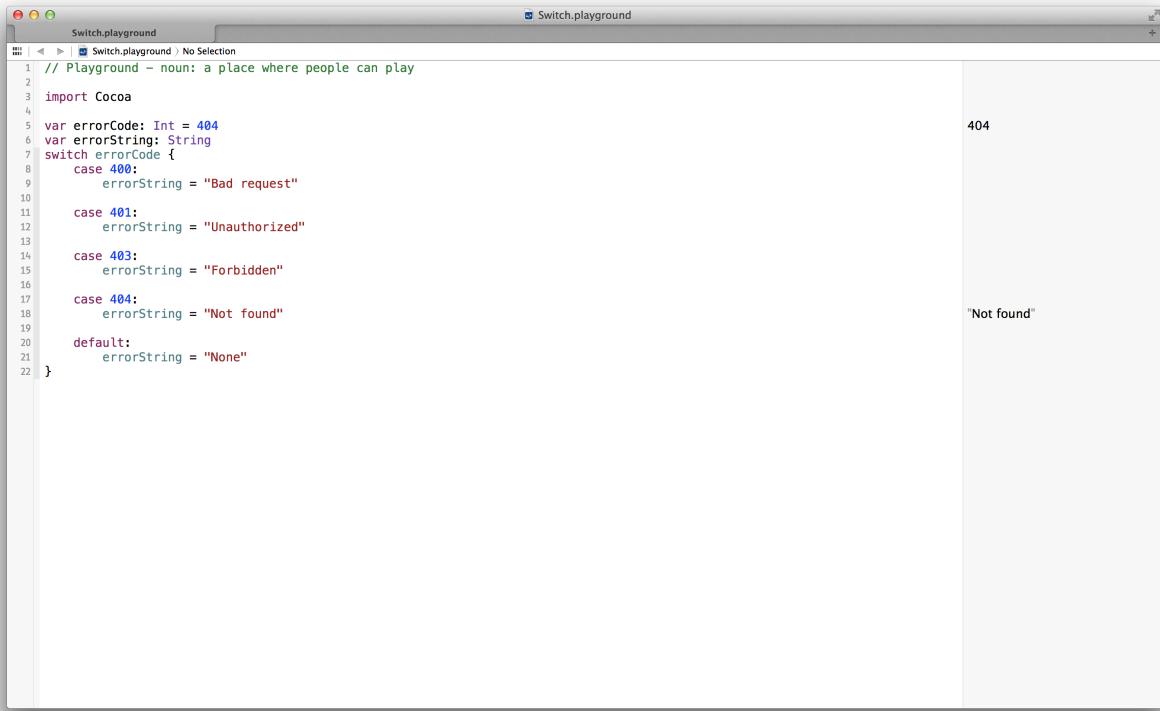
Crie um novo Playground chamado de Switch e salve-o no local desejado. Digite o exemplo abaixo para que seu código fique igual ao nosso.

```
import Cocoa

var statusCode: Int = 404
var errorString: String
switch statusCode {
    case 400:
        errorString = "Bad request."
    case 401:
        errorString = "Unauthorized."
    case 403:
        errorString = "Forbidden."
    case 404:
        errorString = "Not found."
    default:
        errorString = "None."
}
```

A instrução de switch acima usa um código de status de HTTP para corresponder a uma instância de `String` que descreve o erro para o código se houver de fato um erro. Observe que a barra lateral de resultados mostra qual caso correspondeu ao erro. `errorString` é, portanto, atribuído para ser igual a `"Not found."` Depois disso, podemos usar `errorString` conforme necessário.

Figure 6.1 Correspondendo uma string de erro a um código de erro



Digamos que queremos usar a instrução de `switch` acima para criar uma descrição de erro significativa. Altere seu código para corresponder ao nosso.

```

import Cocoa

var statusCode: Int = 404
var errorString: String
var errorString: String = "The request failed with the error:"
switch statusCode {
    case 400+:
        errorString = "Bad request."
    case 401+:
        errorString = "Unauthorized."
    case 403+:
        errorString = "Forbidden."
    case 404+:
        errorString = "Not found."
    default:
        errorString = "None."
    case 400, 401, 403, 404:
        errorString += " There was something wrong with the request."
        fallthrough
    default:
        errorString += " Please review the request and try again."
}

```

Essa nova instrução de `switch` é um pouco diferente do nosso exemplo anterior. Agora inicializamos `errorString` com uma string que representa o primeiro segmento da descrição do nosso erro. Os dois pontos

no final de `errorString` sinaliza visivelmente que essa instância ainda não está finalizada e está pronta para o compartilhamento. O resto da instrução de `switch` tem poucas diferenças importantes. Agora você tem um caso para todos os códigos de status de erro. Você também usou o operador de atribuição de composto `+=` dentro do corpo do caso para adicionar a informação específica de `statusCode` a `errorString`. Se o `statusCode` corresponder a qualquer um dos valores do caso, `errorString` receberá o texto " There was something wrong with the request. ".

Você também introduziu uma *instrução de transferência de controle* chamada de `fallthrough`. A palavra-chave `fallthrough` garante que a instrução de `switch` "caia atravessando" até o fundo de cada caso até o próximo. Se houvesse múltiplos casos, o `switch` circularia por cada caso, construindo a `errorString`. A instrução de transferência de controle de `fallthrough` faz com que casos reais sejam ignorados e garante que o código dentro de cada caso seja executado. No exemplo acima, a instrução de `fallthrough` significa que, se o caso corresponder, a instrução de `switch` não será concluída, mas, em vez disso, prosseguirá para o caso `default`. Se não tivéssemos usado a palavra-chave `fallthrough`, a instrução de `switch` teria finalizado a execução após a primeira correspondência. O resultado da instrução de `switch` acima é a definição da `errorString` como: "The request failed with the error: There was something wrong with the request. Please review the request and try again."

Alternância avançada

Imagine que não ligamos muito para a construção de uma `errorString` bem específica. Um recurso da instrução de `switch` da linguagem Swift que pode nos ajudar é a capacidade de corresponder faixas de casos ao valor que gostaríamos de alternar. Vamos reescrever nossa instrução de `switch` acima para tirar proveito desse recurso.

```
switch statusCode {
    case 400, 401, 403, 404:
        errorString += " There was something wrong with the request."
        fallthrough

    default:
        errorString += " Please review the request and try again."
}

switch statusCode {
    case 100, 101:
        errorString += " Informational, 1xx."

    case 204:
        errorString += " Successful but no content, 204."

    case 300...307:
        errorString += " Redirection, 3xx."

    case 400...417:
        errorString += " Client error, 4xx."

    case 500...505:
        errorString += " Server error, 5xx."

    default:
        errorString = "Unknown. Please review the request and try again."
}
```

A instrução de `switch` acima aproveita a sintaxe ... da *correspondência de faixa* para criar uma faixa inclusiva para cada categoria de código de status de HTTP. Isto é, 300...307 constitui uma faixa que inclui 300, 307 e cada inteiro entre eles. Você notará que nossa primeira faixa é separada por uma vírgula, já que há apenas dois códigos para verificar. O caso seguinte aborda apenas 204 porque 200s geralmente representam códigos de status "com êxito", mas 204 é o código para "nenhum conteúdo". O resultado dessa instrução de `switch` é a definição da `errorString` como igual a: "The request failed with the error: Client error 4xx." Observe que, se apenas não reconhecermos o código, nossa `errorString` refletirá essa informação por meio do nosso caso `default`.

Imagine, no entanto, que nosso aplicativo será usado por profissionais de desenvolvimento web já bem familiarizados com códigos de status de HTTP. Então, embora não precisemos ser extremamente específicos quanto ao texto do código de status, os usuários de nosso aplicativo provavelmente vão gostar de ver os códigos de status numéricos reais em nossa `errorString`. Além disso, suponhamos que precisemos nos proteger contra códigos de erro desconhecidos para que não criemos nossa `errorString` de forma errada. Podemos construir com base em nossa instrução de `switch` anterior e incluir essa informação usando um recurso das instruções de `switch` da linguagem Swift chamado de *ligação de valores*.

```
switch statusCode {
    case 100, 101:
        errorString += " Informational, 1xx."
        errorString += " Informational, \(statusCode)."

    case 204:
        errorString += " Successful but no content, 204."

    case 300...307:
        errorString += " Redirection, 3xx."
        errorString += " Redirection, \(statusCode)."

    case 400...417:
        errorString += " Client error, 4xx."
        errorString += " Client error, \(statusCode)."

    case 500...505:
        errorString += " Server error, 5xx."
        errorString += " Server error, \(statusCode)."

    default:
        errorString = "Unknown. Please review the request and try again."
}

case let unknownCode:
    errorString = "\(unknownCode) is not a known error code."
```

Nós usamos a *interpolação de strings* no exemplo acima para passar nosso `statusCode` para a `errorString` em cada caso. O único caso diferente é o último. Nesse caso, criamos uma constante temporária, chamada de `unknownCode`, que nosso `switch` realizará comparação quando o `statusCode` não corresponder a nenhum dos valores dados nas faixas acima. Por exemplo, se o `statusCode` fosse definido como igual a 200 por qualquer motivo, representando êxito, nosso `switch` definiria a `errorString` como igual a: "200 is not a known error code." Se precisássemos trabalhar no `unknownCode`, por qualquer razão, poderíamos tê-lo declarado com `var` em vez de `let`. Isso também significaria que poderíamos ter modificado o valor do `unknownCode` dentro do corpo do caso final.

O que apresentamos acima é bom, mas pode, na verdade, resultar em problemas. Afinal, um código de status 200 não é *realmente* um erro e, portanto, seria bom se nossa instrução de `switch` não capturasse esses casos. Abaixo, usamos a cláusula `where` para garantir que o `unknownCode` não seja um 2xx, êxito. `where` permite que verifiquemos se há condições adicionais dentro de nossa instrução de `switch`. O comportamento desse recurso cria um tipo de filtro dinâmico dentro do `switch`.

```
switch statusCode {  
    case 100, 101:  
        errorString += " Informational, \(statusCode)."   
  
    case 204:  
        errorString += " Successful but no content, 204."   
  
    case 300...307:  
        errorString += " Redirection, \(statusCode)."   
  
    case 400...417:  
        errorString += " Client error, \(statusCode)."   
  
    case 500...505:  
        errorString += " Server error, \(statusCode)."   
  
case let unknownCode:  
    errorString = "\((unknownCode) is not a known error code."  
case let unknownCode where (unknownCode >= 200 && unknownCode < 300)  
    || unknownCode > 505:  
    errorString = "\((unknownCode) is not a known error code."  
  
default:  
    errorString = "Unexpected error encountered."  
}
```

Lembre-se do exemplo anterior acima, nós simplesmente declaramos uma constante no caso final, o que significou que esse caso correspondeu a tudo que acabou chegando até o final da instrução de `switch`. Essa funcionalidade tornou nossa instrução de `switch` anterior abrangente. Como nosso caso para `unknownCode` agora determina uma faixa específica de códigos de status, ela não é mais abrangente. Assim, adicionamos um caso `default` que configura a `errorString` para indicar que o erro é desconhecido.

Como não estamos utilizando o recurso de `fallthrough` da linguagem Swift, a instrução de `switch` finalizará a execução assim que encontrar um caso correspondente e executar o corpo desse caso. De forma correspondente, se `statusCode` for igual a 204, irá corresponder a esse caso e a `errorString` será definida adequadamente. Assim, esse valor nunca será correspondido nesse caso apesar de a faixa especificada na cláusula `where` incluí-lo.

Correspondência de tuplas e padrão

Agora que você tem seu `statusCode` e sua `errorString`, seria útil uni-los. Embora sejam relacionados logicamente, eles estão armazenados atualmente em variáveis independentes. Uma *tupla* pode ser usada para agrupar os dois.

Uma tupla é um agrupamento de valores considerados pelo desenvolvedor como logicamente relacionados. Esses valores diferentes são agrupados como valor composto. O resultado desse agrupamento é uma lista ordenada de elementos.

Crie sua primeira tupla em Swift para agrupar o `statusCode` e a `errorString`.

```

var statusCode: Int = 420
var errorString: String = "The request failed with the error:"
switch statusCode {
    case 100, 101:
        errorString += " Informational, \(statusCode)."
    case 204:
        errorString += " Successful but no content, \(statusCode)."
    case 300...307:
        errorString += " Redirection, \(statusCode)."
    case 400...417:
        errorString += " Client error, \(statusCode)."
    case 500...505:
        errorString += " Server error, \(statusCode)."
    case let unknownCode where (unknownCode >= 200 && unknownCode < 300) || unknownCode > 505:
        errorString = "\((unknownCode)) is not a known error code."
    default:
        errorString = "Unknown error encountered."
}
println(errorString)

let error = (statusCode, errorString)

```

Você criou uma tupla agrupando `statusCode` e `errorString` entre parêntesis. O resultado foi atribuído à constante `error`.

Os elementos de uma tupla podem ser acessados por seu índice. Digite o texto abaixo para acessar cada elemento armazenado dentro da tupla.

```

...
let error = (statusCode, errorString)
error.0
error.1

```

Você deve ver 420 e "Unknown error encountered." exibidos na barra lateral de resultados para `error.0` e `error.1` respectivamente.

As tuplas da linguagem Swift também podem ter elementos nomeados. Nomear os elementos de uma tupla torna mais fácil a leitura do código. Por exemplo, não é muito fácil controlar quais valores são representados por `error.0` e `error.1`.

Dê um nome informativo para os elementos da sua tupla.

```

...
let error = (statusCode, errorString)
error.0
error.1
let error = (code: statusCode, error: errorString)
error.code
error.error

```

Agora você pode acessar os elementos da sua tupla usando um nome relacionado: `code` para `statusCode` e `error` para `errorString`. Você verá que a sua barra lateral de resultados exibe as mesmas informações.

Correspondência de padrão

Tuplas também são muito úteis para a correspondência de padrões. Imagine, por exemplo, que você queira escrever um `switch` que corresponda a uma faixa específica de códigos de status na tupla `error`, mas você não liga muito para a string de erro. Alternar para uma tupla facilita muito esse procedimento em Swift.

Adicione o código abaixo a sua nova tupla para corresponder a uma faixa específica de códigos de status.

```
...
let error = (code: statusCode, error: errorString)
error.code
error.error

switch error {
    case (400...505, _):
        println("PANIC!")

    default:
        println("Not worth panicking.")
}
```

A nova instrução de switch corresponde a apenas um caso. Observe que a string de erro não é o foco do padrão que você está tentando corresponder. O caractere de sublinhado (_) corresponde a qualquer string de erro, o que permite que você o ignore. Assim, o foco aqui é voltado aos códigos de status específicos de nosso interesse.

Como o código de status na tupla é 420, você deve ver "PANIC!" na barra lateral de resultados.

Conclusão

Conseguimos tratar de grande parte do que as instruções de switch podem nos oferecer. Vamos revisar rapidamente o que vimos.

- sintaxe básica e uso das instruções de switch
- correspondência de faixa
- ligação de valores
- mecanismos de transferência de controle por meio de `fallthrough` e `break`
- a cláusula `where`

Todos esses itens são recursos poderosos e permitem que as instruções de switch sirvam como um meio extremamente eficaz de controlar a lógica de nossos aplicativos.

Desafio

Analise a instrução de switch abaixo. O que será registrado no console?

```
var statusCode: Int = 301
var errorString: String = "The request failed with the error:"
switch statusCode {
    case 100, 101:
        errorString += " Informational, \$(statusCode)."
    case 204:
        errorString += " Successful but no content, \$(statusCode)."
    case 300...307:
        errorString += " Redirection, \$(statusCode)."
    case 400...417:
        errorString += " Client error, \$(statusCode)."
    case 500...505:
        errorString += " Server error, \$(statusCode)."
    case let unknownCode where (unknownCode >= 200 && unknownCode < 300)
        || unknownCode > 505:
        errorString = "\$(unknownCode) is not a known error code."
    default:
        errorString += " Unknown error."
}
println(errorString)
```


7

Loops

Os loops nos ajudam a ganhar impulso em tarefas de natureza repetitiva. Eles executam um conjunto de códigos repetidamente dependendo de um dado número de iterações ou de uma condição estabelecida. Você verá que os loops podem nos salvar de ter que escrever códigos entediantes e repetitivos. Então tome nota! Você vai usá-los muito durante o desenvolvimento.

Neste capítulo, você usará dois tipos de loops: 1) o loop `for` e 2) o loop `while`. O loop `for` é usado muitas vezes quando você quer iterar nos elementos específicos de uma instância ou coleção de instâncias. Os loops `for` são usados frequentemente quando o número de iterações que devem ser realizadas é conhecido ou fácil de derivar. Como distinção, o loop `while` é adequado para tarefas que você vai querer executar repetidamente desde que determinada condição seja atendida. Vamos começar com nosso primeiro loop: o loop `for`.

Loops for-in

Crie um novo Playground chamado de Loops e salve-o onde quiser. Adicione o código a seguir:

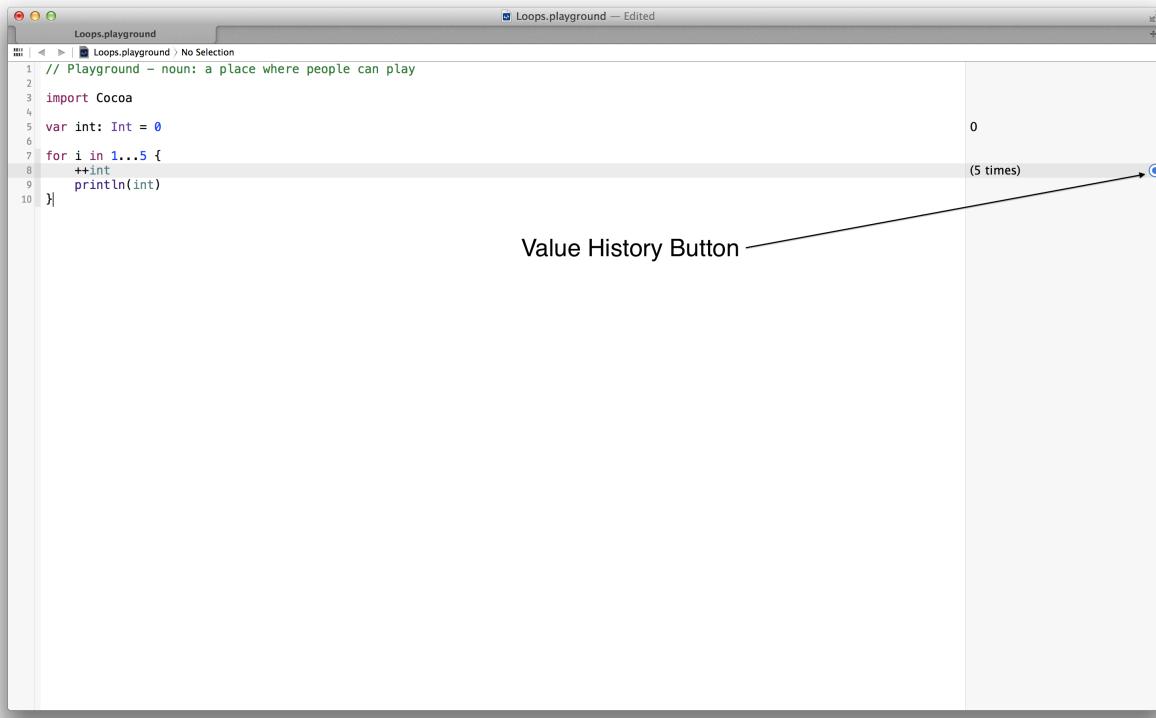
```
import Cocoa  
  
var int: Int = 0  
  
for i in 1...5 {  
    ++int  
    println(int)  
}
```

Primeiramente, você declarou uma variável chamada de `int` que é uma instância de `Int` e é inicializada para ser igual a 0. Em seguida, você escreveu o que, na linguagem Swift, chamamos de loop `for-in`. O loop `for-in` executa um conjunto de códigos para cada item em uma faixa, sequência ou coleção específica. No código acima, você usou `...` para criar uma faixa inclusiva que começou em 1 e continuou até 5. Além disso, você declarou uma constante denominada `i` que representa o valor atual do iterador à medida que o loop atravessa a faixa de 1 a 5. Para cada inteiro nessa faixa, você aumentou o `int` em 1. Você então registrou esse valor no console. Se, em vez disso, você tivesse usado `.., a faixa ainda iniciaria no 1, mas teria terminado antes, no 4.`

No loop acima, você declarou uma constante chamada de `i` que existe apenas dentro do corpo do loop. Na primeira iteração do loop, o valor é o primeiro valor na faixa do loop: 1. Na segunda iteração, o valor de `i` é igual a 2 e assim por diante. O código entre chaves (`{}`) é executado em cada iteração do loop. Esse processo continua até que `i` atinja o final da faixa. Por isso, você pode ler o loop `for-in` acima como: "para `i` na faixa de 1 a 5".

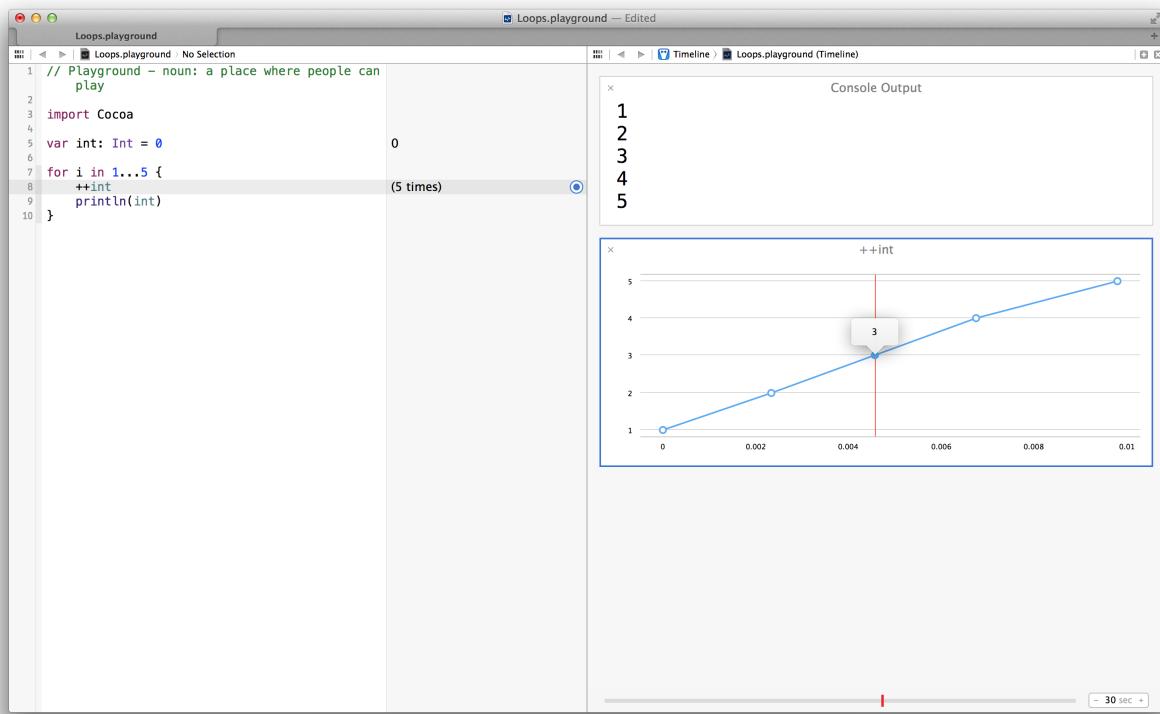
Para ver os resultados de seu loop, revele o editor assistente pressionando `command-option-return` em seu teclado. Como antes, você agora verá o console e o valor de `int` que foi registrado em cada iteração do loop ao longo de sua faixa. Vamos adicionar mais informações clicando no botão *Value History* na borda direita da barra lateral de resultados na linha do código: `++int`. Clicar nesse botão revela a *linha do tempo* do histórico de valores dessa instância. Além disso, você também pode tocar no botão de histórico de valores para revelar ao mesmo tempo tanto o editor assistente como a linha do tempo. Veja os detalhes na figura 7.1 abaixo.

Figure 7.1 Botão de histórico de valores



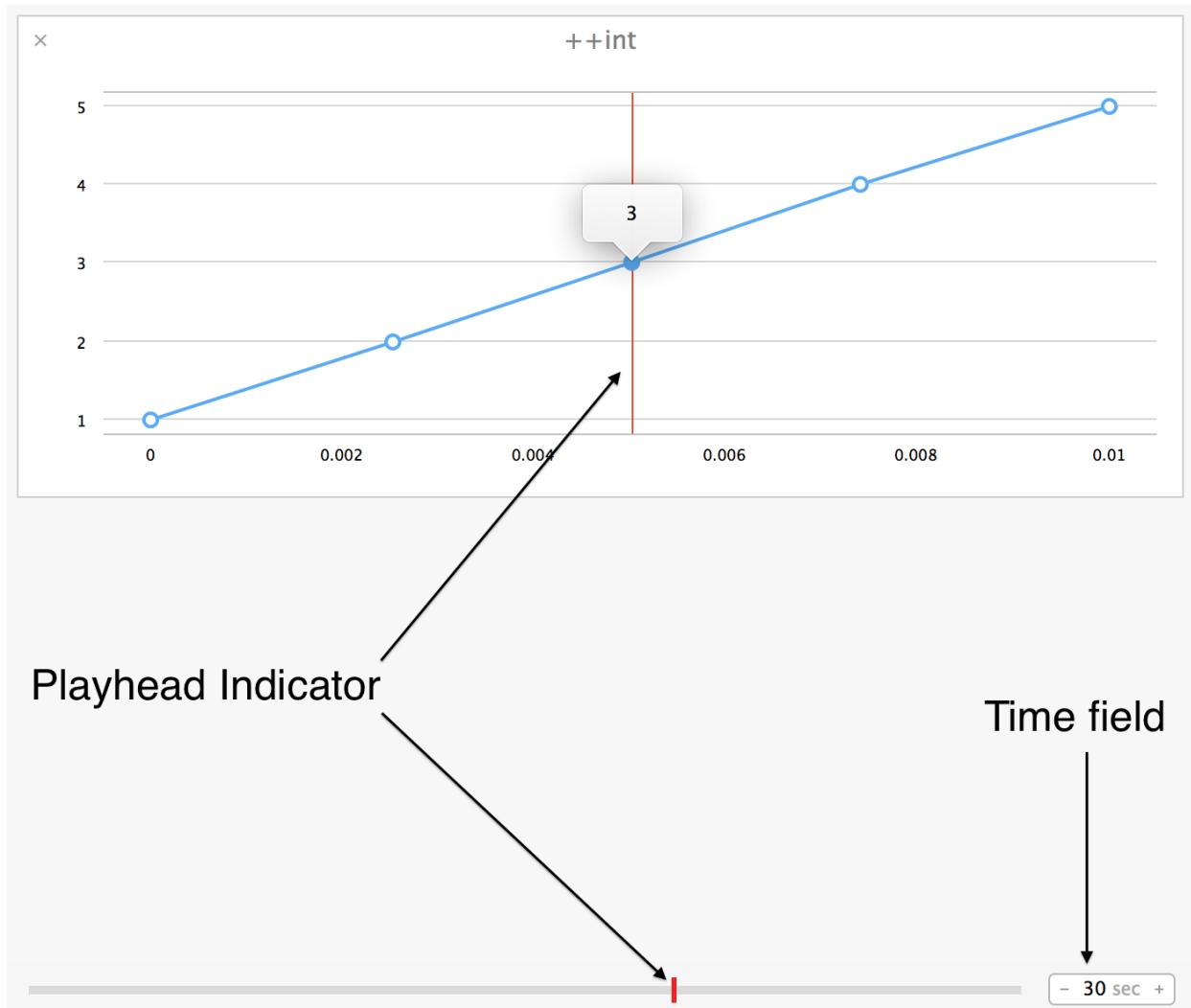
A figura 7.1 exibe o que acontece após clicar no botão de histórico de valores. Observe que você consegue ver tanto o console como a linha do tempo do histórico de valores para `int` no editor assistente. Cada ponto ao longo da linha na linha do tempo mostra o valor de `int` em cada iteração na faixa do loop acima. O eixo x descreve o tempo de execução, e o eixo y refere-se ao valor de `int` ao longo do curso de vida do loop. Clique no terceiro ponto da linha e você verá o Playground se destacar e exibir o valor nesse ponto específico. Como pode imaginar, esse recurso é ótimo para loops mais complicados ou longos.

Figure 7.2 Exibição do histórico de valores na linha do tempo



Está vendo o controle deslizante na parte inferior do editor assistente? Você deve ver uma barra vermelha vertical cuja posição corresponde aproximadamente a sua seleção atual na linha que representa o histórico de valores do `int`. Deslide essa barra vermelha horizontalmente para mudar a localização do playhead da linha do tempo. Você também pode manipular o período de tempo relatado na linha do tempo. Manipule o número de segundos exibidos no campo de tempo exibido no canto inferior direito para aumentar ou diminuir o intervalo de tempo retratado.

Figure 7.3 Playhead e campo de tempo



Como você declarou `i` como iterador no loop `for-in` acima, é possível acessar `i` dentro de cada iteração do loop. Por exemplo, você poderia ter escrito o loop acima da seguinte maneira:

```
for i in 1...5 {
    ++int
    println(int)
    println("int equals \$(int) at iteration \$(i)")
}
```

A amostra de código acima mostra como acessar o iterador `i` dentro de cada iteração do loop. Essa implementação do loop `for-in` exibiria o seguinte no console para a terceira iteração do loop: "int equals 3 at iteration 3".

Se, por qualquer motivo, você não ligar para o uso de um iterador explicitamente declarado na faixa especificada acima, você pode ignorá-lo completamente usando `_` em vez de uma constante com nome. Por exemplo, altere seu loop para corresponder ao apresentado abaixo. Perceba que você praticamente voltou à primeira implementação do loop `for-in` dada no início do capítulo.

```
for i in 1...5
for _ in 1...5 {
    ++int
    println("int equals \$(int) at iteration \$(i)")
    println(int)
}
```

Essa implementação do loop `for-in` é boa porque se preocupa com a garantia de que uma operação específica ocorra em um determinado número de vezes. Ela não dedica nenhuma atenção particular para verificar e reportar o valor do iterador a cada passagem do loop por sua faixa.

Nota rápida sobre inferência de tipo

Observe a mensagem a seguir.

```
for i in 1...5 {
    ++int
    println("int equals \((int) at iteration \(i)")
}
```

Observe que você nunca teve que declarar `i` ao tipo `Int`? Você poderia se quisesse (por exemplo, `for i: Int in 1...5` -- a porção `let` da declaração é assumida pela sintaxe para você), mas não é necessário. O tipo de `i` é inferido com base no contexto. Nesse exemplo, `i` é inferido como sendo do tipo `Int` porque a faixa especificada acima contém inteiros.

Como podemos ver, a inferência de tipo é bastante útil. Ela não requer tanta digitação física no teclado e, quanto menos você tiver que digitar, menos erros de digitação você cometerá. Existem alguns casos em que você precisará declarar o tipo especificamente e com certeza destacaremos esses casos quando surgirem. Em termos gerais, achamos que você deve tirar proveito da inferência de tipo sempre que possível e você verá isso muitas vezes neste livro.

Loops for

A linguagem Swift também suporta o loop clássico `for`:

```
for initialization; condition; increment {
    // Code to execute at each iteration
}
```

Usamos ponto e vírgula para separar as três partes do loop `for`. Cada uma das partes realiza uma função específica nas três etapas de execução desse loop.

1. Quando o loop é inserido, a expressão de inicialização é avaliada para configurar o iterador para o loop.
2. A expressão de condição é avaliada; se for falsa, o loop é finalizado e a execução do código é transferida para depois do loop. Se for verdadeiro, o código entre as chaves (`{}`) do loop é executado.
3. Depois que o código entre as chaves for executado, a expressão de incremento é executada. Dependendo do código aqui, o incrementador pode ser aumentado ou diminuído. Uma vez definido o incrementador, voltaremos à segunda etapa para determinar se o loop deve continuar a iterar.

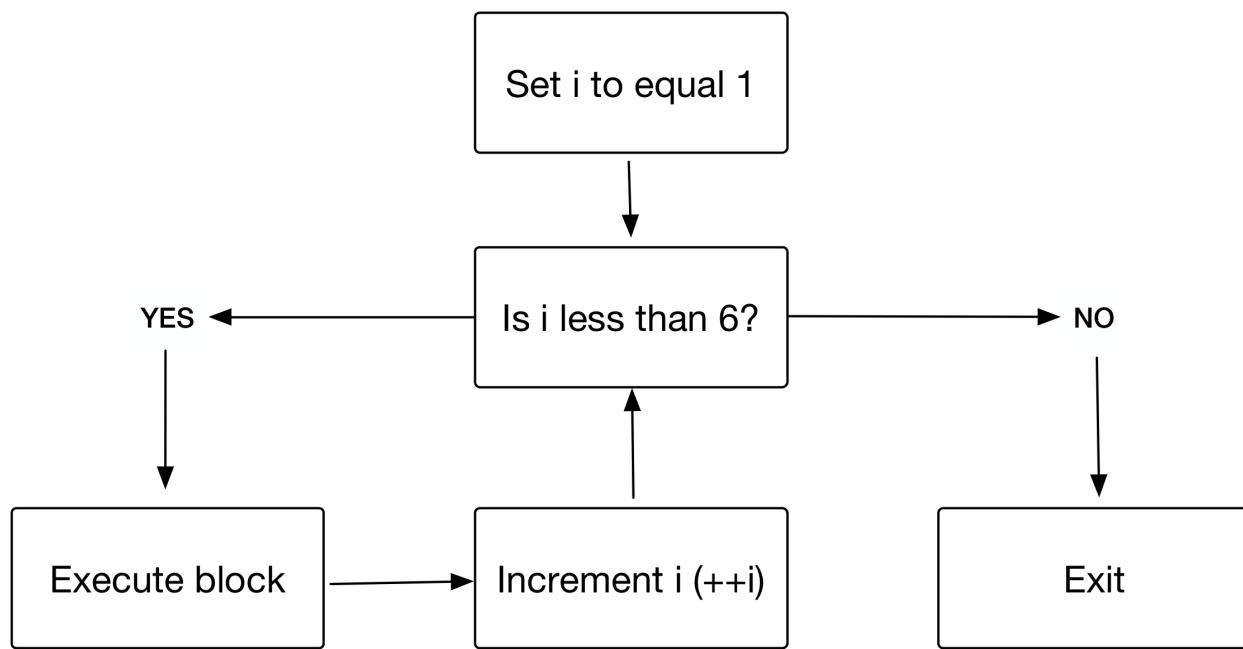
Para deixar esse loop mais concreto, refatore a implementação do loop `for-in` que criamos no início do capítulo para usar a forma mais tradicional.

```
for var i = 1; i <= 5; ++i {
    ++int
    println(int)
}
```

Como podemos ver, a implementação é similar e o resultado é exatamente o mesmo. Como o loop `for-in`, o incrementador `i` declarado no segmento de inicialização do loop `for` está disponível dentro do escopo do loop (isto é, o código entre as chaves do loop).

A figura 7.4 apresenta o diagrama do fluxo de execução descrito na listagem de código acima.

Figure 7.4 Diagrama do loop for



Loops while

O loop clássico `for` discutido acima pode ser expresso, na verdade, como loop `while`.

```

var i = 1
while i < 6 {
  ++int
  println(int)
  ++i
}
  
```

Como o loop `for` acima, esse loop `while` inicializa um incrementador (`var i = 1`), avalia uma condição (`i < 6`), incrementa um contador (`++i`) e, em seguida, retorna ao topo do loop `while` para determinar se o loop deve continuar com a iteração.

Loops `while` são os melhores para os casos em que o número de iterações pelos quais o loop passará é desconhecido por algum motivo no momento em que foi escrito. Por exemplo, imagine um jogo de tiro no espaço bem simples com uma espaçonave atirando de forma contínua desde que ela ainda possua seus escudos. Diversos fatores externos podem diminuir ou aumentar os escudos da nave, mas desde que os escudos da nave tenham um valor superior a 0, a espaçonave continuará atirando. O trecho de código abaixo ilustra uma implementação relativamente simples de como esse comportamento pode ser realizado.

```

while shields > 0 {
  println("Fire blasters!")
}
  
```

Loops do-while

A principal diferença entre o loop `while` e o loop `do-while` é que o primeiro avalia a condição antes de entrar no loop. Essa diferença significa que o loop `while` pode nem ser executado dependendo de sua capacidade de aprovar ou não sua condição em primeiro lugar. Como distinção, o loop `do-while` executa seu loop apenas uma vez e, em seguida, avalia sua condição. A sintaxe para o loop `do-while` demonstra essa diferença.

Você pode refatorar o loop `while` acima para evitar uma situação um tanto deprimente: e se a espaçonave do usuário for criada e, devido a um acidente estranho, perder imediatamente todos os escudos? Talvez nossa espaçonave tenha sido gerada bem na frente de um asteroide que se aproximava. Nesse caso, a espaçonave não conseguiria dar nenhum tiro. Que chato. Nem teríamos a chance de explodir as coisas! Essa seria uma experiência muito ruim para o usuário. Um loop `do-while` ajudará a evitar esse evento decepcionante.

```
do {  
    println("Fire blasters!")  
} while shields > 0
```

O loop `do-while` reorganiza nosso loop `while` anterior para garantir que possamos disparar pelo menos um tiro com a nave antes de a condição do loop ser avaliada. Como podemos ver, o bloco de código que contém a linha `println("Fire blasters!")` é executado antes e, em seguida, a condição do loop `do-while` é avaliada para determinar se o loop deve continuar a iteração. Assim, o loop `do-while` garante que nossa espaçonave vai conseguir disparar pelo menos uma vez.

Instruções de transferência de controle, Redux

Vamos rever as *instruções de transferência de controle* no contexto dos loops. Lembre-se que no capítulo sobre instruções de `switch` vimos que *instruções de transferência de controle* alteram a ordem típica de execução. No contexto de um loop, isso significa que você pode controlar se a execução recicla ou não mais uma vez para o topo do loop ou deixa os loops todos juntos.

Considere o exemplo anterior: imagine que você quer parar o loop onde ele está e começar novamente desde o topo. Use a *instrução de transferência de controle* `continue` para completar essa tarefa. Para entender como isso funciona, refatore o loop `while` que você escreveu acima para que fique igual ao seguinte exemplo e adicione-o a seu Playground.

```
var shields = 5  
var blastersOverheating = false  
var blasterFireCount = 0  
while shields > 0 {  
  
    if blastersOverheating {  
        println("Blasters are overheated! Cooldown initiated.")  
        sleep(5)  
        println("Blasters ready to fire")  
        sleep(1)  
        blastersOverheating = false  
        blasterFireCount = 0  
        continue  
    }  
  
    if blasterFireCount > 100 {  
        blastersOverheating = true  
        continue  
    }  
  
    println("Fire blasters!")  
    ++blasterFireCount  
}
```

Você acabou de adicionar um bocado de código, então vamos analisá-lo. Primeiramente, você adicionou algumas variáveis que monitoram as seguintes informações: 1) `shields` são do tipo `Int` e são inicializados para ser igual a 5; 2) `blastersOverheating` é um `Boolean` inicializado como `false` que monitora se os canhões precisam ou não de tempo para resfriar; e 3) `blasterFireCount` é uma variável do tipo `Int` que monitora a quantidade de tiros que a espaçonave disparou para que saibamos se os canhões vão superaquecer ou não.

Em seguida, você tem duas instruções de `if` que verificam se os canhões estão superaquecendo ou não e a contagem de disparos, respectivamente. Se os canhões estiverem superaquecendo, você registrará essa informação no console.

A função `sleep()` informa o sistema para que aguarde 5 segundos, que é o nosso modo de modelar a fase de resfriamento dos canhões. Você, depois disso, registrou que os canhões estão prontos para disparar novamente (após aguardar por mais 1 segundo, algo feito apenas para facilitar a visualização do que é registrado no console em seguida), definiu `blasterOverheating` como sendo igual a `false` e também restaurou `blasterFireCount` para ser igual a 0. Por último, você usou `continue` para pular para o início do loop. Sendo assim, com nossos escudos intactos e nossos canhões resfriados, sua espaçonave está pronta para abrir fogo.

A segunda condicional verifica se `blasterFireCount` é superior a 100. Se essa condicional for avaliada como `true`, você definirá o Boolean para `blastersOverheating` como sendo `true`. Neste momento, nossos canhões estão superaquecidos e, por isso, você precisa de alguma maneira voltar para o topo do loop para que sua espaçonave não dispare enquanto os canhões estão nesse estado instável. Como vimos no parágrafo anterior, você usou `continue` para voltar ao topo do loop. Como os canhões da espaçonave superaqueceram, a condicional na primeira instrução de `if` avaliará como `true` e os canhões serão desligados por 5 segundos para resfriar.

Após a avaliação da segunda condicional, você registrará no console como antes. Sua última linha, no entanto, será nova, pois será adicionado 1 a `blasterFireCount`. Depois de incrementar essa variável, o loop vai voltar ao topo, avaliar a condição e iterar novamente ou entregar o fluxo de execução para a linha imediatamente seguinte à chave que fecha o loop.

É importante observar que esse código será executado por tempo indefinido. Se nada for alterado e seu computador tiver energia suficiente para funcionar para sempre, esse loop continuará sendo executado. Isso é o que chamamos de *loop infinito*. Mas todos os jogos devem chegar ao fim. Por exemplo, imagine que o jogo termina depois que o usuário destruir 500 demônios espaciais. Vamos adicionar uma *instrução de transferência de controle* `break` para sair do loop quando esse evento ocorrer.

```
var shields = 5
var blastersOverheating = false
var blasterFireCount = 0
var spaceDemonsDestroyed = 0
while shields > 0 {

    if spaceDemonsDestroyed == 500 {
        println("You beat the game!")
        break
    }

    if blastersOverheating {
        println("Blasters are overheated! Cooldown initiated.")
        sleep(5)
        println("Blasters ready to fire")
        sleep(1)
        blastersOverheating = false
        blasterFireCount = 0
        continue
    }

    if blasterFireCount > 100 {
        blastersOverheating = true
        continue
    }
    println("Fire blasters!")
    ++blasterFireCount
    ++spaceDemonsDestroyed
}
```

No exemplo acima, você adicionou uma nova variável chamada de `spaceDemonsDestroyed`. Essa variável monitorará o número de demônios espaciais destruídos pelo usuário, os quais aumentaremos arbitrariamente toda vez que os canhões dispararem. Você é bom de tiro, afinal. Em seguida, você verificou se `spaceDemonsDestroyed` é igual a 500 e, se for o caso, é registrada vitória no console. Observe o uso da instrução de `break` acima. A instrução de transferência de controle `break` sairá do loop `while` e a execução continuará na linha imediatamente seguinte à chave que fecha o loop. Isso faz algum sentido em nosso exemplo: se o usuário tiver destruído 500 demônios espaciais, vence-se o jogo e os canhões não precisam mais ser disparados.

Conclusão

Você agora conhece os loops da linguagem Swift! Reserve um momento para considerar o que aprendeu neste capítulo.

1. O loop `for-in`.
2. O loop tradicional `for`.
3. Inferência de tipo.
4. O loop `while`
5. O loop `do-while`.
6. *Instruções de transferência de controle* no contexto de loops.

Quanta informação! Parabéns. Você deve estar se sentindo muito bem agora. Lembre-se de tentar o desafio abaixo para aprimorar ainda mais suas habilidades.

O próximo capítulo apresentará as Opcionais, um recurso da linguagem Swift que permite que as instâncias sejam do tipo `nil`. O que `nil` significa? Continue lendo para descobrir!

Desafio

Use um loop para contar de 2 em 2 de 0 a 100. Use outro loop para garantir que o primeiro loop seja executado cinco vezes. Dica: uma boa maneira de fazer isso é usando um loop aninhado.

8

Strings

Em programação, o conteúdo textual é representado por strings. Como bem sabemos, "Hello, playground" representa uma coleção ordenada de caracteres, cada um representando uma letra no alfabeto. A combinação dessas letras forma uma palavra. Na verdade, você já viu uma string em ação. Os playgrounds que vimos foram iniciados com a string "Hello, playground" no topo.

Trabalhando com strings

Na linguagem Swift, as strings são criadas com o tipo `String`. Crie um novo playground para criar algumas instâncias de `String`. Chame o playground `Strings.playground` e salve-o em algum local conveniente.

Destaque e exclua tudo que estiver no playground. Você vai criar e trabalhar com suas próprias strings. Adicione o código a seguir para representar uma nova instância do tipo `String`.

```
let playground = "Hello, playground"
```

Você criou uma instância de `String` chamada de `playground`. `playground` foi criado usando a sintaxe literal da string. A sintaxe literal da string delimita uma sequência de texto com as aspas.

Essa instância foi criada com a palavra-chave `let`, tornando-a uma constante. Lembre-se de que, sendo uma constante, a instância não poderá ser alterada. Se você tentar alterá-la, o compilador emitirá um erro.

Cria uma nova string, mas deixe essa instância mutável.

```
let playground = "Hello, playground"  
var mutablePlayground = "Hello, mutable playground"
```

`mutablePlayground` é uma instância mutável do tipo `String`. Em outras palavras, você pode alterar o conteúdo dessa string. Faça isso adicionando o código a seguir.

```
let playground = "Hello, playground"  
var mutablePlayground = "Hello, mutable playground"  
mutablePlayground += "!"
```

Você usou o operador de adição e atribuição (`+=`) para adicionar um ponto de exclamação a `mutablePlayground`. Olhe os resultados da barra lateral de resultados à direita do playground. Você verá que a barra lateral está mostrando que a instância foi alterada: "Hello, mutable playground!".

As strings são compostas de sequências ordenadas de caracteres. Na verdade, os caracteres que compõem as strings em Swift são de um tipo específico: o tipo `Character`. O tipo `Character` da linguagem Swift é usado para representar caracteres Unicode, usados em combinação para formular uma instância de `String`.

Por enquanto, faça um loop na string `mutablePlayground` para ver o tipo `Character` em ação.

```
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"
for c: Character in mutablePlayground {
    println("\(c)")
}
```

O loop itera em cada Character c em mutablePlayground. Cada iteração registra o caractere no console. Cada caractere é registrado no console em sua própria linha porque você usou o `println()`, que exibe uma quebra de linha após registrar o conteúdo.

Você verá o seguinte registro no console quando revelar o editor assistente.

Figure 8.1 Registro de caracteres em uma string

The screenshot shows the Xcode interface with the 'Console Output' tab selected. The output window displays the characters of the string 'Hello, mutable playground!' printed one by one on separate lines, demonstrating the effect of the `println()` statement.

```
X Console Output
H
e
l
l
o
,
m
u
t
a
b
l
e
p
l
a
y
g
r
o
u
n
d
!
```

Unicode

Unicode é um padrão reconhecido internacionalmente que codifica caracteres para que possam ser processados e representados sem obstáculos independentemente da plataforma. Ele é projetado para representar a linguagem humana (e outras formas de comunicação, como emoticons) nos computadores. Com isso, um número único é atribuído a cada caractere nesse padrão.

Os tipos `String` e `Character` da linguagem Swift são construídos com base em Unicode e são eles que fazem a maior parte do serviço pesado. Entretanto, é sempre bom saber como esses tipos funcionam com o Unicode. Com esse conhecimento, você provavelmente evitara desperdício de tempo e frustração no futuro.

Escalares de Unicode

No fundo, strings em Swift são compostas do que chamamos de *escalares de Unicode*. As escalares de Unicode descrevem números únicos de 21 bits designados para representar um caractere específico no padrão Unicode. Por exemplo, U+0061 representa a letra 'a' minúscula. U+1F60E representa o emoticon sorrindo com óculos escuros. O texto U+1F60E é o modo padrão de escrever um caractere Unicode. A parte 1F60E é um número escrito em hexadecimal, ou base 16.

Digite o exemplo abaixo para ver como usamos escalares de Unicode em Swift e no playground.

```
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"

for c: Character in mutablePlayground {
    println("\\"(c)"")
}
let oneCoolDude = "\u{1F60E}"
```

O novo código acima cria uma instância do tipo `String`. Observe que você usou uma nova sintaxe: "`\u{1F60E}`". A sintaxe `\u{n}` é escrita para denotar uma escalar de Unicode específica `n`, na qual `n` é um número hexadecimal único que representa um caractere específico. Portanto, a instância é atribuída como igual ao caractere que representa um emoticon.

O que isso tem a ver com strings mais comuns? Acontece que todas as strings em Swift são compostas dessas escalares de Unicode, como indicado acima. Alguns outros conceitos são úteis ao explicar essa ideia.

Cada caractere em Swift é formado por uma ou mais escalares de Unicode. Uma escalar de Unicode mapeia um caractere fundamental específico em determinada linguagem. Por exemplo, `U+0301` representa a escalar de Unicode para o acento agudo combinador: '̄'. Essa escalar é escrita como combinador porque será colocada no topo ou combinada ao caractere que a precede. Você pode usar essa escalar com a letra 'a' minúscula para criar o caractere 'a' acentuado.

```
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"

for c: Character in mutablePlayground {
    println("\\"(c)"")
}

let oneCoolDude = "\u{1F60E}"
let aAcute = "\u{0061}\u{0301}"
```

Você deve ver a letra 'a' acentuada na barra lateral de resultados.

Observe que você criou esse caractere combinando duas escalares de Unicode. A primeira escalar, `U+0061`, corresponde à letra 'a' minúscula. A segunda escalar, `U+0301`, corresponde ao acento agudo combinador '̄'. As marcas de combinação são associadas a outra escalar de Unicode.

Esse exercício demonstra que cada caractere em Swift é o que chamados de cluster de grafemas estendido. Os clusters de grafemas estendidos são uma sequência de uma ou mais escalares de Unicode. No exemplo mais recente, você decompôs a letra 'a' minúscula e acentuada nas duas escalares de Unicode que a constituem: a letra e o acento.

A linguagem Swift também oferece um mecanismo que pode ser usado para ver as escalares de Unicode reais que constituem sua string. Por exemplo, você pode ver todas as escalares de Unicode utilizadas pela linguagem Swift para criar a instância de `String` chamada de `playground` criada acima. Digite o código a seguir para ver.

```
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"

for c: Character in mutablePlayground {
    println("\\"(c)"")
}

let oneCoolDude = "\u{1F60E}"

let aAcute = "\u{0061}\u{0301}"
for scalar in playground.UnicodeScalars {
    print("\\"(scalar.value) ")
}
```

A constante playground tem uma propriedade chamada de `UnicodeScalars`. Não se preocupe com o conceito de propriedade neste momento; esse assunto será tratado com detalhes no Chapter 16. Por enquanto, tudo que você precisa saber é que propriedade é uma forma do tipo de se ligar aos dados. Nesse caso, `UnicodeScalars` se liga a todas as escalares usadas pela linguagem Swift para criar a string.

Abra o editor assistente para visualizar o console. Você verá o seguinte resultado na parte inferior: 72 101 108 108 111 44 32 112 108 97 121 103 114 111 117 110 100. O que todos esses números significam?

Lembre-se da propriedade das `UnicodeScalars` contida nos dados que representam todas as escalares de Unicode que foram usadas para criar o playground da instância da string. Isso significa que cada número no console corresponde a uma escalar de Unicode que representa um único caractere na string. Cada um desses números é representado como um inteiro de 32 bits sem sinal. Assim, 72 é convertido no valor da escalar de Unicode de U+0048 ou 'H' maiúsculo.

Equivalência canônica e aplicativos

O Unicode oferece uma escalar específica para a letra 'a' minúscula e acentuada. Isso significa que você não precisa decompor esse caractere específico em duas partes: a letra e o acento. Ele já vem previamente composto. A escalar é U+00E1. Crie uma nova string de constante que use essa escalar de Unicode.

```
...
let aAcute = "\u{0061}\u{0301}"

for scalar in playground.UnicodeScalars {
    print("\(scalar.value) ")
}

let aAcutePrecomposed = "\u{00E1}"
```

Como podemos ver, `aAcutePrecomposed` parece ter o mesmo valor que `aAcute`. A grande diferença entre essas duas constantes é que `aAcute` foi criada usando duas escalares de Unicode e `aAcutedPrecomposed` usou apenas uma. De fato, se você verificar se esses dois caracteres são os mesmos, você descobrirá que a linguagem Swift dirá que sim.

```
let aAcute = "\u{0061}\u{0301}"

for scalar in playground.UnicodeScalars {
    print("\(scalar.value) ")
}

let aAcutePrecomposed = "\u{00E1}"

let b = (aAcute == aAcutePrecomposed) // true
```

Por que é esse o resultado? Parece um tanto estranho que essa comparação diga que os dois caracteres são os mesmos já que você usou um conjunto diferente de escalares de Unicode para criar cada um. A resposta a essa questão envolve algo que chamamos de *equivalência canônica*.

A equivalência canônica refere-se à possibilidade de dois clusters de grafemas estendidos serem linguisticamente iguais ou não. Dois caracteres, ou duas strings, são, assim, considerados iguais se tiverem o mesmo significado linguístico e a mesma aparência, independentemente das escalares de Unicode possivelmente diferentes usadas para criá-los. Por exemplo, `aAcute` e `aAcutedPrecomposed` são strings iguais porque ambas representam a letra 'a' minúscula com acento agudo, embora cada caractere tenha sido criado com um conjunto diferente de escalares de Unicode. Sendo assim, embora os clusters de grafemas estendidos resultantes sejam diferentes, eles são tratados como canonicamente equivalentes porque os caracteres resultantes têm o mesmo significado linguístico e a mesma aparência.

Elementos de contagem

A equivalência canônica possui implicações na contagem de elementos de uma string. É possível achar que `aAcute` e `aAcutePrecomposed` teriam contagens de caractere diferentes devido à diferença no número de escalares de Unicode usadas para compor cada string. Escreva o código a seguir para verificar.

```

let aAcute = "\u{0061}\u{0301}"

for scalar in playground.UnicodeScalars {
    print("\(scalar.value) ")
}

let aAcutePrecomposed = "\u{00E1}"
let b = (aAcute == aAcutePrecomposed)
println("aAcute: \(countElements(aAcute)); aAcuteDecomposed: \(countElements(aAcutePrecomposed))")

```

Você usou a função `countElements()` para determinar a contagem de caracteres dessas duas strings. Essa função itera em escalares de Unicode de um caractere ou string para determinar o tamanho. A barra lateral de resultados revela que a contagem de caracteres é igual: o tamanho de ambos é de 1 caractere. A equivalência canônica ajuda a explicar por que essas duas strings têm o mesmo número de caracteres. Ambas representam a letra 'a' minúscula com acento agudo.

Índices e faixas

Como as strings são uma coleção ordenada de caracteres, é possível pensar que podemos acessar um caractere específico em uma string da seguinte maneira:

```

let index = playground[3] // 'y'???
// error: 'subscript' is unavailable: cannot subscript String with an Int

```

Como podemos ver, o compilador da linguagem Swift impediu o acesso a um caractere específico em uma string e subíndice. Essa limitação tem a ver com o fato de as strings e os caracteres em Swift serem armazenados como clusters de grafemas estendidos. O resultado disso é a impossibilidade de indexar uma string com um inteiro porque a linguagem Swift não sabe qual escalar de Unicode corresponde a determinado índice sem passar por todos os caracteres que o precedem. Essa operação pode ser dispendiosa. Portanto, a linguagem Swift o força a ser mais explícito.

A linguagem Swift usa um tipo opaco chamado de `String.Index` para monitorar os índices. Se você quiser achar o caractere em um índice específico, você pode usar a propriedade `startIndex` do tipo `String`. Essa propriedade retorna o índice inicial de uma string como `String.Index`. Você pode, então, usar esse ponto inicial juntamente com `advance()` para mover para a frente até chegar à posição de sua escolha. Imagine, por exemplo, que você quer o quinto caractere da string `playground` criada no início deste capítulo.

```

...
let fromStart = playground.startIndex
let toPosition = 4
let end = advance(fromStart, toPosition)
let fifthCharacter = playground[end] // 'o'

```

Você usou a propriedade `startIndex` na string para obter o primeiro índice da string. Essa propriedade retornou uma instância do tipo `String.Index`. Em seguida, você criou uma constante para se fixar na posição dentro da string até a qual você quer ir. 4 representa o quinto caractere porque a string é indexada a zero (ou seja, 0 é o primeiro índice). Em seguida, você usou `advance()` para avançar desse ponto inicial até a posição desejada. O resultado é um `String.Index` que pode ser usado para subindexar sua instância da string `playground`, que resultou no caractere 'o'.

Faixas, tal como índices, dependem do tipo `String.Index`. Imagine que você queira pegar os primeiros cinco caracteres do `playground`. Você pode usar as constantes `fromStart` e `end` já criadas.

```

...
let fromStart = playground.startIndex
let toPosition = 4
let end = advance(fromStart, toPosition)
let fifthCharacter = playground[end] // 'o'
let range = fromStart ... end
let firstFive = playground[range] // 'Hello'

```

A sintaxe `fromStart ... end` criou uma faixa do tipo `String.Index`. Você foi capaz, então, de usar essa nova faixa como subíndice na string `playground`. Esse subíndice pegou os primeiros cinco caracteres do `playground`. O resultado disso é que `firstFive` é uma constante igual a "Hello".

Conclusão

Neste capítulo, você aprendeu como a linguagem Swift processa e representa strings. Você aprendeu que strings são compostas de caracteres. Ao longo do caminho, você descobriu que os caracteres são, na verdade, compostos de clusters de grafemas estendidos. Essa descoberta o levou a explorar o Unicode e demonstrou como esse padrão mapeia as escalares de Unicode numéricas como caracteres específicos. Finalmente, todo esse conhecimento foi colocado em prática na demonstração de como a linguagem Swift representa índices e faixas no tipo `String`.

Desafio

Desafio de bronze

O tipo `String` possui um método que retornará um `Bool` que indica se uma string é iniciada ou não com certo prefixo. Descubra esse método e use-o para determinar se a string `playground` inicia com a string "Hello".

Desafio de prata

Refaça o desafio de bronze acima substituindo a string "Hello" por uma instância criada totalmente com base em suas escalares de Unicode correspondentes. Você vai precisar fazer uma pequena pesquisa na internet para descobrir os códigos adequados.

9

Opcionais

As opcionais são um recurso especial da linguagem Swift. Elas são usadas para indicar que uma instância pode não ter um valor. Quando você vir uma opcional, saberá que existem duas possibilidades a respeito dessa instância: 1) ela possui um valor e está pronta para ser usada ou 2) ela não possui nenhum valor. Se uma instância não tiver nenhum valor associado a ela, dizemos que ela é `nil` (nula).

Você pode usar opcionais com qualquer tipo para indicar que uma instância pode não ter um valor. Em outras palavras, você pode usar uma opcional com qualquer tipo para sinalizar que ela é potencialmente `nil`. Esse recurso distingue a linguagem Swift da Objective-C, a qual permite que apenas objetos sejam `nil`.

Este capítulo abordará como declarar tipos opcionais para que você possa começar a usar opcionais, mostrará como usar *ligação de opcionais* para verificar se uma opcional é `nil` e como usar seu valor se de fato tiver valor e introduzirá *encadeamento de opcionais* para consultar uma sequência de valores opcionais.

Tipos de opcionais

Opcionais em Swift deixam a linguagem mais segura. Lembre-se de que uma instância declarada como tipo opcional pode potencialmente ser `nil`. Como distinção, se uma instância não for declarada como tipo opcional, essa instância definitivamente não será `nil`. O uso de opcionais significa que o compilador sabe com certeza se uma instância pode ser `nil`. Como resultado, essa declaração explícita torna seu código mais expressivo e seguro. Vamos analisar mais de perto como declararmos um tipo opcional.

De início, crie um novo Playground e chame-o de Opcionais. Comece verificando se o seu código corresponde ao trecho abaixo.

```
import Cocoa  
  
var errorCodeString: String?  
errorCodeString = "404"
```

Primeiramente, você criou uma variável chamada de `errorCodeString` para conter informações de código de erro em formato de string. Em seguida, você declarou explicitamente o tipo da `errorCodeString` como `String`. Observe, contudo, que você especificou o tipo da `errorCodeString` de maneira um pouco diferente do que fez antes. Dessa vez, você colocou um `?` no final da `String`. O `?` significa que `errorCodeString` é uma opcional do tipo `String`. Em outras palavras, você declarou `errorCodeString` com planejamento para considerar a possibilidade de ela não conter nenhum valor.

Agora que você declarou uma opcional e atribuiu a ela um valor, é hora de usá-la de alguma forma. Registre o valor da opcional no console da seguinte maneira:

```
import Cocoa  
  
var errorCodeString: String?  
errorCodeString = "404"  
println(errorCodeString)
```

Como você atribuiu um valor de "404" a `errorCodeString`, você pode registrar o valor no console tranquilamente. Se você não tivesse atribuído um valor a `errorCodeString`, seria registrado `nil` no console. Vá em frente e tente!

Registrar `nil` no console não é muito útil. De maneira mais ampla, você provavelmente vai querer que suas variáveis sejam `nil` e vai querer executar o código adequado da existência ou não de um valor. Nessas circunstâncias, você pode usar uma condicional para ganhar impulso no valor de uma variável. Por exemplo, digamos que se alguma operação tiver gerado erro, queiramos atribuir esse erro a uma nova variável e registrá-la no console. Adicione o código a seguir em seu Playground.

```
import Cocoa

var errorCodeString: String?
errorCodeString = "404"
println(errorCodeString)
if errorCodeString != nil {
    let theError = errorCodeString!
    println(theError)
}
```

Você criou uma nova constante chamada de `theError` para conter o valor de `errorCodeString`. Para isso, você anexou um `!` a `errorCodeString`. O ponto de exclamação faz o que chamamos de *desempacotamento forçado*. O ponto de exclamação acessou um valor subjacente da opcional, o que nos permite pegar "404" e atribuí-lo à nossa constante `theError`. Dizemos que ele faz isso "à força" porque ele tenta acessar o valor subjacente havendo ou não um valor lá. Ou seja, o `!` assume que há um valor; se não houver, desempacotar o valor dessa maneira leva a um erro de tempo de execução. Finalmente, você registrou esse novo valor da constante no console.

O que aconteceria se não tivéssemos desempacotado o valor da `errorCodeString` e simplesmente tivéssemos atribuído a opcional à constante `theError`? O erro ainda seria registrado no console corretamente. Então, por que você desempacotou o valor da opcional e o atribuiu a uma constante? As respostas dessas perguntas envolvem uma melhor compreensão do que é um tipo opcional.

Se você tiver omitido o ponto de exclamação no final da `errorCodeString`, você apenas atribuiria a opcional `String` a uma constante em vez do valor real de `String` para o código de erro. Na verdade, o tipo da `errorCodeString` é `String?`. `String?` não é o mesmo tipo que `String` - se você tiver uma variável `String`, você não pode configurá-la para o valor de uma `String?` sem desempacotar a opcional.

A opcional `errorString` é `nil` quando é declarada na primeira vez porque não foi atribuído nenhum valor a ela. Na próxima linha, você atribuiu "404" a `errorCodeString` e, então, o valor para a opcional se tornou `Some "404"`. Você pode comparar um valor opcional com `nil` para determinar se ele contém um valor ou não. No código acima, você primeiro verificou se a `errorCodeString` possui um valor; se o valor não for igual a `nil`, você saberá que é seguro desempacotar a `errorCodeString`.

A criação de uma constante dentro da condicional é um pouco complicada. Por sorte, existe um jeito melhor de ligar condicionalmente um valor de opcional a uma constante. Chama-se *ligação opcional*.

Ligaçāo opcional

A ligação opcional é um padrão útil para detectar se uma opcional contém um valor. Se houver um valor, o atribuiremos a uma constante ou variável temporária e o disponibilizaremos dentro da primeira ramificação de execução de uma condicional. Assim, ela pode ajudá-lo a tornar seu código mais conciso ao mesmo tempo que retém sua natureza expressa. Veja uma sintaxe básica:

```
if let temporaryConstant = anOptional {
    // Do something with temporaryConstant
} else {
    // There was no value in anOptional; i.e., anOptional is nil
}
```

Com essa sintaxe em mãos, refatore o exemplo acima usando a ligação opcional.

```
import Cocoa

var errorCodeString: String?
errorCodeString = "404"
if let theError = errorCodeString {
    let theError = errorCodeString!
    println(theError)
}
```

Como podemos ver, o exemplo é mais ou menos igual. Na verdade, tudo o que você fez foi mover a constante `theError` do corpo da condicional para a primeira linha. `theError` agora é uma constante temporária disponível dentro da primeira ramificação da condicional. Em outras palavras, se houver um valor dentro da opcional, uma constante temporária é disponibilizada para ser usada na condicional no bloco de código executado se a condição `for` avaliada como `true`. Observe que você não precisou desempacotar a opcional à força. Se a conversão `for` bem-sucedida, essa operação é realizada por você e o valor da opcional é disponibilizado na constante temporária declarada. Também observe que você poderia ter declarado `theError` com a palavra-chave `var` se precisasse manipular o valor dentro da primeira ramificação da condicional.

Opcionais desempacotadas implicitamente

Neste momento, é importante mencionar as opcionais desempacotadas implicitamente, embora você não vá usá-las muito até discutirmos classes e inicialização de classes posteriormente neste livro. Opcionais desempacotadas implicitamente são como tipos opcionais comuns, mas há uma diferença importante: você não precisa desempacotá-las. Como é isso? Tem a ver com o modo como você as declara. O código abaixo refatora o exemplo acima para trabalhar com uma opcional desempacotada implicitamente.

```
import Cocoa

var errorCodeString: String!
errorCodeString = "404"
println(errorCodeString)
```

Essa reimaginação do exemplo deste capítulo redeclara a opcional com o `!`, o que indica que ela é uma opcional desempacotada implicitamente, e dispensa a condicional. A condicional é removida porque usar uma opcional desempacotada implicitamente indica muito mais confiança do que seu correspondente mais humilde. De fato, grande parte do poder e da flexibilidade associados à opcional desempacotada implicitamente está relacionada à ideia de que você não precisa desempacotar a opcional para acessar seu valor.

Observe, contudo, que esse poder e essa flexibilidade apresentam algum risco: acessar o valor de uma opcional desempacotada implicitamente resultará em erro de tempo de execução se ela não tiver um valor. Por isso, sugerimos que você não use a opcional desempacotada implicitamente se achar que há chance de a instância se tornar nula. Correspondentemente, o uso é limitado a alguns casos especiais. Como indicamos acima, o caso principal diz respeito à inicialização de classe. Voltaremos a esse conceito no capítulo que trata de inicialização. Por enquanto, você sabe o suficiente sobre os princípios básicos das opcionais desempacotadas implicitamente para entender o que acontece se achá-las por aí.

Encadeamento de opcionais

Tal como a ligação de opcionais, o encadeamento de opcionais oferece um mecanismo para consultar uma opcional e determinar se ela contém um valor. Uma diferença importante entre os dois é o fato de o encadeamento de opcionais permitir que o programador encadeie diversas consultas em um valor da opcional. Se cada opcional na cadeia contiver um valor, a chamada a cada uma delas terá êxito e toda a cadeia de consulta retornará uma opcional do tipo esperado. Se qualquer opcional na cadeia de consulta for `nil`, toda a cadeia retornará `nil`.

Vamos começar com um exemplo conciso. Adicione o seguinte a seu Playground.

```
var errorCodeString: String?  
errorCodeString = "404"  
if let theError = errorCodeString {  
    println(theError)  
}  
  
var errorCodeInt: Int?  
errorCodeInt = errorCodeString?.toInt()
```

Primeiro você declarou `errorCodeInt` como sendo do tipo `Int?` por meio do ponto de interrogação comum. Em seguida, você atribuiu o valor de inteiro da `errorCodeString` a sua nova variável. Para isso, você usou uma função chamada de `toInt()`. `toInt()` é uma função definida no tipo `String` que converte uma string em sua forma de inteiro, se for possível. Se essa conversão não for possível, a função retorna `nil`. Por exemplo, a string "4" pode ser convertida em um inteiro, mas a string "Four" não. Sendo assim, na verdade, `toInt()` retorna uma opcional do tipo `Int?`.

O ponto de interrogação colocado ao final da `errorCodeString` sinaliza que essa linha de código inicia o processo de encadeamento de opcionais. No exemplo acima, você pode deduzir que `errorCodeInt` contém um valor; no entanto, essa dedução pode ser mais difícil em um programa mais complicado que, com certeza, você encontrará durante o desenvolvimento. Use a ligação de opcionais nesse caso para testar se há um valor no resultado.

```
var errorCodeString: String?  
errorCodeString = "404"  
if let theError = errorCodeString {  
    println(theError)  
}  
  
var errorCodeInt: Int?  
errorCodeInt = errorCodeString?.toInt()  
  
if let errInt = errorCodeInt {  
    println("The integer error code is \u{errInt}.")  
} else {  
    println("errorCodeString was either nil or could not be converted to an integer.")  
}
```

Se `errorCodeInt` possuir um valor, vamos desempacotá-lo por meio de ligação de opcionais e atribuí-lo a uma constante temporária. Depois disso, podemos acessar o valor no interior da primeira ramificação da instrução condicional. Caso contrário, se não houver valor, lidamos com esse cenário na segunda ramificação da instrução condicional.

Até agora, não tiramos proveito do fato de que o encadeamento de opcionais pode ser usado em uma sequência de opcionais de comprimento arbitrário. Como `toInt()` retorna um tipo opcional, `Int?`, você pode encadear outra chamada de função. Imagine que, por alguma razão, você queira adicionar um inteiro arbitrário ao código de erro.

```
var errorCodeString: String?  
errorCodeString = "404"  
if let theError = errorCodeString {  
    println(theError)  
}  
  
var errorCodeInt: Int?  
errorCodeInt = errorCodeString?.toInt()?.advancedBy(100)  
if let errInt = errorCodeInt {  
    println("The integer error code is \u{errInt}.")  
} else {  
    println("errorCodeString is either nil or could not be converted to an integer.")  
}
```

A função `advancedBy()` retorna um `Int` adicionando um valor de inteiro, fornecido dentro de parêntesis como argumento, a outro inteiro. Assim, se ambos `errorCodeString` e `toInt()` possuírem valores, o resultado será avançado em 100. Como antes, é importante observar que o resultado será do tipo `Int?`. Ou seja, o resultado será um inteiro opcional.

Operador de união nula

Uma operação comum ao lidar com opcionais é o desejo de obter um valor se a opcional contiver valor ou usar algum valor padrão se a opcional for nula. Por exemplo, ao analisar sintaticamente `errorCodeString`, você pode querer que o padrão seja 500 (código HTTP para erro interno do servidor) se a string não contiver um inteiro válido. Você pode realizar isso com ligação de opcionais:

```
var actualErrorCode: Int
if let errInt = errorCodeString?.toInt() {
    actualErrorCode = errInt
} else {
    actualErrorCode = 500
}

println("actualErrorCode = \(actualErrorCode)")
```

Existem dois problemas com essa técnica. O primeiro é que precisamos escrever muito código para algo que deveria ser uma operação simples: obter o valor da opcional ou usar 500 se a opcional for nula. O segundo é que usar ligação de opcionais significa que você teve que declarar `actualErrorCode` como var em vez de let, mesmo não querendo alterar o valor. Ambos os problemas podem ser resolvidos por meio do *operador de união nula*:

```
var actualErrorCode: Int
if let errInt = errorCodeString?.toInt() {
    actualErrorCode = errInt
} else {
    actualErrorCode = 500
}

let actualErrorCode = errorCodeString?.toInt() ?? 500
println("actualErrorCode = \(actualErrorCode)")
```

`??` é o operador de união nula. O lado esquerdo deve ser uma opcional - `errorCodeString?.toInt()` em seu caso, que é uma opcional `Int`. O lado direito deve ser uma não opcional do mesmo tipo - 500 em seu caso, que é um `Int`. Se a opcional do lado esquerdo for nula, `??` retorna o lado direito. Se a opcional do lado esquerdo não for nula, `??` retorna o valor contido na opcional.

Tente alterar `errorCodeString` para que ela não contenha um inteiro analisável e confirme se `actualErrorCode` recebe o valor 500:

```
errorCodeString = "404"
errorCodeString = "an error occurred"
```

Conclusão

Este capítulo foi bastante complexo. Você aprendeu um monte de material novo. As opcionais são assunto novo independentemente do seu nível de experiência com desenvolvimento para Mac ou iOS. Entretanto, elas são um recurso poderoso da linguagem Swift. Trata-se de uma necessidade frequente de representar `nil` em uma instância. As opcionais ajudam o desenvolvedor a monitorar se as instâncias são `nil` ou não e oferecem um mecanismo para responder de forma adequada.

Se você ainda não estiver tão confortável com as opcionais, não se preocupe. Você ainda vai vê-las muito nos próximos capítulos. Elas são uma parte importante da linguagem Swift.

Desafio

Anteriormente neste capítulo, dissemos que acessar um valor da opcional quando for `nil` resultará em erro de tempo de execução. Cometa esse erro desempacotando à força uma opcional quando for `nil`, examine o erro e compreenda o que ele está dizendo.

Part III

Coleções e funções

10

Arrays

Uma tarefa importante na programação envolve agrupar valores logicamente relacionados. Por exemplo, imagine que seu aplicativo mantém uma lista de amigos, livros favoritos, destinos turísticos e etc. de um usuário. Muitas vezes, é necessário que esses valores possam ser agrupados e transmitidos em seu código. As coleções tornam essas operações convenientes.

A linguagem Swift oferece dois tipos de coleção. O primeiro a ser abordado se chama `Array`. O segundo, assunto do próximo capítulo, é chamado de `Dictionary`. Os arrays são uma coleção ordenada de valores. Cada posição em um array é identificada por um índice e qualquer valor pode aparecer diversas vezes em um array. Os arrays normalmente são usados quando, por algum motivo, é importante ou útil saber a ordem dos valores, mas não há um pré-requisito que dita que a ordem dos valores deve ser significativa. Diferentemente da Objective-C, o tipo `Array` da linguagem Swift pode conter qualquer tipo de valor – objetos e não objetos.

Para começar, crie e salve um novo Playground em Swift chamado de Arrays.

Criação de um array

Neste capítulo, você criará um array que representará sua "bucket list", ou seja, a lista de coisas que você gostaria de fazer no futuro. Em `Arrays.playground`, adicione o código a seguir:

```
import Cocoa  
var bucketList: Array<String>
```

Você criou uma nova variável chamada de `bucketList` que é do tipo `Array`. Você já deve estar familiarizado com grande parte dessa sintaxe. Por exemplo, a palavra-chave `var` significa que a `bucketList` é uma variável. Isso significa que `bucketList` é mutável, nós a usamos para indicar que o array pode ser alterado. Discutiremos arrays imutáveis mais tarde neste capítulo. A parte nova, contudo, é a seguinte: `<String>`. Esse código diz à `bucketList` quais tipos de instâncias podem ser aceitos. No exemplo acima, seu novo array aceitará instâncias do tipo `String`.

É importante mencionar que este código é igual ao código acima quanto à funcionalidade.

```
import Cocoa  
var bucketList: Array<String>  
var bucketList: [String]
```

Os colchetes indicam à `bucketList` que ela é uma instância de `Array` e a sintaxe da `String` indica à `bucketList` quais tipos de valores são aceitos. Como o array conterá informações referentes a seus objetivos futuros, faz sentido aceitar instâncias do tipo `String`. Entretanto, você poderia ter especificado qualquer tipo desejado.

Sua `bucketList` só está declarada, mas não foi inicializada ainda. Isso significa que ela ainda não está pronta para aceitar instâncias do tipo `String`. Por exemplo, se você tentasse anexar um item a sua `bucketList` com o código: `bucketList.append("Climb Mt. Everest")`, você receberia o seguinte erro: `Playground execution failed`:

error: <EXPR>:7:12: error: variable 'bucketList' passed by reference before being initialized. O que esse erro significa? Resumidamente, ele diz que você está tentando adicionar uma instância a sua bucketList antes de ela ser preparada de maneira adequada.

Para consertar esse erro, altere sua declaração de bucketList para inicializar o array na mesma linha.

```
import Cocoa  
var bucketList: [String] = ["Climb Mt. Everest"]
```

O sinal = é o operador de atribuição, que usamos juntamente com a sintaxe literal do Array ["Climb Mt. Everest"]. Uma literal de Array é uma sintaxe abreviada que inicializa um array para você com quaisquer instâncias incluídas nela. Nesse caso, você inicializou a bucketList com o item "escalar o Monte Everest".

Como com outros tipos, você pode criar uma instância de Array tirando proveito das capacidades de inferência de tipo da linguagem Swift. Remova a declaração de tipo do código acima para usar a inferência de tipo.

```
import Cocoa  
var bucketList: [String] = ["Climb Mt. Everest"]
```

Note que nada mudou na verdade. Sua lista ainda contém o mesmo item e aceitará somente instâncias do tipo String no futuro. Se você tentasse adicionar um inteiro a esse array, talvez usando `append()` que vimos acima, o seguinte erro ocorreria: Type 'String' does not conform to protocol 'IntegerLiteralConvertible'. Em outras palavras, esse erro indica que você não pode adicionar uma instância de Int a seu array porque ele está esperando instâncias da String.

Agora que você sabe como criar e inicializar um array, é hora de aprender como acessar e modificar os elementos de seu array.

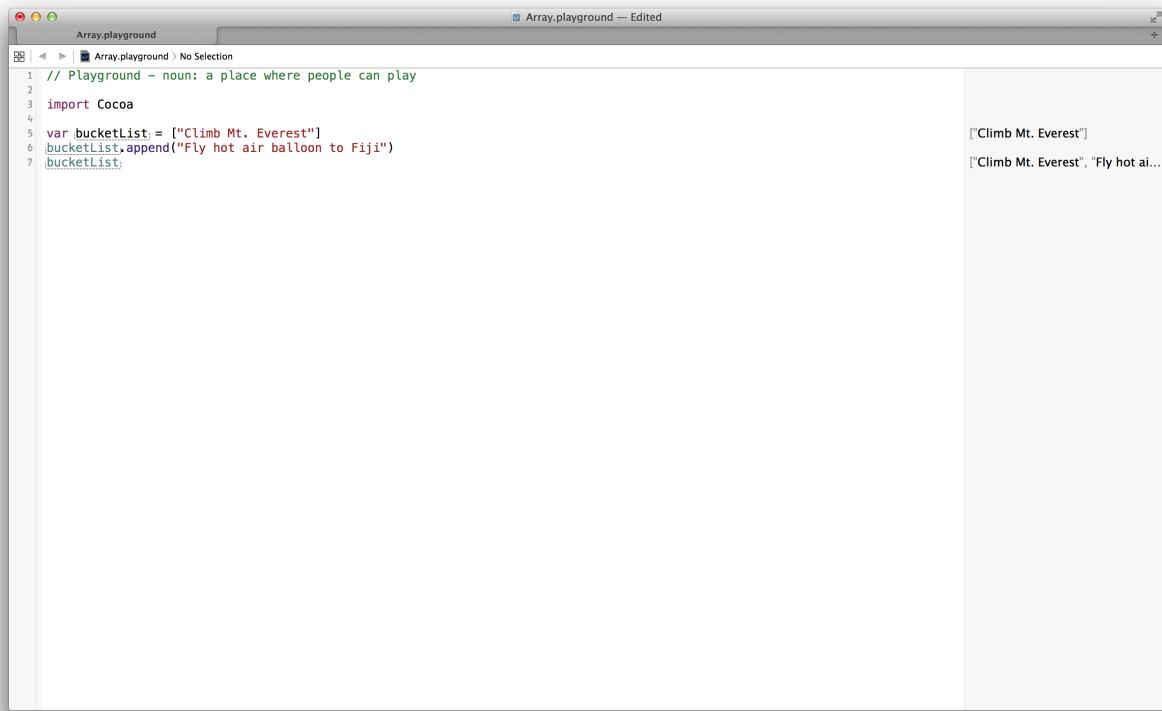
Acesso e modificação de arrays

Então, você tem uma "bucket list"? Ótimo! Infelizmente, você ainda não tem muitas ambições em relação a ela. Mas você é uma pessoa interessante, com grande zelo pela vida; portanto, vamos mudar isso. Para isso, precisamos saber como adicionar valores à bucketList. Na verdade, você já viu como se faz isso. Vá em frente e adicione o código a seguir em seu Playgroud.

```
import Cocoa  
var bucketList = ["Climb Mt. Everest"]  
bucketList.append("Fly hot air balloon to Fiji")  
bucketList
```

Seu Playgroud deve parecer com o seguinte exemplo.

Figure 10.1 Adição de valor a sua lista "bucket list"



Agora, sua lista possui dois itens futuros. Usamos **append()** para adicionar valor a `bucketList`. A função aceita um argumento de qualquer tipo que será o novo elemento adicionado a seu array. No exemplo acima, você adicionou "`Fly hot air balloon to Fiji`" como sua nova ambição da lista. Lembre-se, porém, de que seu array só aceita instâncias do tipo `String`.

Adicione outras aventuras futuras em sua lista usando a função **append()**.

```

import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList

```

Agora você tem seis itens na `bucketList`. Mas e se você mudar de ideia? Ou, pensando positivamente, e se você na verdade realizar um desses itens da lista? Vamos imaginar que você ficou doente no sábado e teve muito tempo livre. Com isso, você passou o dia assistindo à trilogia *O Senhor dos Anéis*. Remova esse item.

```

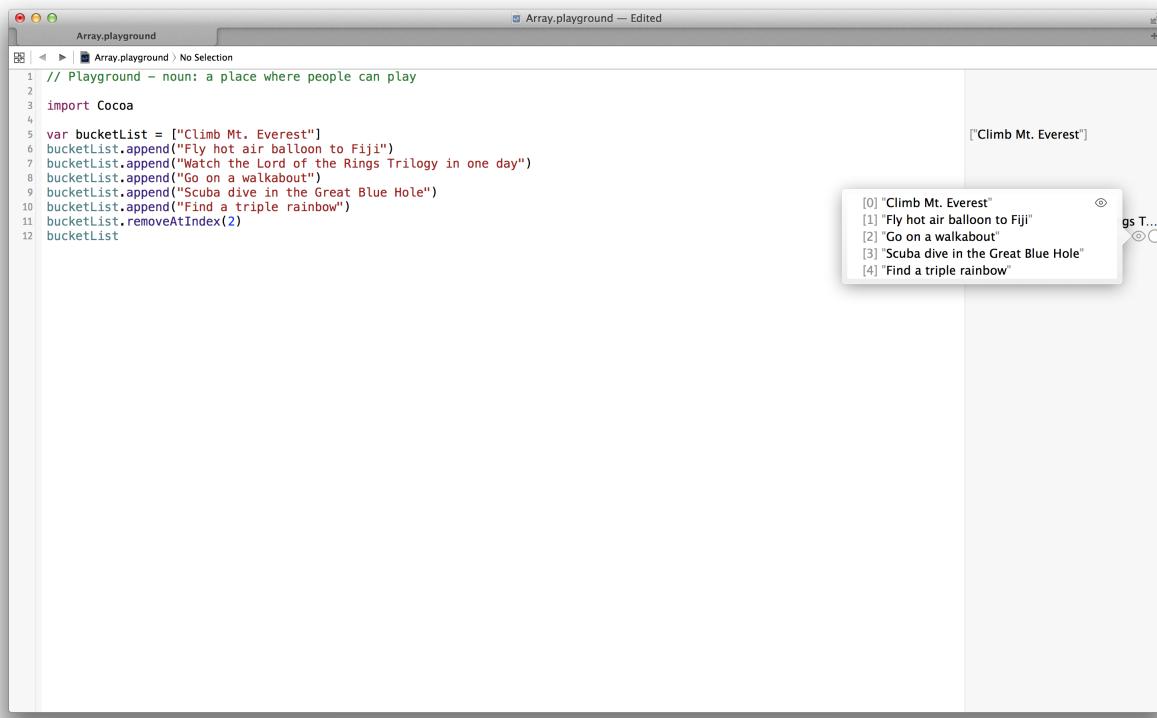
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.removeAtIndex(2)
bucketList

```

Para revelar o conteúdo da `bucketList`, destaque a linha 12 na barra lateral de resultados e clique no botão parecido com um olho -- ele é chamado de "olhada rápida". Como podemos ver na figura abaixo, o item que antes estava no segundo índice foi removido. Também observe que os arrays são indexados a zero. Portanto, `bucketList[0]` é igual a "Climb Mt. Everest".

Figure 10.2 Remoção de um item de sua lista



Agora imagine que você está em uma festa e alguém menciona esse assunto de objetivos futuros. Após ouvir o que todos têm a dizer, você tem certeza de que sua lista é mais interessante. À medida que você começa a falar, o pessoal começa a fazer perguntas. "Se sua lista é tão interessante assim, por que você não nos diz quantas coisas você quer fazer?"

Sem problemas! É fácil descobrir o número de itens em um array. Os arrays são capazes de monitorar o número de itens que há neles por meio da propriedade `count`. Adicione o código a seguir em seu Playgroud para exibir o número de itens da sua lista no console.

```

import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.removeAtIndex(2)
bucketList
println(bucketList.count)

```

"Cinco itens," o pessoal se sobressalta, "você quer fazer muitas coisas mesmo!" Como a festa está chegando ao fim e todo mundo precisa ir para casa e repensar suas próprias listas, o pessoal começa a pedir para que você diga seus três itens mais importantes.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.removeAtIndex(2)
bucketList
println(bucketList.count)
println(bucketList[0...2])
```

A sintaxe entre colchetes ([1...3]) é o que chamamos de subíndice. O subíndice permite que você acesse índices específicos em nosso array. Observe que você exibiu os três primeiros itens no console. Você poderia ter registrado apenas o último item por meio deste código: `println(bucketList[3])`.

O subíndice é um recurso poderoso. Digamos que, durante a conversa, alguém perguntasse: "Onde você quer fazer a excursão a pé?" A pergunta fez com que você percebesse que o terceiro item precisava ser esclarecido. Afinal, você não quer fazer uma excursão a pé *em qualquer lugar*. De fato, você quer fazer a excursão a pé na Austrália. Você pode usar o subíndice para alterar um item em um índice específico (ou faixa de índices).

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.removeAtIndex(2)
bucketList
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList
```

Se você clicar no botão de olhada rápida, verá que o terceiro item agora detalha que você quer fazer a excursão a pé na Austrália. Como isso aconteceu? Você usou o operador de adição e atribuição `+=` para adicionar algum texto ao item no índice 2. Essa atribuição funcionou porque a instância no índice 2 é do mesmo tipo que a instância adicionada a ela; ou seja, "Go on a walkabout" e " in Australia" são ambas do tipo `String`.

Pensar sobre todas essas aventuras o deixou pensativo e agora você está com problemas para dormir. Já que ler geralmente ajuda a dormir, você começou a ler sobre escaladas no Monte Everest. Durante sua leitura, você descobriu que é muito perigoso escalar lá. Você pensa consigo mesmo, "Embora eu me considere bastante aventureiro, prefiro realizar a minha aventura com segurança." Você decide atualizar o item do topo de sua lista.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.removeAtIndex(2)
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList
```

Aí está! Assim está melhor. Agora o item do topo de sua lista é mais seguro, ainda que seja um tanto aventureiro. E, embora agora você esteja feliz com o conteúdo de sua lista, você não está tão feliz assim com o fato de ter digitado `bucketList.append()` cinco vezes. Você pensa consigo mesmo: "Deve haver uma forma melhor de fazer isso!"

E aí você se lembra de algo: "Eu sei usar loops! E se eu fizer um array de todos os itens da lista que quero acrescentar? Aí eu poderia fazer um loop no array e usar **append()** sempre que o loop iterar. Isso significa que eu só terei que digitar `bucketList.append()` uma vez!" É exatamente isso que você fará.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
var newItems = [
    "Fly hot air balloon to Fiji",
    "Watch the Lord of the Rings trilogy in one day",
    "Go on a walkabout",
    "Scuba dive in the Great Blue Hole",
    "Find a triple rainbow"
]

for item in newItems {
    bucketList.append(item)
}
bucketList.removeAtIndex(2)
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList
```

Seu novo código acima cria um array para os itens da lista que você gostaria de acrescentar chamado de `newItems`. Em seguida, você criou um loop para iterar em cada `item` no array e adicioná-los a sua `bucketList`. Como vimos anteriormente, você tirou proveito do fato de que pode usar a variável `item` no escopo local do loop para adicioná-lo ao array de sua lista. "Isso é muito bom," você pensa, "mas aposto que é possível fazer melhor." Na verdade, é possível.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
var newItems = [
    "Fly hot air balloon to Fiji",
    "Watch the Lord of the Rings trilogy in one day",
    "Go on a walkabout",
    "Scuba dive in the Great Blue Hole",
    "Find a triple rainbow"
]
for item in newItems {
    bucketList.append(item)
}
bucketList.extend(newItems)
bucketList
bucketList.removeAtIndex(2)
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList
```

A função `extend()` aceita um argumento que está em conformidade com o protocolo Sequence. Veremos mais sobre protocolos posteriormente neste livro, mas, por enquanto, entenda que os protocolos descrevem um tipo de contrato que define o comportamento de uma instância que está em conformidade com ele. Acontece que o tipo Array está em conformidade com a Sequence, o que significa que você pode passar uma instância do Array para o argumento de `extend()`. Então, quando você passa seu array `newItems` para a função `extend()`, ele adiciona cada item desse novo array a seu array `bucketList` já existente.

Você está prestes a ir dormir, feliz que refatorou seu código tornando-o mais conciso e ainda mantendo-o com a mesma expressividade, mas então vem um pensamento à mente como em um raio de inspiração. "Posso usar o operador de adição e atribuição!" E, de fato, você pode.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
var newItems = [
    "Fly hot air balloon to Fiji",
    "Watch the Lord of the Rings trilogy in one day",
    "Go on a walkabout",
    "Scuba dive in the Great Blue Hole",
    "Find a triple rainbow"
]
bucketList.extend(newItems)
bucketList += newItems
bucketList
bucketList.removeAtIndex(2)
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList
```

O `+=`, que chamamos de operador de adição e atribuição, é uma forma limpa e fácil de adicionar seu array de `newItems` de forma concisa a sua lista já existente.

Finalmente, digamos que você queira alterar o item número três. Embora uma excursão a pé na Austrália seja obviamente muito legal, você acha que, na verdade, seria melhor andar de tobogã no Alasca antes. Use a função `insert()` para adicionar um elemento a seu array em um índice específico.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
var newItems = [
    "Fly hot air balloon to Fiji",
    "Watch the Lord of the Rings trilogy in one day",
    "Go on a walkabout",
    "Scuba dive in the Great Blue Hole",
    "Find a triple rainbow"
]
bucketList.extend(newItems)
bucketList += newItems
bucketList
bucketList.removeAtIndex(2)
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList.insert("Toboggan across Alaska", atIndex: 2)
bucketList
```

A função `insert()` tem dois argumentos. O primeiro argumento aceita a instância que você gostaria de adicionar ao array. Lembre-se de que seu array aceita instâncias de `String`. O segundo argumento aceita o índice de onde você gostaria de adicionar o novo elemento no array.

Com sua lista totalmente formada, você se deita e sonha que está voando em um balão de ar quente até ilhas montanhosas.

Igualdade e identidade de array

Na manhã seguinte, você acorda e vai até a cafeteria mais próxima. Lá, você encontra com um amigo, Myron, com quem você falou na festa na noite anterior. Seu amigo ficou completamente inspirado por sua `bucketList` e quis modelar a própria lista com base na sua. Ele foi para casa após a festa e anotou todos os seus itens, mas agora quer

ter certeza de que está tudo correto. Após atualizar Myron com as alterações que você fez após a festa, é hora de comparar seus arrays de itens da lista para garantir que sejam iguais.

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
var newItems = [
    "Fly hot air balloon to Fiji",
    "Watch the Lord of the Rings trilogy in one day",
    "Go on a walkabout",
    "Scuba dive in the Great Blue Hole",
    "Find a triple rainbow"
]
bucketList.extend(newItems)
bucketList += newItems
bucketList
bucketList.removeAtIndex(2)
println(bucketList.count)
println(bucketList[1...3])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList.insert("Toboggan across Alaska", atIndex: 2)
bucketList

var myronsList = [
    "Climb Mt. Kilimanjaro",
    "Fly hot air balloon to Fiji",
    "Toboggan across Alaska",
    "Go on a walkabout in Australia",
    "Scuba dive in the Great Blue Hole",
    "Find a triple rainbow"
]

let equal = (bucketList == myronsList)
```

Seu novo código usa == para testar a igualdade entre duas instâncias. Como o conteúdo de ambos os arrays são iguais, equal é definido como true.

Arrays imutáveis

Após ler o exemplo acima, uma questão pode ter surgido em sua mente: "E se eu não quiser que meu array mude?" Você usa arrays imutáveis nesses casos. Veja como.

Imaginemos que você estivesse criando um aplicativo que permitisse que os usuários monitorassem seus almoços semana a semana. Entre muitas outras funções úteis, seu aplicativo permitirá que os usuários registrem o que comem e gerará relatórios posteriormente. Você decide colocar essas refeições em um array imutável ao gerar esses relatórios. Afinal, não faz sentido alterar os almoços da semana passada após já ter comido. Adicione o código a seguir em seu Playground.

```
var myronsList = [
    "Climb Mt. Kilimanjaro",
    "Fly hot air balloon to Fiji",
    "Toboggan across Alaska",
    "Go on a walkabout in Australia",
    "Scuba dive in the Great Blue Hole",
    "Find a triple rainbow"
]

let equal = (bucketList == myronsList)
let lunches = ["Cheeseburger",
    "Veggie Pizza",
    "Chicken Caesar Salad",
    "Black Bean Burrito",
    "Falafel wrap"
]
```

Você usou a palavra-chave `let` para criar uma array imutável. Se você tentar modificar o array de alguma maneira, o compilador emitirá um erro dizendo que você não pode mudar um array imutável. De fato, se você tentasse reatribuir um novo array a `lunches`, você também receberia um erro do compilador dizendo que não pode reatribuir uma instância a uma constante criada por meio da palavra-chave `let`.

Conclusão

Este capítulo apresentou o tipo `Array` da linguagem Swift. Ao longo do caminho, você aprendeu a usar `var` para criar um array mutável e a usar `let` para criar um array imutável. Além disso, você aprendeu que o poderoso sistema de inferência de tipo da linguagem Swift permite que você seja mais conciso e ao mesmo tempo tire proveito da segurança e da verificação de tipo dessa linguagem. Você também usou um `for-in` para iterar em um array para adicionar seus itens a outro array. Em seguida, você viu como modificar arrays mutáveis por meio da sintaxe de subíndice do tipo `Array` e um pacote de funções auxiliares (por exemplo, `append()`, `removeAtIndex()`, `insert()`, `extend()` e etc.). Finalmente, você verificou se há igualdade (por meio de `==`) entre dois arrays.

Desafio

Olhe o array abaixo. Use a documentação para encontrar uma função que dirá se o array possui ou não elementos em seu interior. Em seguida, use um loop para inverter a ordem dos elementos desse array. Registre os resultados no console. Finalmente, examine a documentação para ver se há um modo mais conveniente de realizar essa operação.

```
var toDoList = ["Take out garbage", "Pay bills", "Cross off finished items"]
```


11

Dicionários

No capítulo anterior, você se familiarizou com o tipo `Array` da linguagem Swift. O tipo `Array` é uma coleção útil para cenários em que a ordem dos elementos dentro da instância do array é importante. Mas a ordem nem sempre é importante. Talvez você simplesmente queira manter um conjunto de informações em um recipiente e, no futuro, recuperar essas informações conforme necessário. É para isso que servem os dicionários.

Um `Dictionary` é um tipo de coleção que organiza seu conteúdo por pares chave-valor. As chaves em um dicionário correspondem a valores. Uma chave opera de maneira similar ao comprovante que damos ao atendente da chapelaria de um teatro. Você fala com o atendente, dá a ele o seu comprovante e ele usa esse comprovante para achar seu casaco. Semelhantemente, você dá uma chave a uma instância do tipo `Dictionary` e ela retornará o valor associado a essa chave. O ponto principal é que você deve usar um dicionário quando quiser armazenar e recuperar informações com uma chave específica.

Neste capítulo, você vai:

- aprender a criar e inicializar um dicionário;
- executar loop em dicionários;
- acessar e modificar dicionários por meio de suas chaves.

Você também aprenderá mais sobre chaves e como elas funcionam, principalmente no que diz respeito à linguagem Swift. Por último, você verá como criar arrays com chaves e valores do seu dicionário.

Criação de um dicionário

A forma geral de um dicionário em Swift é semelhante a: `var dict: Dictionary<KeyType, ValueType>`. Esse código cria uma instância mutável do tipo `Dictionary` chamada de `dict`. As declarações dos tipos aceitos pelas chaves e pelos valores do dicionário se encontram dentro dos sinais de menor e maior (`<>`), conforme indicado por `KeyType` e `ValueType`. `dict` aceitará apenas instâncias de chaves e valores do tipo apropriado após essa declaração.

A única exigência para as chaves do tipo `Dictionary` da linguagem Swift é que elas devem ser *conversíveis em hash*. Ou seja, cada `KeyType` deve fornecer um mecanismo para permitir que o `Dictionary` garanta que determinada chave seja única. Os tipos básicos da linguagem Swift, como `String`, `Int`, `Float`, `Double` e `Bool` são todos *conversíveis em hash*.

Antes de começar a digitar o código, vamos dar uma olhada nas diferentes maneiras de obter uma instância de um `Dictionary`.

```
var dict1: Dictionary<String, Double> = [:]
var dict2 = Dictionary<String, Double>()
var dict3: [String:Double] = [:]
var dict4 = [String:Double]()
```

A listagem acima demonstra quatro maneiras de obter uma instância de dicionário. A instância de dicionário resultante é vazia em cada um desses casos; ou seja, a instância atualmente não possui chaves nem valores.

Cada uma produz o mesmo resultado: uma instância do tipo `Dictionary` totalmente inicializada com informações de tipo associadas a suas chaves e seus valores futuros. O `KeyType` é configurado para aceitar as chaves do tipo `String` e o `ValueType` é configurado para aceitar os valores do tipo `Double`.

Qual é a diferença entre usar `()` e `[:]` na sintaxe acima? Como cada linha de código cria e prepara uma instância do tipo `Dictionary`, todos são essencialmente iguais. O código `[:]` usa a sintaxe literal para criar uma instância vazia do tipo `Dictionary` que inferirá informações de tipo para suas chaves e valores. Por exemplo, `dict1` especifica o tipo `e`, então, é inicializado como sendo um dicionário vazio. A sintaxe `()` usa o inicializador padrão para o tipo `Dictionary`, que irá preparar uma instância de dicionário vazio. Você verá mais sobre inicializadores posteriormente neste livro.

Como antes, é útil tirar proveito das capacidades de inferência de tipo da linguagem Swift. A inferência de tipo deixa o código mais conciso, com a mesma expressividade. Assim, você continuará usando inferência de tipo neste capítulo.

Preenchimento de um dicionário

De início, crie um novo arquivo chamado de `Dictionary.playground`. Salve-o em um local fácil de achar. Adicione o código a seguir em seu Playground.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
```

No código acima, você criou um dicionário mutável usando a sintaxe literal de `Dictionary`. Essa instância particular conterá avaliações de filmes. As chaves são instâncias do tipo `String` e representam filmes individuais. Essas chaves correspondem a valores que são instâncias do tipo `Integer`, que representam avaliações individuais dos filmes na instância `Dictionary`.

Acesso e modificação de um dicionário

Agora que você possui um dicionário mutável, a pergunta é: como você trabalha com ele? Você vai querer ler e modificar o dicionário. Comece usando `count` para obter alguma informação sobre seu dicionário.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
```

`count` é uma propriedade de somente leitura no tipo `Dictionary` que monitora quantas instâncias são contidas na própria instância do dicionário. Discutiremos as propriedades em mais detalhes em outro capítulo, mas basta dizer que, neste momento, as propriedades são variáveis em um tipo que armazena ou computa alguns dados sobre o tipo em que você está interessado. Nesse caso, usamos `count` para registrar no console: `I have rated 3 movies..`

Agora vamos ler um valor do dicionário `movieRatings`.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
```

Acessamos valores de um dicionário fornecendo a chave associada ao valor que queremos recuperar. No exemplo acima, você forneceu a chave `"Donnie Darko"` para o dicionário de avaliações de filmes. `darkoRating` agora está configurada como igual à pontuação de 4.

Pressione a tecla `option` e clique na instância `darkoRating` - observe que o tipo é `Int?`, mas `movieRatings` tem o tipo `[String : Int]`. Por que a discrepância? O tipo `Dictionary` precisa de uma maneira de indicar que o valor solicitado não está presente. Por exemplo, você ainda não avaliou o filme `Coração Valente`, então: `let braveheartRating = movieRatings["Braveheart"]` resultaria em `braveheartRating` como `nil`.

Agora você modificará um valor no seu dicionário de avaliações de filmes. Imagine, por exemplo, que um usuário insira uma avaliação e, em seguida, reconsidere a avaliação original.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
```

Como podemos ver, o valor associado à chave "Dark City" agora é igual a 5.

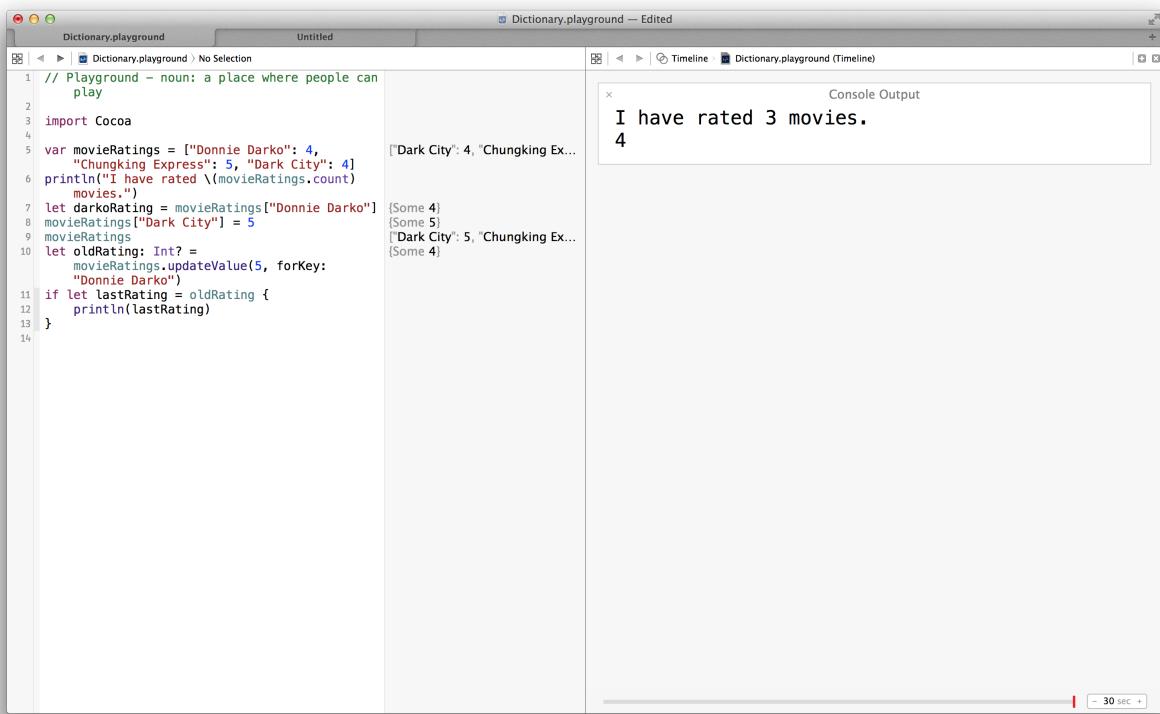
Existe outra maneira útil de atualizar valores associados a chaves de um dicionário. A função **updateValue()** aceita dois argumentos: `value: ValueType`, `forKey key: KeyType`. O primeiro argumento, `value`, aceita o novo valor. O segundo argumento, `forKey`, especifica a chave cujo valor você gostaria de mudar.

Essa função é útil porque informa o último valor correspondido pela chave. Há uma pequena restrição: **updateValue()** retorna uma opcional. Esse tipo de retorno é útil porque a chave pode não existir no dicionário. Como consequência, é útil atribuir o retorno da função **updateValue()** a uma opcional do tipo esperado e usar a ligação opcional de implementação para obter acesso ao valor antigo da chave. Veja abaixo.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
    println(lastRating)
}
```

A figura abaixo mostra que o valor antigo para a avaliação de "Donnie Darko" estava registrado no console.

Figure 11.1 Atualização de um valor



Adição de um valor

Agora que você viu como se atualiza um valor, é natural considerar como gostaria de adicionar um valor a um dicionário. Vamos supor que você tem um usuário que gostaria de adicionar uma nova avaliação de filme ao dicionário `movieRatings`.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \$(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
    println(lastRating)
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
```

Você pode adicionar um novo par chave-valor a seu dicionário fornecendo uma nova chave ao dicionário por meio desta sintaxe: `movieRatings["The Cabinet of Dr. Caligari"]`. Em seguida, use o *operador de atribuição* para associar um valor a essa nova chave. Nesse caso, você atribui ao filme *O Gabinete do Doutor Caligari* uma pontuação de 5.

Remoção de um valor

Naturalmente, após a adição de valores a um dicionário, o assunto seguinte é a remoção de um valor de um dicionário. Imagine, por exemplo, que o usuário achou que já tinha visto o filme *Dark City*, mas depois percebeu que não viu.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \$(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
    println(lastRating)
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
let removedRating: Int? = movieRatings.removeValueForKey("Dark City")
```

A função `removeValueForKey()` aceita uma chave como argumento e remove o par chave-valor que corresponde ao que você fornece. Como resultado, `movieRatings` não contém mais uma entrada para avaliação de *Dark City*. Além disso, essa função retorna o valor ao qual a chave foi associada caso a chave seja encontrada e removida com êxito. `removeValueForKey()` retorna uma opcional do tipo removido. No exemplo acima, `removedRating` é uma opcional `Int`.

Você também pode remover um par chave-valor configurando o valor de uma chave para ser igual a `nil`.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \$(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
    println(lastRating)
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
let removedRating: Int? = movieRatings.removeValueForKey("Dark City")
movieRatings["Dark City"] = nil
```

O resultado é essencialmente o mesmo, sendo que a principal diferença dessa estratégia de remover um par chave-valor de um dicionário é o não retorno do valor da chave removida.

Looping

Você pode usar um `for-in` para executar um loop em um dicionário. O tipo `Dictionary` da linguagem Swift fornece um mecanismo conveniente para executar loop em uma instância que permite acessar cada chave e valor da entrada em cada iteração no loop. Esse mecanismo divide cada elemento em suas partes constituintes fornecendo uma constante temporária que representa a chave e o valor.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
    println(lastRating)
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
movieRatings["Dark City"] = nil
for (key, value) in movieRatings {
    println("The movie \(key) was rated: \(value).")
}
```

Abra o *editor assistente* para visualizar o console. Você verá que cada filme com sua avaliação foi registrado no console. Observe como você usou *interpolação de strings* para combinar os valores de `key` e `value` em uma única string para registrar no console.

Um dicionário monitora suas chaves e valores por meio das propriedades de `keys` e `values`. Você pode usá-los para acessar informações específicas se não precisar de ambas as partes do par chave-valor. Veja o loop abaixo para ver um exemplo.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
    println(lastRating)
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
movieRatings["Dark City"] = nil
for (key, value) in movieRatings {
    println("The movie \(key) was rated: \(value).")
}
for movie in movieRatings.keys {
    println("User has rated \(movie).")
```

Esse novo loop vai iterar na chave de `movieRatings` e, portanto, vai registrar no console cada filme avaliado pelo usuário.

Dicionários imutáveis

A criação de um dicionário imutável funciona praticamente da mesma forma que no caso do array. Use a palavra-chave `let` para dizer ao compilador da linguagem Swift que você não quer que uma instância de `Dictionary` mude. O exemplo abaixo cria um dicionário imutável que lista os nomes das faixas de um álbum musical fictício (um curto... digamos que se trata de um EP) juntamente com a duração em segundos de cada faixa.

```
let album = ["Diet Roast Beef": 268,
            "Dubba Dubbs Stubs His Toe": 467,
            "Smokey's Carpet Cleaning Service": 187,
            "Track 4": 221]
```

Os nomes das faixas são as chaves e as durações em segundos são os valores. Se você tentar mudar esse dicionário, o compilador emitirá um erro e impedirá a mudança.

Tradução de um dicionário em array

Às vezes, é útil retirar informações de um dicionário e colocá-las em um array. Imagine, por exemplo, que parte do seu aplicativo não liga para o modo que certo usuário avaliou os filmes em sua fila. Em vez disso, essa parte do aplicativo só quer saber quais filmes foram avaliados pelo usuário. Por exemplo, se o usuário tiver avaliado um filme, essa parte do aplicativo pode presumir que o usuário já viu tal filme e, portanto, não recomendará ao usuário. Nesse caso, faz sentido enviar a lista de filmes que o usuário avaliou como instância do tipo `Array` para essa parte do aplicativo.

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
println("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating {
    println(lastRating)
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
movieRatings["Dark City"] = nil
for (key, value) in movieRatings {
    println("The movie \(key) was rated: \(value).")
}
for movie in movieRatings.keys {
    println("User has rated \(movie).")
}
let watchedMovies = Array(movieRatings.keys)
```

Você usou a sintaxe `Array()` para criar uma nova instância de `[String]`. Dentro dos parêntesis `()`, você entregou as chaves do dicionário. O resultado disso é o fato de que `watchedMovies` é uma instância de constante do tipo `Array` que representa todos os filmes que um usuário tem no dicionário `movieRatings`.

Conclusão

Este capítulo o apresentou aos dicionários. Os dicionários são um tipo de coleção útil para quando você quer associar uma chave a um valor específico. Você atribui uma chave a um dicionário e ela entrega o valor de volta a você. Como o tipo `Array`, os dicionários podem ser mutáveis e imutáveis. Se o dicionário for mutável, você pode adicionar, excluir e modificar valores nele. Se o dicionário for imutável, esses métodos não estarão disponíveis. Além disso, a linguagem Swift oferece um mecanismo de looping útil que ajuda a iterar em um dicionário e a retirar informações de chaves e valores a cada passagem pelo loop. Esse padrão é útil porque proporciona um controle mais detalhado das partes que constituem o dicionário.

Desafio

Não é incomum colocar instâncias do tipo `Array` dentro de um dicionário. Por exemplo, imagine que você tem um dicionário que representa um estado. Esse dicionário possui chaves que se referem a municípios. Cada chave deve corresponder a um array que contenha todos os CEPs dentro desse município. Resumindo, escolha um estado e atribua a ele três municípios de sua escolha como chaves. Em seguida, adicione cinco CEPs a cada array de CEP do município. Você pode inventá-los se não quiser adicionar CEPs reais. Finalmente, registre apenas os CEPs do dicionário. O resultado deve ficar parecido com:

```
Georgia has the following zip codes: [30306, 30307, 30308, 30309, 30310,
30311, 30312, 30313, 30314, 30315,
30301, 30302, 30303, 30304, 30305]
```

12

Funções

Uma função é um nome relacionado a um conjunto de código usado para realizar alguma tarefa específica. O nome da função é criado para descrever a tarefa que a função realiza. Você já usou algumas funções. Por exemplo, `println()` é uma função oferecida pela linguagem Swift.

Funções executam códigos. Eles podem definir argumentos que podem ser usados para transmitir dados para a função, ajudando-a a realizar seu trabalho. As funções também podem retornar algo após ter completado o trabalho. Você pode pensar na função como uma pequena e ávida máquina. Você liga a máquina e ela começa a se mexer e fazer seu trabalho. Você pode enviar dados a ela e, se for construída para isso, ela retornará um novo bloco de dados resultante do trabalho realizado.

As funções são uma parte extremamente importante da programação. De fato, um programa é basicamente uma coleção de funções relacionadas que se combinam para realizar algum trabalho que fornece algumas funcionalidades. Assim, temos muita coisa para ver neste capítulo; por isso, não tenha pressa e esteja confortável antes de prosseguir. Vamos dar uma olhada em alguns exemplos.

Uma função básica

Crie um novo Playground chamado de `Functions.playground`. Certifique-se de que o seu código corresponde ao que apresentamos abaixo.

```
import Cocoa

func printGreeting() {
    println("Hello, playground.")
}
printGreeting()
```

Define-se uma função com a palavra-chave `func`. Depois da palavra-chave `func`, declara-se o nome da função. No exemplo acima, você especificou uma função chamada de `printGreeting()`. Os parêntesis estão vazios porque essa função não aceita nenhum argumento. Veremos mais sobre isso em breve. A chave de abertura (`{`) indica o início da implementação da função. É aí que você escreve o código que descreve como a função realizará seu trabalho. Quando a função é chamada, o código dentro das chaves de abertura e fechamento é executado.

A função `printGreeting()` é bastante simples. Você possui uma linha de código que usa `println()` para registrar a string “Hello, playground.” no console. Contudo, você precisa chamar a função para de fato executar o código dentro dela. Para isso, você digitou `printGreeting()` na linha após a definição da função. Chamar uma função executou seu código no Playground e “Hello, playground.” foi registrado no console.

Agora que você escreveu e executou uma função simples, é hora de passar para variedades mais sofisticadas.

Parâmetros de função

As funções começam de fato a ganhar mais vida quando possuem argumentos. Usamos argumentos para atribuir algumas entradas a uma função. A função vai então aceitar os dados nos argumentos e usá-los para executar uma

tarefa específica ou produzir algum resultado. Mude sua função existente para que possa criar uma saudação mais pessoal. Para isso, você precisa adicionar um argumento.

```
import Cocoa

func printGreeting() {
    println("Hello, playground.")
}

printGreeting()
func printPersonalGreeting(name: String) {
    println("Hello \(name), welcome to your playground.")
}
printPersonalGreeting("Matt")
```

printPersonalGreeting() aceita um único argumento localizado dentro de parêntesis diretamente após o nome da função. Nesse caso, você adicionou um argumento chamado de `name` que espera ser uma instância do tipo `String`. Você especificou o tipo para `name` após `:`, que seguiu o nome do argumento, como quando você estava aprendendo sobre como especificar os tipos para variáveis e constantes.

Desde que o argumento seja uma instância da `String`, ele será interpolado na string que está registrada no console. Confira. Seu console deve dizer algo como: “Hello Matt, welcome to your playground.” Se você acabar transmitindo uma instância ao argumento que não seja do tipo `String`, o compilador emitirá um erro dizendo que o tipo transmitido é incorreto. Esse comportamento é muito útil. É útil saber como suas entradas serão quando você estiver escrevendo as implementações de suas funções.

As funções podem aceitar múltiplos argumentos. De fato, esse cenário é bastante comum. Crie uma nova função que faça alguns cálculos.

```
import Cocoa

func printPersonalGreeting(name: String) {
    println("Hello \(name), welcome to your playground.")
}
printPersonalGreeting("Matt")

func divisionDescription(num: Double, den: Double) {
    println("\(num) divided by \(den) equals \(num / den)")
}
divisionDescription(9, 3)
```

A função `divisionDescription()` descreve algumas divisões básicas construídas com base nas instâncias do tipo `Double` que você forneceu aos dois argumentos da função: `num` e `den`. Observe que você fez alguns cálculos dentro de `\()` da string exibida no console. Você agora deve ver “9 divided by 3 equals 3” registrado no console.

Nomes de parâmetros

As funções acima que possuem argumentos possuem nomes de parâmetro para seus argumentos. Por exemplo, a função `divisionDescription()` possui dois argumentos e dois nomes de parâmetro correspondentes: `num` e `den`. Entretanto, esses nomes são apenas locais dentro do corpo da função. Na verdade, você não vê esses parâmetros quando usa `divisionDescription()`. Eles estão lá para referência, mas desaparecem quando você atribui valores aos argumentos da função. Às vezes, contudo, é útil ter nomes dos parâmetros visíveis fora do corpo da sua função.

Parâmetros com nome podem deixar suas funções mais legíveis. Por exemplo, os argumentos de `divisionDescription`, na verdade, não possuem nomes de parâmetros informativos. Isso é mais ou menos bom dentro do corpo da função; a implementação da função deixa claro para que servem esses argumentos. Mas e se a função for usada em outro arquivo na base do código de seu aplicativo? Pode ser um pouco difícil inferir o que atribuir aos argumentos da função, o que deixará a função menos útil.

Altere os argumentos da `divisionDescription` para ter nomes de parâmetros explícitos.

```

import Cocoa

func printPersonalGreeting(name: String) {
    println("Hello \(name), welcome to your playground.")
}
printPersonalGreeting("Matt")

func divisionDescription(num: Double, den: Double) {
    println("\(num) divided by \(den) equals \(num / den)")
}
divisionDescription(9, 3)
func divisionDescription(numerator num: Double, denominator den: Double) {
    println("\(num) divided by \(den) equals \(num / den)")
}
divisionDescription(numerator: 9, denominator: 3)

```

Agora, `divisionDescription()` possui dois parâmetros com nomes explícitos. Esses nomes de parâmetros explícitos, `numerator` e `denominator`, são mais claros do que os mais sucintos `num` e `den`. Além disso, você deixa os nomes dos parâmetros visíveis quando usa a função `divisionDescription()`. Essa visibilidade para os nomes de parâmetro deixa a função mais legível.

A linguagem Swift oferece uma sintaxe abreviada para parâmetros externos de nomeação. A sintaxe de parâmetros externos abreviada economiza tempo de digitação. Coloque o símbolo # na frente do nome do parâmetro para usá-lo como nome de parâmetro externo. Refatore sua função para usar a sintaxe abreviada.

```

import Cocoa

func printPersonalGreeting(name: String) {
    println("Hello \(name), welcome to your playground.")
}
printPersonalGreeting("Matt")

func divisionDescription(numerator num: Double, denominator den: Double) {
    println("\(num) divided by \(den) equals \(num / den)")
}
divisionDescription(numerator: 9, denominator: 3)

func divisionDescription(#numerator: Double, #denominator: Double) {
    println("\(numerator) divided by \(denominator) equals \(numerator / denominator)")
}
divisionDescription(numerator: 9, denominator: 3)

```

Como podemos ver, a sintaxe abreviada faz com que você não precise digitar dois nomes de parâmetros para a função. Anteriormente, você digitou um parâmetro externo com nome explícito e um parâmetro local com nome. A sintaxe abreviada simplifica o processo de nomeação de parâmetros para você.

Parâmetros variadic

Um parâmetro variadic aceita zero ou mais valores de entrada para seu argumento. As funções podem ter um parâmetro variadic. Além disso, ele deve ser o parâmetro final na lista.

Para criar um parâmetro variadic, use três pontos após o tipo do parâmetro, por exemplo, `names: String....`. Os valores fornecidos ao argumento são disponibilizados dentro do corpo da função como array. No exemplo anterior, `names` está disponível dentro do corpo da função e possui o tipo `[String]`.

Refatore a função `printPersonalGreeting()` para ter um parâmetro variadic.

```
import Cocoa

func printPersonalGreeting(name: String) {
    println("Hello \(name), welcome to your playground.")
}
printPersonalGreeting("Matt")

func printPersonalGreetings(names: String...) {
    for name in names {
        println("Hello \(name), welcome to the playground.")
    }
}
printPersonalGreetings("Aaron", "Brian", "John", "Matt")

func divisionDescription(#numerator: Double, #denominator: Double) {
    println("\(numerator) divided by \(denominator) equals \(numerator / denominator)")
}
divisionDescription(numerator: 9, denominator: 3)
```

A `printPersonalGreeting()` foi substituída por uma versão no plural: `printPersonalGreetings()`. Verifique o console. Você verá que a função registrou uma saudação pessoal para cada nome fornecido ao parâmetro variadic.

Valores padrão de parâmetros

Os parâmetros da linguagem Swift podem aceitar valores padrão. Se uma função possuir parâmetros com valores padrão, coloque-os no final da lista de parâmetros da função. De maneira um tanto conveniente, se um parâmetro possuir valor padrão, você pode omitir o parâmetro ao chamar a função. Obviamente, omitir o parâmetro significa que a função assumirá que o valor padrão do parâmetro deve ser usado. Refatore a função `divisionDescription()` para ter um parâmetro novo com valor padrão.

```
import Cocoa

func printPersonalGreeting(name: String) {
    println("Hello \(name), welcome to your playground.")
}
printPersonalGreeting("Matt")

func printPersonalGreetings(names: String...) {
    for name in names {
        println("Hello \(name), welcome to the playground.")
    }
}
printPersonalGreetings("Aaron", "Brian", "John", "Matt")

func divisionDescription(#numerator: Double, #denominator: Double) {
    println("\(numerator) divided by \(denominator) equals \(numerator / denominator)")
}
divisionDescription(numerator: 9, denominator: 3)
func divisionDescription(#numerator: Double,
                       #denominator: Double, punctuation: String = ".") {
    println("\(numerator) divided by \(denominator) equals \(numerator / denominator)\(\punctuation)")
}
divisionDescription(numerator: 9, denominator: 3)
divisionDescription(numerator: 9, denominator: 3, punctuation: "!")
```

Observe o novo código: `punctuation: String = "."`. Você adicionou um novo parâmetro chamado de `punctuation`, adicionou o tipo esperado e atribuiu um valor padrão por meio da sintaxe `= ". "`. Assim, a string criada pela função finalizará com um ponto por padrão.

Você também pode usar esse novo parâmetro dentro do corpo da função. Também observe que atribuir ao parâmetro um valor padrão automaticamente criou um parâmetro externo com nome explícito. Como sugerido pelo trecho do código acima, você pode ignorar o parâmetro final e usar o padrão do valor dado pela definição da função.

Você também pode substituir esse valor padrão por um novo sinal de pontuação. A primeira chamada da função `divisionDescription()` registrará a descrição com um ponto e a segunda registrará a descrição com um ponto de exclamação.

Parâmetros in-out

Às vezes, há um motivo para uma função modificar o valor de um parâmetro. Os parâmetros in-out são úteis se você quiser que o impacto de uma função sobre uma variável vá além do corpo da função. Existem duas restrições: 1) os parâmetros in-out não podem ter valores padrão e 2) parâmetros variadic não podem ser marcados com `inout`.

Suponha que você tenha uma função que aceitará uma mensagem de erro como argumento e adicionará outras informações com base em certas condições. Considere o exemplo abaixo.

```
var error = "The request failed:"
func appendErrorString(#errorCode: Int, inout #errorString: String) {
    if errorCode == 400 {
        errorString += " bad request."
    }
}
appendErrorString(errorCode: 400, errorString: &error)
```

A função `appendErrorString` tem dois argumentos. O primeiro é o código de erro com o qual a função comparará e o segundo é um parâmetro `inout` chamado de `errorString`. Ambos os parâmetros usam a sintaxe abreviada e cada um espera uma instância de `String`. Como podemos ver no exemplo acima, um parâmetro in-out é indicado pela sintaxe `inout` na lista de parâmetros.

Quando você usa a função, a variável transmitida ao argumento para o parâmetro `inout` é precedida pelo símbolo `&`. Isso é feito para indicar que a variável será modificada pela função. No exemplo acima, a `errorString` será modificada para: “The request failed: bad request.”

Os parâmetros in-out não são iguais a uma função que retorna um valor. Se você quiser que sua função produza algo para ser usado posteriormente, há uma maneira mais elegante de lidar com essa situação.

Retorno de uma função

As funções podem oferecer informações após terminarem de executar o código dentro de sua implementação. Essas informações são chamadas de retorno da função. Muitas vezes, você quer escrever uma função que faça algum trabalho e retorne alguns dados a você. Por exemplo, refatorar `divisionDescription` para retornar uma instância do tipo `String`.

```
import Cocoa

func printPersonalGreetings(names: String...) {
    for name in names {
        println("Hello \(name), welcome to the playground.")
    }
}
printPersonalGreetings("Aaron", "Brian", "John", "Matt")

func divisionDescription(#numerator: Double,
                      #denominator: Double, punctuation: String = ".") {
    println("\(numerator) divided by \(denominator) equals
           \(numerator / denominator)(punctuation)")
}
divisionDescription(numerator: 9, denominator: 3)
divisionDescription(numerator: 9, denominator: 3, punctuation: "!")
func divisionDescription(#numerator: Double,
                      #denominator: Double, punctuation: String = ".") -> String {
    return "\(numerator) divided by \(denominator) equals
           \(numerator / denominator)(punctuation)"
}
println(divisionDescription(numerator: 9, denominator: 5, punctuation: "!"))
```

O comportamento dessa nova função é muito similar à implementação anterior, com uma característica importante: essa nova implementação retorna um valor. Esse valor retornado é indicado pela sintaxe `-> String`. A sintaxe `->` indica que a função retorna um valor imediatamente. Sempre que vir essa sintaxe, você pode esperar que a função retornará algum valor. Como podemos supor, as funções da linguagem Swift especificam o tipo da instância que retornam após a sintaxe `->`. Como você quer registrar uma string no console, sua função retornará uma instância do tipo `String`. Por último, você chama essa função em uma chamada a `println()` para registrar a instância de string no console.

As definições de função da linguagem Swift podem ser aninhadas. As funções aninhadas são declaradas e implementadas dentro da definição de outra função. Esse recurso é útil quando você quer que uma função realize algum trabalho, mas não é importante que ela tenha vida própria fora da função que a delimita. Nesse cenário, a função aninhada não estará disponível fora de sua função mais geral. Veja o exemplo abaixo.

```
import Cocoa

func printPersonalGreetings(names: String...) {
    for name in names {
        println("Hello \(name), welcome to the playground.")
    }
}
printPersonalGreetings("Aaron", "Brian", "John", "Matt")

func divisionDescription(#numerator: Double,
                      #denominator: Double, punctuation: String = ".") -> String {
    return "\((numerator) divided by \((denominator)) equals
           \((numerator / denominator))\((punctuation))"
}
println(divisionDescription(numerator: 9, denominator: 5, punctuation: "!"))

func areaOfTriangle(base: Double, height: Double) -> Double {
    let numerator = base * height
    func divide() -> Double {
        return numerator / 2
    }
    return divide()
}
areaOfTriangle(3, 5)
```

A função `areaOfTriangle()` recebe dois argumentos do tipo `Double`: `base` e `height`. `areaOfTriangle()` também retorna um `Double`. Dentro da implementação dessa função, você declarou e implementou outra função chamada de `divide()`. Essa função também não aceita argumentos e retorna um duplo. Por último, a função `areaOfTriangle()` chama a `divide()` e retorna o resultado dessa função.

Escopo

A função `divide()` acima utiliza uma constante chamada `numerator`. Por que isso funciona? Essa constante é definida dentro do escopo delimitador da função `divide()`. Qualquer coisa escrita entre as chaves (`{}`) de uma função é considerada delimitada pelo escopo dessa função. O escopo de uma função descreve a visibilidade que uma instância ou função terá. Ele pode ser comparado a um tipo de horizonte. Qualquer coisa definida dentro do escopo de uma função estará visível para essa função; qualquer coisa que não estiver além do campo de visão da função. `numerator` é definido dentro do escopo delimitador da função `divide()`. Assim, `numerator` está visível para a função `divide()` porque compartilham o mesmo escopo delimitador. Como distinção, a função `divide()` é definida dentro do escopo da função `areaOfTriangle()` e não estará visível fora dele. O compilador emitirá um erro se você tentar chamar a função `divide()` fora do escopo da função `areaOfTriangle()`.

Múltiplos retornos

As funções podem retornar mais de um valor. A linguagem Swift usa um tipo de dados chamado de *tupla* para isso, o qual você aprendeu no Chapter 6. Para entender melhor como usar tuplas, você vai criar uma função que aceita um array de inteiros e classifica-os em arrays de inteiros pares e ímpares.

```

func areaOfTriangle(base: Double, height: Double) -> Double {
    let numerator = base * height
    func divide() -> Double {
        return numerator / 2
    }
    return divide()
}
areaOfTriangle(3, 5)
func sortEvenOdd(numbers: [Int]) -> (evens: [Int], odds: [Int]) {
    var evens = [Int]()
    var odds = [Int]()
    for number in numbers {
        if number % 2 == 0 {
            evens.append(number)
        } else {
            odds.append(number)
        }
    }
    return (evens, odds)
}

```

Você primeiro declarou uma função chamada de `sortEvenOdd()`. Especificamos essa função para aceitar um array de inteiros como seu único argumento. A função retorna o que chamamos de *tupla com nome*. Sabemos que a tupla tem nome porque suas partes constituintes têm nome: `evens` será um array de inteiros e `odds` também será um array de inteiros.

Em seguida, dentro da implementação da função, você inicializou os arrays `evens` e `odds` para prepará-los para armazenar seus respectivos inteiros. Você então executou um loop no array de inteiros fornecido ao argumento da função, `numbers`. A cada iteração no loop, usamos o operador `%`, chamado de operador de divisão módulo. Diferentemente do dividendo normal, a divisão módulo retorna o resto da divisão. Essa operação dividirá o número atual no array por 2 e verificará se o resultado é igual a 0. Se o resultado for igual a 0, o inteiro será par (porque dividi-lo por 2 não resulta em nenhum resto) e você o adicionará ao array `evens`. Se o resultado não for igual a 0, a divisão resultou em um resto e o inteiro na iteração atual é ímpar. Sendo assim, você adicionará esse inteiro ao array `odds`.

Adicione o novo código abaixo para exercitar essa função.

```

func areaOfTriangle(base: Double, height: Double) -> Double {
    let numerator = base * height
    func divide() -> Double {
        return numerator / 2
    }
    return divide()
}
areaOfTriangle(3, 5)

func sortEvenOdd(numbers: [Int]) -> (evens: [Int], odds: [Int]) {
    var evens = [Int]()
    var odds = [Int]()
    for number in numbers {
        if number % 2 == 0 {
            evens.append(number)
        } else {
            odds.append(number)
        }
    }
    return (evens, odds)
}

let aBunchOfNumbers = [10,1,4,3,57,43,84,27,156,111]
let theSortedNumbers = sortEvenOdd(aBunchOfNumbers)
println("The even numbers are: \(theSortedNumbers.evens);")
    the odd numbers are: \(theSortedNumbers.odds)")

```

Primeiro, você criou uma instância do tipo `Array` para abrigar uma série de inteiros. Em seguida, você atribuiu esse array a `sortEvenOdd()` e atribuiu o valor de retorno a uma constante chamada de `theSortedNumbers`. Como o valor de retorno é especificado acima como (`evens: [Int]`, `odds: [Int]`), esse é o tipo que o compilador infere para a sua constante recém-criada. Finalmente, você registrou o resultado no console. Note que você usou a interpolação de strings juntamente com uma tupla. Você pode acessar os membros de uma tupla por nome se eles forem definidos. Por exemplo, `theSortedNumbers.evens` inseriu o conteúdo do array `evens` na string registrada no console. O resultado do seu console deve corresponder à Figure 12.1.

Figure 12.1 Usando uma tupla

```
x                                     Console Output
Hello Matt, welcome to your playground!
5.0 divided by 3.0 equals 1.66666666666667
The even numbers are: [10, 4, 84, 156]; the
odd numbers are: [1, 3, 57, 43, 27, 111]
```

Tipos de retorno de opcionais

Às vezes, você quer uma função para retornar uma opcional. Em outras palavras, você acha que uma função precisará retornar `nil` ocasionalmente, mas também terá um valor para retornar em outras vezes. Nesse caso, você usará um retorno de opcionais. Imagine, por exemplo, que você precisa de um método que olhe o nome inteiro de uma pessoa, retire e retorne o nome do meio da pessoa. Nem todos têm nome do meio. Assim, sua função precisará de um mecanismo que retorne o nome do meio da pessoa se houver e retorne `nil` caso contrário. Opcionais são perfeitas para esses casos. Veja mais abaixo.

```
func sortEvenOdd(numbers: [Int]) -> (evens: [Int], odds: [Int]) {
    var evens = [Int]()
    var odds = [Int]()
    for number in numbers {
        if number % 2 == 0 {
            evens.append(number)
        } else {
            odds.append(number)
        }
    }
    return (evens, odds)
}

let aBunchOfNumbers = [10,1,4,3,57,43,84,27,156,111]
let theSortedNumbers = sortEvenOdd(aBunchOfNumbers)
println("The even numbers are: \(theSortedNumbers.evens);
       the odd numbers are: \(theSortedNumbers.odds)")
func grabMiddleName(name: (String, String?, String)) -> String? {
    return name.1
}

let middleName = grabMiddleName(("Matt", nil, "Mathias"))
if let theName = middleName {
    println(theName)
```

No código acima, você criou uma função chamada de `grabMiddleName()`. Essa função é um pouco diferente quando comparada ao que vimos antes. Nesse exemplo, sua função aceita um argumento: uma tupla do tipo `(String,`

`String?, String)`. A tupla conterá três instâncias de `String`, uma para cada nome (primeiro nome, nome do meio e último nome). A instância listada para o nome do meio é declarada como tipo opcional porque uma pessoa pode não ter um nome do meio. Da mesma forma, o tipo de retorno da função também é opcional por esse mesmo motivo.

O argumento um da função `grabMiddleName()` é chamado de `name`. Acessamos esse argumento dentro da implementação da função usando o índice do nome que você quer retornar. Como a tupla é indexada a zero, usamos 1 para acessar o nome do meio fornecido no argumento. Tudo está bem aqui porque essa instância terá o tipo adequado: `String?`.

Você então pode usar `grabMiddleName()` fornecendo a ela um primeiro nome, um nome do meio e um último nome (sinta-se livre para mudar os nomes). Observe que, como você declarou que o componente do nome do meio da tupla será do tipo `String?`, você pode passar `nil` para essa parte da tupla. Você não pode fazer isso para a primeira e a última partes da tupla. Além disso, considere por que nada é registrado no console: o nome do meio para a pessoa escolhida é `nil`. Como tal, o boolean usado na ligação de opcionais não avalia como verdadeiro. Em outras palavras, o valor de `middleName` é `nil` após a função executar seu código.

Tente atribuir ao nome do meio uma instância de `String` válida e observe o resultado.

Tipos de função

Cada função com a qual você trabalhou neste capítulo tinha um tipo específico. Todas as funções têm. Os tipos de função são constituídos dos tipos de retorno e parâmetro da função. Considere a função `sortEvenOdd()` descrita acima. Essa função aceita um array de inteiros como argumento e retorna uma tupla com dois arrays de inteiros. Assim, o tipo de função para `sortEvenOdd()` se parece com: `([Int]) -> ([Int], [Int])`.

Os parâmetros da função são listados à esquerda entre parêntesis e o tipo de retorno vem após `->`. Você pode ler esse tipo de função como: “Uma função com um parâmetro que aceita um array de inteiros e retorna uma tupla com dois arrays contendo inteiros.”. Daqui em diante, é importante saber que uma função sem argumento e sem retorno tem o seguinte tipo: `() -> ()`.

Os tipos de função são particularmente úteis porque você pode atribuí-los a variáveis. Esse recurso será especialmente útil no próximo capítulo quando você vir que pode usar as funções nos argumentos e retornos de outras funções. Por enquanto, vamos dar uma olhada em como você pode atribuir um tipo de função a uma constante.

```
let evenOddFunction: ([Int]) -> ([Int], [Int]) = sortEvenOdd
```

O exemplo acima cria uma constante chamada de `evenOddFunction` cujo valor é o tipo de função da função `sortEvenOdd()`. Muito legal, né? Agora você pode passar essa constante por aí como qualquer outra. Você pode até mesmo usar essa constante para chamar a função: `evenOddFunction([1,2,3])` classificará os números no array fornecido ao único argumento da função em uma tupla de dois arrays - um para inteiros pares e um para inteiros ímpares.

Conclusão

Você realizou muitas coisas neste capítulo. Vimos muito material aqui e pode ser necessário revisá-lo. Como de costume, sugerimos que você digite todos os códigos mostrados. De fato, tente estender e aplicar os exemplos a outros casos; tente também quebrar os exemplos e consertá-los. Se você ainda estiver um pouco confuso sobre as funções, não se preocupe. Elas também são um grande foco do próximo capítulo.

Desafio

Escreva uma função chamada de `beanSifter` que aceite uma lista de compras (como array de strings) e "peneire" o feijão das outras compras. Essa função deve aceitar um argumento com parâmetro externo chamado de `groceryList` e deve retornar uma tupla com nome do tipo `(beans: [String], otherGroceries: [String])`. Aqui está um exemplo de como você deve ser capaz de chamar sua função e qual deve ser o resultado:

```
let result = beanSifter(grocerList: ["green bean",
    "milk",
    "black bean",
    "pinto bean",
    "apples"])

result.beans == ["green bean", "black bean", "pinto bean"]
result.otherGroceries == ["milk", "apples"]
```

DICA: Pode ser necessário utilizar uma função no tipo `String` chamada de `hasSuffix()`.

13

Fechamentos

É hora de revelar um segredo. Você já sabe o que são fechamentos; na verdade, você acabou de aprender um bocado sobre eles no último capítulo. Veja, na verdade, as funções são apenas um caso especial de fechamento. Ou seja, funções *são* fechamentos. E, como as funções, os fechamentos são pacotes discretos de funcionalidades que podem ser usados em seu aplicativo para realizar tarefas específicas. Como veremos neste capítulo, você pode passar fechamentos para argumentos de função e até mesmo retornar fechamentos de outras funções.

No capítulo anterior, você trabalhou principalmente com funções globais e aninhadas. Em ambos os casos, as funções criadas tinham nome. É isso que queremos dizer quando mencionamos que funções são um caso especial de fechamentos: funções podem ser consideradas um fechamento com nome.

Os fechamentos também são diferentes das funções por possuírem uma sintaxe compacta e leve. Como veremos neste capítulo, os fechamentos permitem que você escreva uma construção “tipo função” sem ter que se preocupar com nome e com uma declaração de função completa. Isso facilita a passagem de fechamentos por argumentos e retornos de função.

Vamos começar. Se ainda não tiver feito, crie um novo Playground chamado de `Closures.playground` e salve-o em um local conveniente.

Sintaxe de fechamento

Imaginemos que você é um organizador de comunidade gerenciando uma série de organizações. Você quer monitorar quantos voluntários existem para cada organização e criou uma instância do `Array` para essa tarefa. Deixe seu código igual ao exemplo abaixo.

```
import Cocoa  
var volunteerCounts = [1,3,40,32,2,53,77,13]
```

Na prática, você acabou de inserir o número de voluntários para cada organização conforme foram informados. Isso significa que o array está completamente desorganizado. Seria melhor se seu array de voluntários fosse classificado do menor para o maior. A boa notícia: A linguagem Swift possui uma função chamada de `sorted()` que permite que você especifique como uma instância de `Array` será organizada.

`sorted()` aceita dois argumentos. O primeiro argumento aceita a instância do tipo `Array` que você gostaria de classificar e o segundo argumento aceita um fechamento que descreve como a classificação deve ser feita. O próprio fechamento aceita dois argumentos que devem ser do mesmo tipo. Esses argumentos são comparados para produzir um `Bool` que representa se a instância no primeiro argumento deve ser classificada ou não antes da instância no segundo argumento. Use `<` se você quiser que o argumento `um` seja classificado antes do argumento `dois`; use `>` se quiser que o argumento `dois` venha antes do argumento `um`.

Como seu array de voluntários por organização está cheio de inteiros, o tipo de função para `sorted()` será assim no seu código: `([Int], (Int, Int) -> Bool) -> [Int]`. Em palavras, podemos ler o tipo de função da seguinte maneira. “`Sorted` é uma função que aceita dois argumentos: 1) um array de Inteiros e 2) um fechamento que aceita

dois inteiros para comparar e retorna um valor boolean especificando qual inteiro deve vir antes. Sorted retorna um novo array de inteiros que foram ordenados de acordo com a maneira como o fechamento teve de fazer a organização.” Isso significa que você pode passar uma função para o segundo argumento da função **sorted()**.

Agora, quais são os tipos de função e como usá-las já devem estar um pouco mais claros. Adicione o código a seguir para classificar seu array.

```
import Cocoa  
  
var volunteerCounts = [1,3,40,32,2,53,77,13]  
  
func sortAscending(i: Int, j: Int) -> Bool {  
    return i < j  
}  
let volunteersSorted = sorted(volunteerCounts, sortAscending)
```

Primeiro, você criou uma função chamada de **sortAscending()** que tem o tipo exigido. Ela compara dois inteiros e retorna um Bool que indica se **Int i** deve ser colocado ou não antes de **Int j**. Por exemplo, **sortAscending()** retornará **true** se **i** tiver que ser colocado antes de **j**. Lembre-se: como essa função global é um simples fechamento com nome, você pode fornecer essa função como o valor do segundo argumento na função **sorted()**.

Depois disso, você chamou **sorted()** e passou essa função em seu array de contagens de voluntários para o primeiro argumento e a função **sortAscending()** para o segundo argumento. Como **sortAscending()** retorna um novo array, atribui-se esse resultado a um novo array de constante chamado de **volunteersSorted**. Essa instância servirá de novo registro do número de voluntários das organizações classificados pelo número de voluntários.

Observe a barra lateral de resultados do seu Playground. Você verá que os valores dentro de **volunteersSorted** são classificados do menor para o maior.

Figure 13.1 Classificação de contagens de voluntários

The screenshot shows an Xcode playground window titled "Closures.playground — Edited". The code in the playground is:

```
// Playground – noun: a place where people can play
import UIKit
var volunteerCounts = [1,3,40,32,2,53,77,13]
func sortAscending(i: Int, j: Int) -> Bool {
    return i < j
}
let volunteersSorted = sorted(volunteerCounts, sortAscending)
```

The output of the sorted function is displayed in a callout box:

- [0] 1
- [1] 2
- [2] 3
- [3] 13
- [4] 32
- [5] 40
- [6] 53
- [7] 77

Sintaxe de expressão de fechamento

A sintaxe de fechamento segue esta forma geral:

```
{(parameters) -> return type in
    // code
}
```

Escrevemos uma expressão de fechamento entre chaves ({}). Os parâmetros do fechamento são listados entre parêntesis imediatamente após a chave de abertura. Um tipo de retorno do fechamento vem depois dos parâmetros e usa a sintaxe comum. A palavra-chave `in` é usada para separar os parâmetros do fechamento e o tipo de retorno das instruções dentro do corpo.

Refatore o exemplo acima para usar uma expressão de fechamento. Você criará um fechamento em linha em vez de definir uma função separada fora da função `sorted()`.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

func sortAscending(i: Int, j: Int) -> Bool {
    return i < j
}

let volunteersSorted = sorted(volunteerCounts, {
    (i: Int, j: Int) -> Bool in
    return i < j
})
```

Esse código é um pouco mais claro e elegante do que a primeira passagem. Em vez de fornecer uma função definida em outro lugar do Playground, você implementou um fechamento em linha no segundo argumento da função `sorted()`. Você definiu os parâmetros e o tipo dos parâmetros (`Int`) dentro dos parêntesis do fechamento e também especificou o tipo de retorno. Em seguida, você implementou o corpo do fechamento fornecendo o teste lógico (isto é, `i` é menor que `j`?) que informará o retorno do fechamento. O resultado é o mesmo de antes: o array classificado é atribuído a `volunteersSorted`.

O exemplo acima é ótimo, mas é um pouco prolixo. Os fechamentos podem tirar proveito do sistema de inferência de tipo da linguagem Swift como esperado. Limpe o fechamento removendo as informações de tipo.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = sorted(volunteerCounts, {
    (i: Int, j: Int) -> Bool in
    return i < j
})

let volunteersSorted = sorted(volunteerCounts, { i, j in i < j })
```

Existem três novos acontecimentos. Primeiro, as informações de tipo foram removidas tanto para os parâmetros como para o retorno. Em seguida, a palavra-chave `return` foi removida. Essa remoção é possível porque o fechamento só tem uma expressão (anteriormente, era: retorno `i < j`). Se fosse necessário trabalhar mais, você não conseguiria removê-la. Além disso, remover `return` funciona porque a operação (`i` é menor que `j`?) retorna um valor Boolean. Por último, você moveu toda a expressão de fechamento para apenas uma linha. O resultado é exatamente o mesmo.

O seu fechamento está ficando bastante compacto, mas pode ficar ainda melhor. A linguagem Swift fornece nomes de argumento abreviados aos quais você pode se referir em expressões de fechamento em linha. Esses nomes de argumento abreviados se comportam de maneira semelhante a quando você declarou os argumentos explicitamente no exemplo acima: eles têm os mesmo tipos e valores. As capacidades de inferência de tipo do compilador ajudam a conhecer o número e os tipos de argumentos aceitos por seu fechamento, o que significa que não é necessário nomeá-los.

Por exemplo, o compilador sabe que `sorted()` aceita um fechamento no segundo argumento. Esse próprio fechamento aceita dois parâmetros do mesmo tipo dentro do array passado para o primeiro argumento da função `sorted()`. Como o fechamento em questão possui dois argumentos, seus valores são comparados para determinar a ordem; você pode referir-se aos valores de ambos da seguinte maneira: `$0` para o primeiro e `$1` para o segundo. Ajuste seu código para tirar proveito da sintaxe abreviada.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = sorted(volunteerCounts, { i, j in i < j })

let volunteersSorted = sorted(volunteerCounts, { $0 < $1 })
```

Como a sua expressão de fechamento em linha utiliza a sintaxe de argumento abreviada, você não precisa declarar os parâmetros explicitamente como você fez para `i` e `j`. O compilador sabe que os valores nos argumentos do

fechamento são do tipo correto e sabe o que inferir com base no operador `<`. Além disso, se o seu fechamento tiver mais de dois argumentos, você pode referir-se a eles como: `$2`, `$3` e assim por diante.

Antes que pense que esse fechamento não poderia ficar menor, espere, há mais! Se um fechamento for passado ao argumento final de uma função, ele pode ser escrito em linha fora de e após os parêntesis da função. Veja abaixo.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = sorted(volunteerCounts, { $0 < $1 })

let volunteersSorted = sorted(volunteerCounts) { $0 < $1 }
```

A sintaxe de fechamento final é especialmente útil se o corpo do fechamento for longo. No código acima, um fechamento final apenas diminui a quantidade de digitação, e muito pouco. Não há muito a ganhar aqui ao usar um fechamento final.

Na verdade, “conclusão é a alma do negócio.” No fim, o código acima funciona bem nessa forma sucinta. Afinal, você está preocupado apenas com uma coisa (um inteiro é menor que outro?) e isso pode ser facilmente expresso de maneira sucinta. Não se empolgue com esses truques. É importante ter certeza de que seu código será legível e de fácil manutenção.

Funções como tipos de retorno

Na linguagem Swift, as funções são o que chamamos de objeto de primeira classe. Isso significa que as funções podem retornar outras funções como tipo de retorno. Lembre-se da cidade de Knowhere com a qual você trabalhou no início do livro. É hora de criar uma função para melhorar sua cidade. Você vai construir algumas estradas.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = sorted(volunteerCounts) { $0 < $1 }

func makeTownGrand() -> (Int, Int) -> Int {
    func buildRoads(lightsToAdd: Int, toLights: Int) -> Int {
        return toLights + lightsToAdd
    }
    return buildRoads
}
```

A função `makeTownGrand()` não aceita nenhum argumento. É como se fosse seu avô. Contudo, ela retorna uma função. Essa função aceita dois argumentos, ambos inteiros, e retorna um inteiro. Dentro do corpo da função `makeTownGrand()`, você implementa a função que retornar. Em termos de detalhes de implementação, essa função retornada é uma função aninhada chamada de `buildRoads()`. Como podemos ver, seus argumentos e o tipo de retorno correspondem ao que foi declarado acima em `makeTownGrand()`.

Exercite sua nova função e construa algumas estradas. Adicione o código a seguir.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = sorted(volunteerCounts) { $0 < $1 }

func makeTownGrand() -> (Int, Int) -> Int {
    func buildRoads(lightsToAdd: Int, toLights: Int) -> Int {
        return toLights + lightsToAdd
    }
    return buildRoads
}
var stopLights = 4
let townPlan = makeTownGrand()
stopLights = townPlan(4, stopLights)
println("Town has \(stopLights) stop lights.")
```

Primeiro, configura-se uma variável chamada de `stopLights`. Essa instância é declarada como variável porque você vai construir algumas estradas que serão adicionadas ao número de semáforos da cidade. Em seguida, você criou uma constante chamada de `townPlan` que se refere à função criada pela função `makeTownGrand()`. Depois disso, você chamou essa função e passou a ela o número de semáforos a serem adicionados (o primeiro argumento) e o número atual de semáforos (o segundo argumento). O resultado dessa função, uma instância do tipo `Int`, é reatribuído à variável `stopLights`. Por último, você exibiu esse novo valor no console. Verifique seu console agora. Deverá aparecer “Town has 8 stop lights.”

Funções como argumentos

As funções podem servir de argumentos a outras funções. Lembre-se, por exemplo, de que você inicialmente atribuiu a `sorted()` a função `sortAscending()` como argumento no início do capítulo.

Imagine que sua cidade não pode construir estradas arbitrariamente. A praticidade sugere que sua cidade só pode construir estradas quando tiver um orçamento apropriado. Ajuste sua função `makeTownGrand()` anterior para aceitar um parâmetro de orçamento e um parâmetro de condição. O parâmetro de orçamento servirá de orçamento da sua cidade e o parâmetro de condição avaliará se esse orçamento é adequado ou não para construir estradas novas.

```
import Cocoa

var volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = sorted(volunteerCounts) { $0 < $1 }

func makeTownGrand() -> (Int, Int) -> Int {
    func buildRoads(lightsToAdd: Int, toLights: Int) -> Int {
        return toLights + lightsToAdd
    }
    return buildRoads
}

let townPlan = makeTownGrand()
stopLights = townPlan(4, stopLights)

func makeTownGrand(budget: Int, condition: Int -> Bool) -> ((Int, Int) -> Int)? {
    if condition(budget) {
        func buildRoads(lightsToAdd: Int, toLights: Int) -> Int {
            return toLights + lightsToAdd
        }
        return buildRoads
    } else {
        return nil
    }
}
func evaluateBudget(budget: Int) -> Bool {
    return budget > 10000
}
var stopLights = 4
if let townPlan = makeTownGrand(1000, evaluateBudget) {
    stopLights = townPlan(4, stopLights)
}
println("Town has \(stopLights) stop lights.")
```

Existem três mudanças importantes.

Primeiramente, a nova função `makeTownGrand()` aceita dois argumentos. O primeiro é uma instância de `Int` que representa o orçamento da cidade. O segundo é chamado de `condition` e aceita uma função. Essa função será usada para determinar se o orçamento da cidade é suficiente. Assim, aceitará um inteiro e retornará um Boolean. Se o orçamento de inteiros for alto o suficiente, essa função retornará `true`. Se o orçamento não for alto o suficiente, essa função retornará `false`.

Percebeu que a função `makeTownGrand()` tem um tipo de retorno diferente? O tipo de retorno agora é `((Int, Int) -> Int)?`. A implementação anterior de `makeTownGrand()` retornou uma função que aceitou dois inteiros e retornou

um inteiro. Nessa versão revisada, `makeTownGrand()` retorna a mesma coisa, mas de forma opcional. O motivo disso é simples: se a cidade tiver o orçamento apropriado, `buildRoads()` será criada e retornada. Se, por outro lado, o orçamento não for suficiente, `buildRoads()` não será criada e `nil` será retornado.

A implementação de `makeTownGrand()` executa a função passada para o parâmetro `condition`. Se for avaliado como `true`, `buildRoads()` é criada e retornada. Se `condition` avaliar como `false`, retorna-se `nil`.

Em seguida, cria-se a função apropriadamente chamada de `evaluateBudget()`. Essa função aceita um inteiro e retorna um Boolean. A implementação dessa função simplesmente avalia o inteiro para verificar se é superior ao limite mínimo arbitrário: 10000 unidades de alguma moeda.

Depois disso, você usou a ligação de opcionais para definir condicionalmente `townPlan`. Se o orçamento fornecido à função `makeTownGrand()` for suficientemente grande, `buildRoads()` será criada, retornada e atribuída a `townPlan`. Nesse caso, o número de `stopLights` da sua cidade será aumentado em 4. Contudo, se o orçamento não for grande o suficiente, `makeTownGrand()` retornará `nil`. Nesse caso, o número de semáforos da sua cidade não será aumentado.

Verifique seu console. Infelizmente, o orçamento da sua cidade não parece ser grande o suficiente. Um orçamento de 1000 certamente é menor que o requisito de 10000. Assim, `makeTownGrand()` retornou `nil` e `buildRoads()` nunca foi executado. Dessa forma, sua cidade não conseguirá construir nenhuma estrada.

Valores de captura de fechamentos

Os fechamentos e as funções podem monitorar as informações internas englobadas por uma variável definida em seu escopo delimitador. Por exemplo, imagine que a cidade de Knowhere está crescendo. Como o crescimento pode ser irregular, você decidiu criar uma função que permitisse atualizar os dados da população da cidade dependendo do crescimento recente. O planejador da cidade decidiu que faz mais sentido atualizar os dados do censo da cidade sempre que a população tiver um crescimento de 500 pessoas.

```
println("Town has \(stopLights) stop lights.")

func makeGrowthTracker(forGrowth growth: Int) -> () -> Int {
    var totalGrowth = 0
    func growthTracker() -> Int {
        totalGrowth += growth
        return totalGrowth
    }
    return growthTracker
}
var currentPopulation = 5422
let growBy500 = makeGrowthTracker(forGrowth: 500)
```

`makeGrowthTracker()` é uma função que constrói uma função `growthTracker()`. `makeGrowthTracker()` aceita um argumento, um inteiro que representa o crescimento a ser monitorado e retorna uma função que não aceita argumentos e retorna um inteiro. Esse inteiro é um total parcial do crescimento que está afetando sua cidade. A função `growthTracker()` captura o valor da variável `totalGrowth` de seu escopo delimitador. Depois de criar `growthTracker()`, a variável `totalGrowth` será aumentada pela quantidade especificada no argumento passado à função `makeGrowthTracker()`. Exercite e teste essa função chamando-a algumas vezes.

```
println("Town has \(stopLights) stop lights.")

func makeGrowthTracker(forGrowth growth: Int) -> () -> Int {
    var totalGrowth = 0
    func growthTracker() -> Int {
        totalGrowth += growth
        return totalGrowth
    }
    return growthTracker
}
var currentPopulation = 5422
let growBy500 = makeGrowthTracker(forGrowth: 500)
growBy500()
growBy500()
growBy500()
currentPopulation += growBy500() // currentPopulation is now 7422
```

Como podemos ver, `growBy500()` é chamada quatro vezes. Isso é traduzido em um crescimento de 2000 pessoas para sua cidade. Observe que as três primeiras chamadas a `growBy500()` não atribuem o resultado a nenhuma constante ou variável. Isso é bom porque a função está mantendo um total parcial interno do crescimento da sua cidade. Assim, tudo o que você precisa fazer para atualizar a população da sua cidade é atribuir o resultado da função à variável `currentPopulation` quando o planejador da cidade estiver pronto.

Fechamentos são tipos de referência

Fechamentos são tipos de referência. Ao atribuir uma função a uma constante ou variável, você está, na verdade, configurando essa constante ou variável para apontar para a função. Você não está criando uma cópia distinta dessa função. Uma consequência importante desse fato é que qualquer informação capturada pelo escopo da função será alterada se você chamar a função por meio da sua nova constante ou variável. Por exemplo, imagine que você criou uma nova constante e a definiu como igual a sua função `growBy500()`.

```
func makeGrowthTracker(forGrowth growth: Int) -> () -> Int {  
    var totalGrowth = 0  
    func growthTracker() -> Int {  
        totalGrowth += growth  
        return totalGrowth  
    }  
    return growthTracker  
}  
var currentPopulation = 5422  
let growBy500 = makeGrowthTracker(forGrowth: 500)  
growBy500()  
growBy500()  
growBy500()  
currentPopulation += growBy500() // currentPopulation is now 7422  
let anotherGrowBy500 = growBy500  
anotherGrowBy500() // totalGrowth now equal to 2500
```

`anotherGrowBy500` agora aponta para a mesma função apontada pela `growBy500`. A importância disso está no fato de que, se você chamar a função digitando `anotherGrowBy500()`, a variável `totalGrowth` será incrementada em 500. Sendo assim, o crescimento total da população de Knowhere aumenta até 2500 no código acima.

Como distinção, considere que uma cidade vizinha adorou a função do planejador da sua cidade. Essa cidade é um tanto maior e o prefeito está confiante que usar a função de crescimento da sua cidade ajudará a reduzir custos. Assim, a cidade precisará de sua própria função de monitoramento do crescimento porque o crescimento dessa cidade é bem maior do que o da sua pequena cidade. Crie outra população da cidade e use a função `makeGrowthTracker()` para criar outra função de monitoramento de crescimento para a cidade maior. Essa nova função de crescimento aumentará a população da cidade em 10000.

```
let anotherGrowBy500 = growBy500  
anotherGrowBy500() // totalGrowth now equal to 2500  
var someOtherPopulation = 4061981  
let growBy10000 = makeGrowthTracker(forGrowth: 10000)  
someOtherPopulation += growBy10000()  
currentPopulation
```

Agora você tem outra população sendo monitorada e tem uma nova função de monitoramento de crescimento chamada de `growBy10000()` para ajudá-lo a fazer isso. Como acima, você usa `growBy10000()` para aumentar a população da cidade: `someOtherPopulation += growBy10000()`. A população da cidade é aumentada para 4.071.981 após essa linha. Observe que a população da sua cidade não muda; ela continua tendo 7.422 pessoas. Isso acontece porque você usou a função `makeGrowthTracker()` para criar uma nova função de monitoramento de crescimento. Essa nova função de monitoramento de crescimento é separada e diferente de `growBy500()`.

Conclusão

Fechamentos e funções apresentam uma forte relação uns com os outros: as funções são fechamentos com nome. Ambos são construções poderosas, capazes de servir de argumentos e tipos de retorno para outras funções. Você

pode usar as funções e os fechamentos para construir outras funções ou fechamentos. Além disso, os fechamentos são extremamente úteis na simplificação do seu código, para torná-lo mais elegante e legível.

É importante se familiarizar com o uso dessas ferramentas. Particularmente, certifique-se de entender a gramática das funções e dos fechamentos. Essa habilidade o ajudará a ler o código dos outros mais rapidamente e será de grande ajuda ao escrever o seu próprio código. Fechamentos e funções são imprescindíveis para trabalhar de forma eficaz com a linguagem Swift.

Desafio

A linguagem Swift adota alguns padrões da programação funcional. Por exemplo, a linguagem Swift oferece diversas funções de ordem superior de amplo conhecimento dos programadores amantes da programação funcional: **filter**, **map** e **reduce**. Essas funções de ordem superior aceitam pelo menos uma função como entrada. Essa qualidade é o que as torna funções de ordem superior quanto à definição. Na verdade, você já trabalhou com funções de ordem superior neste capítulo.

Digamos que alguém perguntou a você quantos voluntários você está monitorando. Como você descobriria esse número? Olharia dentro do Playground e simplesmente adicionaria os números em cada índice no array? Cruzes! Você é um programador! Escreva algum código. Ou, quem sabe você escreveria um loop for-in para fazer a contagem por você. Eca! Isso requer muita digitação para uma tarefa tão simples. Acontece que **reduce** apresenta um modo ainda melhor. Dê uma olhada.

```
let totalVolunteers = volunteerCounts.reduce(0, combine: {  
    (i: Int, j: Int) -> Int in  
    return i + j  
})
```

Você pode usar **reduce** em tipos de coleção como arrays. A função disso é *reduzir* os valores na coleção até um único valor que é retornado da função. O primeiro argumento se refere a uma quantidade inicial (ou outro valor) que pode ser adicionada ao início. O segundo argumento é um fechamento que define como os valores dentro da coleção devem ser combinados. No cenário apresentado acima, tudo o que você quer é adicionar os inteiros em cada índice do array. Quando a função estiver concluída, `totalVolunteers` é definido como igual a 221.

Agora, o desafio: use o que aprendeu neste capítulo e limpe a implementação de **reduce()** apresentada acima. Na verdade, a implementação pode ser abreviada significativamente. Sua solução deve ser expressa em uma linha. Quando tiver acabado, vá em frente e dê uma olhada em outras funções de ordem superior e pratique.

Part IV

Enumerações, estruturas e classes

14

Enumerações

No decorrer deste livro até aqui, você usou todos os tipos embutidos fornecidos pela linguagem Swift, como `Int`, `String`, arrays e dicionários. Os dois próximos capítulos exibirão as capacidades fornecidas pela linguagem para criar seus próprios tipos. O foco deste capítulo são as enumerações, que permitem que você crie instâncias com autorização para ser um caso de uma lista predefinida. Se você tiver usado enumerações em outras línguas, você já estará familiarizado com boa parte deste capítulo, mas as enums da linguagem Swift também têm recursos avançados que as tornam únicas.

Enumerações básicas

Crie um novo Playground chamado de `Enumerations.playground`. Digite o código abaixo no playground para definir uma enumeração de possíveis alinhamentos de texto:

```
enum TextAlignement {
    case Left
    case Right
    case Center
}
```

Define-se uma enumeração com a palavra-chave `enum` seguida do nome da enumeração. A chave de abertura (`{`) abre o corpo da enum e deve conter pelo menos uma instrução de `case` que declara os possíveis valores para a enum. O nome da enumeração (`TextAlignment`, nesse caso) agora pode ser usado como tipo, da mesma maneira que `Int` ou `String` ou os diversos outros tipos usados até agora. Os tipos são escritos com a primeira letra em maiúsculo por convenção. Se múltiplas palavras forem necessárias, use o camel-casing; por exemplo, `UpperCamelCasedType`. As variáveis e as funções começam com a primeira letra em minúsculo e também usam camel-casing conforme necessário.

Como a enumeração declara um novo tipo, você pode criar instâncias desse tipo:

```
enum TextAlignement {
    case Left
    case Right
    case Center
}

var alignment: TextAlignement = TextAlignement.Left
```

O compilador ainda pode inferir o tipo para `alignment`, embora `TextAlignment` seja um tipo definido por você. Portanto, você pode omitir o tipo explícito da variável `alignment`:

```
var alignment: TextAlignement = TextAlignement.Left
var alignment = TextAlignement.Left
```

A habilidade do compilador de realizar inferência de tipo em enumerações não é limitada apenas a declarações de variáveis. Se tiver uma variável conhecida por ser um tipo enum particular, você pode omitir o tipo da instrução de caso ao atribuir um novo valor à variável:

```
var alignment = TextAlignement.Left
alignment = .Right
```

Observe que você teve de especificar o tipo e o valor de enum ao criar pela primeira vez a variável `alignment`, porque essa linha atribui a `alignment` tanto seu tipo como seu valor. Na próxima linha, você pode omitir o tipo e simplesmente reatribuir `alignment` para ser igual a um valor diferente dentro de seu tipo. Você também pode omitir o tipo de enum ao passar os valores a funções ou compará-los:

```
alignment = .Right

if alignment == .Right {
    println("we should right-align the text!")
}
```

Embora os valores de enum possam ser comparados em instruções `if`, normalmente usamos uma instrução `switch` para lidar com valores de enum. Insira o código abaixo que exibe o alinhamento de maneira humanamente legível:

```
var alignment = TextAlign.Alignment.Left
alignment = .Right

if alignment == .Right {
    println("we should right align the text!")
}
switch alignment {
case .Left:
    println("left aligned")

case .Right:
    println("right aligned")

case .Center:
    println("center aligned")
}
```

Lembre-se do Chapter 6 que dizia que as instruções de `switch` devem ser abrangentes. Nesse capítulo, cada instrução de `switch` escrita incluía uma cláusula `default`. Ao ativar os valores de enumeração, isso não é necessário: o compilador conhece todos os valores possíveis que podem ser verificados pela enumeração e, se você tiver incluído um caso para cada um, o `switch` será abrangente. Você pode incluir uma cláusula `default` ao ativar um tipo de enum:

```
switch alignment {
case .Left:
    println("left aligned")

case .Right:
    println("right aligned")

case .Center:
default:
    println("center aligned")
}
```

Esse código funciona da mesma maneira, mas recomendamos evitar cláusulas `default` ao ativar tipos de enum. Ele não é à prova de alterações futuras. Suponhamos que você escreveu esse código e, depois, voltou a ele e percebeu que seria melhor adicionar outra opção de alinhamento para texto justificado:

```
enum TextAlign {
    case Left
    case Right
    case Center
    case Justify
}

var alignment = TextAlign.Alignment.Left
alignment = .Right
var alignment = TextAlign.Alignment.Justify
```

Observe que seu programa ainda funciona, mas agora exibe o valor errado! A variável `alignment` é definida como `Justify`, mas a instrução de switch exibe "center aligned". Volte à instrução de switch e mude de volta para listar cada caso explicitamente em vez de usar `default`:

```
switch alignment {
    case .Left:
        println("left aligned")

    case .Right:
        println("right aligned")

    default:
        case .Center:
            println("center aligned")
}
```

Agora, em vez de seu programa executar e exibir a resposta errada, você recebe um erro de tempo de compilação dizendo que sua instrução de switch não é abrangente. Pode parecer estranho sugerir que um erro de compilador é algo desejável, mas essa é exatamente a situação. Se você usar uma cláusula `default` ao ligar uma enum e, depois, adicionar um novo caso à enum, a instrução de switch irá para `default` quando encontrar o novo caso. Listar cada caso no switch significa que o compilador pode ajudá-lo a encontrar todos os lugares em seu código que precisarão ser atualizados se você adicionar posteriormente um caso adicional à sua enum.

```
switch alignment {
    case .Left:
        println("left aligned")

    case .Right:
        println("right aligned")

    case .Center:
        println("center aligned")

    case .Justify:
        println("justified")
}
```

Enumerações de valores brutos

Se você já tiver usado enumerações em alguma linguagem como C ou C++, você pode se surpreender ao descobrir que, diferentemente dessas linguagens, as enums da Swift não têm um tipo de inteiro subjacente. Você pode, contudo, escolher ter o mesmo comportamento usando o que a linguagem Swift chama de "valor bruto". Para usar valores brutos de `Int` para sua próxima enumeração de alinhamento de texto, mude a declaração da enum:

```
enum TextAlignment {
    enum TextAlignment : Int {
        case Left
        case Right
        case Center
        case Justify
    }
}
```

Especificar um valor bruto para `TextAlignment` dá um valor bruto diferente desse tipo (`Int`) para cada caso. O comportamento padrão para valores brutos inteiros determina que o primeiro caso recebe um valor bruto igual a 0, o próximo caso recebe um valor bruto igual a 1 e assim por diante, que pode ser confirmado em:

```
var alignment = TextAlignment.Justify

println("Left has raw value \(TextAlignment.Left.rawValue)")
println("Right has raw value \(TextAlignment.Right.rawValue)")
println("Center has raw value \(TextAlignment.Center.rawValue)")
println("Justify has raw value \(TextAlignment.Justify.rawValue)")
println("The alignment variable has raw value \(alignment.rawValue)")
```

Como alternativa, você pode especificar o valor bruto para cada caso manualmente:

```
enum TextAlignement : Int {
    case Left
    case Right
    case Center
    case Justify
    case Left    = 20
    case Right   = 30
    case Center  = 40
    case Justify = 50
}
```

Quando uma enumeração de valor bruto é útil? O motivo mais comum para usar um valor bruto é a necessidade de armazenar ou transmitir a enum. Em vez de ter que escrever funções para transformar uma variável que contém uma enum, você pode usar `rawValue` para converter a variável em seu valor bruto. Isso levanta outra questão: se você tiver um valor bruto, como convertê-lo de volta ao tipo de enum? Todo tipo de enum com valor bruto pode ser criado com um argumento `rawValue:`, que retorna uma enum Opcional (veja o Chapter 9):

```
println("Justify has raw value \(TextAlignment.Justify.rawValue)")
println("The alignment variable has raw value \(alignment.rawValue)")

// Create a raw value.
let myRawValue = 20

// Try to convert the raw value into a TextAlignement
if let myAlignment = TextAlignement(rawValue: myRawValue) {
    // Conversion succeeded!
    println("successfully converted \(myRawValue) into a TextAlignement")
} else {
    // Conversion failed.
    println("\(myRawValue) has no corresponding TextAlignement case")
}
```

O que está acontecendo aqui? Começamos com `myRawValue` que é uma variável do tipo `Int`. Em seguida, tentamos converter esse valor bruto em um caso `TextAlignement` usando `TextAlignement(rawValue:)`. Como `TextAlignement(rawValue:)` tem um tipo de retorno de `TextAlignement?`, você está usando a ligação de opcionais para determinar se recebe de volta um valor `TextAlignement` ou `nil`. O valor bruto usado aqui acaba correspondendo a `TextAlignement.Left`, então a conversão tem êxito. Tente mudá-la para um valor bruto que não existe e observe que você recebe uma mensagem dizendo que a conversão não é possível:

```
let myRawValue = 20
let myRawValue = 100
```

Até aqui, você usou `Int` como o tipo para os valores brutos. A linguagem Swift permite que uma variedade de tipos seja usada, incluindo todos os tipos numéricos embutidos e `String`. Crie uma nova enum que use `String` como o tipo de valor bruto:

```
enum ProgrammingLanguage : String {
    case Swift      = "Swift"
    case ObjectiveC = "Objective-C"
    case C          = "C"
    case Cpp         = "C++"
    case Java        = "Java"
}

let myFavoriteLanguage = ProgrammingLanguage.Swift
println("My favorite programming language is \(myFavoriteLanguage.rawValue)")
```

Você não precisou especificar os valores quando usou pela primeira vez o valor bruto do tipo `Int` - o compilador definiu automaticamente o primeiro caso como 0, o segundo caso como 1 e assim por diante. Isso não acontece com

`String` (ou qualquer tipo de valor bruto que não seja integral): você deve sempre fornecer valores brutos distintos para todos os casos.

Métodos

Um método é uma função associada a um tipo. Em algumas linguagens, os métodos só podem ser associados a classes (que discutiremos no Chapter 15). Na linguagem Swift, os métodos também pode ser associados a enums. Crie uma nova enum que represente o estado de uma lâmpada:

```
enum LightBulb {
    case On
    case Off
}
```

Um dos aspectos que você pode querer saber é a temperatura da lâmpada. (Por questões de simplicidade, presuma que a lâmpada começa a aquecer imediatamente ao ser ligada e resfria até a temperatura ambiente do local imediatamente após ser desligada.) Adicione um método para computar a temperatura da superfície:

```
enum LightBulb {
    case On
    case Off

    func surfaceTemperatureForAmbientTemperature(ambient: Double) -> Double {
        switch self {
            case .On:
                return ambient + 150.0

            case .Off:
                return ambient
        }
    }
}
```

Você adicionou uma função dentro da definição da enumeração `LightBulb`. Devido ao local da definição dessa função, ela agora é um método associado ao tipo `LightBulb`, mais conhecido como "método em `LightBulb`". A função parece aceitar um único argumento (`ambient`); entretanto, como ela é um método, também aceita um argumento implícito chamado de `self` do tipo `LightBulb`. Crie uma variável para representar uma lâmpada e chame seu novo método:

```
enum LightBulb {
    case On
    case Off

    func surfaceTemperatureForAmbientTemperature(ambient: Double) -> Double {
        switch self {
            case .On:
                return ambient + 150.0

            case .Off:
                return ambient
        }
    }
}

var bulb = LightBulb.On
let ambientTemperature = 77.0

var bulbTemperature = bulb.surfaceTemperatureForAmbientTemperature(ambientTemperature)
println("the bulb's temperature is \(bulbTemperature)")
```

Primeiro crie a lâmpada `bulb`, uma instância do tipo `LightBulb`. Quando você tem uma instância do tipo, você pode chamar métodos nessa instância usando a sintaxe `instance.methodName(arguments)`. Outro método que parece

ser útil é um método de alternar a lâmpada. Para alternar a lâmpada, você precisa modificar `self` (para alterná-la de ligado para desligado e de desligado para ligado). Tente adicionar o método `toggle`, que não aceita argumentos nem retorna nada:

```
enum LightBulb {
    case On
    case Off

    func surfaceTemperatureForAmbientTemperature(ambient: Double) -> Double {
        switch self {
            case .On:
                return ambient + 150.0

            case .Off:
                return ambient
        }
    }

    func toggle() {
        switch self {
            case .On:
                self = .Off

            case .Off:
                self = .On
        }
    }
}
```

Após digitar isso, você receberá um erro de compilador dizendo que não é possível atribuir nada a `self` dentro de um método. Na linguagem Swift, uma enumeração é um tipo de valor, e os métodos nos tipos de valor não podem realizar mudanças em `self` (discutiremos mais sobre os tipos de valor no Chapter 15). Se quiser permitir que um método altere `self`, você precisa marcá-lo como método `mutating`:

```
func toggle() {
    mutating func toggle() {
        switch self {
            case .On:
                self = .Off

            case .Off:
                self = .On
        }
    }
}
```

Agora você pode alternar sua lâmpada e ver a temperatura quando a lâmpada estiver desligada:

```
var bulbTemperature = bulb.surfaceTemperatureForAmbientTemperature(ambientTemperature)
println("the bulb's temperature is \(bulbTemperature)")

bulb.toggle()
bulbTemperature = bulb.surfaceTemperatureForAmbientTemperature(ambientTemperature)
println("the bulb's temperature is \(bulbTemperature)")
```

Valores associados

Tudo o que você fez até agora com enumerações está na mesma categoria geral de definição de casos estáticos que enumeram possíveis valores ou estados. A linguagem Swift também ostenta uma propriedade de enumeração muito mais poderosa: casos com valores associados. Os valores associados permitem que você adicione dados (de qualquer tipo especificado por você) a um caso; e diferentes casos em uma enumeração podem ter diferentes tipos de valores associados.

Crie uma enumeração que permita o monitoramento das dimensões de algumas formas básicas. Cada tipo de forma possui diferentes tipos de propriedades: para representar um quadrado, você precisa de um único valor (o comprimento de um lado); para representar um retângulo, você precisa de uma largura e de uma altura.

```
enum ShapeDimensions {
    // Square's associated value is the length of one side
    case Square(Double)

    // Rectangle's associated value defines its width and height
    case Rectangle(width: Double, height: Double)
}
```

Você definiu um novo tipo de enumeração, `ShapeDimensions`, com dois casos. O caso de `Square` tem um valor associado do tipo `Double`. O caso de `Rectangle` tem um valor associado do tipo `(width:Double, height:Double)`, uma tupla com nome (vista pela primeira vez no Chapter 12). Para criar instâncias de `ShapeDimensions`, você deve especificar tanto o caso como um valor associado adequado para o caso:

```
enum ShapeDimensions {
    // Square's associated value is the length of one side
    case Square(Double)

    // Rectangle's associated value defines its width and height
    case Rectangle(width: Double, height: Double)
}

var squareShape = ShapeDimensions.Square(10.0)
var rectShape = ShapeDimensions.Rectangle(width: 5.0, height: 10.0)
```

Você pode usar uma instrução de `switch` para descompactar um valor associado e utilizá-lo. Adicione um método a `ShapeDimensions` para computar a área:

```
enum ShapeDimensions {
    // Square's associated value is the length of one side
    case Square(Double)

    // Rectangle's associated value defines its width and height
    case Rectangle(width: Double, height: Double)

    func area() -> Double {
        switch self {
            case let .Square(side):
                return side * side

            case let .Rectangle(width: w, height: h):
                return w * h
        }
    }
}
```

Na implementação de `area()`, ativamos `self` como fizemos anteriormente neste capítulo. A diferença é que os casos do `switch` usam *correspondência de padrão* da linguagem Swift para ligar o valor associado de `self` a uma nova variável (ou variáveis). Chame o método de `area()` nas instâncias que você criou anteriormente para vê-lo em ação:

```
var squareShape = ShapeDimensions.Square(10.0)
var rectShape = ShapeDimensions.Rectangle(width: 5.0, height: 10.0)

println("square's area = \(squareShape.area())")
println("rectangle's area = \(rectShape.area())")
```

Nem todos os casos de `enum` precisam ter valores associados. Por exemplo, você poderia adicionar um caso de `Point`, mas os pontos geométricos não têm nenhuma dimensão! Tudo bem; deixe de fora o tipo de valor associado. Também atualize o método de `area()` para incluir a área de um `Point`:

```
enum ShapeDimensions {
    // Point has no associated value - it is dimensionless
    case Point

    // Square's associated value is the length of one side
    case Square(Double)

    // Rectangle's associated value defines its width and height
    case Rectangle(width: Double, height: Double)

    func area() -> Double {
        switch self {
            case let .Point:
                return 0

            case let .Square(side):
                return side * side

            case let .Rectangle(width: w, height: h):
                return w * h
        }
    }
}
```

Crie uma instância de um ponto e confirme se ela funciona:

```
var squareShape = ShapeDimensions.Square(10.0)
var rectShape = ShapeDimensions.Rectangle(width: 5.0, height: 10.0)
var pointShape = ShapeDimensions.Point

println("square's area = \(squareShape.area())")
println("rectangle's area = \(rectShape.area())")
println("point's area = \(pointShape.area())")
```

assert() e precondition()

Pode ser que haja um pequeno problema com essa implementação. O que aconteceria se algum louco (possivelmente você, em um momento de fraqueza) criasse uma forma com dimensão negativa? Experimente e veja:

```
var squareShape = ShapeDimensions.Square(10.0)
var rectShape = ShapeDimensions.Rectangle(width: 5.0, height: 10.0)
var rectShape = ShapeDimensions.Rectangle(width: -5.0, height: 10.0)
var pointShape = ShapeDimensions.Point

println("square's area = \(squareShape.area())")
println("rectangle's area = \(rectShape.area())")
println("point's area = \(pointShape.area())")
```

Agora `area()` retorna um valor negativo, o que definitivamente não faz sentido! Como podemos resolver esse problema? A solução mais simples é documentar um aviso em um comentário: "Não crie uma forma com dimensão NEGATIVA", mas isso não ajuda se ninguém estiver prestando atenção. Uma solução mais complicada para isso pode ser o manuseio correto de dimensões negativas, talvez presumindo que elas sejam positivas e aceitando seus valores absolutos, mas isso também é problemático: como a forma com dimensão negativa não deve existir, ela provavelmente representa um bug no código.

As falhas de software podem ser categorizadas em diversas maneiras diferentes, mas uma das mais comuns é dividi-las em duas categorias: erros cometidos pelo programador e erros não cometidos pelo programador. Um erro cometido pelo programador é um bug no código, como a criação de um retângulo com largura negativa. Um erro não cometido pelo programador é um problema que não é causado por um bug no código, mas que ainda assim pode fazer com que o programa se comporte de maneira incorreta. Podemos citar como exemplo de um erro não cometido pelo programador a tentativa de buscar um website usando um dispositivo não conectado à internet (como

um celular durante um voo) - o programa não conseguirá buscar o website, mas não por causa de uma falha do programador.

As soluções dessas duas categorias de falhas são muito diferentes. Erros não cometidos pelo programador precisam ser resolvidos no programa; muitas vezes, a *incapacidade* de resolver um erro não cometido pelo programador é um erro cometido pelo programador (por exemplo, se um aplicativo tentando buscar um website gerar uma falha quando não houver conexão à internet, essa falha definitivamente será devido a um bug)! Os erros cometidos pelo programador geralmente não podem ser resolvidos, porque você nem mesmo sabe que eles existem, ou teria consertado. O melhor a fazer é tentar descobrir esses erros e parar imediatamente o programa antes que ele continue a funcionar em um estado desconhecido.

A linguagem Swift fornece duas funções que podem ser usadas para capturar erros cometidos pelo programador: `assert()` e `precondition()`. Quando um programa é construído em modo de depuração (o modo padrão ao executar em Xcode), as duas funções fazem o mesmo: verificam uma condição que você espera que seja verdadeira e, se for falsa, prendem em uma *armadilha*, que imediatamente o leva ao depurador. Quando um programa é construído em modo de lançamento (por exemplo, quando você envia um aplicativo para Mac ou iOS para a App Store), as chamadas de `assert()` são removidas, mas as chamadas de `precondition()` permanecem, e o aplicativo será interrompido se uma precondição falhar.

Quando devemos usar `assert()` em vez de `precondition()`? Uma regra geral é usar `assert()` para verificar se há condições que somente serão falsas se alguma coisa "local" der errado. A definição de "local" é um pouco nebulosa e requer certo julgamento. Você pode achar que local significa "qualquer código controlado por você", mas uma definição mais comum seria "dentro do mesmo tipo". Por exemplo, se outro método de `ShapeDimensions` tiver chamado `area()`, faria sentido usar `assert()` para confirmar que `area()` retornou um valor não negativo. `precondition()`, por outro lado, deve ser usado para verificar se há erros causados fora do escopo da verificação; por exemplo, os arrays da linguagem Swift usam `precondition()` (ou algo muito parecido) para garantir que você não tente ler além do final do array.

Quanto a `ShapeDimensions`, faz sentido usar `precondition` para verificar se uma forma possui dimensões não negativas, já que um erro desse tipo seria causado pelo código usando `ShapeDimensions`, não o código dentro do próprio `ShapeDimensions`. Faça isso agora:

```
func area() -> Double {
    switch self {
        case let .Square(side):
            precondition(side >= 0, "side cannot be negative")
            return side * side

        case let .Rectangle(width: w, height: h):
            precondition(w >= 0, "width cannot be negative")
            precondition(h >= 0, "height cannot be negative")
            return w * h
    }
}
```

Após adicionar as precondições, abra a linha do tempo e verá: `precondition failed: width cannot be negative`. Agora que `ShapeDimensions` prende os erros cometidos pelo programador em armadilhas, volte e conserte a causa raiz:

```
var rectShape = ShapeDimensions.Rectangle(width: -5.0, height: 10.0)
var rectShape = ShapeDimensions.Rectangle(width: 5.0, height: 10.0)
```

Conclusão

Agora você está familiarizado com as enumerações em Swift e com a variedade de opções que elas apresentam:

- Enums simples para listar os casos de um grupo de coisas relacionadas.

- Enums de valor bruto para grupos de coisas relacionadas em que cada caso possui um valor bruto constante e todos os casos possuem o mesmo tipo de valor bruto.
- Enums de valor associado, um recurso poderoso da linguagem Swift que permite adicionar dados a instâncias de uma enumeração.

Além disso, você aprendeu a criar métodos, que são funções associadas a um tipo. Os métodos também aparecerão proeminentemente no próximo capítulo. Finalmente, você sabe usar `assert()` e `precondition()` para se proteger contra erros cometidos pelo programador.

Desafio

Adicione outro caso à enum `ShapeDimensions` para um triângulo retângulo. Você pode ignorar a direção do triângulo - basta acompanhar o comprimento dos lados do triângulo. Há alguma precondição que faça sentido para essa forma além de verificar se os lados não são negativos? (Dica: lembre-se da geometria que você aprendeu no ensino médio e do teorema de Pitágoras.)

Adicione um método de `perimeter()` à enum `ShapeDimensions`. Esse método deve computar o perímetro de uma forma (a soma do comprimento de todas as extremidades). Lembre-se de lidar com todos os casos!

Para os mais curiosos: erro ao lidar com o Cocoa

Uma expressão comum para escrever funções que podem falhar com erros não cometidos pelo programador em Objective-C, nas plataformas Mac e iOS, é retornar uma opcional (que será nil se a função tiver falhado) e aceitar um parâmetro extra do tipo `NSError` que é definido de acordo com os detalhes do erro. Abaixo, apresentamos um exemplo dessa situação na linguagem Swift para uma função que tenta buscar um website. Observe que este código é sintaticamente válido em Swift, mas não será executado em seu estado atual - ele assume que há outras funções e tipos não exibidos aqui.

```
var error: NSError? = nil
let url = "http://www.bignerdranch.com"

let webSiteContents: Contents? = fetchWebSite(url, error: &error)

if let contents = webSiteContents {
    println("successfully fetched \(contents)")
} else {
    println("fetching \(url) failed with error: \(error)")
}
```

Esse padrão é um tanto estranho devido à incapacidade da linguagem Objective-C de retornar múltiplos valores. Como a linguagem Swift suporta o retorno de múltiplos valores por meio de tuplas, você pode achar que, em vez disso, essa expressão seria traduzida como `fetchWebSite` retornando uma tupla de (`Contents?`, `NSError?`), como:

```
let url = "http://www.bignerdranch.com"

let (webSiteContents: Contents?, error: NSError?) = fetchWebSite(url)

if let contents = webSiteContents {
    println("successfully fetched \(contents)")
} else {
    println("fetching \(url) failed with error: \(error!)")
```

Essa é uma grande melhoria, mas ainda existem problemas. O tipo de retorno de `fetchWebSite` não expressa claramente a intenção da função; por exemplo, a função pode retornar tanto um `Contents` como um `NSError` ou mesmo (`nil`, `nil`)! Em vez disso, as enumerações em Swift com valores associados podem ser usadas para esclarecer o significado: a função retornará um `Contents` ou um `NSError`, mas nunca ambos ou nenhum deles:

```
enum FetchResult {
    case Success(Contents)
    case Failure(NSError)
}

let url = "http://www.bignerdranch.com"

let result: FetchResult = fetchWebSite(url)

switch result {
case let .Success(contents):
    println("successfully fetched \(contents)")

case let .Failure(error):
    println("fetching \(url) failed with error: \(error)")
}
```

Isso resulta em um código ligeiramente maior, mas é muito mais legível, porque os possíveis resultados de `fetchWebSite` são listados claramente no tipo de retorno. Agora é impossível que `fetchWebSite` retorne um valor ambíguo: você tem a garantia de receber o conteúdo ou um erro. Essa garantia é verificada pelo compilador e é óbvio tanto para quem chama (o código aqui) como para o implementador do próprio `fetchWebSite` quais são os possíveis valores de retorno.

15

Structs e classes

As estruturas, normalmente chamadas de structs, e as classes são os alicerces sobre os quais você construirá seus aplicativos. Elas oferecem um importante mecanismo para modelar o que você quer representar em seu código.

Neste capítulo, você passará do Playground e criará um Command Line Tool. O Command Line Tool descreverá um projeto que representa uma cidade sendo invadida por muitos monstros. Você usará tanto structs como classes para modelar essas entidades e atribuirá a elas propriedades de armazenamento de dados e funções para que possam trabalhar. Ao longo do caminho, você verá as diferenças e similaridades entre classes e structs, o que o ajudará a decidir quando usar cada uma.

Um projeto novo

Crie um novo projeto clicando no ícone do Xcode. A primeira tela que aparece é a tela de boas-vindas. Consulte a Figure 15.1 para obter detalhes. Clique no botão que diz "Create a new Xcode Project".

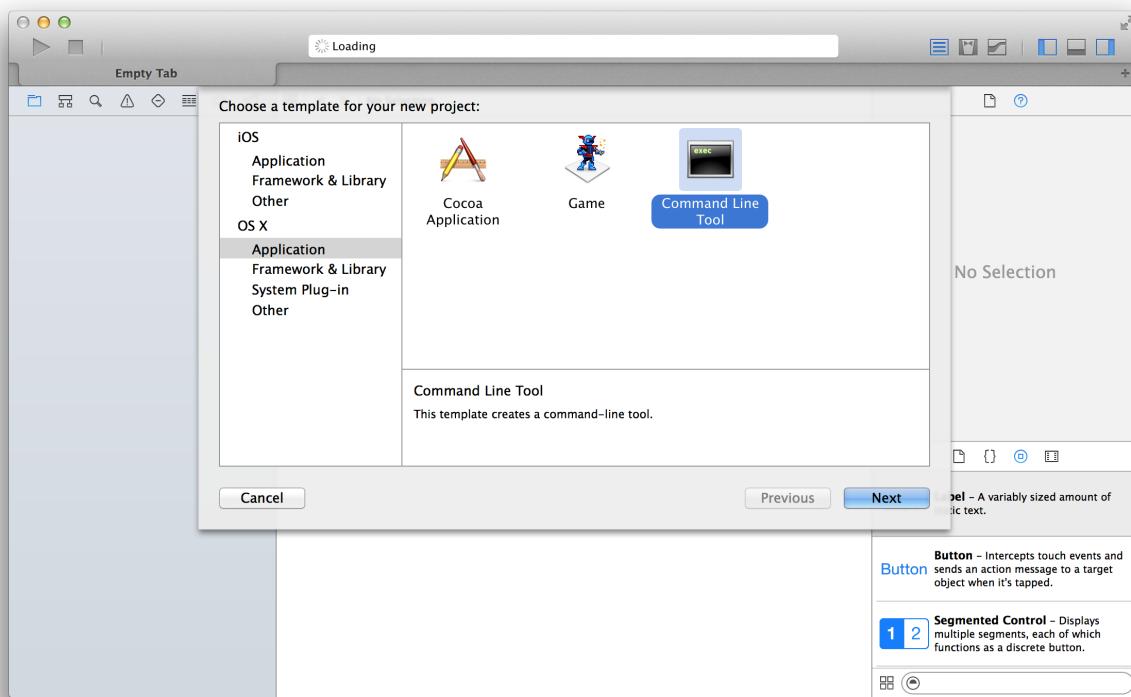
Figure 15.1 Janela de boas-vindas



Em seguida, você verá uma tela para selecionar um template de projeto. O modelo formata seu projeto com uma série de predefinições e configurações comuns a um dado estilo de aplicativo. À esquerda da janela, é possível

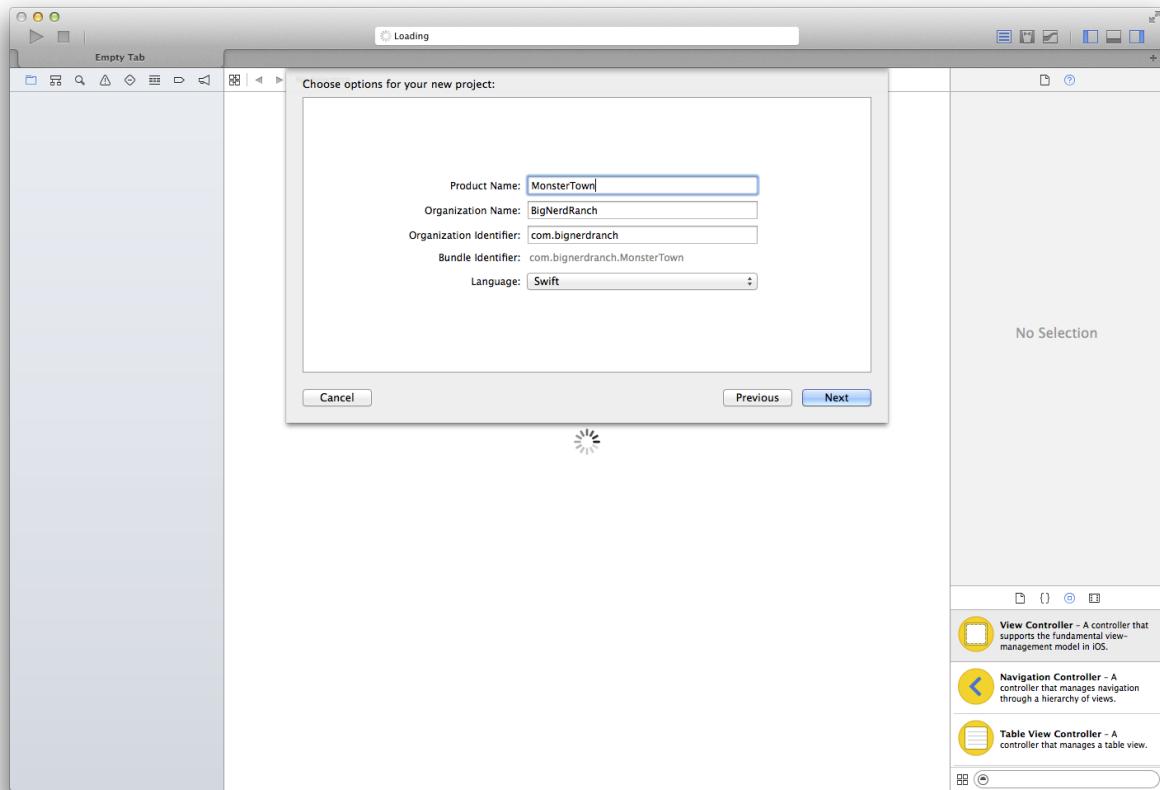
ver duas seções: 1) iOS e 2) OS X. Selecione "Application" dentro da seção OS X; uma janela muito similar à Figure 15.2 vai aparecer. Escolha a opção de template "Command Line Tool". Esse template criará um arquivo de projeto bem básico para você.

Figure 15.2 Escolha de um novo projeto



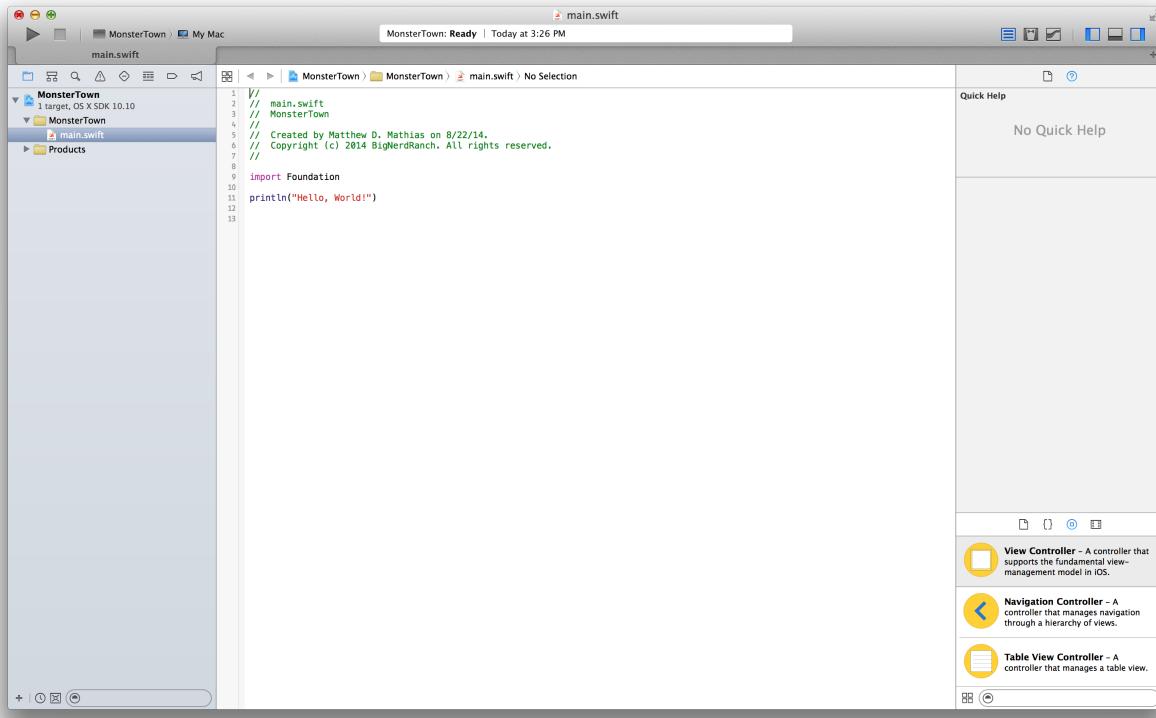
Agora você precisa nomear seu projeto. Você verá uma janela semelhante à Figure 15.3. No campo "Product Name", digite "MonsterTown". Você pode digitar o que quiser em "Organization Name" do projeto. Usamos BigNerdRanch. O "Organization Identifier" normalmente usa a anotação "Reverse Domain Name Service" (DNS reverso) e é usado com "Product Name" para criar um "Bundle Identifier". A ID do pacote é usada para identificar seu aplicativo no iTunes Connect quando você estiver pronto para distribuí-lo. A última coisa a fazer nesta janela é selecionar Swift na opção "Language".

Figure 15.3 Nomeando seu projeto



Por último, o Xcode perguntará onde salvar o projeto. Selecione um local que você considere bom e clique em "Create". Agora que você criou seu projeto, verá que ele está aberto no Xcode com o arquivo `main.swift` selecionado. Veja a Figure 15.4.

Figure 15.4 main.swift



Observe que o arquivo `main.swift` já tem o código a seguir:

```

import Foundation

println("Hello, World!")

```

O código `import Foundation` leva o framework do Foundation para o arquivo `main.swift`. Esse framework consiste em uma série de classes projetadas principalmente para trabalhar em e com a linguagem Objective-C. Em seguida, note o código `println("Hello, World!")` com o qual você já deve estar bastante familiarizado. Ele registra a string "Hello, World!" no console. Vá em frente e execute o seu programa. Isso pode ser feito de diversas maneiras: 1) clique em "Product" na barra de ferramentas e selecione "Run", 2) toque em command-R em seu teclado ou 3) clique no botão play no canto superior esquerdo do Xcode.

Depois de executar o programa, você verá que "Hello, World!" está registrado no console. Isso é ótimo, mas você já viu esse comportamento antes. Vamos deixar seu programa mais interessante. Primeiro, remova o código listado em `main.swift`. Você não vai precisar desse código.

```

import Foundation

println("Hello, World!")

```

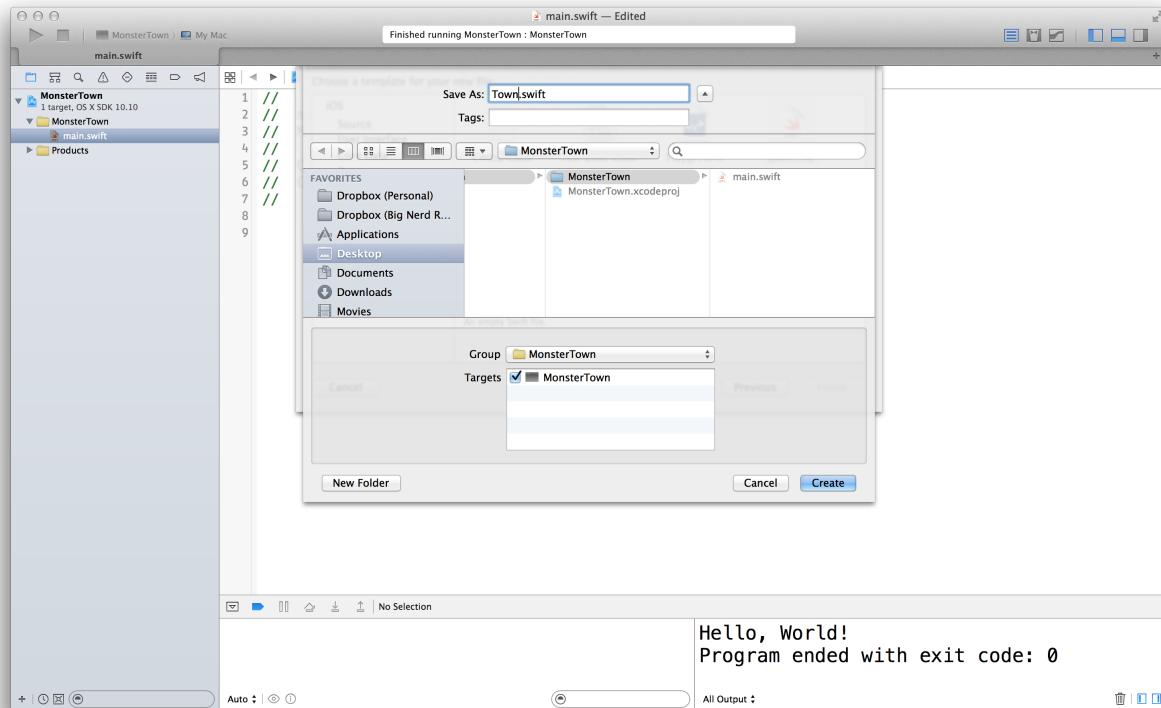
Em seguida, você passará para a criação das suas próprias structs e classes.

Estruturas

Uma struct é um tipo que agrupa um conjunto de blocos de dados relacionados na memória. É usada quando um desenvolvedor quer agrupar dados em um tipo comum. Existem outras considerações a serem feitas ao escolher entre uma classe e uma struct. Veremos essas considerações ao longo deste capítulo.

Adicione um novo arquivo ao seu projeto. Clique em File > New > File... Você também pode usar as teclas command-N no teclado para isso. Uma nova janela aparecerá e pedirá que você selecione um template para o novo arquivo. Selecione "Source" na seção OS X à esquerda e, em seguida, escolha "Swift File" e clique em "Next". Depois disso, você terá de nomear o novo arquivo e definir a localização. Chame esse arquivo de Town.swift e lembre-se de marcar a caixa para adicioná-lo ao destino MonsterTown. Veja a Figure 15.5 para referência.

Figure 15.5 main.swift



No lado esquerdo da janela do aplicativo Xcode, encontramos o navegador de projetos. No navegador de projetos, é possível ver uma listagem com seus arquivos: 1) main.swift e 2) Town.swift. Selecione o arquivo Town.swift. Quando esse arquivo aparecer, observe que ele estará completamente em branco (exceto o código com comentário no topo do arquivo). Adicione o exemplo abaixo para criar uma declaração para a struct da sua cidade.

```
struct Town {  
}
```

A palavra-chave `struct` indica que você está declarando uma struct com nome "Town". Você adicionará o código entre chaves (`{}`) para definir o comportamento dessa estrutura. Por exemplo, você pode adicionar variáveis a sua nova struct para que ela retenha dados.

Tecnicamente, essas variáveis são chamadas de propriedades, que é o assunto do próximo capítulo. As propriedades podem ser variáveis ou constantes, como vimos antes quando usamos as palavras-chaves `var` e `let`. Adicione algumas propriedades a sua struct.

```
struct Town {  
    var population = 5422  
    var numberofStopLights = 4  
}
```

Se lembrarmos dos capítulos anteriores, você estava tentando modelar uma cidade em um Playground. Como esse exemplo era relativamente pequeno, não foi tão limitador. Entretanto, é melhor englobar a definição da cidade

dentro de seu próprio tipo. Nesse caso, agora você está definindo uma nova struct chamada de Town. Town tem duas propriedades: 1) population e 2) numberofStopLights. Ambas essas propriedades são mutáveis - a população e o número de semáforos de uma cidade podem mudar. Essas propriedades também são valores padrão por uma questão de simplicidade. Quando uma nova instância da struct Town é criada, ela tem como padrão uma population de 5422 e 4 semáforos.

Crie uma nova instância de Town para ver sua struct em ação. Alterne para seu arquivo `main.swift` e adicione este código:

```
var myTown = Town()  
println("Population: \(myTown.population),  
       number of stop lights: \(myTown.numberofStopLights)")
```

Esse código faz três coisas.

Primeiro, cria-se uma instância do tipo Town. Para isso, você digitou o nome do tipo do qual você queria obter uma instância (nesse caso, é Town) seguido de parêntesis vazios (). Em seguida, define-se essa nova instância como igual a uma variável chamada de `myTown`.

Depois disso, você usou interpolação de strings para exibir os valores das duas propriedades da struct Town no console. Observe que você usou a sintaxe de ponto para acessar os valores das propriedades. Por exemplo, a sintaxe `myTown.population` recupera a população da instância `myTown`. Assim, o resultado deve dizer: "Population: 5422, stop lights: 4".

Métodos de instância

A função `println()` acima é um ótimo jeito de exibir a descrição de `myTown`, mas uma cidade deve saber como se autodescrever. Cria uma função na struct Town que exibe os valores de suas propriedades no console. Navegue até seu arquivo `Town.swift` e adicione a seguinte definição de função.

```
struct Town {  
    var population = 5422  
    var numberofStopLights = 4  
  
    func printTownDescription() {  
        println("Population: \(population); number of stop lights: \(numberofStopLights)")  
    }  
}
```

A função `printTownDescription()` é chamada de método porque é uma função associada a um tipo particular. `printTownDescription()` não aceita nenhum argumento e não retorna nada. Em vez disso, o objetivo principal é registrar uma descrição das propriedades da cidade no console.

Para utilizar seu novo método de instância, é preciso chamar a função em uma instância de Town. Navegue até `main.swift` e troque a função `println()` pelo seu novo método de instância.

```
var myTown = Town()  
println("Population: \(myTown.population),  
       number of stop lights: \(myTown.numberofStopLights)")  
myTown.printTownDescription()
```

Usa-se a sintaxe de ponto para chamar uma função em uma instância: `myTown.printTownDescription()`. Vá em frente e execute seu programa; você verá o mesmo resultado no console que antes.

Métodos mutáveis

Sua função `printTownDescription()` é ótima para exibir as informações atuais da sua cidade, mas e se você precisasse de uma função que mudasse essas informações? Você precisará escrever uma função *mutável* para isso. Se um método de instância em uma struct mudar alguma propriedade da struct, esse método precisará ser marcado como mutável. Em `Town.swift`, adicione uma nova função ao tipo Town para aumentar a população da instância de uma cidade.

```

struct Town {
    var population = 5422
    var number0fStopLights = 4

    func printTownDescription() {
        println("Population: \population; number of stop lights: \number0fStopLights")
    }

    mutating func increasePopulation(amount: Int) {
        population += amount
    }
}

```

Observe que você marcou o método de instância `increasePopulations()` com a palavra-chave `mutating`. Como no Chapter 14, isso significa que esse método pode alterar os valores na struct. A questão aqui é que tanto as estruturas como as enumerações são tipos de valor e requerem mutação.

O método tem um argumento, `amount`, que é do tipo `Int`. Esse parâmetro é usado para aumentar a população da cidade: `population += amount`. Alterne para `main.swift` para exercitar essa função.

```

var myTown = Town()
myTown.increasePopulation(500)
myTown.printTownDescription()

```

Como antes, você usou a sintaxe de ponto para chamar a função em sua cidade. Se construir e executar o programa, você verá que a população de `myTown` foi aumentada em 500: Population: 5922; number of stop lights: 4.

Classes

Uma classe de monstros

Agora que você tem uma struct que representa uma cidade, é hora de deixar as coisas um pouco mais interessantes. Imagine, por exemplo, que sua cidade fica na Transilvânia. Nesse caso, sua cidade teria de se preocupar com uma variedade de monstros que perambulam pelo país. Comece criando um novo arquivo chamado de `Monster.swift`. Adicione o código abaixo para que seu arquivo corresponda.

```

class Monster {

}

```

Você perceberá que a sintaxe para definir uma nova instância de classe é muito similar à sintaxe usada para definir uma nova instância de struct. A diferença entre elas está no uso da palavra-chave `class`. Como antes, a definição da classe fica entre chaves: {}.

Por questões de herança (discutidas na próxima seção), a classe `Monster` é definida em termos bastante gerais. Isso significa que o tipo `Monster` descreverá a forma geral de um monstro.

```

class Monster {
    var town: Town? = Town()
    var name = "Monster"

    func terrorizeTown() {
        if let t = town {
            println("\(name) is terrorizing a town!")
        } else {
            println("\(name) hasn't found a town to terrorize yet...")
        }
    }
}

```

Historicamente, sabemos que monstros fazem uma coisa muito bem: perambulam e aterrorizam cidades. Sendo assim, o tipo `Monster` tem uma propriedade opcional para a cidade que será aterrorizada pelo monstro. Lembre-se

de que opcionais são usadas quando uma instância pode se tornar `nil`. Assim, a propriedade `town` é uma opcional do tipo `Town?` porque o monstro pode ou não já ter encontrado uma cidade para aterrorizar. Você também criou uma propriedade para o nome do `Monster` e atribuiu a ele um valor padrão genérico.

Em seguida, você definiu um stub básico para uma função chamada de `terrorizeTown()`. Essa função será chamada em uma instância para aterrorizar a cidade do monstro. Observe que você usou a ligação de opcionais para verificar se a instância tem uma `town`. Se tiver, ela registrará no console o nome do monstro assolando a cidade. Se a instância ainda não tiver uma cidade, o método registrará essa informação.

Como cada tipo de monstro aterrorizará a cidade de maneira diferente, as subclasses devem oferecer suas próprias implementações desse método.

Altere para seu `main.swift` para exercitar a classe `Monster`. Adicione o código abaixo para obter uma instância do tipo, atribua a ela uma cidade e chame `terrorizeTown()` para ela.

```
var myTown = Town()
myTown.increasePopulation(500)
myTown.printTownDescription()
let gm = Monster()
gm.town = myTown
gm.terrorizeTown()
```

Primeiro, cria-se uma instância do tipo `Monster` chamada de `gm` (para um monstro genérico). Essa instância é declarada como constante porque não há necessidade de ela ser mutável. Em seguida, atribui-se `myTown` à propriedade `town` de `gm`. Finalmente, chama-se a função `terrorizeTown()` na instância `Monster`. Você verá o registro de `Monster is terrorizing a town!` no console.

Herança

Um dos principais recursos das classes, o qual estruturas não possuem, é a herança. A herança refere-se a um relacionamento no qual uma classe, uma subclasse, é definida nos termos de outra, uma superclasse. A natureza dessa relação significa que uma subclasse *herda* as propriedades e os métodos de sua superclasse. De certo modo, a herança define a genealogia de tipos de classe.

O fato das classes poderem tirar proveito da herança é o principal motivo de ter definido `Monster` como classe em primeiro lugar. Crie um tipo `Zombie` que seja herdeiro do tipo `Monster` para ver a relação de forma mais prática.

Uma subclasse de zumbis

Crie um novo arquivo em Swift chamado `Zombie`. Esse arquivo conterá a definição de uma nova classe que descreverá um zumbi. O tipo `Zombie` será herdeiro do tipo `Monster`. Adicione a seguinte declaração de classe para ver como.

```
class Zombie: Monster {
    var walksWithLimp = true

    override func terrorizeTown() {
        town?.changePopulation(-10)
        super.terrorizeTown()
    }
}
```

A classe acima define um novo tipo chamado de `Zombie`. Ele é herdeiro do tipo `Monster`, que é indicado pelos dois pontos (`:`) após `Zombie`. Ser herdeiro de `Monster` significa que `Zombie` tem todas as propriedades e métodos de `Monster` (isto é, a propriedade `town` e a função `terrorizeTown()`).

`Zombie` também adiciona uma nova propriedade. A propriedade é chamada de `walksWithLimp` do tipo `Bool`. Esse tipo é inferido com base no valor padrão da propriedade: `true`.

Por último, `Zombie` sobrescreve o método `terrorizeTown()`. Observe o uso de `override`. A palavra-chave `override` é necessária quando uma subclasse fornece sua própria definição de um método definido em uma superclasse. Na verdade, não adicionar `override` resultará em um erro de compilador.

Lembre-se de que a propriedade `town` de `Zombie` é uma opcional do tipo `Town?`, que é herdeiro do tipo `Monster`. O fato de `town` ser uma opcional significa que ela pode acabar sendo `nil`. Assim, você precisa se certificar de que a instância de `Zombie` tenha uma cidade para aterrorizar antes de chamar quaisquer métodos na `town`. Como podemos fazer essa verificação?

Uma possível solução é usar a ligação de opcionais. Você pode sentir-se tentado a fazer algo como:

```
if let terrorTown = town {
    // do something to terrorTown
}
```

No código acima, a instância `Zombie` tem uma `town`, então o valor na opcional é desempacotado e colocado na constante `terrorTown`. Depois disso, esse valor estará pronto para ser aterrorizado, mas com uma restrição importante: a semântica das structs do valor indica que a instância `terrorTown` não será igual à instância `town`. O problema é que nenhuma mudança feita na `terrorTown` será refletida na propriedade `town` da instância `Zombie`. Além dessa limitação, esse código também pode ser mais conciso.

Como vimos no capítulo sobre Opcionais, o encadeamento de opcionais permite que essa verificação seja feita em uma única linha. Mantém-se a mesma expressividade, mas de forma mais concisa. Por exemplo, `town?.changePopulation(-10)` realiza a mesma função que a ligação de opcionais. Se a opcional `town` tiver um valor, o método `changePopulation()` é chamado nessa instância.

Esse código o ajuda a garantir que é seguro chamar uma função na instância. Sendo assim, quando uma instância do tipo `Zombie` aterrorizar uma cidade, a população será reduzida em 10 pessoas.

Finalmente, observe a linha: `super.terrorizeTown()`. `super` é um prefixo usado para acessar a implementação de um método de uma superclasse. Nesse caso, você usou `super` para chamar a implementação de `terrorizeTown()` do tipo `Monster`. Portanto, você verá mais uma vez o registro de `Monster is terrorizing a town!` no console.

Como `super` baseia-se na ideia de herança, ele não está disponível para tipos de valor como enums ou structs. Normalmente, ele é invocado para pegar emprestado ou sobrescrever a funcionalidade de uma superclasse.

Impedimento da sobrescrita

Às vezes, queremos impedir que as subclasses sejam capazes de sobrescrever métodos ou propriedades. Na prática, isso raramente será necessário, mas acontece ocasionalmente. Nesses casos, usamos a palavra-chave `final` para impedir que um método ou propriedade seja sobrescrito.

Imagine, por exemplo, que você não quer especificamente que as subclasses do tipo `Zombie` forneçam a própria implementação da função `terrorizeTown()`. Talvez você tenha um bom motivo para querer que todas as subclasses de `Zombie` aterrorizem suas cidades da mesma forma. Adicione a palavra-chave `final` à declaração dessa função.

```
class Zombie: Monster {
    class func makeSpookyNoise() -> String {
        return "Brains..."
    }

    var walksWithLimp = true
    override func terrorizeTown() {
        town?.changePopulation(-10)
        super.terrorizeTown()
    }
    final override func terrorizeTown() {
        town?.changePopulation(-10)
        super.terrorizeTown()
    }
    func changeName(name: String, walksWithLimp: Bool) {
        self.name = name
        self.walksWithLimp = walksWithLimp
    }
}
```

Agora, as subclasses da classe `Zombie` não serão capazes de sobrescrever a função `terrorizeTown()`. Vá em frente e crie uma nova subclasse de `Zombie`; chame-a de `ZombieBoss`. Tente sobrescrever a função `terrorizeTown()`.

```
class ZombieBoss: Zombie {
    override func terrorizeTown() {
        println("terrorizing town...")
    }
}
```

Você verá o seguinte erro na linha onde tentou sobrescrever a função `terrorizeTown(): Instance method overrides a 'final' instance method.` Esse erro indica que não é possível sobrescrever essa função porque ela está marcada como `final` na superclasse. Vá em frente e exclua esse novo arquivo; você não o usará mais.

Nossa cidade está com um problema de zumbis

Agora é uma boa hora de exercitar o tipo `Zombie`. Escolha o arquivo `main.swift` no navegador de projetos à esquerda. Escreva um código para criar uma instância da classe `Zombie`. Observe que é necessário excluir o código que exibe a descrição da cidade para deixar o console menos confuso; também devemos excluir o código que criou uma instância genérica do tipo `Monster` já que ele não será mais necessário.

```
var myTown = Town()
myTown.increasePopulation(500)
myTown.printTownDescription()
let gm = Monster()
gm.town = myTown
gm.terrorizeTown()
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
```

Primeiro, cria-se uma nova instância do tipo `Zombie` chamada de `fredTheZombie`. Em seguida, atribui-se a instância preexistente do tipo `Town`, `myTown`, à propriedade `town` do tipo `Zombie`. Nesse momento, `fredTheZombie` está livre para aterrorizar uma cidade, o que será feito com entusiasmo.

Depois que `fredTheZombie` tiver aterrorizado por tempo suficiente, é bom verificar os resultados com `printTownDescription()`. Observe, contudo, que isso não é tão simples como antes. Infelizmente, por motivos que serão discutidos em breve, não é possível chamar a função `printTownDescription()` na instância `myTown`. Sendo assim, é melhor exibir a descrição da cidade de `fredTheZombie`.

Como a propriedade `town` de `fredTheZombie` é uma opcional do tipo `Town?`, é preciso desempacotá-la antes de poder chamar a função `printTownDescription()` nela. Como antes, você pode realizar essa operação com encadeamento de opcionais com o operador de ponto de interrogação: `fredTheZombie.town?`. Esse código garante que o zumbi tenha uma cidade dentro da opcional antes de você tentar usar a função de descrição.

Depois que `fredTheZombie` tiver terminado de aterrorizar sua cidade, o resultado do console deve indicar:
"Population: 5912; number of stop lights: 4".

Nomes de parâmetros de métodos

Neste momento, é possível que tenha percebido que as convenções seguidas pela linguagem Swift para os nomes de parâmetros externos são diferentes entre as funções globais usadas no início do livro e os métodos usados neste capítulo. Os parâmetros de uma função global não são nomes externos atribuídos como padrão. Por exemplo, lembre-se da função global `divisionDescription` -- ou seja, uma função que não seja definida em um tipo específico -- do Chapter 12:

```
func divisionDescription(num: Double, den: Double) {
    println("\(num) divided by \(den) equals \(num / den)")
}
divisionDescription(9, 3)
```

divisionDescription tem dois argumentos: `num` e `den`. Observe que você não usou esses nomes de parâmetros quando chamou a função. Você precisaria usar nomes de parâmetros explícitos ou sintaxe abreviada se quisesse esse comportamento.

Os métodos em tipos funcionam de maneira um pouco diferente. O primeiro argumento de um método de tipo ou instância não recebe um nome externo, mas todos os outros argumentos recebem nomes externos como se tivesse prefixado-os com `#`. Se um método de tipo ou instância tiver um argumento, a mesma regra se aplica: o argumento não recebe um nome externo.

Para ver essa diferença em ação, abra o arquivo `Zombie.swift` para adicionar uma nova função a essa classe. Adicione uma nova função para mudar o nome e o status de manco de uma instância da classe `Zombie`.

```
class Zombie: Monster {
    class func makeSpookyNoise() -> String {
        return "Brains..."
    }

    var walksWithLimp = true

    override func terrorizeTown() {
        town?.changePopulation(-10)
        super.terrorizeTown()
    }
    func changeName(name: String, walksWithLimp: Bool) {
        self.name = name
        self.walksWithLimp = walksWithLimp
    }
}
```

A nova função `changeName` tem a seguinte assinatura: `changeName(_: walksWithLimp:)`. Ela é uma função simples que permite que um desenvolvedor mude as propriedades `name` e `walksWithLimp` de um `Zombie`. O caractere de sublinhado (`_`) indica que o primeiro parâmetro da função não tem nome; ou seja, você não usará o nome desse parâmetro ao chamar a função. Entretanto, `walksWithLimp` é usada ao chamar a função, conforme indicado pela assinatura da função.

Alterne para `main.swift` para exercitar essa função e ver como ela é chamada. Adicione uma chamada a essa nova função no final do arquivo.

```
...
fredTheZombie.changeName("Fred the Zombie", walksWithLimp: false)
```

Chama-se `changeName(_: walksWithLimp:)` em `fredTheZombie`. Observe que o nome do primeiro argumento é omitido, mas o nome do segundo é usado. Escreve-se o nome do primeiro argumento no nome da função real -- a parte "Name" de `changeName`. Essa convenção é um padrão comum ao nomear métodos em frameworks de iOS e Mac OS X. A palavra 'name' corresponde à primeira propriedade que você gostaria de modificar com essa função e `walksWithLimp` corresponde à segunda propriedade. Nomear a função dessa maneira facilita a visualização de que "Fred the Zombie" é a `String` que você quer atribuir à propriedade `name`.

A discrepância entre as funções globais e os métodos de tipo podem parecer estranhas. O motivo dessa diferença está relacionado à relação da linguagem Swift com a Objective-C e a C. Por enquanto, tudo o que você precisa saber é que as funções globais não atribuem nomes de parâmetros externos por padrão, enquanto métodos de tipo não atribuem nomes de parâmetros externos livres para o primeiro argumento, mas atribuem nomes externos a todos os parâmetros subsequentes.

Tipos de valor vs. tipos de referência

No exemplo acima de uma instância `Zombie` aterrorizando uma cidade, você chamou `printTownDescription()` na `town` da `fredTheZombie`. É possível achar que essa instância é a mesma instância de `Town` que está em `myTown`, mas isso não é verdade. Daqui em diante, chame `printTownDescription()` em `myTown`.

```
var myTown = Town()
myTown.increasePopulation(500)
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
myTown.printTownDescription()
```

Você verá que o console registra valores diferentes para a `population`. Ou seja, `myTown` tem uma população de 5922 e não 5912 como a cidade de `fredTheZombie`. Por que esses dois valores são diferentes? A resposta tem a ver com a diferença entre os tipos de valor e os tipos de referência.

Os tipos de valor são tipos cujos valores são copiados quando são atribuídos a outra instância ou passados ao argumento de uma função. Isso significa que atribuir uma instância de um tipo de valor a outra, na verdade, atribui uma cópia da primeira instância à segunda instância. Os tipos de valor desempenham uma função importante na linguagem Swift. Por exemplo, arrays e dicionários são ambos tipos de valor. Todas as enums e structs que você escrever também são tipos de valor.

Note que `Town` é uma struct. Assim, quando `myTown` foi atribuído a `fredTheZombie` (`fredTheZombie.town = myTown`), uma cópia de `myTown` foi criada e foi essa cópia que foi atribuída a `town` de `fredTheZombie`. Assim, quando `fredTheZombie` aterroriza sua cidade, é apenas sua instância de `Town` que é atualizada. A `myTown` original é mantida inalterada.

Tipos de referência não são copiados quando são atribuídos a uma instância ou passados a um argumento de uma função. Em vez disso, passa-se uma referência à mesma instância. Classes e funções são tipos de referência. Sendo assim, como `Zombie` é uma classe, ele é, portanto, um tipo de referência. Crie mais duas instâncias de `Zombie` dentro de `main.swift` para ver como os tipos de referência se diferenciam dos tipos de valor.

```
var myTown = Town()
myTown.increasePopulation(500)
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
myTown.printTownDescription()
let z1 = Zombie()
z1.walksWithLimp = false
let z2 = z1
println("z1 walks with limp? \$(z1.walksWithLimp); z2 walks with limp? \$(z2.walksWithLimp)")
```

Primeiro, você criou uma nova instância de `Zombie` e colocou uma referência a essa instância em uma constante chamada de `z1`. Em seguida, você definiu `walksWithLimp` como `false` em `z1`. Em seguida, criou uma nova constante atribuindo `z1` a `z2`. Isso define `z2` para apontar para a mesma instância do tipo `Zombie` apontado por `z1`. Esse ponto é confirmado pelo resultado do console gerado pela função `println()` acima: "z1 walks with limp? false; z2 walks with limp? false".

Para sublinhar o ponto sobre os tipos de referência, mude o valor de `walksWithLimp` em `z2`. Lembre-se de definir `walksWithLimp` em `z2` antes de registrar o resultado no console.

```
let z1 = Zombie()
z1.walksWithLimp = false
let z2 = z1
z2.walksWithLimp = true
println("z1 walks with limp? \$(z1.walksWithLimp); z2 walks with limp? \$(z2.walksWithLimp)")
```

Como podemos ver, `z1` e `z2` andam mancando: "z1 walks with limp? true; z2 walks with limp? true". Esse resultado pode parecer confuso; afinal, você apenas definiu `walksWithLimp` em `z2`. Todavia, lembre-se de que os tipos de referência não criam uma cópia da instância durante a atribuição. Em vez disso, a nova constante ou variável recebe uma referência à mesma instância que a constante ou variável no lado direito da atribuição. Assim, ambas as instâncias `z1` e `z2` têm a mesma referência à instância do tipo `Zombie` na memória. Mudar o valor de `walksWithLimp` em uma dessas constantes também altera o valor da outra porque ambos apontam para a mesma instância de `Zombie`.

Curiosamente, é possível alterar o valor de `walksWithLimp` em `z1` e `z2` mesmo tendo sido declarados como constantes. Por que isso funciona? Entender que os tipos de referência na verdade não armazenam a instância do próprio tipo ajuda a esclarecer essa dúvida. Em outras palavras, as duas constantes, `z1` e `z2`, contêm uma referência à mesma instância da classe `Zombie`. Ao trocar um valor nessa instância, você na verdade deixa a referência a ela intacta e inalterada. Essa distinção é importante porque `z1` e `z2` são mera constante ligadas a uma referência que aponta para uma instância subjacente em algum lugar da memória. Se você tentar reatribuir a referência contida por uma dessas constantes, talvez criando uma nova instância da classe `Zombie`, o compilador emitirá um erro. Assim, é importante lembrar que há uma distinção entre a referência a uma instância e a própria instância.

Identidade vs. igualdade

Agora que você entendeu a diferença entre tipos de valor e de referência, você está apto para aprender sobre igualdade e identidade. Igualdade se refere a duas instâncias com os mesmos valores em suas características observáveis; por exemplo, duas instâncias do tipo `String` que tenham o mesmo texto. Identidade, por outro lado, refere-se à possibilidade de duas variáveis ou constantes apontarem ou não para a mesma instância na memória. Por exemplo, dê uma olhada no código abaixo.

```
let x = 1
let y = 1
x == y // true
```

Duas constantes, `x` e `y`, são criadas. Ambas são do tipo `Int` e contêm o mesmo valor, 1. Como esperado, a verificação de igualdade, realizada por meio de `==`, avalia como `true`, o que faz sentido porque `x` e `y` contêm o mesmo valor. Isso é exatamente o que queremos saber com a verificação de igualdade: duas instâncias têm o mesmo valor? Todos os tipos de dados básicos da linguagem Swift (`String`, `Int`, `Float`, `Double`, `Array` e `Dictionary`) podem ser verificados quanto à igualdade.

`z1` e `z2` são ambos tipos de referência porque apontam para uma instância da classe `Zombie`. Assim, você pode verificar a identidade nessas duas constantes usando o operador de identidade: `===`. Veja abaixo.

```
z1 === z2 // true
```

A verificação de identidade acima funciona porque tanto `z1` como `z2` apontam para o mesmo local na memória onde se encontra uma instância da classe `Zombie`.

Mas se você quisesse verificar a identidade em `x` e `y`? Você pode achar que é possível usar o operador de identidade: `x === y`. Esse código gerará um erro no compilador. Por quê? Esse erro acontece porque os tipos de valor são passados por valor. Como o tipo `Int` é implementado na linguagem Swift como `struct`, tanto `x` como `y` são tipos de valor. Sendo assim, você não pode comparar essas duas constantes com base em sua localização na memória.

A propósito, o que acontece se você tentar verificar a igualdade em `z1` e `z2`: `z1 == z2`? Você verá um erro do compilador indicando que você não pode invocar `==` com uma lista de argumentos do tipo (`Zombie`, `Zombie`). Em português, isso quer dizer que o compilador não sabe como chamar a função `==` na classe `Zombie`. Se você quiser verificar se há igualdade em classes criadas por você, é necessário ensinar às suas classes como fazer isso implementando a função `==`. Fazer isso envolve estar em conformidade com um protocolo chamado de `Equatable`, que é o assunto que introduziremos em outro capítulo.

Como observação final, é importante perceber que as duas constantes ou variáveis podem ser iguais (isto é, ter o mesmo valor), mas podem não ser idênticas (isto é, podem apontar para instâncias distintas de um dado tipo). Particularmente, isso não funciona ao contrário. Se duas variáveis ou constantes apontarem para a mesma instância na memória, elas também serão iguais.

O que devemos usar?

As estruturas e classes são adequadas para definir diversos tipos personalizados. Havia um tempo em que as structs eram tão diferentes das classes que os casos de uso de ambos esses tipos eram óbvios. Na linguagem Swift, no

entanto, a funcionalidade adicionada às structs tornou o comportamento das structs mais similar ao das classes. Essa similaridade deixou a decisão de qual usar um tanto mais complicada.

Entretanto, existem importantes diferenças entre structs e classes que ajudam a dar uma orientação sobre qual deve ser usada e quando. Apresentamos aqui algumas recomendações.

1. Se você quiser que um tipo seja passado por valor, use uma struct. Isso garante que o tipo seja copiado quando atribuído ou passado ao argumento de uma função.
2. Se você não quiser que o tipo seja herdado, use uma struct. As structs não suportam herança e, portanto, não podem possuir subclasses.
3. Se o comportamento que você gostaria de representar em um tipo for relativamente simples e englobar apenas alguns valores simples, considere começar com uma struct. É possível alterar o tipo para que vire uma classe a qualquer momento no futuro.
4. Use uma classe em todos os outros casos.

Os itens acima representam um conjunto de orientações gerais; não são de forma alguma regras explícitas. De fato, regras fixas e inalteráveis sobre esse assunto são difíceis de definir. Alguns exemplos podem ajudá-lo a esclarecer um pouco mais essa questão. As structs normalmente são usadas ao modelar tamanhos de formas (por exemplo, retângulos têm uma largura e uma altura), faixas (por exemplo, uma corrida tem um início e um fim) e pontos em algum sistema de coordenadas (por exemplo, um ponto em um espaço bidimensional tem um valor X e um valor Y). Também são ótimas para definir estruturas de dados; na verdade, os tipos `String`, `Array` e `Dictionary` são todos definidos como structs na biblioteca padrão da linguagem Swift.

Esses exemplos não são de forma alguma extremamente abrangentes. Existe muito espaço para experimentação e criatividade aqui. Em geral, sugerimos começar com uma struct apenas se você tiver total certeza de que precisa dos benefícios de um tipo de referência. Os tipos de valor são um pouco mais fáceis de abordar porque são um tanto "mais seguros" de trabalhar (por exemplo, você não precisa se preocupar com o que acontece com uma instância se você mudar os valores em uma cópia).

Desafio

Desafio de bronze

Atualmente, existe um bug no tipo `Zombie`. Se uma instância de `Zombie` aterrorizar uma cidade com uma população de 0, a população diminuirá até -10. Esse resultado não faz sentido. Conserte esse bug trocando a função `terrorizeTown()` no tipo `Zombie` para apenas diminuir a população da cidade se a ela for superior a 0. Também se certifique de que a população da cidade seja definida como 0 se a quantidade a ser reduzida for superior à população atual.

Desafio de prata

Crie outra subclasse do tipo `Monster`. Chame-a de `Vampire`. Sobrescreva a função `terrorizeTown()` para que toda vez que uma instância do tipo `Vampire` aterrorizar uma cidade, ela adicione um novo servo de vampiro a um array de vampiros no tipo `Vampire`. Esse array de servos de vampiro devem ficar vazios por padrão. Aterrorizar uma cidade também deve diminuir a população de vampiros de uma cidade em um. Por último, exerçite esse tipo `Vampire` em `main.swift`.

Para os mais curiosos: métodos de tipo

Neste capítulo, você definiu alguns métodos de instância que foram chamados em instâncias de um tipo. Por exemplo, `terrorizeTown()` é um método de instância que pode ser chamado em instâncias do tipo `Monster`. Você também pode definir métodos que sejam chamados no próprio tipo. Eles são chamados de métodos de tipo. Os

métodos de tipo são úteis para trabalhar com informações de nível de tipo. Por exemplo, imagine a seguinte struct chamada de `Square`.

```
struct Square {
    static func numberOfSides() -> Int {
        return 4
    }
}
```

Para tipos de valor, você indicou que está definindo um método de tipo com a palavra-chave `static`. O método `numberOfSides()` apenas retorna o número de lados que um `Square` pode ter.

Como distinção, os métodos de tipo em classes usam a palavra-chave `class`. Imagine, por exemplo, que você queira um método de tipo para a classe `Zombie` que represente o barulho assustador que todos os zumbis fazem.

```
class Zombie: Monster {
    class func makeSpookyNoise() -> String {
        return "Brains..."
    }

    var walksWithLimp = true

    override func terrorizeTown() {
        town?.changePopulation(-10)
    }
}
```

Para usar esses métodos de tipo, basta chamá-los no próprio tipo. Dê uma olhada no código abaixo para ver exemplos disso:

```
let sides = Square.numberOfSides() // sides is 4
let spookyNoise = Zombie.makeSpookyNoise() // spookyNoise is "Brains..."
```

Os métodos de tipo podem trabalhar com informações de nível de tipo em um dado tipo. Isso significa que os métodos de tipo podem chamar outros métodos de tipo e podem até mesmo trabalhar com propriedades de tipo, que iremos ver no capítulo sobre Propriedades. Observe, contudo, que os métodos de tipo não podem chamar métodos de instância nem trabalhar com nenhuma propriedade de instância. O motivo por trás dessa limitação é que uma instância não está disponível para ser usada no nível de tipo.

Para os mais curiosos: Currying de função

Depois de trabalhar com este capítulo, você pode estar se perguntando sobre a palavra-chave `mutating`. Por que ela é necessária para permitir que você modifique uma struct ou enum? Acontece que entender um novo conceito chamado de *currying de função* ajuda a explicar a resposta.

Crie um novo playground chamado de `Curry.playground` e salve-o onde quiser.

O currying de função permite reescrever uma função existente que aceita múltiplos parâmetros como uma nova função que aceita um parâmetro e retorna outra função. A função retornada aceita os parâmetros restantes da função original e também retorna o que a função original retorna. Esse processo de funções de aninhamento, cada um com o número de parâmetros restantes, continua até que não haja nenhum parâmetro restante.

A função reescrita é chamada de *função curried*. Uma função curried aplica parcialmente uma função existente. Ou seja, uma função curried permite que você ligue valores aos argumentos de uma função antes de chamá-la. Esse recurso de funções curried é similar ao fornecimento de valores padrão aos parâmetros de uma função, mas é muito mais dinâmico.

Considere uma simples função que retorna uma saudação de `String`.

```
func greetName(greeting: String, name: String) -> String {
    return "\(greeting) \(name)"
}
```

A função **greetName** aceita dois argumentos: um `greeting` e um `name`. Ela constrói e retorna uma saudação com base nesses dois argumentos. Essa função é simples de usar.

```
func greetName(greeting: String, name: String) -> String {  
    return "\(greeting) \(name)"  
}  
  
let personalGreeting = greetName("Hello,", "Matt")  
println(personalGreeting)
```

Você pode reescrever **greetName** para que seja uma função curried.

```
func greetName(greeting: String, name: String) -> String {  
    return "\(greeting) \(name)"  
}  
  
let personalGreeting = greetName("Hello,", "Matt")  
println(personalGreeting)  
  
func greetingForName(greeting: String) -> (String) -> String {  
    func greet(name: String) -> String {  
        return "\(greeting) \(name)"  
    }  
    return greet  
}
```

A função **greetingForName** aceita um argumento, o `String` `greeting`, e retorna uma função. Essa própria função retornada aceita um `String`, representando o `name` da saudação, e retorna uma saudação de `String` para o `name` dado. Você definiu uma função aninhada chamada de **greet** dentro da implementação para **greetingForName**. O tipo da função de **greet** corresponde ao tipo especificado pela **greetingForName**; ele aceita um `String` e retorna um `String`. Observe que você combinou o parâmetro `greeting` com o parâmetro `name` das duas funções para construir a saudação personalizada. Finalmente, você retornou a função **greet**.

Adicione o código abaixo para usar a função curried.

```
func greetName(greeting: String, name: String) -> String {  
    return "\(greeting) \(name)"  
}  
  
let personalGreeting = greetName("Hello,", "Matt")  
println(personalGreeting)  
  
func greetingForName(greeting: String) -> (String) -> String {  
    func greet(name: String) -> String {  
        return "\(greeting) \(name)"  
    }  
    return greet  
}  
  
let greeterFunction = greetingForName("Hello,")  
let greeting = greeterFunction("Matt")  
println(greeting)
```

Chama-se a função **greetingForName** e passa-se a saudação desejada ("Hello,"). O resultado é atribuído a uma constante chamada de `greeterFunction`. `greeterFunction` contém uma função que corresponde ao tipo de retorno de **greetingForName**: ele aceita um `String` e retorna um `String`. A saudação específica, "Hello,", é passada juntamente no escopo delimitador da função **greet** que é retornada por **greetingForName**.

Para criar uma saudação personalizada para um nome específico, chama-se a função **greet** e passa-se a ela um nome ("Matt") a seu único parâmetro. O resultado dessa função é atribuído a `greeting`, o qual é registrado no console. Você verá o mesmo resultado registrado no console.

Felizmente, a linguagem Swift oferece uma sintaxe mais conveniente para escrever funções curried. O exemplo abaixo é equivalente ao que você acabou de escrever.

```
func greetName(greeting: String, name: String) -> String {
    return "\(greeting) \(name)"
}

let personalGreeting = greetName("Hello,", "Matt")
println(personalGreeting)

func greetingForName(greeting: String) -> (String) -> String {
    func greet(name: String) -> String {
        return "\(greeting) \(name)"
    }
    return greet
}

let greeterFunction = greetingForName("Hello,")
let greeting = greeterFunction("Matt")
println(greeting)

func greeting(greeting: String)(name: String) -> String {
    return "\(greeting) \(name)"
}

let friendlyGreeting = greeting("Hello,")
let newGreeting = friendlyGreeting(name: "Matt")
println(newGreeting)
```

A sintaxe da função `greeting` parece um pouco diferente do que acabamos de ver. Dessa vez, você separou cada argumento delimitando cada parâmetro dentro de seus próprios parêntesis. Observe que essa sintaxe é muito mais concisa, mas funciona da mesma maneira.

Chamar essa função curried funciona de forma muito similar ao que acabamos de escrever. Você chamou `greeting` e passou uma saudação de `String` para seu primeiro argumento. A função resultante foi atribuída a uma constante chamada de `friendlyGreeting`. Em seguida, você chamou a função em `friendlyGreeting` e passou um nome para a saudação. Observe que o segundo argumento, `name`, recebeu automaticamente um nome externo que você tinha que usar.

Verifique o console agora. Você verá que os resultados são iguais.

Agora que você entendeu como o currying de função trabalha, você está apto a examinar como a palavra-chave `mutating` funciona. De início, crie uma nova struct chamada de `Person`.

```
...
let friendlyGreeting = greeting("Hello,")
let newGreeting = friendlyGreeting(name: "Matt")
println(newGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeName(fn: String, ln: String) {
        firstName = fn
        lastName = ln
    }
}
```

Nada muito especial ou estranho está acontecendo aqui. A struct `Person` possui propriedades relacionadas ao primeiro e ao último nome de uma pessoa. Ela também define uma função mutável para alterar essas propriedades.

Crie uma nova instância de `Person`.

```
...
let friendlyGreeting = greeting("Hello,")
let newGreeting = friendlyGreeting(name: "Matt")
println(newGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeName(fn: String, ln: String) {
        firstName = fn
        lastName = ln
    }
}
var p = Person()
```

Não há nada de novo aqui também, mas é quando tudo começa a ficar interessante. Acontece que os métodos de instância da linguagem Swift, os mesmos que você aprendeu neste capítulo, são, na verdade, funções curried. Digite o código a seguir para ver isso em ação.

```
...
let friendlyGreeting = greeting("Hello,")
let newGreeting = friendlyGreeting(name: "Matt")
println(newGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

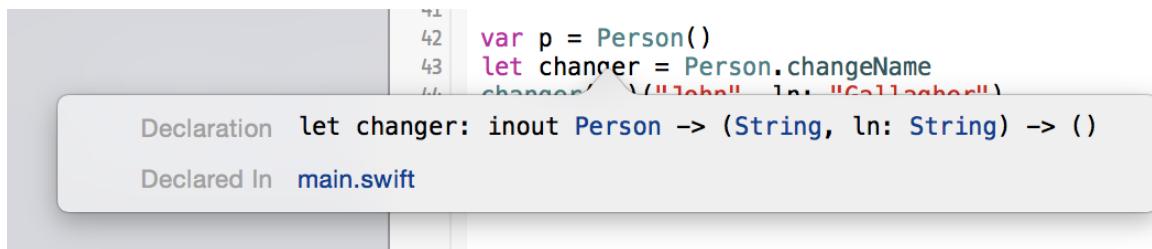
    mutating func changeName(fn: String, ln: String) {
        firstName = fn
        lastName = ln
    }
}

var p = Person()
let changer = Person.changeName
```

Você pode acessar a função **changeName** na struct **Person**. Observe que você não está chamando a função **changeName**. Em vez disso, você está atribuindo-a a uma constante chamada de **changer**.

Afinal, o que é **changer**? Para descobrir, segure a tecla option e clique nela. Você verá algo semelhante à Figure 15.6.

Figure 15.6 Assinatura de uma função curried



O que a assinatura significa? Resumidamente, ela indica que **changer** é uma função curried. Mais especificamente, **changer** contém uma função cujo único argumento é uma instância da struct **Person** passada como parâmetro **inout**. Essa função retorna uma função que aceita dois argumentos, uma **String** para o novo primeiro nome e uma **String** para o novo último nome. A função resultante não retorna nada.

Lembre-se do Chapter 12 em que aprendemos que o parâmetro `inout` permite que uma função modifique o valor passado para o parâmetro. As mudanças no parâmetro `inout` feitas dentro da função também persistem fora da função após ela ser chamada. Em outras palavras, as modificações substituem o valor original do parâmetro.

Juntando todas essas informações, uma função mutável é simplesmente uma função curried cujo primeiro argumento é `self` passado como parâmetro `inout`. Como os tipos de valor são copiados quando são passados, para métodos não mutáveis, `self` é, na verdade, uma cópia do valor. Para fazer mudanças, `self` precisa ser declarado como `inout` e `mutating` é o modo como a linguagem Swift permite realizar isso.

O tipo a seguir demonstra esse ponto para ver `change` em ação.

```
...
let friendlyGreeting = greeting("Hello,")
let newGreeting = friendlyGreeting(name: "Matt")
println(newGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeName(fn: String, ln: String) {
        firstName = fn
        lastName = ln
    }
}

var p = Person()
let changer = Person.changeName
changer(&p)("John", ln: "Gallagher")
println(p.firstName) // John
```

Chama-se a função de `changer`, passando a instância de `Person` que você quer modificar. Lembre-se de que você precisa prefixar os parâmetros `inout` com um & para garantir que consiga passar a referência da instância para a função. Em seguida, atribui-se duas strings ("John" e "Gallagher") aos dois parâmetros finais da função curried, um para cada um dos nomes. Essas strings são usadas para modificar os valores da instância `Person` para essas propriedades. Finalmente, exibe-se o resultado da chamada de função para confirmar que o primeiro nome de `p` foi alterado para John.

O que quisemos mostrar com esta seção foi o que a palavra-chave `mutating` faz. Na prática, você provavelmente não vai querer usar currying de função para mudar uma `struct`. Vá em frente e remova o código de currying de função e utilize a função `changeName` de maneira mais direta. O resultado será exatamente o mesmo.

```
...
let friendlyGreeting = greeting("Hello,")
let newGreeting = friendlyGreeting(name: "Matt")
println(newGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeName(fn: String, ln: String) {
        firstName = fn
        lastName = ln
    }
}

var p = Person()
let changer = Person.changeName
changer(&p)("John", ln: "Gallagher")
p.changeName("John", ln: "Gallagher")
println(p.firstName) // John
```


16

Propriedades

O capítulo anterior apresentou as propriedades de maneira muito limitada. O foco era as estruturas e as classes. Entretanto, os tipos precisam de propriedades para que possam modelar dados em seu aplicativo. Sendo assim, você atribuiu algumas propriedades armazenadas muito básicas a seus tipos personalizados para que tivessem dados para representar. Este capítulo desenvolverá ainda mais seu entendimento das propriedades e aprofundará sua compreensão de como usá-las com seus tipos personalizados.

Como vimos, as propriedades são usadas para associar valores a um dado tipo. As propriedades podem aceitar valores de constante ou variável para um tipo. Classes, estruturas e enumerações podem todas ter propriedades. Resumidamente, as propriedades são usadas para modelar as características da entidade que o tipo deve representar.

As propriedades podem ter duas versões: 1) armazenadas e 2) computadas. As propriedades armazenadas podem receber valores padrão, e as propriedades computadas podem retornar o resultado de algum cálculo com base em informações disponíveis. É possível observar as propriedades para ver quando forem alteradas e subsequentemente executar um código específico quando a propriedade for definida com um novo valor. É possível até mesmo estabelecer regras que determinem a visibilidade das propriedades a outros arquivos em seu aplicativo. De fato, as propriedades são muito poderosas e flexíveis. Vamosvê-las.

Propriedades armazenadas básicas

As propriedades armazenadas são propriedades na forma mais básica. Para ver como elas funcionam, você vai expandir o comportamento dos tipos desenvolvidos no Chapter 15. Faça uma cópia do seu projeto `MonsterTown.xcodeproj` e salve-a em um local conveniente.

Depois de fazer uma cópia do projeto, abra o arquivo `Town.swift`. Lembre-se da propriedade `population: var population = 5422`. Esse código indica três itens importantes.

- `var` marca essa propriedade como variável, o que significa que ela pode ser modificada.
- `population` tem um valor padrão igual a 5422.
- `population` é uma propriedade armazenada cujo valor pode ser lido e definido.

`population` é referida como propriedade armazenada porque é usada para conter um tipo específico de informação: a população da cidade. É isso que as propriedades armazenadas fazem; elas armazenam dados.

Um ponto importante que deve ser reiterado é o fato de a propriedade armazenada acima, `population`, ser do tipo "leitura/gravação". Em outras palavras, é possível ler e definir o valor da propriedade. Também é possível criar propriedades armazenadas como "somente leitura"; ou seja, você pode criar uma propriedade que não permite que o usuário altere o valor.

Use `let` com uma propriedade armazenada para criar uma propriedade "somente leitura". Imagine, por exemplo, que a cidade específica que você precisa modelar em seu aplicativo seja sempre da região sul dos EUA. Uma propriedade armazenada de constante é uma boa opção para esse caso.

```
struct Town {
    let region = "South"
    var population = 5422
    var numberOfWorkLights = 4

    func printTownDescription() {
        println("Population: \(population); number of stop lights: \(numberOfWorkLights)")
    }

    mutating func changePopulation(amount: Int) {
        population += amount
    }
}
```

`region` é uma propriedade armazenada que armazena a região em que a cidade se encontra. Nesse caso, você fez de `region` uma constante, o que significa que a propriedade não pode ser alterada. Essa imutabilidade efetivamente faz com que `region` seja uma propriedade somente leitura.

Tipos aninhados

Os tipos aninhados são tipos definidos dentro de outro tipo delimitador. Muitas vezes, são usados para dar suporte à funcionalidade de um tipo e não devem ser usados separados desse tipo. As enumerações são usadas frequentemente assim.

Dentro de `Town.swift`, crie uma nova enumeração chamada de `Size`. Você usará essa enumeração, juntamente com outra propriedade nova que será adicionada depois, para calcular se uma cidade pode ou não ser designada como pequena, média ou grande. Lembre-se de definir a enum dentro da definição da struct `Town`:

```
struct Town {
    let region = "South"
    var population = 5422
    var numberOfWorkLights = 4

    enum Size: String {
        case Small = "Small"
        case Medium = "Medium"
        case Large = "Large"
    }

    func printTownDescription() {
        println("Population: \(population); number of stop lights: \(numberOfWorkLights)")
    }

    mutating func changePopulation(amount: Int) {
        population += amount
    }
}
```

Observe que `Size` é definido entre chaves (`{}`) da definição da struct `Town`. `Size` tem valores brutos de `String` associados a cada caso que descreve o tamanho da cidade. Os valores padrão do caso ajudarão a descrever o tamanho da cidade ao exibir no console.

`Size` também tem uma função que transformará o caso da enum atual em um valor de `String` útil.

`Size` não será usado fora de `Town`. Na verdade, a definição sugere que `Size` usará uma instância do tipo `Town` para determinar em qual tamanho de cidade a instância se encontra. Essa realidade sugere que a instância de `Town` precisará de um valor em sua propriedade `population` antes de usar o tipo aninhado. No entanto, todas as propriedades com as quais trabalhamos até aqui calcularam o valor da propriedade quando a instância foi criada. A próxima seção introduz um novo tipo de propriedade que retarda a computação de seu valor para que seja calculado quando as informações corretas estiverem disponíveis.

Propriedades armazenadas lentas

Às vezes, o valor de uma propriedade armazenada não pode ser atribuído imediatamente. O desenvolvedor pode não querer computar os valores das propriedades imediatamente quando a instância for criada porque a computação pode ser dispendiosa em relação à memória ou ao tempo. Outra situação potencial pode envolver uma propriedade que dependa de fatores externos ao tipo específico, que serão desconhecidos antes de a instância ser criada. Essas circunstâncias requerem algo chamado de "carregamento lento".

Em termos de propriedades, o carregamento lento significa que o cálculo do valor da propriedade não será realizado até que seja necessário pela primeira vez. Essa demora adiará a computação do valor da propriedade até que a instância seja inicializada. Assim, as propriedades `lazy` devem ser declaradas com `var` porque seus valores mudarão.

Crie uma nova propriedade `lazy` chamada de `townSize`. Essa propriedade será do tipo `Size`; ou seja, seu valor será uma instância da enum `Size`. Lembre-se de definir essa nova propriedade dentro do tipo `Town`.

```
enum Size: String {
    case Small = "Small"
    case Medium = "Medium"
    case Large = "Large"
}
lazy var townSize: Size = {
    precondition(self.population >= 0, "Town cannot have negative population.")
    switch self.population {
        case 0...10000:
            return Size.Small
        case 10001...100000:
            return Size.Medium
        default:
            return Size.Large
    }
}()
```

Comparada às propriedades que você escreveu anteriormente, `townSize` parece diferente por diversos motivos. Primeiro, observe que você marcou `townSize` como `lazy`. Isso significa que o valor de `townSize` só será calculada depois de ser acessado pela primeira vez. Retardar a computação dessa propriedade faz sentido porque o valor depende da `population` da instância.

Além disso, o tipo da propriedade é `Size`, como mencionado acima. A diferença aqui é que o valor dessa propriedade não será definido de forma direta. Por exemplo, você não escreverá o código assim: `myTown.townSize = Size.Small`. Em vez disso, você tirará proveito do tipo aninhado `Size` juntamente com um fechamento para calcular o tamanho da cidade dada sua população.

Por último, observe esta linha de código: `lazy var townSize: Size = {}`. O que está acontecendo aqui? Para entender, é importante se lembrar de que funções e fechamentos são "tipos de primeira classe". Uma consequência desse fato é que as propriedades podem se referir a funções e fechamentos. Com esse conhecimento revigorado e em mãos, dê uma olhada na definição de `townSize`.

`townSize` define um valor de propriedade padrão, isto é, o tamanho da cidade, com o resultado retornado por um fechamento. Um fechamento funciona bem aqui porque o valor da `population` da cidade é necessário para determinar o tamanho dessa cidade. Assim, o fechamento, que é definido entre chaves `({})`, usa uma instrução `switch` para determinar o tamanho da cidade. Logo, o fechamento alterna a `population` (`self.population`) da instância.

Observe que você adicionou uma `precondition` antes da instrução `switch`. Essa `precondition` garante que a instância da cidade tenha uma população viável para `switch`. Ou seja, ela garante que a cidade não tenha uma `population` negativa, que é impossível e que não seria capturada pelos casos de `switch`.

Dentro da instrução `switch`, você especificou três casos: 1) `0...10000` é uma cidade pequena, 2) `10001...100000` é uma cidade média e 3) um caso `default` que capturará qualquer `population` maior que `100000` e descreverá a cidade como grande. Cada caso retorna uma instância da enum `Size`.

Também observe que o fechamento para `townSize` termina com parêntesis vazios (). Esses parênteses, combinados com a marcação lenta, garantem que a linguagem Swift chamará o fechamento e atribuirá o resultado que retorna a `townSize` quando a propriedade for acessada pela primeira vez. Se você tivesse omitido os parêntesis, simplesmente estaria atribuindo um fechamento à propriedade `townSize`. Assim, o fechamento será executado na primeira vez que acessar a propriedade `townSize`.

Finalmente, `townSize` precisa ser marcada de `lazy` porque o fechamento precisa alternar, ou `switch`, a propriedade `population` da instância. Se um fechamento trabalhar com as propriedades de uma instância, o compilador exigirá que o fechamento se refira a `self` ao acessar qualquer propriedade nessa instância. Isso é devido ao fato de que se referir a `self` dentro de um fechamento pode provocar vazamento de memória. (Não se preocupe com problemas de gerenciamento de memória por enquanto; discutiremos esse assunto mais tarde no livro.) Além disso, lembre-se de que `lazy` significa que a computação do valor da propriedade é adiada até que a instância seja inicializada. Portanto, como o fechamento precisa se referir a `self` para obter acesso à propriedade `population` das instâncias, isso significa que a propriedade `townSize` precisa ser marcada como `lazy` para garantir que a instância (que `self` se refere) esteja completamente preparada para trabalhar.

Alterne para `main.swift` para exercitar essa propriedade `lazy`.

```
var myTown = Town()
let ts = myTown.townSize
println(ts.rawValue)
myTown.changePopulation(500)
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
let z1 = Zombie()
z1.walksWithLimp = false
let z2 = z1
z2.walksWithLimp = true
println("z1 walks with limp? \$(z1.walksWithLimp); z2 walks with limp? \$(z2.walksWithLimp)")
fredTheZombie.changeName("Fred the Zombie", walksWithLimp: false)
```

Depois de criar uma instância, você criou uma constante chamada de `ts` para conter as informações de tamanho da `myTown`. Essa linha acessa a propriedade lenta `townSize` e faz com que seu fechamento seja executado. Após o fechamento alternar a `population` de `myTown`, uma instância da enum `Size` é atribuída a `ts`. Em seguida, você usou a propriedade `rawValue` em `ts` para registrar o valor bruto no console. Como resultado, você deve ver `Small` registrado no console.

É importante observar que as propriedades marcadas com `lazy` só serão calculadas uma vez. Esse recurso de `lazy` indica que alterar o valor da `population` de `myTown` nunca fará com que `townSize` de `myTown` seja recalculado.

Aumente a `population` de `myTown` em 1000000 e verifique o tamanho de `myTown` registrando-a no console. Vá em frente e inclua a `population` de `myTown` na instrução de registro para que se refira ao que está informado para `townSize`.

```
var myTown = Town()

let ts = myTown.townSize
println(ts.rawValue)

myTown.changePopulation(500)
myTown.changePopulation(1000000)
println("Size: \$(myTown.townSize.rawValue); population: \$(myTown.population)")
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
let z1 = Zombie()
z1.walksWithLimp = false
let z2 = z1
z2.walksWithLimp = true
println("z1 walks with limp? \$(z1.walksWithLimp); z2 walks with limp? \$(z2.walksWithLimp)")
fredTheZombie.changeName("Fred the Zombie", walksWithLimp: false)
```

Você verá o seguinte registro no console: `Size: Small; population: 1005422`. Como previsto, o tamanho de `myTown` não mudou mesmo após o aumento intenso de sua `population`. Essa discrepância é devida à natureza `lazy` de `townSize`. A propriedade será calculada apenas no primeiro acesso e não será recalculada depois disso.

Essa discrepancia entre a `population` de `myTown` e `townSize` é indesejável. Além disso, parece que `townSize` não deveria ser marcado com `lazy`, especialmente se `lazy` significar que `myTown` não será capaz de recalibrar seu `townSize` para refletir as mudanças na `population`. Na verdade, uma *propriedade computada* é uma opção bem melhor.

Propriedades computadas

Você pode usar as propriedades computadas com qualquer classe, struct ou enum que definir. As propriedades computadas não armazenam valores como as propriedades com as quais você trabalhou até aqui. Em vez disso, uma propriedade computada oferece um getter e um setter opcional para recuperar ou definir o valor da propriedade. Para deixar isso mais prático, troque sua definição da propriedade `townSize` no tipo `Town` por uma propriedade computada somente leitura.

```
lazy var townSize: Size = {
    precondition(self.population >= 0, "Town cannot have negative population.")
    switch self.population {
        case 0...10000:
            return Size.Small
        case 10001...100000:
            return Size.Medium
        default:
            return Size.Large
    }
}

var townSize: Size {
    get {
        precondition(self.population >= 0, "Town cannot have negative population.")
        switch self.population {
            case 0...10000:
                return Size.Small
            case 10001...100000:
                return Size.Medium
            default:
                return Size.Large
        }
    }
}
```

`townSize` agora está definido como propriedade computada. Ela oferece um getter personalizado que usa a mesma instrução `switch` usada na definição anterior baseada em fechamento dessa propriedade. Observe que você declarou explicitamente o tipo da propriedade computada como sendo `Size`. Você deve fornecer propriedades computadas com suas informações de tipo. Essas informações ajudam o compilador a saber qual getter da propriedade deve retornar. Também é importante mencionar que as propriedades computadas devem ser declaradas como variáveis com a palavra-chave `var`.

Como antes, essa propriedade é acessada por meio da sintaxe de ponto: `myTown.townSize`. Dessa forma, o getter será executado para `townSize`, o que resultará no uso da `population` de `myTown` para calcular o `townSize`. Execute o programa novamente da maneira que está atualmente. Você verá `Size: Large; population: 1005422` registrado no console após aumentar a população da cidade em 1000000.

`townSize` é uma propriedade computada somente leitura. Em outras palavras, `townSize` não pode ser definida diretamente. Ela só pode recuperar e retornar um valor com base no cálculo definido no getter. Uma propriedade

somente leitura é perfeita nesse caso se você quiser que `myTown` calcule sua `townSize` com base na `population` da instância.

Getter e setter

As propriedades computadas também são declaradas com um getter e um setter, tornando a propriedade uma propriedade de leitura/gravação. Abra seu `Monster.swift` e adicione a propriedade computada a seguir à declaração de `Monster`.

```
class Monster {
    var town: Town? = Town()
    var name = "Monster"
    var victimPool: Int {
        get {
            return town?.population ?? 0
        }
        set(newVictimPool) {
            town?.population = newVictimPool
        }
    }
    func terrorizeTown() {
        if let t = town {
            println("\(name) is terrorizing a town!")
        } else {
            println("\(name) hasn't found a town to terrorize yet...")
        }
    }
}
```

Imagine que você precise que cada instância de `Monster` seja capaz de monitorar o pool de vítimas em potencial. Naturalmente, esse número corresponderá à população da cidade assombrada pelo monstro. De maneira correspondente, `victimPool` é uma nova propriedade computada com um getter e um setter. Como antes, você a declarou como `var` e atribuiu a ela informações de tipo específicas. Nesse caso, `victimPool` é um `Int`.

As informações de tipo da propriedade é seguida de chave (`{}`). A definição da propriedade é escrita entre essa chave de abertura e a chave de fechamento (`}`). Você definiu um getter para a propriedade por meio do mesmo `get`. O getter usa o operador de união `nil` para verificar se a instância `Monster` está ocupando uma cidade naquele momento. Se estiver, ele retorna o valor da população dessa cidade. Se a instância ainda não tiver achado uma cidade para aterrorizar, simplesmente retorna 0.

O setter para a propriedade computada é escrito dentro do bloco `set`. Observe a nova sintaxe: `set(newVictimPool)`. A especificação de `newVictimPool` entre parêntesis significa que você está fornecendo um novo valor com nome explícito. Você pode se referir a essa variável dentro da implementação do setter. Por exemplo, você usou o encadeamento de opcionais para garantir que a instância `Monster` encontrasse uma cidade e, então, definiu que a população dessa cidade tinha que corresponder a `newVictimPool`. Se você não tivesse nomeado explicitamente o novo valor, a linguagem Swift teria dado uma variável chamada de `newValue` para conter as mesmas informações.

Volte para `main.swift` para usar essa nova propriedade computada. Adicione o código abaixo no final do arquivo.

```
println("Victim pool: \(fredTheZombie.victimPool)")
fredTheZombie.victimPool = 500
println("Victim pool: \(fredTheZombie.victimPool)")
```

A primeira linha nova exerce o getter para a propriedade computada. Execute o programa e você verá `Victim pool: 1005412` registrado no console. A próxima linha nova usa o setter para alterar o `victimPool` de `fredTheZombie`: `fredTheZombie.victimPool = 500`. Por último, `victimPool` é registrado novamente no console por meio do getter da propriedade. Dê uma olhada no console. O `victimPool` deve ser atualizado para ser igual a 500.

Observadores de propriedades

A linguagem Swift tem um recurso interessante chamado de *observação de propriedades*. Os observadores de propriedades monitoram e respondem a mudanças em uma dada propriedade. A observação de propriedades está disponível para qualquer propriedade armazenada definida e também está disponível para qualquer propriedade herdada. Você não pode usar os observadores de propriedades com propriedades computadas definidas por você. Essa realidade não é tão limitadora assim se considerarmos que você tem total controle sobre a definição do setter e do getter dessa propriedade computada e pode, portanto, responder a qualquer alteração do jeito que quiser.

Imagine que os cidadãos de sua cidade sitiada estejam ficando muito inquietos. Eles exigem que o prefeito faça alguma coisa para proteger o povo contra a doença monstruosa que se espalha pelo campo. A primeira ação do prefeito é monitorar os ataques contra o povo da cidade. Os observadores de propriedades são perfeitos para essa tarefa.

É possível observar as mudanças feitas em uma propriedade de duas maneiras:: 1) quando uma propriedade está prestes a mudar: por meio de `willSet`; e 2) quando uma propriedade muda de fato: por meio de `didSet`. Para monitorar quantos ataques a cidade está sofrendo, o prefeito decide prestar muita atenção ao momento em que a `population` da cidade muda. Assim, um observador `didSet` deve ser usado, já que ele será notificado logo após a propriedade receber um novo valor.

Adicione o código abaixo a `Town.swift` para monitorar as mudanças feitas à propriedade `population`.

```
var population: Int = 5422 {
    didSet(oldPopulation) {
        println("The population has changed to \(population) from \(oldPopulation).")
    }
}
```

A sintaxe para observadores de propriedade são similares aos getters e setters das propriedades computadas. A resposta à mudança é definida entre chaves. No exemplo acima, você criou um nome de parâmetro personalizado para a população antiga: `oldPopulation`. O observador `didSet` dá a você uma noção sobre o valor antigo da propriedade. Como distinção, o observador `willSet` dá a você uma noção sobre o novo valor da propriedade. Se você não tivesse especificado um nome novo, a linguagem Swift teria dado o parâmetro `oldValue` automaticamente. De mesma forma, no caso do observador `willSet`, a linguagem Swift gera um parâmetro `newValue` para você.

Esse observador de propriedade simplesmente registra a informação de `population` da cidade no console sempre que ela mudar. Por exemplo, você verá um registro dessa mudança depois que `fredTheZombie` aterrorizar a cidade. Execute o programa e dê uma olhada no console. Você deve ver um registro sempre que a `population` mudar.

```
Small
The population has changed to 1005422 from 5422.
Size: Large; population: 1005422
The population has changed to 1005412 from 1005422.
Population: 1005412; number of stop lights: 4
z1 walks with limp? true; z2 walks with limp? true
Victim pool: 1005412
The population has changed to 500 from 1005412.
Victim pool: 500
```

Propriedades de tipos

Até agora, você trabalhou apenas com propriedades de instâncias. Sempre que você criar uma nova instância de um tipo, essa instância receberá suas próprias propriedades que são distintas das outras instâncias desse tipo. As propriedades de instâncias são úteis para armazenar e computar valores em uma instância de um tipo, mas e aqueles valores que pertencem ao próprio tipo?

Você pode definir *propriedades de tipos*. Elas são propriedades universais do tipo; os valores nessas propriedades serão compartilhados com todas as instâncias do tipo. Normalmente, essas propriedades armazenam informações que serão iguais em todas as instâncias. Por exemplo, todas as instâncias do tipo `Square` terão exatamente quatro lados.

Os tipos de valores (isto é, estruturas e enumerações) podem ambos aceitar propriedades de tipo armazenadas e computadas. Como com os métodos de tipo, as propriedades de tipo nos tipos de valor começarão com a palavra-chave `static`.

Lembre-se que anteriormente no capítulo você criou uma propriedade somente leitura de constante na `Town` para a `region` da cidade. Esse uso de uma propriedade de instância de constante é um pouco estranho. Afinal, cada instância de `Town` será na mesma região: a região sul. Uma propriedade de tipo funcionaria melhor para a propriedade `region`. Altere a `Town` para refletir essa revisão.

```
let region = "South"
static let region = "South"
```

As propriedades de tipos armazenadas devem receber um valor padrão. Por exemplo, `region` recebeu o valor `South`. Esse requisito faz sentido porque os tipos não têm inicializadores, que são o assunto do Chapter 17. Em outras palavras, a propriedade de tipo armazenada precisa ter todas as informações necessárias para fornecer seu valor a qualquer chamador.

Diferentemente dos tipos de valor, as classes só podem ter propriedades de tipos computadas. A sintaxe para uma propriedade de tipo de classe usa a palavra-chave `class`.

Em um capítulo anterior, você criou um método de tipo no tipo `Zombie` para criar um barulho assustador: `makeSpookyNoise()`. Observe que `makeSpookyNoise()` não aceita nenhum argumento. Esse fato torna o método de tipo um ótimo candidato a ser reescrito como propriedade de tipo computada. Revise o método de tipo para que seja uma propriedade de tipo computada.

```
class Zombie: Monster {
    class func makeSpookyNoise() -> String {
            return "Brains..."
    }
    class var spookyNoise: String {
            return "Brains..."
    }
}

var walksWithLimp = true

override func terrorizeTown() {
    town?.changePopulation(-10)
}

func changeName(name: String, walksWithLimp: Bool) {
    self.name = name
    self.walksWithLimp = walksWithLimp
}
}
```

Como você revisou o método de tipo para ser uma propriedade computada, você alterou o nome para `spookyNoise`. A definição da propriedade computada é muito similar ao método de tipo. As principais diferenças são a substituição da palavra-chave `func` por `var` e a remoção dos parêntesis indicando que essa propriedade não é um método. Do contrário, a propriedade de tipo computada seria escrita exatamente como qualquer outra propriedade computada. Por exemplo, você ainda tem que declarar a propriedade de tipo computada como `var`.

Um novo elemento no código acima é o uso da sintaxe de getter abreviada. Se você não quiser fornecer um setter para uma propriedade computada, é possível omitir o bloco `get` da definição da propriedade computada e simplesmente retornar o valor computado conforme necessário.

Alterne para `main.swift`. Adicione uma linha no final do arquivo para exibir a propriedade `spookyNoise` do tipo `Zombie` no console. Você verá que `Brains...` foi registrado.

```
println("Victim pool: \(fredTheZombie.victimPool)")
fredTheZombie.victimPool = 500
println("Victim pool: \(fredTheZombie.victimPool)")
println(Zombie.spookyNoise)
```

Controle de acesso

A intuição fundamental por trás do controle de acesso é o fato de você nem sempre querer que elementos do código de seu programa sejam visíveis para os outros elementos. Na verdade, você vai querer com frequência ter um controle mais preciso do acesso a seu código. Você pode conceder níveis específicos de acesso a diversos componentes do seu código.

O controle de acesso pode ser visto como a definição do nível de acesso a seu código obtido por outras partes de um programa. Por exemplo, você pode esconder ou expor um método em uma classe dependendo do que você quer compartilhar. Imagine que você tem uma propriedade usada apenas dentro da definição de uma classe. Poderia ser problemático se outro tipo externo modificasse essa propriedade por engano. Com o controle de acesso, você pode gerenciar a visibilidade dessa propriedade para escondê-la das outras partes do programa. Isso englobará os dados da propriedade e impedirá a intromissão de código externo nela.

O controle de acesso é organizado com base em dois conceitos importantes e relacionados: módulos e arquivos-fonte. Em relação aos arquivos e à organização do seu projeto, esses dois conceitos são fundamentais para seu aplicativo.

Um módulo é uma unidade de código distribuída em conjunto. Você deve se lembrar de ver `import UIKit` ou `import Cocoa` no topo dos seus Playgrounds. São frameworks, os quais agrupam uma série de tipos relacionados que ajudam a realizar uma série de tarefas relacionadas. Por exemplo, `UIKit` é um framework projetado para facilitar o desenvolvimento de interfaces de usuário. Os módulos são levados a outro módulo usando a palavra-chave `import` da linguagem Swift, conforme sugerido pelos exemplos acima.

Arquivos-fonte, por outro lado, são unidades mais discretas. Eles representam um único arquivo e permanecem dentro de um módulo específico. Em geral, você definirá um único tipo dentro de um arquivo-fonte. Isso ajudará a manter seu projeto organizado; apesar disso, essa prática não é uma exigência. É possível definir múltiplas funções e tipos dentro de um único arquivo, embora não seja uma estratégia de organização comum.

Table 16.1 Controle de acesso em Swift

Nível de acesso	Descrição
Público	O acesso público torna as entidades visíveis a todos os arquivos no módulo ou aqueles que importam o módulo.
Interno	O acesso interno (padrão) torna as entidades visíveis a todos os arquivos no mesmo módulo.
Privado	O acesso privado torna as entidades visíveis apenas dentro do arquivo-fonte que as define.

Como você deve inferir a partir da tabela, o acesso `public` é o que menos restringe e o acesso `private` é aquele com maior nível de restrição do controle de acesso. Em geral, o nível de acesso do tipo precisa ser consistente com os níveis de acesso de suas propriedades e métodos. Uma propriedade não pode ter um nível de restrição de controle de acesso menor do que seu tipo. Por exemplo, uma propriedade com nível de controle de acesso `internal` não pode ser declarada em um tipo com acesso `private`. Da mesma forma, a restrição do controle de acesso de uma função não pode ser menor do que a do controle de acesso listado para seus parâmetros. Em ambos os casos, a ideia é que uma entidade não pode ter menos visibilidade do que é declarado para suas partes constituintes.

A linguagem Swift especifica que `internal` é o nível padrão de controle de acesso para seu aplicativo. Ter um nível de acesso padrão significa que você não precisa declarar especificamente os controles de acesso para cada tipo, propriedade e método em seu código. Embora isso seja bastante conveniente, existem situações em que é melhor gerenciar o acesso.

Com esse conhecimento em mãos, é hora de pôr em prática. Crie uma propriedade `isFallingApart` Boolean definida no tipo `Zombie`. Atribua a ela o valor padrão `false`. Como esperado, essa propriedade monitorará a

integridade física de uma instância. De maneira correspondente, essa propriedade realmente não precisa ser exposta ao resto do programa; ela é um detalhe de implementação da classe `Zombie`.

```
class Zombie: Monster {
    class var spookyNoise: String {
        return "Brains..."
    }

    var walksWithLimp = false
    private var isFallingApart = false

    override func terrorizeTown() {
        town?.changePopulation(10)
        if !isFallingApart {
            town?.changePopulation(-10)
        }
    }

    func changeName(name: String, walksWithLimp: Bool) {
        self.name = name
        self.walksWithLimp = walksWithLimp
    }
}
```

Depois de criar a propriedade, você também fez uso dela na função `terrorizeTown()`. Se a instância estiver caindo aos pedaços, ela não poderá aterrorizar a cidade. Sendo assim, devemos verificar se `isFallingApart` é `false`. Se for `false`, a instância estará livre para aterrorizar a cidade. Se a instância estiver caindo aos pedaços, ela não será capaz de aterrorizar a cidade.

Conclusão

Este capítulo apresentou uma grande quantidade de material. Aproveite para reservar um momento para assimilar todas as ideias. Relembrando de maneira muito geral, você aprendeu sobre:

- Sintaxe de propriedades
- Propriedades armazenadas vs. computadas
- Propriedades somente leitura e leitura/gravação
- Carregamento lento e propriedades lentas
- Observadores de propriedades
- Propriedades de tipo
- Controle de acesso

Como vimos ao longo deste livro até aqui, as propriedades são mesmo um conceito central na programação em Swift. É uma boa ideia se familiarizar com os itens listados acima. Os desafios descritos abaixo o ajudarão a dominar os conceitos importantes.

Desafios

Desafio de bronze

O prefeito da sua cidade está ocupado. Os nascimentos e deslocamentos não requerem a atenção do prefeito. Afinal, a cidade está em crise! Registre apenas mudanças na população da cidade se a nova população for inferior ao valor antigo.

Desafio de prata

Crie um novo tipo chamado de `Mayor`. Ele deve ser uma struct. O tipo `Town` deve ter uma propriedade chamada `mayor` que contenha uma instância do tipo `Mayor`. Além disso, a cidade deve informar o `mayor` toda vez que a propriedade de `population` mudar. Se a `population` da cidade diminuir, a instância de `Mayor` deve registrar esta instrução no console: "*I'm deeply saddened to hear about this latest tragedy. I promise that my office is looking into the nature of this rash of violence.*" Se a população aumentar, o `mayor` não deve fazer nada. (DICA: Você deve definir um novo método de instância no tipo `Mayor` para completar esse desafio.)

Desafio de ouro

Prefeitos são também pessoas. Uma instância do tipo `Mayor` naturalmente ficará nervosa sempre que a cidade perder parte da `population` devido a um ataque de `Zombie`. Crie uma propriedade de instância armazenada no tipo `Mayor` chamado de `anxietyLevel`. Ela deve ser do tipo `Int` e também deve iniciar com um valor padrão igual a 0. Aumente a propriedade `anxietyLevel` sempre que uma instância de `Mayor` for informada de um ataque de `Zombie`. Por último, como o prefeito não vai querer mostrar que está ansioso, marque essa propriedade como `private`. Verifique que essa propriedade não pode ser acessada em `main.swift`.

17

Inicialização

A inicialização é a operação de configuração de uma instância de um tipo. Ela envolve a atribuição de um valor inicial a cada propriedade armazenada e pode abranger qualquer outro trabalho de preparação. Depois desse processo, a instância estará preparada e disponível para o uso.

Os tipos que você criou até aqui foram todos criados mais ou menos da mesma maneira. Os valores para as propriedades receberam valores armazenados padrão ou foram computados sob demanda. A inicialização não foi personalizada e não foi considerada especificamente.

É muito comum querer controlar como uma instância de um tipo é criada. Por exemplo, seria ideal que a instância tivesse todos os valores corretos em suas propriedades logo após você recebê-la. Anteriormente, você atribuiu valores padrão a uma instância para suas propriedades armazenadas e, depois disso, alterou os valores dessas propriedades depois de ter criado uma instância. Essa estratégia não é elegante. *Inicializadores* ajudam a criar uma instância com os valores adequados.

Sintaxe do inicializador

As estruturas e classes devem ter valores iniciais em suas propriedades armazenadas no momento que a inicialização é concluída. Esse requisito explica o porquê de você ter atribuído valores padrão a todas as suas propriedades armazenadas. Se você não tivesse atribuído valores padrão a essas propriedades armazenadas, o compilador emitiria erros dizendo que as propriedades do tipo não estão prontas para o uso. A definição de um inicializador no tipo garante que as propriedades tenham valores quando a instância for criada.

A sintaxe para escrever um inicializador é um pouco diferente do que já vimos. Os inicializadores são escritos com a palavra-chave `init`. Embora sejam métodos em um tipo, os inicializadores não são precedidos da palavra-chave `func`. O exemplo informal abaixo ilustra a sintaxe do inicializador.

```
struct CustomType {  
    init() {  
        // Initialization code here...  
    }  
}
```

No exemplo acima, o inicializador não aceita argumentos, conforme indicado pelos parêntesis vazios. A implementação do inicializador é definida entre chaves, como você já fez com funções e métodos comuns no decorrer deste livro. Ao contrário de outros métodos, os inicializadores não retornam valores. Em vez disso, eles possuem a função de atribuir valores às propriedades armazenadas de um tipo. Essa sintaxe geral não é diferente de estruturas (ou enumerações) e classes.

Inicialização de structs

Inicializadores padrão

Faça uma cópia do projeto `MonsterTown` e salve-a em um local conveniente. Quando você estiver pronto, abra o projeto e navegue até `main.swift`.

Você se lembra de como obtinha instâncias do seu tipo `Town` anteriormente? Você não criava seu próprio inicializador, mas atribuía valores padrão às propriedades armazenadas do tipo. Isso significava que você podia tirar proveito de um inicializador "vazio" livre oferecido pelo compilador da linguagem Swift. Assim, tudo o que você precisava fazer era digitar `var myTown = Town()`. Essa sintaxe chama o inicializador vazio e livre e define as propriedades da nova instância como os valores padrão especificados na definição do tipo.

Se você não fornecer um inicializador para sua struct personalizada, a linguagem Swift dará a você um inicializador "memberwise" livremente. Esse inicializador memberwise incluirá argumentos para todas as propriedades armazenadas que precisem de valores. Lembre-se de que um dos principais objetivos da inicialização é atribuir valores a todas as propriedades armazenadas do tipo para que a nova instância fique pronta para ser usada. O compilador reforçará a exigência de que sua nova instância possua valores em suas propriedades armazenadas por meio de valores padrão ou por inicialização de memberwise.

Em `main.swift`, substitua o uso do inicializador vazio no tipo `Town` por uma chamada ao inicializador memberwise livre. Deixe o restante desse arquivo inalterado.

```
var myTown = Town()
var myTown = Town(population: 10000, numberofStopLights: 6)
myTown.printTownDescription()
```

Clique no botão play no canto superior esquerdo para compilar e executar seu programa. Notou que o seu resultado é um pouco diferente? A descrição de `myTown` é diferente dos valores padrão que você atribuiu às propriedades armazenadas do tipo `Town`. Você verá `Population: 10000; number of stop lights: 6` registrado no console. Como os valores dessas propriedades mudaram com base nos valores padrão que você atribuiu?

Como podemos ver acima, a instância `myTown` foi criada com o inicializador memberwise livre. As propriedades armazenadas do tipo `Town` são listadas no inicializador, o que permitiu que você especificasse novos valores para as propriedades da instância. Depois disso, a exibição da descrição de `myTown` no console revelou que os valores padrão dessas propriedades foram substituídos pelos valores que você atribuiu ao inicializador.

Você também perceberá que os nomes da propriedade de `Town` são usados como nomes de parâmetro externo na chamada a esse inicializador. A linguagem Swift oferece nomes de parâmetro externo a cada inicializador automaticamente, um para cada parâmetro dado pelo inicializador. Essa convenção é importante porque todos os inicializadores da linguagem Swift possuem o mesmo nome: `init`. Portanto, o nome da função não pode ser usado para identificar qual inicializador específico deve ser chamado. Os nomes dos parâmetros e seus tipos ajudam o compilador a diferenciar os inicializadores, possibilitando que ele saiba qual inicializador deve chamar.

Inicializadores personalizados

É hora de escrever seu próprio inicializador para o tipo `Town`. Observe que, se você escrever qualquer um de seus próprios inicializadores para o seu tipo, a linguagem Swift não dará a você nenhum inicializador livre. A expectativa aqui é que se você está se responsabilizando pela inicialização do seu tipo, logo também é responsável por garantir que todas as propriedades das instâncias recebam os valores adequados.

Você removerá todos os valores padrão das propriedades. Eles foram úteis antes de você aprender sobre os inicializadores, já que garantiram que as propriedades para as instâncias do seu tipo tivessem valores quando uma instância fosse criada. Contudo, agora, eles não contribuem muito para a struct `Town`. Além disso, você reverterá `region` a uma propriedade de instância - a infestação de monstros está começando a se espalhar além da região sul.

Abra o arquivo `Town.swift` e remova os valores padrão das propriedades.

```

struct Town {
    static let region = "South"
    let region

    var population: Int = 5422 {
    var population: Int {
        didSet(oldPopulation) {
            println("The population has changed to \((population) from \((oldPopulation).")
        }
    }
    var numberofStopLights = 4
    var numberOfStopLights

    enum Size: String {
        case Small = "Small"
        case Medium = "Medium"
        case Large = "Large"
    }
    ...
}

```

Depois de excluir os valores padrão desses tipos, é possível que você tenha percebido que o compilador emitiu um erro: `Type annotation missing in pattern`. Anteriormente, você tirou proveito da inferência de tipo para essas propriedades, que funcionou bem com os valores padrão atribuídos a elas. No entanto, sem os valores padrão, o compilador não sabe qual informação de tipo atribuir às propriedades. Logo, como solução, devemos declarar explicitamente o tipo.

```

struct Town {
    let region
    let region: String
    var population: Int {
        didSet(oldPopulation) {
            println("The population has changed to \((population) from \((oldPopulation).")
        }
    }
    var numberofStopLights
    var numberOfStopLights: Int

    enum Size: String {
        case Small = "Small"
        case Medium = "Medium"
        case Large = "Large"
    }
    ...
}

```

Lembre-se de que o compilador da linguagem Swift dá a você um inicializador memberwise livre em structs que não definem um inicializador personalizado. Você vai criar esse inicializador manualmente. Depois, você vai chamar esse inicializador a partir de outro inicializador definido dentro desse mesmo tipo. Por enquanto, adicione o seguinte inicializador a seu tipo `Town`.

```

...
var numberOfStopLights: Int
init(region: String, population: Int, stopLights: Int) {
    self.region = region
    self.population = population
    numberOfStopLights = stopLights
}
enum Size: String {
    case Small = "Small"
    case Medium = "Medium"
    case Large = "Large"
}
...

```

O método `init` escrito acima aceita três argumentos, um para cada propriedade armazenada no tipo `Town`. Um item de interesse no inicializador acima é a passagem dos valores atribuídos aos argumentos do inicializador para

as propriedades de fato do tipo. Por exemplo, o valor passado ao argumento `region` do inicializador foi definido como o valor para a propriedade `region`. Como o nome do parâmetro no inicializador é o mesmo que o nome da propriedade, você precisou acessar explicitamente a propriedade por meio de `self`. `numberOfStopLights` não tem esse problema e, por isso, você conseguiu simplesmente definir o valor do argumento do inicializador de `stopLights` para a propriedade `numberOfStopLights`.

Outro item de interesse é que você foi capaz de definir o valor da propriedade `region` embora ela tenha sido declarada como constante. Você conseguiu fazer isso porque o compilador da linguagem Swift permite que você modifique o valor de uma propriedade constante em qualquer altura durante a inicialização. Lembre-se, o objetivo da inicialização é garantir que as propriedades de um tipo tenham valores após a conclusão da inicialização.

Neste momento, você pode estar vendo erros em `main.swift` e `Monster.swift` referentes aos inicializadores livres que o compilador estava atribuindo por padrão. Se você olhar o `main.swift` mais de perto, verá que o inicializador `memberwise` atribuído pelo compilador usou o nome real da propriedade de `numberOfStopLights` para o nome do argumento no inicializador. No inicializador acima, você abreviou esse nome de parâmetro para `stopLights`. Alterne para `main.swift`, altere o nome do parâmetro e adicione o parâmetro `region`:

```
var myTown = Town(population: 10000, numberOfStopLights: 6)
var myTown = Town(region: "West", population: 10000, stopLights: 6)
```

Em `Monster.swift`, você estava contando com o inicializador vazio livre dado pelo compilador porque todas as propriedades de `Town` tinham valores padrão. O inicializador vazio livre se foi agora que você está exigindo que `Town` seja inicializado explicitamente. De qualquer maneira, não faz nenhum sentido um monstro criar sua própria cidade, então altere `Monster` para começar com uma cidade para aterrorizar:

```
class Monster {
    var town: Town? = Town()
    var town: Town? = nil
}
```

Compile e execute o programa. O erro deve desaparecer e você verá o mesmo resultado de registro no console.

Delegação de inicialização

É possível definir inicializadores para chamarem outros inicializadores no mesmo tipo. Esse procedimento é chamado de *delegação de inicializadores*. Em geral, é usado para fornecer múltiplos caminhos para criar uma instância de um tipo.

Em tipos de valor (isto é, enumerações e estruturas), a delegação de inicializadores é relativamente simples. Como os tipos de valor não suportam herança, a delegação de inicializadores envolve apenas a chamada de outro inicializador definido no tipo. Já para classes, é um tanto mais complicado, como veremos em breve.

Alterne para `Town.swift` para escrever um novo inicializador nesse tipo que utilize delegação de inicializadores. Adicione o código a seguir:

```
init(region: String, population: Int, stopLights: Int) {
    self.region = region
    self.population = population
    numberOfStopLights = stopLights
}
init(population: Int, stopLights: Int) {
    self.init(region: "N/A", population: population, stopLights: stopLights)
}
enum Size: String {
    case Small = "Small"
    case Medium = "Medium"
    case Large = "Large"
}
```

No código acima, você definiu um novo inicializador no tipo `Town`. Esse inicializador, contudo, é diferente daquele que você criou antes. Ele aceita apenas dois argumentos: 1) `population` e 2) `stopLights`. E a propriedade `region`? Como está sendo definida?

Dê uma olhada na implementação desse novo inicializador. Você está chamando o outro inicializador de `Town` no `self.init(region: "N/A", population: population, stopLights: stopLights)`. Observe que você passou os argumentos fornecidos para `population` e `stopLights`. Como você não tinha um argumento para `region`, você precisou fornecer seu próprio valor. Nesse caso, você especificou a string "`N/A`" para indicar que não havia informações de região atribuídas ao inicializador.

A delegação de inicializadores ajuda a evitar a duplicação do código. Em vez de ter que digitar novamente o mesmo código para atribuir os valores passados nos argumentos do inicializador às propriedades do tipo, você pode simplesmente chamar outro inicializador no tipo. Além de ter que digitar a mesma coisa duas vezes, se você precisasse alterar o código duplicado, teria que alterar ambos os lugares (o que abre espaço para possíveis bugs quando você só atualiza um). A delegação de inicializadores também ajuda a definir um caminho pelo qual um tipo cria uma instância.

Como você definiu seu próprio inicializador `memberwise`, o compilador não dará nenhum inicializador livre. Essa realidade não é tão limitante assim. Ela pode até mesmo ser um benefício. Por exemplo, você pode querer usar esse novo inicializador se não houver informação de região disponível para determinada cidade que queira criar. Nesses casos, você usaria seu novo inicializador útil com argumentos para `population` e `stopLights` para definir as propriedades correspondentes enquanto atribui a `region` um valor de placeholder.

Utilize esse novo inicializador em `main.swift`.

```
var myTown = Town(region: "West", population: 10000, stopLights: 6)
var myTown = Town(population: 10000, stopLights: 4)
myTown.printTownDescription()
```

Se você compilar e executar seu aplicativo, o resultado será o mesmo, mas com uma importante diferença. Você não está mais definindo especificamente a `region` da instância.

Inicialização de classes

A sintaxe geral para a inicialização em classes é muito similar à inicialização em tipos de valores. Entretanto, existem algumas regras diferentes para classes que devem ser observadas. Essas regras adicionais são devidas principalmente ao fato de que as classes podem ser herdeiras de outras classes, o que necessariamente adiciona certa complexidade à inicialização.

Em particular, classes adicionam conceitos de inicializadores designados e de conveniência. Inicializadores designados são responsáveis por garantir que todas as propriedades de uma instância tenham valores antes da conclusão da inicialização, tornando, assim, a instância pronta para o uso. Inicializadores de conveniência auxiliam inicializadores designados, complementando-os ao chamar uma classe para seu inicializador designado. O papel dos inicializadores de conveniência, em geral, é criar uma instância de uma classe para um caso de uso muito específico.

Inicializadores padrão

Você já viu exemplos de uso do inicializador padrão de uma classe. As classes recebem um inicializador vazio padrão se você fornecer valores padrão a todas as propriedades e não escrever seu próprio inicializador. As classes não recebem um inicializador `memberwise` livre como as structs. Isso explica por que você atribuiu valores padrão a suas classes anteriormente; foi possível tirar proveito do inicializador vazio livre. Assim, você pôde receber uma instância de `Zombie` da seguinte maneira: `let fredTheZombie = Zombie()`, com os parêntesis indicando que você estava usando o inicializador padrão.

Inicialização e herança de classe

Abra o `Monster.swift`. Modifique a classe para dar a ela um inicializador. Perceba que você também está removendo o valor padrão de "`Monster`" da propriedade `name`.

```
class Monster {
    var town: Town? = nil
    let name = "Monster"
    let name: String

    var victimPool: Int {
        get {
            return town?.population ?? 0
        }
        set(newVictimPool) {
            town?.population = newVictimPool
        }
    }
    init(town: Town?, monsterName: String) {
        self.town = town
        name = monsterName
    }
    func terrorizeTown() {
        if let t = town {
            println("\(name) is terrorizing a town!")
        } else {
            println("\(name) hasn't found a town to terrorize yet...")
        }
    }
}
```

Esse inicializador tem dois argumentos: um para uma instância opcional do tipo `Town` e outro para o nome do monstro. Os valores para esses argumentos são atribuídos às propriedades da classe dentro da implementação do inicializador. Novamente, observe que o argumento para `town` no inicializador corresponde ao nome da propriedade na classe, então é necessário definir o valor da propriedade acessando-a por meio de `self`. Você também foi capaz de contornar acessando `name` por meio da variável `self` porque o parâmetro do inicializador tinha um nome diferente.

Agora que você adicionou esse inicializador, é possível ver que há dois erros de compilador em `main.swift`. Alterne para esse arquivo para examinar os erros.

Você verá que o uso anterior de `Zombie()` para obter uma instância dessa classe já não satisfaz o compilador. Por que não? O erro diz: `Missing argument for parameter 'town' in call.`

O erro indica que o compilador está esperando que o inicializador de `Zombie` inclua um parâmetro para `town`. Ainda assim, essa expectativa é estranha já que você não forneceu um inicializador à classe `Zombie` que exigia `town`. Na verdade, você não forneceu nenhum inicializador a essa classe; em vez disso, você estava contando com o inicializador vazio dado gratuitamente pelo compilador quando as suas propriedades têm valores padrão.

`Zombie` não recebe mais o inicializador vazio e livre que você estava utilizando mais cedo. Para entender essa mudança, é necessário entender a *herança automática de inicializadores*.

Herança automática de inicializadores

As classes normalmente não herdam os inicializadores de suas superclasses. Esse recurso da linguagem Swift existe para impedir que as subclasses forneçam inadvertidamente inicializadores que não definirão valores em todas as propriedades do tipo de subclasse, já que as subclasses frequentemente adicionam propriedades adicionais que não existem na superclasse. Esse comportamento impede que os tipos sejam inicializados parcialmente com inicializadores incompletos.

Entretanto, há circunstâncias em que uma classe herdará automaticamente os inicializadores de sua superclasse. Se sua subclasse tiver fornecido valores padrão para todas as propriedades adicionadas por ela, existirão dois cenários em que essa subclasse herdará os inicializadores de sua superclasse.

1. **Cenário 1:** Uma subclasse que não define nenhum inicializador designado herdará aqueles de sua superclasse.
2. **Cenário 2:** Uma subclasse que implementa todos os inicializadores designados de sua superclasse -- explicitamente ou por meio do cenário 1 acima -- herdará todos os inicializadores de conveniência de sua superclasse.

Acontece que seu tipo `Zombie` está incluído no primeiro desses dois cenários. Ele herda o único inicializador designado do tipo `Monster` porque não define seu próprio inicializador designado. A assinatura desse inicializador é: `init(town:monsterName:)`. Como o tipo `Zombie` herda um inicializador, o compilador não vai mais fornecer o inicializador livre que você estava usando antes. Sendo assim, do ponto de vista do compilador, a classe `Zombie` não tem um inicializador vazio disponível para o uso.

Existem dois lugares nos quais o inicializador de `Zombie` precisa ser atualizado para remover o erro. O primeiro é para `fredTheZombie` e o segundo é de um capítulo anterior sobre comparação de tipos de referência por meio de `z1` e `z2`. Você não precisa mais dessas instâncias, vá em frente e as exclua.

```
let fredTheZombie = Zombie()
let fredTheZombie = Zombie(town: myTown, monsterName: "Fred")
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
let z1 = Zombie()
z1.walksWithLimp = false
let z2 = z1
z2.walksWithLimp = true
println("z1 walks with limp? \(z1.walksWithLimp), z2 walks with limp? \(z2.walksWithLimp)")
```

Agora, quando você criar uma instância do tipo `Monster` ou `Zombie`, atribuirá um valor à instância em relação às suas propriedades `town` e `name`. Compile e execute o aplicativo; os erros devem desaparecer e os resultados devem ser iguais.

Inicializadores designados

As classes usam inicializadores designados como inicializador primário para uma classe. Como parte dessa função, os inicializadores designados são responsáveis por garantir que todas as propriedades da classe recebam valores antes da conclusão da inicialização. Se uma classe tiver uma superclasse, os inicializadores designados também devem chamar o inicializador designado de sua superclasse.

Você já escreveu um inicializador designado para a classe `Monster`. Lembre-se:

```
init(town: Town?, monsterName: String) {
    self.town = town
    name = monsterName
}
```

Os inicializadores designados não têm adornos, o que significa que não são indicados por nenhuma palavra-chave especial colocada antes de `init`. Essa sintaxe distingue inicializadores designados de inicializadores de conveniência, que usam a palavra-chave `convenience`. Esse inicializador garante que todas as propriedades em `Monster` recebam valores antes da conclusão da inicialização. Atualmente, o tipo `Zombie` atribui valores padrão a todas as suas propriedades (exceto aquelas herdadas de `Monster`). Sendo assim, o inicializador definido para `Monster` funciona bem para `Zombie`.

Remova os valores padrão das propriedades de `Zombie`.

```
class Zombie: Monster {
    class var spookyNoise: String {
        return "Brains..."
    }

    var walksWithLimp = false
    var walksWithLimp: Bool
    private var isFallingApart = false
    private var isFallingApart: Bool

    override func terrorizeTown() {
        if !isFallingApart {
            town?.changePopulation(-10)
        }
    }
}
```

A remoção desses valores padrão aciona um erro de compilador: `Class 'Zombie' has no initializers.` Como indicado pela mensagem de erro, a classe `Zombie` precisa de um inicializador para atribuir valores a suas propriedades. Atualmente, não há como todas as suas propriedades receberem valores antes de a inicialização ser concluída.

Adicione um novo inicializador à classe `Zombie` para resolver esse problema.

```
class Zombie: Monster {
    class var spookyNoise: String {
        return "Brains..."
    }

    var walksWithLimp: Bool
    private var isFallingApart: Bool
    init(limp: Bool, fallingApart: Bool, town: Town?, monsterName: String) {
        walksWithLimp = limp
        isFallingApart = fallingApart
        super.init(town: town, monsterName: monsterName)
    }
    override func terrorizeTown() {
        if !isFallingApart {
            town?.changePopulation(-10)
        }
    }
}
```

Seu novo inicializador acima resolve o erro porque agora você está garantindo que as propriedades de `Zombie` têm valores até o final da inicialização. Essa tarefa é realizada de duas formas. Primeiro, o novo inicializador define os valores das propriedades `walksWithLimp` e `isFallingApart` por meio dos argumentos `limp` e `fallingApart`. Essas propriedades são específicas da classe `Zombie` e, portanto, o inicializador designado precisa inicializá-las com valores adequados. Assim, os valores desses parâmetros são atribuídos às propriedades.

Segundo, chama-se o inicializador designado da superclasse do tipo `Monster`. `super` aponta para a superclasse da subclasse. Logo, a sintaxe: `super.init(town: town, monsterName: monsterName)` passa os valores dos parâmetros `town` e `monsterName` do inicializador na classe `Zombie` para o inicializador designado na classe `Monster`. Com isso, chama-se esse inicializador em `Monster`, o que garantirá que as propriedades de `Zombie` para `town` e `name` sejam definidas.

Ainda assim, precisamos resolver mais um erro. A classe `Zombie` não está sendo inicializada corretamente em `main.swift`. Alterne para esse arquivo e você verá que há um erro no compilador que diz que está faltando um argumento no inicializador de `Zombie`. Conserte o erro atualizando esse inicializador antigo com o novo.

```
let fredTheZombie = Zombie(town: myTown, monsterName: "Fred")
let fredTheZombie = Zombie(
    limp: false, fallingApart: false, town: myTown, monsterName: "Fred")
```

`fredTheZombie` agora está sendo inicializado com todas as informações que precisa para estar pronto para o uso.

Inicializadores de conveniência

Diferentemente dos inicializadores designados, os inicializadores de conveniência não são responsáveis por garantir que todas as propriedades de uma classe tenham um valor. Em vez disso, eles fazem o trabalho definidos para eles e, depois, entregam essas informações para outro inicializador de conveniência ou para um inicializador designado. Todos os inicializadores de conveniência chamam outro inicializador na mesma classe. Mais cedo ou mais tarde, um inicializador de conveniência deve chamar o inicializador designado de sua classe. A relação entre inicializadores de conveniência e inicializadores designados em determinada classe define, assim, um caminho pelo qual as propriedades armazenadas de uma classe receberam seus valores iniciais.

Crie um inicializador de conveniência no tipo `Zombie`. Esse inicializador apenas fornecerá argumentos para indicar se a instância `Zombie` deve ou não mancar e se a instância está caindo aos pedaços ou não.

```
init(limp: Bool, fallingApart: Bool, town: Town?, monsterName: String) {
    walksWithLimp = limp
    isFallingApart = fallingApart
    super.init(town: town, monsterName: monsterName)
}
convenience init(limp: Bool, fallingApart: Bool) {
    self.init(limp: limp, fallingApart: fallingApart, town: nil, monsterName: "")
    if walksWithLimp {
        println("This zombie has a bad knee.")
    }
}
override func terrorizeTown() {
    if !isFallingApart {
        town?.changePopulation(-10)
    }
}
```

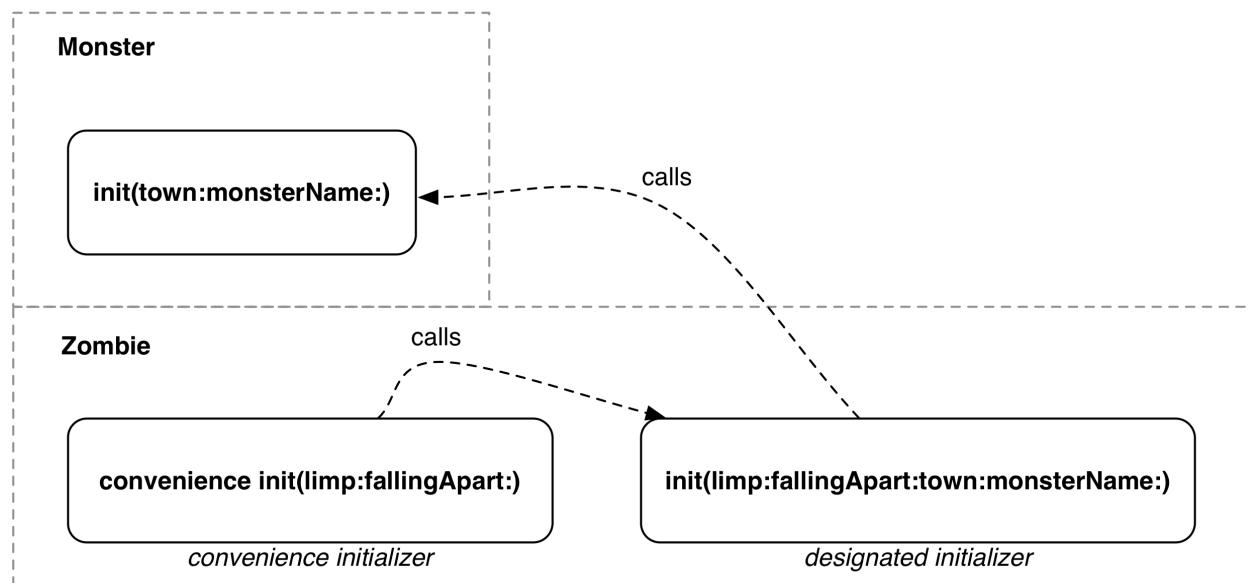
Marca-se um inicializador como inicializador de conveniência com a palavra-chave `convenience`. Essa palavra-chave informa o compilador que o inicializador precisará delegar a outro inicializador na classe, chamando no fim um inicializador designado. Depois dessa chamada, uma instância da classe estará pronta para o uso.

O inicializador de conveniência acima chama primeiro o inicializador designado na classe `Zombie`. Ele passa os valores dos parâmetros que recebeu: `limp` e `fallingApart`. No caso dos parâmetros para os quais o inicializador de conveniência recebeu valores, `town` e `monsterName`, você passou `nil` e uma string vazia respectivamente.

O inicializador de conveniência pode considerar a instância como totalmente preparada para o uso após esse ponto. Portanto, você pode verificar o valor da propriedade `walksWithLimp` na instância. Se você tivesse tentado realizar essa verificação antes de chamar o inicializador designado de `Zombie`, o compilador teria emitido um erro: `Use of 'self' in delegating initializer before self.init is called`. Esse erro está dizendo que o inicializador de delegação está tentando usar `self`, que é necessário para acessar a propriedade `walksWithLimp`, antes de ficar pronto para o uso.

A Figure 17.1 exibe em um gráfico as relações entre os inicializadores designados e de conveniência discutidos acima.

Figure 17.1 Delegação de inicializadores



Agora você pode criar instâncias do tipo `Zombie` com esse inicializador de conveniência. Alterne para `main.swift` e use-o para criar uma instância. Lembre-se, contudo, de que as instâncias de `Zombie` criadas com esse inicializador de conveniência apresentarão `nil` na propriedade `town` e uma string vazia ("") na propriedade `name`.

```
...
var fredTheZombie: Zombie? = Zombie(
    limp: false, fallingApart: false, town: myTown, monsterName: "Fred")
fredTheZombie?.terrorizeTown()
fredTheZombie?.town?.printTownDescription()

var convenientZombie = Zombie(limp: false, fallingApart: false)
...
```

Inicializadores exigidos

Uma classe pode exigir que as subclasses forneçam um inicializador específico. Por exemplo, imagine que você queira que todas as subclasses da classe `Monster` forneçam um inicializador básico. Para isso, marca-se o inicializador em questão com a palavra-chave `required` para indicar que todas as subclasses desse tipo devem fornecer determinado inicializador.

Alterne para `Monster.swift` para criar um inicializador exigido.

```
class Monster {
    var town: Town? = nil
    let name: String

    var victimPool: Int {
        get {
            return town?.population ?? 0
        }
        set(newVictimPool) {
            town?.population = newVictimPool
        }
    }
    init(town: Town?, monsterName: String) {
    required init(town: Town?, monsterName: String) {
        self.town = town
        name = monsterName
    }

    func terrorizeTown() {
        if let t = town {
            println("\(name) is terrorizing a town!")
        } else {
            println("\(name) hasn't found a town to terrorize yet...")
        }
    }
}
```

O único inicializador designado na classe `Monster` é agora exigido. As subclasses devem implementar esse inicializador. Essa exigência faz sentido já que o nome e a cidade a ser aterrorizada são dois componentes essenciais de um monstro.

Infelizmente, a mudança aciona um erro de compilador: `'required' initializer 'init(town:monsterName:)' must be provided by subclass of 'Monster'`. Esse erro indica que você ainda não está implementando esse recém-exigido inicializador na classe `Zombie`. Navegue até `Zombie.swift` para implementar o inicializador.

```

convenience init(limp: Bool, fallingApart: Bool) {
    self.init(limp: limp, fallingApart: fallingApart, town: nil, monsterName: "")
    if walksWithLimp {
        println("This zombie has a bad knee.")
    }
}
required init(town: Town?, monsterName: String) {
    walksWithLimp = false
    isFallingApart = false
    super.init(town: town, monsterName: monsterName)
}
override func terrorizeTown() {
    if !isFallingApart {
        town?.changePopulation(-10)
    }
}

```

Para implementar o inicializador exigido de uma superclasse, você deve prefixar a implementação do inicializador da subclasse com a palavra-chave `required`. Diferentemente de outras funções que obrigam que você sobrescreva se herdá-las de sua superclasse, você não precisa marcar os inicializadores `required` com a palavra-chave `override`. Isso fica implícito ao marcar o inicializador com `required`.

A sua implementação desse inicializador `required` faz com que ele seja um inicializador designado para a classe `Zombie`. Como esse inicializador se tornou um inicializador designado?

Lembre-se de que inicializadores designados são responsáveis por inicializar as propriedades do tipo e por delegar até o inicializador da superclasse. Essa implementação faz exatamente duas coisas. Você pode, portanto, usar esse inicializador para instanciar a classe `Zombie`.

Anulação da inicialização

A anulação da inicialização refere-se ao processo de remover instâncias de uma classe da memória quando não são mais necessárias. Como resultado, essa anulação não é disponibilizada para uso por tipos de valor. Os detalhes do gerenciamento de memória são tratados com mais detalhes no Chapter 21.

Na linguagem Swift, um *anulador de inicialização* é chamado imediatamente antes do momento em que a instância é removida da memória. Ele cria uma oportunidade para fazer a manutenção final antes de a instância ser desalocada. Conceitualmente, é o oposto da inicialização.

Os anuladores de inicialização são escritos com `deinit`.

```

...
required init(town: Town?, monsterName: String) {
    walksWithLimp = false
    isFallingApart = false
    super.init(town: town, monsterName: monsterName)
}

override func terrorizeTown() {
    if !isFallingApart {
        town?.isBeingTerrorized = true
        town?.changePopulation(-10)
    }
}
deinit {
    println("Zombie named \(name) is no longer with us.")
}
...

```

O novo código acima adiciona um anulador de inicialização à classe `Zombie`. Esse anulador não aceita nenhum argumento, que é uma característica de todos os anuladores de inicialização. Além disso, é importante notar que uma classe pode ter apenas um anulador de inicialização.

O seu novo anulador de inicialização simplesmente registra um adeus para a instância `Zombie` que está prestes a ser desalocada da memória. Observe que o anulador de inicialização acessa o nome do `Zombie`. Os anuladores de inicialização têm total acesso às propriedades e aos métodos de uma instância.

Abra `main.swift` para acionar o método `deinit` de `Zombie`. Você definirá `fredTheZombie` como `nil` no final do arquivo. Isso levará ao processo de remoção dessa instância da memória.

```
...
let fredTheZombie = Zombie(
    limp: false, fallingApart: false, town: myTown, monsterName: "Fred")
var fredTheZombie: Zombie? = Zombie(
    limp: false, fallingApart: false, town: myTown, monsterName: "Fred")
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printTownDescription()
fredTheZombie?.terrorizeTown()
fredTheZombie?.town?.printTownDescription()

println("Victim pool: \(fredTheZombie!.victimPool)")
fredTheZombie!.victimPool = 500
println("Victim pool: \(fredTheZombie!.victimPool)")

println("Victim pool: \(fredTheZombie?.victimPool)")
fredTheZombie?.victimPool = 500
println("Victim pool: \(fredTheZombie?.victimPool)")

println(Zombie.spookyNoise)
fredTheZombie = nil
```

Lembre-se de que apenas tipos opcionais podem ser ou se tornar `nil` na linguagem Swift. Portanto, você teve que declarar `fredTheZombie` como opcional: `Zombie?`. Essa alteração também significa que agora você tem que usar o encadeamento de opcionais para desempacotar o valor da opcional. Você também precisou declarar `fredTheZombie` com `var` em vez de `let` porque essa instância pode opcionalmente mudar para se tornar `nil`.

Compile e execute o programa agora. Você verá que agora deu adeus a `fredTheZombie` ao desalocar a instância.

Inicializadores suscetíveis a falhas

Às vezes, é útil definir um tipo cuja inicialização possa falhar. Nesses casos, uma instância totalmente preparada do tipo com valores para todas as suas propriedades não é o resultado da inicialização; em vez disso, você precisa de um modo de relatar para o chamador que você não conseguiu inicializar a instância. Usa-se um *inicializador suscetível a falhas* para lidar com essas situações.

Existem muitos motivos possíveis para a inicialização falhar. O inicializador do tipo pode receber parâmetros inválidos. Talvez a inicialização de um tipo dependa de um recurso externo que não está disponível. Por exemplo, `let image = UIImage(named: "non-existing-image")` falha ao criar uma instância `UIImage` porque o recurso de imagem não existe. Quando isso acontece, o inicializador suscetível a falhas de `UIImage` vai retornar `nil`, `return nil`, para indicar que a inicialização falhou.

Um inicializador de cidade suscetível a falhas

Os inicializadores suscetíveis a falhas retornam uma instância opcional do tipo. Anexa-se um ponto de interrogação após a palavra-chave `init` para indicar que o inicializador é suscetível a falhas (por exemplo, `init?`). Também é possível usar um ponto de exclamação depois de `init` para criar um inicializador suscetível a falhas que retorne uma opcional desempacotada implicitamente (por exemplo, `init!`).

Alterne para `Town.swift` para atribuir um inicializador suscetível a falhas à struct `Town`. Se uma instância de `Town` estiver sendo criada com uma população igual a 0, a inicialização falhará. Esse resultado faz sentido porque você não pode ter uma cidade sem população.

`Town` tem dois inicializadores. Lembre-se de que você delegou do inicializador `init(population:stopLights:)` para o inicializador `init(region:population:stopLights:)` nesse tipo. Por enquanto, torne apenas `init(region:population:stopLights:)` suscetível a falhas. Troque a definição de maneira correspondente.

```

struct Town {
    ...
        init(region: String, population: Int, stopLights: Int) {
        self.region = region
        self.population = population
        numberOfStopLights = stopLights
    }
        init?(region: String, population: Int, stopLights: Int) {
        if population <= 0 {
            return nil
        }
        self.region = region
        self.population = population
        numberOfStopLights = stopLights
    }
    ...
}

```

Observe que você agora usou a sintaxe de inicializador suscetível a falhas: **init?(region: String, population: Int, stopLights: Int)**. Depois dessa declaração, verifica-se se o valor dado para `population` é inferior ou igual a 0. Se `population` for inferior ou igual a 0, `return nil`. O inicializador falha nesse caso. Ele criará uma instância opcional do tipo `Town` com o valor `nil`.

Abra `main.swift` para ver o seu novo inicializador suscetível a falhas em ação. Para isso, lembre-se de inicializar uma instância de `Town` com o valor igual a 0 para o parâmetro `population`.

```

var myTown = Town(population: 10000, stopLights: 4)
var myTown = Town(population: 0, stopLights: 4)
myTown.printTownDescription()
...

```

Reserve um momento para considerar o que está acontecendo antes de compilar e executar o programa.

O inicializador `init(population:stopLights:)` delega atualmente a um inicializador suscetível a falhas. Esse processo sugere que `init(population:stopLights:)` pode receber `nil` de volta do inicializador designado para o qual ele delega. Receber `nil` de volta do inicializador designado será inesperado porque o próprio `init(population:stopLights:)` não é suscetível a falhas.

Conserte esse problema tornando `init(population:stopLights:)` também um inicializador suscetível a falhas.

```

struct Town {
    ...
        init?(region: String, population: Int, stopLights: Int) {
        if population <= 0 {
            return nil
        }
        self.region = region
        self.population = population
        numberOfStopLights = stopLights
    }

        init(population: Int, stopLights: Int) {
        self.init(region: "N/A", population: population, stopLights: stopLights)
    }
        init?(population: Int, stopLights: Int) {
        self.init(region: "N/A", population: population, stopLights: stopLights)
    }
    ...
}

```

Se você tentar compilar o programa agora, verá que há uma série de erros que precisam ser consertados. Você pode encontrar esses erros no arquivo `main.swift`.

A linha do código `myTown.printTownDescription()` tem um erro que diz: 'Town?' does not have a member named 'printTownDescription'. Lembre-se de que mudar os inicializadores na struct `Town` para que sejam suscetíveis a falha significa que agora eles retornam opcionais. Os inicializadores agora retornam `Town?` e não `Town`.

Isso significa que você tem que desempacotar as opcionais antes de usá-las. Use o encadeamento de opcionais para consertar os erros em `main.swift`.

```
var myTown = Town(population: 0, stopLights: 4)
myTown.printTownDescription()

let ts = myTown.townSize
println(ts.rawValue)

myTown?.changePopulation(1000000)
println("Size: \(myTown.townSize.rawValue); population: \(myTown.population)")
myTown?.printTownDescription()

let ts = myTown?.townSize
println(ts?.rawValue)

myTown?.changePopulation(1000000)
println("Size: \(myTown?.townSize.rawValue); population: \(myTown?.population)")
...
```

A representação de `nil` na linguagem Swift tende a ser muito impactante para seu código. Por exemplo, o uso do inicializador suscetível a falhas significa que você teve que usar o encadeamento de opcionais ao longo do `main.swift`. Essas mudanças podem trazer complexidade e adicionar mais código para seu projeto. Isso aumenta as chances de cometer um erro problemático. Sendo assim, aconselhamos usar opcionais apenas em absoluta necessidade.

Compile e execute o programa agora. Você removeu os erros, logo, seu projeto é executado normalmente.

Inicializadores suscetíveis a falhas em classes

Os inicializadores suscetíveis a falhas funcionam um pouco diferente em classes quando comparados a tipos de valor (como enumerações e estruturas). Em tipos de valor, um inicializador suscetível a falhas pode falhar em qualquer altura e aconteceria `return nil` nesse momento. Os inicializadores suscetíveis a falha tem de atribuir valores iniciais em todas as propriedades da classe antes de falhar.

Esse requisito significa que você não pode escrever o código a seguir:

```
class MyClass {
    let myProperty: String
    init?(myProperty: String) {
        if myProperty.isEmpty {
            return nil
        }
        self.myProperty = myProperty
    }
}
```

Códigos como o acima acionarão um erro no compilador dizendo que todas as propriedades armazenadas de uma classe devem ser inicializadas antes de retornar `nil`. `myProperty` não recebe um valor inicial antes de o inicializador falhar. Como podemos evitar esse erro? Uma tendência atual é o uso de opcional desempacotada implicitamente.

```
class MyClass {
    let myProperty: String
    let myProperty: String!
    init?(myProperty: String) {
        if myProperty.isEmpty {
            return nil
        }
        self.myProperty = myProperty
    }
}
```

O pseudocódigo acima torna `myProperty` uma opcional desempacotada implicitamente. Observe que o erro foi removido. O valor inicial de `myProperty` agora passou a ser `nil` porque é uma opcional desempacotada

implicitamente, que é um valor inicial válido. Portanto, o inicializador pode falhar antes de atribuir a `myProperty` um valor diferente de nil.

Como `myProperty` é uma constante, você não será capaz de atribuir nil a ela depois da inicialização. Por exemplo, este código acionaria um erro: `c?.myProperty = nil`. Assim, você terá certeza de que `myProperty` tem um valor válido se a inicialização tiver êxito. Além disso, a sintaxe da opcional desempacotada implicitamente significa que você pode acessar o valor de `myProperty` sem ter que usar ligação ou encadeamento de opcionais.

Entretanto, ainda é importante ter em mente a orientação comum dada em relação a opcionais de desempacotamento forçado. Se `MyClass` for inicializado com uma instância `String` que faça com que o inicializador falhe (por exemplo, uma `String` vazia), o desempacotamento à força da instância da opcional resultante provocará uma falha de tempo de execução. Por exemplo, este código não é seguro e deve ser evitado: `c!.myProperty`.

Conclusão

A inicialização em Swift é um processo bem definido com muitas regras. Felizmente, o compilador o lembrará do que você precisa fazer para atender às exigências e escrever um inicializador válido. Em vez de memorizar todas as regras da inicialização, é mais útil pensar na inicialização em Swift em termos de tipos de valor e classes.

Em relação aos tipos de valor, como structs, a inicialização é responsável principalmente por garantir que todas as propriedades armazenadas da instância tenham sido inicializadas e tenham recebido os valores adequados. Isso também é verdade para as classes, mas a inicialização é um pouco mais complicada nesse caso.

A inicialização para as classes pode ser considerada um desdobramento de duas fases sequenciais.

Na primeira fase, o inicializador designado da classe acaba sendo chamado (diretamente ou delegado por um inicializador de conveniência). Nessa altura, todas as propriedades declaradas nessa classe são inicializadas com valores aproximados dentro da definição do inicializador designado. Em seguida, um inicializador designado delega até o inicializador designado de sua superclasse. O inicializador designado na superclasse, então, garante que todas as suas próprias propriedades armazenadas sejam inicializadas com valores aproximados, que é um processo que continua até alcançar a classe no topo da cadeia de herança. A primeira fase termina aqui.

Então, a segunda fase começa e proporciona uma oportunidade para uma classe personalizar ainda mais os valores contidos por suas propriedades armazenadas. Por exemplo, um inicializador designado pode modificar as propriedades em `self` após chamar o inicializador designado da superclasse. Os inicializadores designados também podem chamar métodos de instância em `self`. É finalmente nesse momento que a inicialização reentra no inicializador de conveniência, dando a ele a oportunidade de realizar qualquer personalização na instância que quiser.

A instância estará totalmente inicializada depois dessas duas fases e todas as suas propriedades e métodos estarão disponíveis para o uso.

A ideia por trás desse processo de inicialização bem definido é garantir a inicialização bem-sucedida de uma classe. O compilador garante esse procedimento e emitirá erros se alguma etapa ao longo do processo não for cumprida. No fim, não é importante lembrar-se de cada etapa no processo, desde que você siga a orientação do compilador.

Desafios

Desafio de prata

Atualmente, o inicializador exigido em `Monster` está implementado como inicializador designado na subclasse `Zombie`. Torne esse inicializador um inicializador de conveniência na classe `Zombie`. Essa mudança envolverá a delegação da classe `Zombie` até seu inicializador designado.

Desafio de ouro

Atualmente, a classe `Monster` pode ser inicializada com qualquer instância `String` para o parâmetro `monsterName`, até mesmo uma `String` vazia. Isso resultaria em uma instância de `Monster` sem nome, o que não faz muito sentido. Conserte esse problema na classe `Monster` garantindo que `monsterName` não possa ficar vazia.

Sua solução envolverá atribuir um inicializador suscetível a falhas a `Monster`. Observe também que essa mudança terá impacto na inicialização da subclasse `Zombie`. Também faça os ajustes necessários nessa classe.

Para os mais curiosos

Parâmetros de inicializadores

Como funções e métodos, os inicializadores podem fornecer nomes de parâmetros externos explícitos. Os nomes de parâmetros externos fazem distinção entre os nomes de parâmetros disponíveis para chamadores e os nomes de parâmetros locais usados na implementação do inicializador. Como os inicializadores seguem convenções de nomeação diferentes de funções (isto é, nomes de inicializadores são sempre `init`), os nomes de parâmetros e os tipos ajudam a determinar qual inicializador deve ser chamado. Sendo assim, a linguagem Swift fornece nomes de parâmetros externos para todos os argumentos do inicializador por padrão.

Você pode fornecer seus próprios nomes de parâmetros externos conforme necessário. Por exemplo, imagine uma struct `WeightRecordInLBS` que deve poder ser inicializada com quilogramas.

```
struct WeightRecordInLBS {
    let weight: Double

    init(weightInKilos kilos: Double) {
        weight = kilos * 2.20462
    }
}
```

O inicializador acima fornece `weightInKilos` como parâmetro externo explícito e atribui `kilos` como parâmetro local. Em sua implementação, simplesmente se converte `kilos` em libras realizando a multiplicação com a conversão correta. Então, o inicializador seria usado da seguinte maneira: `let wr = WeightRecordInLBS(weightInKilos: 84)`.

Você pode até mesmo usar `_` como nome de parâmetro externo explícito se não quiser expor o nome de um parâmetro. Por exemplo, nossa struct fictícia `WeightRecordInLBS` obviamente define um registro de peso em libras. Sendo assim, faria sentido para o inicializador aceitar como padrão as libras em seu argumento.

```
struct WeightRecordInLBS {
    let weight: Double

    init(_ pounds: Double) {
        weight = pounds
    }

    init(weightInKilos kilos: Double) {
        weight = kilos * 2.20462
    }
}
```

O novo inicializador acima pode ser usado da seguinte maneira: `let wr = WeightRecordInLBS(185)`. Como o tipo representa um registro de peso em libras de forma muito explícita, não há necessidade de ter um parâmetro com nome na lista de argumentos. Usar um `_` pode deixar o seu código mais conciso, o que é conveniente quando está bastante explícito o que será passado para o argumento.

Part V

Conceitos avançados

18

Protocolos

No Chapter 16, você aprendeu sobre o uso dos controles de acesso para esconder informações. Esconder informações é uma forma de encapsulamento, que permite que você projete seu software para que ele permita que você altere uma parte sem afetar o resto do programa. A linguagem Swift suporta outra forma de encapsulamento: um protocolo ajuda a especificar e trabalhar com a *interface* de um tipo sem saber o próprio tipo. Uma interface é um conjunto de propriedades e métodos fornecidos por um tipo.

Formatação de uma tabela de dados

Os protocolos são um conceito mais abstrato do que muitos assuntos que aprendemos até aqui. Você criará uma função para formatar dados em uma tabela que parecerá uma simples planilha. Em seguida, usará um protocolo para tornar essa função flexível o suficiente para lidar com diferentes *fontes de dados*. Aplicativos para Mac e iOS normalmente separam a apresentação de dados da fonte que fornece os dados. Essa separação é um padrão extremamente útil que permite, por exemplo, que a Apple forneça classes que lidem com apresentação, deixando que você determine como os dados devem ser armazenados.

Crie um novo Playground chamado de `Protocols.playground`. Comece com uma função que aceite um array cujos elementos individuais sejam também arrays, um "array de arrays", e exiba o número em uma tabela. Cada elemento do array `data` é um array de inteiros que representa as colunas de uma simples linha, logo o número total de linhas é `data.count`.

```
func printTable(data: [[Int]]) {
    for row in data {
        // create an empty string
        var out = ""

        // append each item in this row to our string
        for item in row {
            out += "\\(item) |"
        }

        // done - print it!
        println(out)
    }
}

let data = [
    [30, 6],
    [40, 18],
    [50, 20],
]

printTable(data)
```

Abra a visão da linha do tempo e você verá uma tabela simples exibindo os dados. Adicione a habilidade de rotular cada linha. É um pouco complicado rotular as linhas porque queremos que todos os rótulos das linhas fiquem alinhados, então será necessário determinar qual é o maior rótulo e preencher as linhas menores com espaços:

```

func printTable(data: [[Int]]) {
func printTable(rowLabels: [String], data: [[Int]]) {
    // Determine length of longest row label
    let maxRowLabelWidth = maxElement(rowLabels.map { countElements($0) })

    for row in data {
        // create an empty string
        var out = ""
        for (i, row) in enumerate(data) {
            // Pad the row label out so they are all the same length
            let paddingAmount = maxRowLabelWidth - countElements(rowLabels[i])
            var out = rowLabels[i] + Repeat(count: paddingAmount, repeatedValue: " ") + "|"

            // append each item in this row to our string
            for item in row {
                out += " \(item) |"
            }

            // done - print it!
            println(out)
        }
    }

let rowLabels = ["Joe", "Karen", "Fred"]
let data = [
    [30, 6],
    [40, 18],
    [50, 20],
]
printTable(rowLabels, data)

```

É necessário determinar a largura do maior rótulo de linha para alinhar todas as linhas corretamente. A instância `maxRowLabelWidth` armazena esse valor; ele é calculado usando `map` para converter cada rótulo de linha em um inteiro com tamanho correspondente e, então, passando esse resultado para a função `maxElement` que retorna o tamanho máximo (maior). Em seguida, quando você iterar cada linha dos dados, terá que preencher as linhas menores que a largura máxima. A instância `paddingAmount` contém a quantidade de preenchimento necessário para uma linha, e `Repeat(count: paddingAmount, repeatedValue: " ")` cria uma string contendo espaços `paddingAmount`.

Finalmente, adicione a habilidade de rotular cada coluna. Rotular colunas é ainda mais difícil; todas as colunas devem ser alinhadas verticalmente, o que pode exigir que todos os itens de dados sejam preenchidos. Você precisará monitorar a largura de cada rótulo de coluna e preencher todos os itens nessa coluna até o tamanho certo.

```

func printTable(rowLabels: [String], data: [[Int]]) {
func printTable(rowLabels: [String], columnLabels: [String], data: [[Int]]) {
    // Determine length of longest row label
    let maxRowLabelWidth = maxElement(rowLabels.map { countElements($0) })

    // Create first row containing column headers
    var firstRow: String = Repeat(count: maxRowLabelWidth, repeatedValue: " ") + " |"

    // Also keep track of the width of each column
    var columnWidths = [Int]()

    for columnLabel in columnLabels {
        let columnHeader = " \(columnLabel) |"
        firstRow += columnHeader
        columnWidths.append(countElements(columnHeader))
    }
    println(firstRow)

    for (i, row) in enumerate(data) {
        // Pad the row label out so they are all the same length
        let paddingAmount = maxRowLabelWidth - countElements(rowLabels[i])
        var out = rowLabels[i] + Repeat(count: paddingAmount, repeatedValue: " ") + " |"

        // append each item in this row to our string
        for item in row {
            out += " \(item) |"
        }
        for (j, item) in enumerate(row) {
            let itemString = " \(item) |"
            let paddingAmount = columnWidths[j] - countElements(itemString)
            out += Repeat(count: paddingAmount, repeatedValue: " ") + itemString
        }

        // done - print it!
        println(out)
    }
}

let rowLabels = ["Joe", "Karen", "Fred"]
let columnLabels = ["Age", "Years of Experience"]
let data = [
    [30, 6],
    [40, 18],
    [50, 20],
]
printTable(rowLabels, data)
printTable(rowLabels, columnLabels, data)

```

Você criou e exibiu `firstRow`, que contém todos os cabeçalhos de coluna. Ele usa o mesmo truque com `Repeat()` para adicionar espaços na primeira coluna (que contém os cabeçalhos de linha). À medida que construía a primeira linha, você também registrou a largura de cada cabeçalho de coluna no array `columnWidths`. Depois, quando anexou cada item de dado na linha de resultado, usou o array `columnWidths` e `Repeat()` para preencher cada item de forma que eles tenham a mesma largura do respectivo cabeçalho de coluna.

Verifique o resultado da linha do tempo novamente. Você verá uma tabela de dados muito bem formatada:

	Age	Years of Experience	
Joe	30	6	
Karen	40	18	
Fred	50	20	

Contudo, há pelo menos um grande problema com a função `printTable`: ela é muito difícil de usar! Você precisa ter arrays diferentes para rótulos de linha, rótulos de coluna e para os dados e precisa certificar-se manualmente de que o número de rótulos de linha e de rótulos de coluna corresponda ao número de elementos no array de dados. É muito mais provável que você queira representar esse tipo de informação usando estruturas e classes. Substitua a parte do

código em que você chama `printTable` com alguns *objetos modelo*, que são tipos que representam os dados com os quais o seu aplicativo trabalha:

```
let rowLabels = ["Joe", "Karen", "Fred"]
let columnLabels = ["Age", "Years of Experience"]
let data = [
    [30, 6],
    [40, 18],
    [50, 20],
]

printTable(rowLabels, columnLabels, data)
struct Person {
    let name: String
    let age: Int
    let yearsOfExperience: Int
}

struct Department {
    let name: String
    var people = [Person]()

    mutating func addPerson(person: Person) {
        people.append(person)
    }
}

var department = Department(name: "Engineering", people: [])
department.addPerson(Person(name: "Joe", age: 30, yearsOfExperience: 6))
department.addPerson(Person(name: "Karen", age: 40, yearsOfExperience: 18))
department.addPerson(Person(name: "Fred", age: 50, yearsOfExperience: 20))
```

Agora você tem um `Department` e gostaria de poder exibir os detalhes dos funcionários usando a função `printTable`. Você poderia modificar `printTable` para aceitar um `Department` em vez dos três argumentos aceitos atualmente. No entanto, a implementação atual de `printTable` poderia ser usada para exibir qualquer tipo de dados de tabela, e seria bom manter essa funcionalidade. Um protocolo pode ajudar a preservar essa funcionalidade.

Protocolos

O *protocolo* permite definir a interface que um tipo deve satisfazer. O tipo que satisfaz um protocolo é considerado *em conformidade* com o protocolo. Defina um protocolo que especifique a interface que precisamos para a função `printTable`. A função `printTable` precisa saber quantas linhas e colunas existem, qual é o rótulo para cada linha e coluna e qual dado deve ser exibido em cada célula. Para o compilador da linguagem Swift, o local onde você coloca esse protocolo no arquivo do Playground não é importante, mas provavelmente faz mais sentido colocá-lo no topo do arquivo, antes de `printTable`.

```
protocol TabularDataSource {
    var numberOfRows: Int { get }
    var numberOfColumns: Int { get }

    func labelForRow(row: Int) -> String
    func labelForColumn(column: Int) -> String

    func itemForRow(row: Int, column: Int) -> Int
}
```

Você já deve estar familiarizado com a sintaxe de um protocolo; ela é muito similar à definição de uma estrutura ou classe, exceto pela omissão de todas as definições de função e propriedade computada. O protocolo `TabularDataSource` declara que qualquer tipo em conformidade deve ter duas propriedades, `numberOfRows` e `numberOfColumns`. A sintaxe `{ get }` indica que a propriedade pode ser lida, mas não escrita. Se a propriedade tivesse que ser do tipo leitura/gravação, você usaria `{ get set }`. Observe que uma propriedade de protocolo marcada com `{ get }` não exclui a possibilidade de que um tipo em conformidade tenha uma propriedade do tipo

leitura/gravação; o protocolo apenas requer que ela seja legível. Além disso, `TabularDataSource` especifica que um tipo em conformidade deve ter três métodos listados com os tipos exatos que são listados.

Um protocolo define o mínimo de propriedades e métodos que um tipo deve ter. O tipo pode ter mais do que o protocolo lista, mas o protocolo define a linha de base. Podem existir propriedades e métodos sem problemas, desde que todos os requisitos do protocolo estejam presentes.

Você fará com que `Department` esteja em conformidade com o protocolo `TabularDataSource`. Comece declarando que `Department` está em conformidade com `TabularDataSource`:

```
struct Department {
    struct Department : TabularDataSource {
        let name: String
        var people = [Person]()

        mutating func addPerson(person: Person) {
            people.append(person)
        }
    }
}
```

A sintaxe de conformidade com um protocolo é adicionar `: ProtocolName` após o nome do tipo. (Isso é muito similar ao modo como declaramos uma superclasse. Veremos como protocolos e superclasses podem ser usados juntos mais tarde.) Seu arquivo de playground agora tem um erro e, se você abrir a linha do tempo, verá os detalhes. Você alegou que `Department` está em conformidade com `TabularDataSource`, mas `Department` não possui nenhuma das propriedades e dos métodos exigidos por `TabularDataSource`. Adicione as implementações de todos eles:

```
struct Department : TabularDataSource {
    let name: String
    var people = [Person]()

    mutating func addPerson(person: Person) {
        people.append(person)
    }

    var numberOfRows: Int {
        return people.count
    }

    var numberOfColumns: Int {
        return 2
    }

    func labelForRow(row: Int) -> String {
        return people[row].name
    }

    func labelForColumn(column: Int) -> String {
        switch column {
            case 0: return "Age"
            case 1: return "Years of Experience"
            default: fatalError("Invalid column!")
        }
    }

    func itemForRow(row: Int, column: Int) -> Int {
        let person = people[row]
        switch column {
            case 0: return person.age
            case 1: return person.yearsOfExperience
            default: fatalError("Invalid column!")
        }
    }
}
```

O `Department` tem uma linha para cada pessoa, então a propriedade `numberOfRows` retorna o número de pessoas no departamento. Cada pessoa tem duas propriedades que devem ser exibidas, então `numberOfColumns` retorna o número dois. O rótulo de cada linha é o nome da pessoa que será exibido na linha. `labelForColumn` e `itemForRow` são um pouco mais interessantes: você usou a instrução `switch` para retornar um de dois cabeçalhos de coluna. (Por que existe um caso `default`? Volte ao Chapter 6.)

Você ainda precisa voltar e modificar `printTable` para aceitar e trabalhar com um `TabularDataSource`. Os protocolos não definem apenas as propriedades e os métodos que devem ser fornecidos por um tipo em conformidade. Eles próprios também podem ser usados como tipos: é possível ter variáveis, argumentos de função e valores de retorno que tenham o tipo de um protocolo. Altere `printTable` para aceitar uma fonte de dados do tipo `TabularDataSource`, agora que esse protocolo fornece os mesmos dados que os antigos argumentos forneciam (incluindo todos os cabeçalhos de coluna e linha e a quantidade de dados disponíveis):

```
func printTable(rowLabels: [String], columnLabels: [String], data: [[Int]]) {
func printTable(dataSource: TabularDataSource) {
    // Determine length of longest row label
let maxRowLabelWidth = maxElement(rowLabels.map { countElements($0) })
let maxRowLabelWidth = maxElement(map(0 ..< dataSource.numberOfRows, {
    countElements(dataSource.labelForRow($0))
}))
```

// Create first row containing column headers
var firstRow: String = Repeat(count: maxRowLabelWidth, repeatedValue: " ") + " |"

// Also keep track of the width of each column
var columnWidths = [Int]()

```
for columnLabel in columnLabels {
    for c in 0 ..< dataSource.numberOfColumns {
        let columnLabel = dataSource.labelForColumn(c)
        let columnHeader = " \(columnLabel) |"
        firstRow += columnHeader
        columnWidths.append(countElements(columnHeader))
    }
    println(firstRow)
```

```
for (i, row) in enumerate(data) {
    for r in 0 ..< dataSource.numberOfLines {
        // Pad the row label out so they are all the same length
let paddingAmount = maxRowLabelWidth - countElements(rowLabels[i])
var out = rowLabels[i] + Repeat(count: paddingAmount, repeatedValue: " ") + " |"
        let rowLabel = dataSource.labelForRow(r)
        let paddingAmount = maxRowLabelWidth - countElements(rowLabel)
        var out = rowLabel + Repeat(count: paddingAmount, repeatedValue: " ") + " |"

        // append each item in this row to our string
        for (j, item) in enumerate(row) {
            let itemString = " \(item) |"
            let paddingAmount = columnWidths[j] - countElements(itemString)
            for c in 0 ..< dataSource.numberOfColumns {
                let item = dataSource.itemForRow(r, column: c)
                let itemString = " \(item) |"
                let paddingAmount = columnWidths[c] - countElements(itemString)
                out += Repeat(count: paddingAmount, repeatedValue: " ") + itemString
            }
        }
        // done - print it!
        println(out)
    }
}
```

O tipo `Department` agora está em conformidade com `TabularDataSource`, e `printTable` foi modificado para aceitar um `TabularDataSource`. Portanto, você pode exibir o departamento:

```

var department = Department(name: "Engineering", people: [])
department.addPerson(Person(name: "Joe", age: 30, yearsOfExperience: 6))
department.addPerson(Person(name: "Karen", age: 40, yearsOfExperience: 18))
department.addPerson(Person(name: "Fred", age: 50, yearsOfExperience: 20))

printTable(department)

```

Conformidade de protocolo

Como dissemos antes, a sintaxe de conformidade de protocolo é exatamente igual à sintaxe que usamos para declarar a superclasse de uma classe, conforme vimos no Chapter 15. Isso suscita algumas questões:

1. Quais tipos podem estar em conformidade com protocolos?
2. Um tipo pode estar em conformidade com múltiplos protocolos?
3. Uma classe pode ter uma classe superior e ainda assim estar em conformidade com protocolos?

Todos os tipos podem estar em conformidade com protocolos. Você fez com que uma estrutura (`Department`) estivesse em conformidade com um protocolo; enums e classes também podem estar em conformidade com protocolos. A sintaxe para declarar que uma enum está em conformidade com um protocolo é exatamente a mesma que para uma struct: dois pontos e o nome do protocolo após a declaração do tipo. (As classes podem ser um pouco mais complicadas; veremos isso em breve).

Um tipo também pode estar em conformidade com múltiplos protocolos. Um dos protocolos definidos pela linguagem Swift é o `Printable`, que é usado por funções como `println` para exibir instâncias. O `Printable` tem um único requisito: o tipo deve aceitar uma propriedade que pode ser obtida, chamada de `description`, que retorna uma `String`. Modifique `Department` para que esteja em conformidade tanto com `TabularDataSource` como com `Printable` usando uma vírgula para separar os protocolos:

```

struct Department : TabularDataSource {
struct Department : TabularDataSource, Printable {
    let name: String
    var people = [Person]()

    var description: String {
        return "Department(\(name))"
    }

    // ... rest of Department stays the same
}

```

Você implementou `description` como propriedade computada somente leitura. Agora é possível ver o nome do departamento ao exibi-lo:

```

printTable(department)
println(department)

```

Finalmente, as classes também podem estar em conformidade com protocolos. Se a classe não tiver uma classe superior, a sintaxe será igual àquela para structs e enums. Por exemplo,

```

class ClassName : ProtocolOne, ProtocolTwo {
    // ...
}

```

Se a classe tiver uma classe superior, o nome da classe superior vem antes seguido de todos os protocolos com os quais a classe está em conformidade; por exemplo,

```

class ClassName : ParentClass, ProtocolOne, ProtocolTwo {
    // ...
}

```

Herança de protocolos

A linguagem Swift suporta *herança de protocolos*. Um protocolo herdeiro de outro requer que os tipos em conformidade forneçam implementações para todas as propriedades e métodos exigidos por ele mesmo e pelo protocolo do qual é herdeiro. Isso é diferente da herança de classe, que define uma relação próxima entre a superclasse e a subclasse. A herança de protocolos meramente adiciona as exigências do protocolo pai para o protocolo filho. Por exemplo, modifique TabularDataSource para que seja herdeiro do protocolo Printable:

```
protocol TabularDataSource {
    protocol TabularDataSource : Printable {
        var numberOfRows: Int { get }
        var numberOfColumns: Int { get }

        func labelForRow(row: Int) -> String
        func labelForColumn(column: Int) -> String

        func itemForRow(row: Int, column: Int) -> Int
    }
}
```

Agora, qualquer tipo que estiver em conformidade com TabularDataSource também deve estar em conformidade com Printable, o que significa que ele deve fornecer todas as propriedades e métodos listados em TabularDataSource, além da propriedade `description` exigida por Printable. Utilize isso em `printTable` para exibir um cabeçalho na tabela:

```
func printTable(dataSource: TabularDataSource) {
    println("Table: \(dataSource.description)")

    // ... rest of function remains unchanged
}
```

Os protocolos podem ser herdeiros de diversos outros protocolos, da mesma maneira que tipos podem estar em conformidade com múltiplos protocolos. Como deve imaginar, a sintaxe para a herança de múltiplos protocolos é assim: protocolos pais adicionais são separados com vírgulas, da seguinte maneira:

```
protocol MyProtocol : MyOtherProtocol, Printable {
    // ... requirements of MyProtocol
}
```

Composição de protocolos

A herança de protocolos é uma ferramenta poderosa que permite que você crie facilmente um novo protocolo que adicione requisitos a um protocolo ou conjunto de protocolos já existente. Entretanto, o uso de herança de protocolos pode potencialmente fazer com que você tome decisões ruins ao criar seus tipos. Na verdade, isso é exatamente o que aconteceu com TabularDataSource. Você fez com que TabularDataSource herdesse de Printable, porque queria poder exibir uma descrição da fonte de dados; mas não há nada inherentemente Printable em relação a uma fonte de dados em tabela. Volte e conserte essa tentativa equivocada de exibir fontes de dados:

```
protocol TabularDataSource : Printable {
    protocol TabularDataSource {
        ...
    }
}
```

O compilador agora reclama, e com razão, quando você tenta obter a `description` da fonte de dados passada a `printTable()`. Você pode usar a *composição de protocolos* para declarar que o argumento para `printTable` deve estar em conformidade tanto com TabularDataSource como com Printable:

```
func printTable(dataSource: TabularDataSource) {
    func printTable(dataSource: protocol<TabularDataSource, Printable>) {
        println("Table: \(dataSource.description)")

        // ... rest of function remains unchanged
    }
}
```

A sintaxe da composição de protocolos usa a palavra-chave `protocol` para indicar ao compilador que você está combinando múltiplos protocolos em um único requisito. Você pode usar a composição de protocolos com mais de dois protocolos adicionando outros protocolos, separados por vírgulas, entre `<>`. O exemplo acima exige que `dataSource` seja tanto `TabularDataSource` como `Printable`: `dataSource: protocol<TabularDataSource, Printable>`.

Considere outra possibilidade. Você também poderia criar um novo protocolo que fosse herdeiro tanto de `TabularDataSource` como de `Printable`, da seguinte maneira:

```
protocol PrintableTabularDataSource: TabularDataSource, Printable { }
```

Você também poderia usar esse protocolo como o tipo do argumento para `printTable()`. Tanto `PrintableTabularDataSource` como `protocol<TabularDataSource, Printable>` exigem que os tipos em conformidade implementem todas as propriedades e métodos exigidos por `TabularDataSource` e `Printable`. Qual é a diferença entre eles?

A diferença é que `PrintableTabularDataSource` é um tipo distinto: você teria que voltar e modificar `Department` para declarar que ele está em conformidade com `PrintableTabularDataSource` mesmo que ele já esteja cumprindo todos os requisitos. Por outro lado, você pode considerar `protocol<TabularDataSource, Printable>` como um tipo de protocolo "temporário". Você não precisou voltar e anotar `Department`; ele já está em conformidade tanto com `TabularDataSource` como com `Printable`, logo também está em conformidade com `protocol<TabularDataSource, Printable>`.

Métodos mutáveis

No Chapter 14 e no Chapter 15, vimos que métodos em tipos de valor, structs e enums não podem modificar `self` a menos que o método seja marcado como `mutating`. Os métodos em protocolos não são mutáveis por padrão. Lembre-se do tipo `LightBulb` do Chapter 14:

```
enum LightBulb {
    case On
    case Off

    mutating func toggle() {
        switch self {
            case .On:
                self = .Off

            case .Off:
                self = .On
        }
    }
}
```

Imagine que você queira definir um protocolo com uma instância "alternável":

```
protocol Toggleable {
    func toggle()
}
```

Declarar que `LightBulb` está em conformidade com `Toggleable` resultará em erro de compilador. A mensagem recebida inclui uma nota que explica o problema:

```
error: type 'LightBulb' does not conform to protocol 'Toggleable'

note: candidate is marked 'mutating' but protocol does not allow it
mutating func toggle() { ^ }
```

A nota indica que, em `LightBulb`, o método `toggle` é marcado como mutável, mas o protocolo `Toggleable` espera uma função não mutável. Você pode consertar esse problema marcando `toggle` como mutável na definição do protocolo:

```
protocol Toggleable {  
    mutating func toggle()  
}
```

Uma classe em conformidade com o protocolo `Toggleable` não precisaria marcar o método `toggle` como mutável. Os métodos em classes sempre podem mudar as propriedades de `self`, porque são tipos de referência.

Conclusão

Neste capítulo, você aprendeu bastante sobre os protocolos da linguagem Swift.

- Como declarar um protocolo com requisitos de propriedade e método
- Como modificar um tipo existente para estar em conformidade com um protocolo
- Uso de herança de protocolos para criar um novo protocolo que adicione requisitos a protocolos existentes
- Quando usar `mutating` em um método de um protocolo

Além disso, você foi apresentado ao padrão de um protocolo de *fonte de dados*, que é um padrão muito comum no desenvolvimento para iOS e Mac OS X. Esse assunto não acaba aqui - os protocolos têm outros recursos poderosos relacionados aos genéricos da linguagem Swift, assunto do Chapter 20.

Desafios

Desafio de prata

A função `printTable` está com um bug - ela falha se algum dado for maior que o rótulo da coluna em que ele se encontra. Tente mudar a idade de Joe para 1000 para ver isso em ação. Corrija o bug. (Para a versão mais fácil desse desafio, só faça com que a função não falhe. Para a versão mais difícil desse desafio, certifique-se de que todas as linhas e colunas da tabela permaneçam alinhadas corretamente.)

Desafio de ouro

Crie um tipo novo, `BookCollection`, que esteja em conformidade com `TabularDataSource`. Ao chamar `printTable` em uma coleção de livros, uma tabela de livros deve aparecer, com colunas para título, autor e pontuação média na Amazon. (A menos que todos os livros usados tenham títulos e nomes de autor muito curtos, você precisará ter concluído o desafio anterior!)

19

Extensões

Imagine que você está trabalhando com um tipo na biblioteca padrão da linguagem Swift, digamos que seja o tipo `Double`. Por algum motivo, o seu aplicativo usa esse tipo frequentemente e seria ótimo se o tipo `Double` suportasse alguns métodos adicionais. Infelizmente, a implementação de `Double` não está disponível, então você não pode adicionar essa funcionalidade ao tipo diretamente. O que você pode fazer?

A linguagem Swift possui um recurso chamado de *Extensões*, que foi projetado justamente para esses casos. As extensões permitem que você adicione funcionalidades a um tipo existente. Você pode estender structs, enums e classes. As extensões podem adicionar propriedades computadas, métodos, conformidade de protocolos e uma série de outros recursos a um tipo existente.

Neste capítulo, você usará extensões para adicionar funcionalidades a um tipo existente cuja definição e cujos detalhes de implementação não estão disponíveis para você. Você também usará extensões para adicionar funcionalidades a um tipo personalizado de sua própria criação. Em ambos os casos, você adicionará funcionalidades a esses tipos de maneira modular, o que significa que você agrupará funcionalidades similares em uma única extensão.

Extensão de um tipo existente

Crie um novo playground chamado de `Extensions.playground`. Salve-o onde quiser e abra-o.

Você modelará o comportamento de um carro nesse playground. A velocidade é uma característica importante de qualquer veículo. Como a velocidade pode ter valores decimais, é sensato representá-la como `Double`.

Como você usará o tipo `Double` frequentemente, seria útil se referir a ele de maneira contextualmente relevante. A palavra-chave `typealias` da linguagem Swift oferece um modo de dar um outro nome a um tipo existente. Dê um nome alternativo ao tipo `Double` no playground.

`typealias Velocity = Double`

A palavra-chave `typealias` permite que você defina a `Velocity` como um nome de tipo alternativo para `Double`. `Velocity` é um nome bom para `Double` nesse exemplo, já que você usará esse tipo para modelar a velocidade de um veículo. Agora você pode se referir a `Double` pelo nome de tipo `Velocity`. Essa intercambiabilidade ajuda a contextualizar `Double` tornando-o mais relevante ao uso.

Agora que você configurou uma `typealias`, é hora de estender o tipo para suportar a conversão entre unidades comumente usadas para velocidade. Adicione o seguinte a seu playground.

```
typealias Velocity = Double
extension Velocity {
    var kph: Velocity { return self * 1.60934 }
    var mph: Velocity { return self }
}
```

A palavra-chave `extension` indica que você está estendendo o tipo `Velocity`. As extensões em Swift não permitem adicionar propriedades armazenadas a um tipo. Nesse caso, você está adicionando duas propriedades computadas a `Velocity`: `kph` e `mph`. Essas propriedades em `Velocity` convertem a velocidade de um veículo de quilômetros por hora (`kph`) a milhas por hora (`mph`) e vice-versa. Observe que a extensão trata `mph` como unidade padrão; essa propriedade computada simplesmente retorna `self`, enquanto `kph` realiza a conversão.

Embora a intercambiabilidade proporcionada por uma typealias possa ser benéfica, ela também pode ser um pouco complicada. Pode acontecer de você querer usar o tipo `Double` nesse arquivo e não o tipo `Velocity`. Como `Velocity` pode ser usado de maneira intercambiável com `Double`, a extensão definida em `Velocity` também está disponível para o tipo `Double`. Isso pode ser confuso, mas a adição da extensão ao `Double` usando `Velocity` typealias oferece um contexto útil. Ela documenta que as propriedades computadas são apenas significativas quando usadas com a typealias `Velocity`, embora estejam disponíveis a todos os `Doubles`.

Lembre-se de que um dos objetivos deste capítulo é definir o comportamento de um veículo. Os protocolos são um dos recursos mais úteis da linguagem Swift para ajudar a definir a interface de um tipo. Como podemos ver, é possível adicionar conformidade de protocolos a um tipo com uma extensão.

Crie um novo protocolo chamado de `VehicleType` para descrever algumas características básicas de um veículo.

```
typealias Velocity = Double

extension Velocity {
    var kph: Velocity { return self * 1.60934 }
    var mph: Velocity { return self }
}

protocol VehicleType {
    var topSpeed: Velocity { get }
    var numberOfDoors: Int { get }
    var hasFlatbed: Bool { get }
}
```

`VehicleType` declara três propriedades: 1) `topSpeed`, 2) `numberOfDoors` e 3) `hasFlatbed`. Cada propriedade requer apenas que o tipo em conformidade implemente um "getter" para a propriedade. Um tipo em conformidade com esse protocolo deverá fornecer essas propriedades que descrevem algumas características gerais de um veículo.

Extensão de seu próprio tipo

Você precisará criar um novo tipo antes de poder adicionar conformidade de protocolos a ele por meio de uma extensão. Crie uma nova struct para representar o tipo `Car`. Posteriormente, usaremos uma extensão em `Car` para adicionar conformidade ao protocolo `VehicleType`.

```
typealias Velocity = Double

extension Velocity {
    var kph: Velocity { return self * 1.60934 }
    var mph: Velocity { return self }
}

protocol VehicleType {
    var topSpeed: Velocity { get }
    var numberOfDoors: Int { get }
    var hasFlatbed: Bool { get }
}

struct Car {
    let make: String
    let model: String
    let year: Int
    let color: String
    let nickname: String
    var gasLevel: Double {
        willSet {
            precondition(newValue <= 1.0 && newValue >= 0.0, "New value must be between 0 and 1.")
        }
    }
}
```

Você definiu acima uma nova struct chamada de `Car`. O tipo `Car` definiu uma série de propriedades armazenadas que serão específicas a uma dada instância. Todas as propriedades são constantes, com uma exceção: `gasLevel`.

`gasLevel` é uma propriedade armazenada mutável com um observador de propriedade. O observador `willSet` será chamado sempre que você for definir um novo valor para `gasLevel`. Você usou uma `precondition()` dentro dessa implementação para garantir que o `newValue` sendo atribuído à propriedade `gasLevel` esteja entre 0 e 1. Esses valores indicam o nível do tanque de combustível de uma instância em termos de pontos percentuais.

Uso de extensões para adicionar conformidade de protocolos

As extensões podem oferecer um ótimo mecanismo para agrupar funcionalidades relacionadas. O agrupamento de funcionalidades relacionadas em uma única extensão pode ser útil para tornar seu código mais legível e de manutenção mais fácil. Esse padrão também ajuda o tipo a manter sua interface organizada.

Estenda o tipo `Car` para estar em conformidade com o protocolo `VehicleType`.

```
...
struct Car {
    let make: String
    let model: String
    let year: Int
    let color: String
    let nickname: String
    var gasLevel: Double {
        willSet {
            precondition(newValue <= 1.0 && newValue > 0.0, "New value must be between 0 and 1.")
        }
    }
}
extension Car: VehicleType {
    var topSpeed: Velocity { return 180 }
    var numberofDoors: Int { return 4 }
    var hasFlatbed: Bool { return false }
}
```

Sua nova extensão criada acima estende `Car` para que esteja em conformidade com `VehicleType`. A sintaxe para estar em conformidade com um protocolo é igual a que vimos antes. A diferença é o uso de uma extensão para obter essa conformidade de protocolo: `extension Car: VehicleType`.

Implementam-se as propriedades exigidas do protocolo dentro do corpo da extensão. Cada propriedade recebeu um "getter" simples. Por questões de conveniência, você simplesmente retornou alguns valores padrão para cada uma das propriedades do protocolo.

Adição de um inicializador com uma extensão

Lembre-se de que structs dão um inicializador `memberwise` livre se você não fornecer um. Se você quiser escrever um novo inicializador para sua struct, mas não quiser perder o inicializador `memberwise`, adicione o inicializador a um tipo dado com uma extensão. Adicione o inicializador a `Car` em uma nova extensão no tipo.

```
...
extension Car: VehicleType {
    var topSpeed: Velocity { return 180 }
    var numberofDoors: Int { return 4 }
    var hasFlatbed: Bool { return false }
}
extension Car {
    init(carMake: String, carModel: String, carYear: Int) {
        self.init(make: carMake,
                  model: carModel,
                  year: carYear,
                  color: "Black",
                  nickname: "N/A",
                  gasLevel: 1.0)
    }
}
```

A nova extensão no tipo `Car` adiciona um inicializador que aceita argumentos apenas para `make`, `model` e `year` de uma instância. Esse novo inicializador utiliza o inicializador `memberwise` livre na struct `Car`. Os novos argumentos do inicializador são passados para o inicializador `memberwise` e você também fornece valores padrão para os argumentos que faltam. A combinação desses dois inicializadores garante que uma instância do tipo `Car` tenha valores para todas as suas propriedades.

Como podemos ver, o inicializador `memberwise` é preservado em `Car` porque esse novo inicializador, na verdade, é definido e implementado em uma extensão. Esse padrão pode ser bastante útil.

Para ver o novo inicializador definido no trabalho de extensão, vá em frente e crie uma instância de `Car`.

```
...
extension Car {
    init(carMake: String, carModel: String, carYear: Int) {
        self.init(make: carMake,
                  model: carModel,
                  year: carYear,
                  color: "Black",
                  nickname: "N/A",
                  gasLevel: 1.0)
    }
}
var c = Car(carMake: "Ford", carModel: "Fusion", carYear: 2013)
```

O código acima cria uma nova instância `c`. Essa instância é criada com o inicializador definido em uma extensão em `Car`. Dê uma olhada na barra lateral de resultados. Você verá que as propriedades de `c` possuem os valores que você atribuiu a esse novo inicializador; os valores padrão que você atribuiu ao inicializador `memberwise` também devem estar visíveis na barra lateral.

Tipos aninhados e extensões

As extensões da linguagem Swift também podem adicionar tipos aninhados a um tipo existente. Digamos, por exemplo, que você queira adicionar uma enumeração a sua struct `Car` para ajudar a classificar o tipo de carro que uma instância pode ser. Crie uma nova extensão no tipo `Car` e adicione um tipo aninhado.

```
...
var c = Car(carMake: "Ford", carModel: "Fusion", carYear: 2013)
extension Car {
    enum CarKind: Printable {
        case Coupe, Sedan
        var description: String {
            switch self {
                case .Coupe:
                    return "Coupe"

                case .Sedan:
                    return "Sedan"
            }
        }
    }
    var kind: CarKind {
        if number0fDoors == 2 {
            return .Coupe
        } else {
            return .Sedan
        }
    }
}
```

A extensão em `Car` adiciona um tipo aninhado chamado de `CarKind`. `CarKind` é uma enumeração com dois casos: um para `Coupe` e outro para `Sedan`. A extensão também adiciona uma propriedade computada em `Car` chamada

de `kind`. Essa propriedade representará qual tipo de carro é a instância de `Car`. O tipo aninhado também está em conformidade com o protocolo `Printable` para facilitar o registro no console.

`kind` retorna valores da enumeração aninhada com base em quantas portas a instância tem. Se a instância tiver duas portas, então será um cupê. Se tiver mais, será um sedã.

Exercite o novo tipo aninhado na extensão acessando a propriedade computada `kind` na instância criada anteriormente.

```
...
extension Car {
    enum CarKind: Printable {
        case Coupe, Sedan
        var description: String {
            switch self {
                case .Coupe:
                    return "Coupe"
                case .Sedan:
                    return "Sedan"
            }
        }
        var kind: CarKind {
            if numberOfDoors == 2 {
                return .Coupe
            } else {
                return .Sedan
            }
        }
    }
    c.kind.description
}
```

Você verá "Sedan" registrado no console.

Extensões com funções

Você pode usar uma extensão para atribuir uma função a um tipo existente. Por exemplo, note que `Car` não tem uma função de encher o tanque. Crie uma extensão para adicionar essa funcionalidade a `Car`.

```
...
c.kind.description
extension Car {
    mutating func emptyGas(amount: Double) {
        precondition(amount <= 1 && amount > 0, "Amount to remove must be between 0 and 1.")
        gasLevel -= amount
    }

    mutating func fillGas() {
        gasLevel = 1.0
    }
}
```

Sua nova extensão adiciona duas funções, `emptyGas()` e `fillGas()`, a `Car`. Observe que ambas as funções são marcadas com a palavra-chave `mutating`. Por quê? Lembre-se de que o tipo `Car` é uma struct. Se uma função quiser mudar o valor de qualquer uma das propriedades da struct, ela deve ser declarada com a palavra-chave `mutating`.

A função `emptyGas()` aceita um argumento: a quantidade de combustível que será removida do tanque. Você usou uma `precondition` dentro de `emptyGas()` para garantir que a quantidade removida do tanque esteja entre 0 e 1. A implementação de `fillGas()` simplesmente define a propriedade `gasLevel` em `Car` como cheia, ou `1.0`.

Exercite essas novas funções em seu tipo existente.

```
extension Car {  
    mutating func emptyGas(amount: Double) {  
        precondition(amount <= 1 && amount > 0, "Amount to remove must be between 0 and 1.")  
        gasLevel -= amount  
    }  
  
    mutating func fillGas() {  
        gasLevel = 1.0  
    }  
}  
c.emptyGas(0.3)  
c.gasLevel  
c.fillGas()  
c.gasLevel
```

Depois de usar a função `emptyGas()`, você verá na barra lateral que o nível de combustível é igual a `0.7`. Depois de encher o nível de combustível, você verá que ele será igual a `1.0`.

Conclusão

As extensões permitem que você adicione funcionalidades a um tipo existente. É possível usá-las para adicionar funcionalidades a um tipo que você não criou. Esse recurso é especialmente útil quando você não tem o arquivo-fonte desse tipo. Também é possível adicionar funcionalidades a um tipo que você mesmo criou. A extensão dos seus próprios tipos dessa forma permite agrupar funcionalidades relacionadas em uma única extensão.

Em geral, é possível usar extensões para estender tipos com:

- Propriedades computadas
- Novos inicializadores
- Conformidade de protocolos
- Novos métodos
- Tipos incorporados

Você viu cada um desses itens neste capítulo.

Desafios

Desafio de bronze

Estenda o tipo `Int` para que ele tenha a propriedade computada `timesFive`. A propriedade computada deve retornar o resultado da multiplicação do inteiro por 5. Deve ser possível usá-la da seguinte maneira:

```
5.timesFives // 25
```

Desafio de prata

O método `emptyGas()` tem alguns bugs. Por exemplo, se o `gasLevel` atual for inferior à quantidade a ser removida, o novo valor dessa propriedade será negativo. Um valor negativo não faz sentido e, na verdade, interromperá a execução do programa (lembre-se da `precondition` no observador de propriedade do `gasLevel`). Revise a implementação de `emptyGas` para garantir que o `gasLevel` não diminua até um valor negativo.

20

Genéricos

Até agora, todos os tipos criados e todas as funções escritas por você funcionaram em tipos concretos, como `Int`, `String` e `Monster`. Você pode ter notado, contudo, que a linguagem Swift permite criar arrays que contenham qualquer tipo, incluindo `Int`, `String` e `Monster`. Como esse array é implementado? Como podemos escrever um código que funcione da mesma forma com diversos tipos? Os "genéricos" são a resposta a essas duas perguntas.

Os genéricos em Swift permitem que você escreva tipos e funções que utilizem tipos que nem você nem o compilador conhecem ainda. Muitos tipos embutidos usados durante este livro, incluindo Opcionais, Arrays e Dicionários são implementados usando genéricos. Neste capítulo, você investigará como escrever tipos genéricos (parecidos com `Array`) e, também, como usar genéricos para escrever funções flexíveis, além de como genéricos são relacionados a protocolos.

Estrutura de dados dos genéricos

Você criará uma *pilha* de genéricos, que é uma estrutura de dados respeitável em ciência da computação. Uma pilha é uma estrutura de dados que é a última a entrar e a primeira a sair (LIFO). Ela suporta duas operações básicas. Você pode *enviar* um item para a pilha, o que adiciona o item à pilha, e pode *remover* para pegar da pilha o último item a ser enviado, o que remove o item da pilha.

De início, crie um novo playground chamado `Generics.playground` e crie uma estrutura `Stack` que armazene apenas inteiros:

```
struct Stack {
    var items = [Int]()

    mutating func push newItem: Int) {
        items.append(newItem)
    }

    mutating func pop() -> Int? {
        if items.isEmpty {
            return nil
        } else {
            return items.removeLast()
        }
    }
}
```

Essa estrutura tem três elementos de interesse. A propriedade armazenada `items` é um array que você está usando para conter os itens atualmente em uma pilha. O método `push()` envia um novo item para a pilha anexando-o ao final do array de `items`. Finalmente, o método `pop()` remove o item do topo da pilha chamando o método `removeLast()` de um array, o que simultaneamente remove o último item e o retorna. Observe que `pop()` retorna uma opcional `Int` porque a pilha pode estar vazia (nesse caso, não haveria nada para remover).

Crie uma instância `Stack` para ver isso em ação:

```
var intStack = Stack()
intStack.push(1)
intStack.push(2)

println(intStack.pop()) // prints Optional(2)
println(intStack.pop()) // prints Optional(1)
println(intStack.pop()) // prints nil
```

Você criou uma nova instância Stack, enviou dois valores para ela e, em seguida, tentou remover três valores dela. Como esperado, as chamadas de remoção retornam os inteiros que você enviou na ordem inversa, e a remoção retorna nula quando não restar mais nenhum item na pilha.

Agora, modifique Stack para que seja uma estrutura de dados de genéricos que possa conter qualquer tipo, não só Int:

```
struct Stack {
    struct Stack<T> {
        var items = [Int]()
        var items = [T]()

        mutating func push(newItem: Int) {
            mutating func push(newItem: T) {
                items.append(newItem)
            }

            mutating func pop() -> Int? {
                mutating func pop() -> T? {
                    if items.isEmpty {
                        return nil
                    } else {
                        return items.removeLast()
                    }
                }
            }
        }
    }
}
```

Você adicionou um *tipo de placeholder*, chamado de T, à declaração de Stack. A sintaxe da linguagem Swift para declarar um genérico usa os sinais de menor e maior (<>) e segue imediatamente o nome do tipo. A letra entre os sinais de menor e maior representa o tipo de placeholder: <T>. Não é necessário usar a letra T; contudo, o uso de letras maiúsculas isoladas (como T e U) é uma convenção comum. O tipo de placeholder T pode ser usado dentro da estrutura de Stack em qualquer lugar onde um tipo concreto poderia ser usado. É possível ver esse uso no momento que você substitui todas as ocorrências de Int por T, incluindo uma declaração de propriedade, o tipo do argumento em push() e o tipo do valor de retorno de pop().

Agora há um erro de compilador, em que você criou uma Stack porque não especificou o tipo que deve ser substituído pelo tipo de placeholder T. O processo do compilador para substituir um tipo concreto por um placeholder é chamado de *especialização*. Conserte o erro especificando que intStack deve ser uma instância de Stack especializada para Int, usando a mesma sintaxe com sinais de maior e menor:

```
var intStack = Stack()
var intStack = Stack<Int>()
```

Agora, você pode criar Stacks de qualquer tipo. Crie uma Stack de Strings:

```
println(intStack.pop()) // prints Optional(1)
println(intStack.pop()) // prints nil

var stringStack = Stack<String>()
stringStack.push("this is a string")
stringStack.push("another string")

println(stringStack.pop()) // prints Optional("another string")
```

É importante observar que, embora `intStack` e `stringStack` sejam ambas as instâncias de `Stack`, elas não têm o mesmo tipo. `intStack` é uma `Stack<Int>`; passar qualquer coisa diferente de `Int` para `intStack.push()` resultaria em um erro de tempo de compilação. Da mesma maneira, `stringStack` é uma `Stack<String>`, o que é diferente de `Stack<Int>`.

As estruturas de dados de genéricos são muito comuns e extremamente úteis. As classes e enumerações também podem virar genéricos com a mesma sintaxe usada para estruturas. Além disso, os tipos não são o único elemento da linguagem Swift que podem ser genéricos; as funções e os métodos também podem ser genéricos.

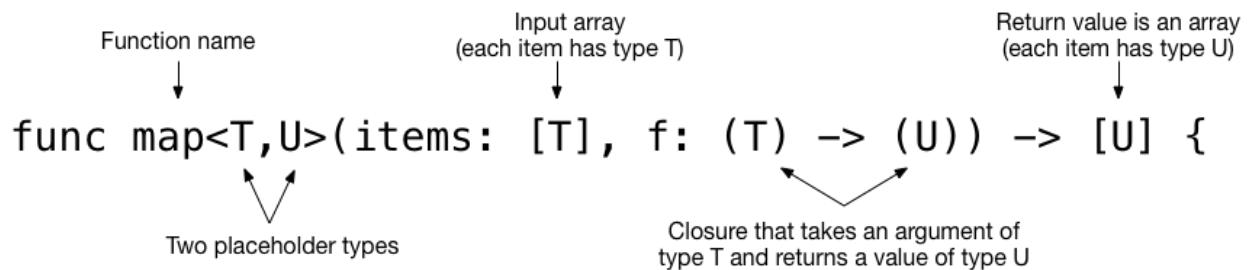
Funções e métodos genéricos

Lembre-se da função `map` do Chapter 13. `map` aceita uma sequência de itens e aplica um fechamento para cada elemento na sequência e retorna um array com os resultados. Com o que você acabou de aprender sobre genéricos, agora você mesmo pode implementar essa função. Adicione o seguinte código a seu playground:

```
func myMap<T,U>(items: [T], f: (T) -> (U)) -> [U] {
    var result = [U]()
    for item in items {
        result.append(f(item))
    }
    return result
}
```

A declaração de `myMap` pode parecer horrível se você não tiver sido exposto a genéricos em outras linguagens. A única coisa nova é que ele declara dois tipos de placeholder, `T` e `U`, não apenas um. Vamos analisar essa linha:

Figure 20.1 Declaração de `myMap`



`myMap` pode ser usado da mesma forma que `map`. Crie um array de strings e mapeie-o a um array de seus comprimentos:

```
func myMap<T,U>(items: [T], f: (T) -> (U)) -> [U] {
    // ... unchanged
}

let strings = ["one", "two", "three"]
let stringLengths = myMap(strings, { countElements($0) })
println(stringLengths) // prints [3, 3, 5]
```

O fechamento passado a `myMap` deve aceitar um único argumento que corresponda ao tipo contido no array de `items`, mas o tipo do seu valor de retorno pode ser qualquer coisa. Nessa chamada a `myMap`, `T` é substituído por `String` e `U` é substituído por `Int`. Observe que, em projetos reais, não é necessário declarar sua própria função de mapeamento – apenas use o `map` embutido.

Os métodos também podem ser genéricos, mesmo dentro de tipos que já sejam genéricos. A função `myMap` que você escreveu só funciona em arrays, mas parece sensato querer mapear uma `Stack`. Crie um método de `map` na `Stack`:

```
struct Stack<T> {
    var items = [T]()

    mutating func push(newItem: T) {
        items.append(newItem)
    }

    mutating func pop() -> T? {
        if items.isEmpty {
            return nil
        } else {
            return items.removeLast()
        }
    }

    func map<U>(f: (T) -> (U)) -> Stack<U> {
        var mappedItems = [U]()
        for item in items {
            mappedItems.append(f(item))
        }
        return Stack<U>(items: mappedItems)
    }
}
```

O método de `map` apenas declara um tipo de placeholder, `U`, mas ele usa tanto `T` como `U`. `T` está disponível porque `map` está dentro da estrutura de `Stack`, o que disponibiliza o tipo de placeholder `T`. O corpo de `map` é quase idêntico a `myMap`, sendo a única diferença o retorno de uma nova `Stack` em vez de um array. Experimente o seu próprio método:

```
var intStack = Stack<Int>()
intStack.push(1)
intStack.push(2)
var doubledStack = intStack.map({ 2 * $0 })

println(intStack.pop()) // prints Optional(2)
println(intStack.pop()) // prints Optional(1)
println(intStack.pop()) // prints nil

println(doubledStack.pop()) // prints Optional(4)
println(doubledStack.pop()) // prints Optional(2)
```

Restrições de tipo

Uma das coisas mais importantes que você deve ter em mente ao escrever funções e tipos de dados genéricos é que, por padrão, você não sabe nada sobre o tipo concreto que será usado. Você criou pilhas de `Int` e `String`, mas também poderia ter criado pilhas de qualquer outro tipo. O impacto prático dessa falta de conhecimento é que não dá para fazer muito com um valor de um tipo de placeholder. Por exemplo, você nem ao menos pode verificar se dois deles são iguais; este código não compilaria:

```
func checkIfEqual<T>(first: T, second: T) -> Bool {
    return first == second
}
```

Essa função poderia ser chamada com qualquer tipo, incluindo tipos para os quais a igualdade não faz sentido, como fechamentos. (É difícil descrever o que significaria dois fechamentos serem "iguais". A linguagem Swift não permite a comparação.) As funções genéricas seriam bastante incomuns se você nunca pudesse presumir nada sobre os tipos de placeholder. Para resolver esse problema, a linguagem Swift permite o uso de *restrições de tipo*, que colocam restrições nos tipos concretos que podem ser passados a funções genéricas. Existem dois tipos de restrições de tipo: a restrição de que um tipo seja uma subclasse de determinada classe ou a restrição de que um tipo esteja em conformidade com um protocolo (ou composição de protocolos). O protocolo `Equatable` é um protocolo fornecido pela linguagem Swift que declara que é possível verificar a igualdade de dois valores. Você pode escrever `checkIfEqual` incluindo uma restrição de que `T` deve ser `Equatable`:

```
func checkIfEqual<T: Equatable>(first: T, second: T) -> Bool {
    return first == second
}

println(checkIfEqual(1, 1))
println(checkIfEqual("a string", "a string"))
println(checkIfEqual("a string", "a different string"))
```

Todo tipo de placeholder pode ter uma restrição de tipo. Por exemplo, escreva uma função para verificar se dois valores Printable têm a mesma descrição:

```
func checkIfDescriptionsMatch<T: Printable, U: Printable>(first: T, second: U) -> Bool {
    return first.description == second.description
}

println(checkIfDescriptionsMatch(Int(1), UInt(1)))
println(checkIfDescriptionsMatch(1, 1.0))
println(checkIfDescriptionsMatch(Float(1.0), Double(1.0)))
```

A restrição de que `T` e `U` sejam `Printable` garante que `first` e `second` tenham uma propriedade chamada de `description` que retorne uma `String`. Embora os dois argumentos possam ter tipos diferentes, você ainda pode comparar a descrição deles.

Protocolos de tipos associados

Agora que você sabe que tipos, funções e métodos podem ser transformados em genéricos, é natural se perguntar se protocolos também podem. A resposta é não; contudo, os protocolos suportam um recurso similar e relacionado: *tipos associados*. Você explorará protocolos com tipos associados examinando dois protocolos definidos pela biblioteca padrão da linguagem Swift. Primeiramente, o protocolo `GeneratorType`:

```
protocol GeneratorType {
    typealias Element

    mutating func next() -> Element?
}
```

O protocolo `GeneratorType` requer um único método de mutação, `next()`, o qual retorna um valor do tipo `Element?`. O objetivo de `GeneratorType` é a possibilidade de chamar `next` repetidamente e com ele gerar novos valores a cada vez. Se o gerador não conseguir mais gerar novos valores, `next` retornará nulo. A nova sintaxe presente neste protocolo é `typealias Element`. Essa sintaxe indica que um tipo em conformidade com `GeneratorType` deve fornecer um tipo concreto associado que será usado como tipo `Element`. No topo de seu playground, crie uma nova struct chamada de `StackGenerator` que esteja em conformidade com `GeneratorType`:

```
struct StackGenerator<T> : GeneratorType {
    typealias Element = T

    var stack: Stack<T>

    mutating func next() -> Element? {
        return stack.pop()
    }
}

struct Stack<T> {
    // ...
}
```

`StackGenerator` inclui uma `Stack` e gera valores removendo itens da pilha. O tipo do `Element` retornado por `next` é `T`, então a `typealias` é definida adequadamente. Crie uma nova pilha, adicione alguns itens, crie um gerador e execute um loop em seus valores para ver `StackGenerator` em ação:

```
var myStack = Stack<Int>()
myStack.push(10)
myStack.push(20)
myStack.push(30)

var myStackGenerator = StackGenerator(stack: myStack)
while let value = myStackGenerator.next() {
    println("got \(value)")
}
```

StackGenerator é um pouco mais prolixo do que precisa. A linguagem Swift pode inferir o tipo dos tipos associados de um protocolo; portanto, você pode remover a typealias explícita indicando que next retorna um T?:

```
struct StackGenerator<T> : GeneratorType {
    typealias Element = T

    var stack: Stack<T>

    mutating func next() -> Element?
    mutating func next() -> T? {
        return stack.pop()
    }
}
```

O próximo protocolo de tipo associado a ser examinado é SequenceType. A definição de SequenceType é:

```
protocol SequenceType {
    typealias Generator : GeneratorType
    func generate() -> Generator
}
```

SequenceType tem um tipo associado chamado de Generator. A sintaxe de : GeneratorType é uma restrição de tipo no tipo associado. Ela tem o mesmo significado das restrições de tipo em genéricos: para que um tipo esteja em conformidade com SequenceType, ele deve ter um tipo associado Generator que esteja em conformidade com o protocolo GeneratorType. SequenceType também requer tipos em conformidade para implementar um único método, generate(), o qual retorna um valor do tipo associado GeneratorType. Como você já tem um gerador adequado para pilhas, modifique Stack para que esteja em conformidade com SequenceType:

```
struct Stack<T> {
struct Stack<T> : SequenceType {
    var items = [T]()

    mutating func push newItem: T) {
        items.append(newItem)
    }

    mutating func pop() -> T? {
        if items.isEmpty {
            return nil
        } else {
            return items.removeLast()
        }
    }

    func map<U>(f: (T) -> (U)) -> Stack<U> {
        var mappedItems = [U]()
        for item in items {
            mappedItems.append(f(item))
        }
        return Stack<U>(items: mappedItems)
    }

    func generate() -> StackGenerator<T> {
        return StackGenerator(stack: self)
    }
}
```

Você utilizou novamente a inferência de tipo da linguagem Swift para evitar ter que declarar explicitamente typealias Generator = StackGenerator<T>, embora não seja um erro declará-lo.

O protocolo SequenceType é o que a linguagem Swift usa internamente para seus loops for ... in. Agora que Stack está em conformidade com SequenceType, você pode executar um loop em seu conteúdo:

```
while let value = myStackGenerator.next() {
    println("got \(value)")
}

for value in myStack {
    println("for-in loop: got \(value)")
}
```

StackGenerator remove os valores de sua pilha sempre que next() é chamado, o que é uma operação bastante destrutiva. Quando StackGenerator retornar nulo de next(), a propriedade da pilha estará vazia. No entanto, você conseguiu criar um gerador manualmente a partir de myStack e então usou myStack novamente em um loop for ... in. Esse reuso é possível porque Stack é um tipo de valor, o que significa que sempre que um StackGenerator é criado, ele recebe uma cópia da pilha, deixando a original intocada.

Uma observação final: se um protocolo tiver um tipo associado, você não poderá usar esse protocolo como tipo concreto. Por exemplo, você não pode declarar uma variável com o tipo GeneratorType ou declarar uma função que aceite um argumento do tipo GeneratorType, porque GeneratorType tem um tipo associado. Contudo, os protocolos com tipos associados são fundamentais para usar *cláusulas where* em declarações genéricas.

Cláusulas where de restrições de tipo

Escreva uma nova função que aceite cada elemento de um array e que os envie a uma pilha:

```
func pushItemsOntoStack<T>(inout stack: Stack<T>, fromArray array: [T]) {
    for item in array {
        stack.push(item)
    }
}

pushItemsOntoStack(&myStack, fromArray: [1, 2, 3])
for value in myStack {
    println("after pushing: got \(value)")
}
```

pushItemsOntoStack recebe seu primeiro argumento, Stack, como argumento inout para que possa chamar o método mutável push(). Essa função é útil, mas não é tão geral como poderia ser. Agora você sabe que qualquer tipo que esteja em conformidade com SequenceType pode ser usado em um loop for ... in, então por que essa função requer um array? Ela deve ser capaz de aceitar qualquer tipo de sequência; até mesmo outra Stack, agora que Stack está em conformidade com SequenceType. Uma primeira tentativa produzirá um erro de compilador:

```
func pushItemsOntoStack<T>(inout stack: Stack<T>, fromArray array: [T]) {
func pushItemsOntoStack<T, S: SequenceType>(inout stack: Stack<T>, fromSequence sequence: S) {
    for item in array {
        for item in sequence {
            stack.push(item)
        }
    }
}
```

Você fez com que pushItemsOntoStack virasse genérico com dois tipos de placeholder: T, que é o tipo dos elementos da pilha, e S, que é algum tipo que está em conformidade com o protocolo SequenceType. A restrição em S garante que você possa executar um loop nele com a sintaxe for ... in; no entanto, isso não é suficiente. Para enviar os itens recebidos de sequence para a pilha, você precisa garantir que o tipo dos itens que chegam da sequência correspondam ao tipo dos elementos da pilha. Ou seja, você precisa adicionar outra restrição para que os próprios elementos produzidos por S sejam do tipo T. A linguagem Swift suporta restrições desse tipo usando uma cláusula where:

```
func pushItemsOntoStack<T, S: SequenceType>(inout stack: Stack<T>, fromSequence sequence: S) {
    func pushItemsOntoStack<T, S: SequenceType where S.Generator.Element == T>(
        inout stack: Stack<T>, fromSequence sequence: S) {
            for item in sequence {
                stack.push(item)
            }
    }
}
```

`pushItemsOntoStack` é uma função genérica com dois tipos de placeholder. O primeiro tipo de placeholder, `T`, não tem nenhuma restrição - ele pode ser qualquer coisa. O segundo tipo de placeholder, `S`, tem uma restrição para que o tipo concreto usado esteja em conformidade com o protocolo `SequenceType`. Além dos tipos de placeholder, a cláusula `where` impõe outras restrições. `S.Generator.Element` refere-se ao tipo `Element` associado ao tipo `Generator` associado a `S`. A restrição `S.Generator.Element == T` exige que o tipo concreto usado para o tipo associado `Element` corresponda ao tipo concreto usado para `T`.

A sintaxe das cláusulas `where` genéricas é, muitas vezes, difícil de ler à primeira vista, mas um exemplo deve torná-la mais clara. Se você passar uma pilha de `Ints` como primeiro argumento para `pushItemsOntoStack`, o segundo argumento deve ser uma sequência que produza `Ints`. Dois tipos que você já conhece, que são sequências produtoras de `Int`, são os tipos `Stack<Int>` e `[Int]`; experimente-os:

```
var myOtherStack = Stack<Int>()
pushItemsOntoStack(&myOtherStack, fromSequence: [1, 2, 3])
pushItemsOntoStack(&myStack, fromSequence: myOtherStack)
for value in myStack {
    println("after pushing items onto stack, got \(value)")
}
```

Você criou uma nova pilha de inteiros vazia: `myOtherStack`. Em seguida, enviou todos os inteiros de um array para `myOtherStack`. Finalmente, enviou todos os inteiros de `myOtherStack` para `myStack`. Você conseguiu usar a mesma função genérica em ambos os casos porque tanto arrays como pilhas estão em conformidade com `SequenceType`.

Conclusão

Os genéricos são um recurso extremamente poderoso da linguagem Swift. Você agora entende como a biblioteca padrão da linguagem Swift inclui estruturas de dados, como arrays, que podem conter valores de qualquer tipo e como criar suas próprias estruturas de dados genéricas. Você também sabe como escrever funções e métodos genéricos. Finalmente, você aprendeu sobre os protocolos de tipos associados e como usar as restrições de tipo para escrever funções genéricas que aceitem tipos que fornecem um comportamento específico. Se você ainda não entendeu os genéricos, não se preocupe - eles são um conceito simultaneamente complexo e abstrato. Reserve um momento para compreender a classe `Stack` que você escreveu durante este capítulo e tente fazer os desafios.

Desafios

Desafio de bronze

Adicione um método `filter` a sua estrutura de `Stack`. Ele deve aceitar um único argumento, um fechamento que aceite um `T` e retorne um `Bool`, e retornar uma nova `Stack<T>` que contenha todos os elementos para os quais o fechamento retorne como verdadeiro.

Desafio de prata

Escreva uma função genérica chamada de `findAll` que aceite um array de qualquer tipo `T` que esteja em conformidade com o protocolo `Equatable` e um único elemento (também do tipo `T`). `findAll` deve retornar um array de inteiros correspondendo a cada local onde o elemento foi encontrado no array. Por exemplo, `findAll([5,3,7,3,9], 3)` deve retornar `[1,3]` porque o item 3 existe nos índices 1 e 3 no array. Tente sua função tanto com inteiros quanto com strings.

Desafio de ouro

Modifique a função `findAll` que você escreveu para o desafio de prata para que ela aceite um `CollectionType` genérico em vez de um array. Dica: Você precisará mudar o tipo de retorno de `[Int]` para um array de um tipo associado do protocolo `CollectionType`.

Para os mais curiosos: Polimorfismo paramétrico

No Chapter 15, você aprendeu sobre a herança de classes. Qualquer função que espere um argumento de uma classe também pode aceitar argumentos que sejam subclasses dessa classe. Essa habilidade de aceitar uma classe ou qualquer subclasse dessa classe é chamada muitas vezes de *polimorfismo*, mas é conhecida mais precisamente de *polimorfismo de tempo de execução* ou *polimorfismo de subclasse*. Polimorfismo, ou muitas formas, significa que você escreveu uma única função que pode aceitar diferentes tipos.

O polimorfismo de tempo de execução é uma ferramenta muito poderosa usada frequentemente pelos frameworks que a Apple oferece para o desenvolvimento para iOS e Mac OS X. Infelizmente, ele tem algumas desvantagens. As classes relacionadas por herança são ligadas firmemente umas nas outras; pode ser difícil mudar uma sem afetar as outras. O polimorfismo de tempo de execução também acarreta uma penalidade pequena, mas observável, devido ao modo como o compilador deve implementar as funções que aceitam argumentos de classe.

O recurso da linguagem Swift que adiciona restrições aos genéricos permite que você use outra forma de polimorfismo: *polimorfismo de tempo de compilação*, também conhecido como *polimorfismo paramétrico*. As funções genéricas com restrições permanecem verdadeiras para a definição de polimorfismo: é possível escrever uma única função que aceite diferentes tipos. As funções polimórficas de tempo de compilação lidam com ambos os problemas listados acima, que assolam o polimorfismo de tempo de execução. Muitos tipos diferentes podem estar em conformidade com um protocolo, permitindo que sejam usados em qualquer função genérica que exija um tipo em conformidade com tal protocolo. Os tipos também podem não ser relacionados, facilitando mudar qualquer um deles sem afetar os outros. Além disso, o polimorfismo de tempo de compilação geralmente não tem nenhuma penalidade de desempenho. No playground, você chamou `pushItemsOntoStack` uma vez com um array e uma vez com uma pilha. O compilador, na verdade, produziu duas versões diferentes de `pushItemsOntoStack` no executável, o que significa que a função em si não precisa fazer nada no tempo de execução para lidar com os diferentes tipos de argumento.

Ainda é muito cedo para fazer generalizações sobre a qualidade do estilo e das expressões da linguagem Swift. Arriscamos dizer que, com base no desenvolvimento da biblioteca padrão da linguagem Swift, achamos que os genéricos e o polimorfismo de tempo de compilação desempenharão uma função importante no futuro. Na próxima vez que começar a escrever uma hierarquia de classes, considere se o problema que está tentando resolver poderia ser resolvido melhor com uma solução que tenha protocolos e genéricos.

21

Gerenciamento de memória e ARC

Todos os programas de computador usam memória. A maioria dos programas de computador usa a memória de forma dinâmica: à medida que o programa é executado, ele aloca e desaloca memória conforme necessário. A posição da linguagem Swift em relação ao gerenciamento de memória é relativamente única. A maioria dos problemas é resolvida automaticamente para você, mas a linguagem não usa coletor de lixo (uma ferramenta comum para gerenciar memória automaticamente em linguagens de programação). Em vez disso, a linguagem Swift usa um sistema de contagem de referência. Neste capítulo, você investigará como esse sistema funciona e saberá a que você precisa estar atento para evitar vazamentos de memória.

Alocação de memória

O gerenciamento e a alocação de memória para tipos de valor - enumerações e estruturas - são muito simples. Quando você criar uma nova instância de um tipo de valor, uma quantidade adequada de memória será reservada automaticamente para sua instância. Independentemente do que você fizer para passar a instância, incluindo passá-la a uma função e armazená-la em uma propriedade, uma cópia da instância será criada. A linguagem Swift recupera a memória quando a instância deixa de existir. Você não precisa fazer nada para gerenciar a memória de tipos de valor.

O restante deste capítulo aborda o gerenciamento da memória para tipos de referência - especificamente instâncias de classes. Quando criamos uma nova instância de classe, a memória é alocada para ser usada pela instância, como acontece com os tipos de valor. No entanto, a diferença é o que acontece quando você passa a instância da classe. Passar uma instância de classe a uma função ou armazená-la em uma propriedade cria uma referência adicional à mesma memória, em vez de copiar a própria instância. Ter múltiplas referências à mesma memória significa que, quando qualquer uma delas muda a instância de classe, essa mudança fica aparente para todas as referências.

A linguagem Swift não requer que você gerencie manualmente a memória como em linguagens como a C. Em vez disso, cada instância de classe tem uma *contagem de referência*, que é o número de referências à memória que constituem a instância de classe. A instância permanece ativa se a contagem de referência for superior a 0. Uma vez que a contagem de referência é igual a 0, a instância é desalocada e o método `deinit` é executado.

Em um passado nem tão distante, os aplicativos desenvolvidos para iOS e Mac OS X em Objective-C usavam "contagem de referência manual". A contagem de referência manual exigia que você, o programador, gerenciasse as contagens de referência de todas as suas instâncias de classes. Cada classe tinha um método de *reter* o objeto (aumentando sua contagem de referência) e um método de *liberar* a instância (diminuindo sua contagem de referência). Como pode imaginar, a contagem de referência manual resultava em muitos bugs: se você retiver uma instância excessivamente, ela nunca será desalocada (causando o que chamamos de vazamento de memória), mas se você liberar uma instância excessivamente, provavelmente ocorrerá uma falha.

Em 2011, a Apple introduziu a *contagem de referência automática*, ou ARC, para a linguagem Objective-C. Sob a ARC, o compilador fica responsável por analisar o seu código e inserir chamadas de retenção e liberação em todos os lugares adequados. A linguagem Swift também é construída com base na ARC. Você não precisa fazer nada para gerenciar a contagem de referência de instâncias de classes - o compilador faz isso para você. Contudo, ainda é importante entender como o sistema funciona. Existem alguns erros comuns que podem provocar problemas de gerenciamento de memória.

Ciclos de referências fortes

Crie uma nova ferramenta de linha de comando com o nome CyclicalAssets. Adicione um arquivo novo a seu projeto com o nome Person.swift e insira a seguinte definição da classe Person:

```
import Foundation

class Person : Printable {
    let name: String

    var description: String {
        return "Person(\(name))"
    }

    init(name: String) {
        self.name = name
    }

    deinit {
        println("\(self) is being deallocated")
    }
}
```

A classe Person tem uma única propriedade definida como inicializador. Ela está em conformidade com o protocolo Printable por implementar a propriedade computada description. Você adicionou uma implementação de deinit para poder ver quando uma pessoa estiver sendo desalocada; isto é, a memória estiver sendo recuperada porque a contagem de referência diminuiu até 0. Modifique main.swift para criar uma opcional Person:

```
import Foundation

//println("Hello, world!")
var bob: Person? = Person(name: "Bob")
println("created \(bob)")

bob = nil
println("the bob variable is now \(bob)")
```

Você criou uma nova Person?, exibiu o nome da pessoa e definiu como nil. (Isso foi deixado como uma opcional para que você pudesse defini-la como nula e, portanto, ver a execução de deinit.) Compile e execute o seu programa. Você verá o seguinte resultado:

```
created Optional(Person(Bob))
Person(Bob) is being deallocated
the bob variable is now nil
```

A variável bob é uma opcional que contém uma instância de classe - um tipo de referência. Por padrão, todas as referências criadas são *referências fortes*, o que significa que aumentam a contagem de referência da instância a qual se referem. Portanto, a Person chamada de Bob tem uma contagem de referência igual a 1 depois de ser criada e atribuída à variável bob. Ao definir bob como nulo, a contagem de referência de Bob é diminuída. Então, é possível ver a mensagem Person Bob is being deallocated, porque a contagem de referência diminuiu até 0.

Em seguida, crie um novo arquivo em Swift chamado de Asset.swift e insira uma classe Asset:

```

import Foundation

class Asset : Printable {
    let name: String
    let value: Double
    var owner: Person?

    var description: String {
        if let actualOwner = owner {
            return "Asset(\(name), worth \(value), owned by \(actualOwner))"
        } else {
            return "Asset(\(name), worth \(value), not owned by anyone)"
        }
    }

    init(name: String, value: Double) {
        self.name = name
        self.value = value
    }

    deinit {
        println("\(self) is being deallocated")
    }
}

```

A classe Asset é muito similar à classe Person. Asset tem as propriedades name e value, está em conformidade com Printable e exibe a mensagem quando é desalocada. Ela também tem uma propriedade armazenada variável, owner, que vai se referir a Person, que tem esse ativo. owner é opcional porque um ativo pode muito bem existir sem que alguém seja seu dono. Crie alguns ativos em main.swift:

```

import Foundation

var bob: Person? = Person(name: "Bob")
println("created \(bob)")

var laptop: Asset? = Asset(name: "Shiny Laptop", value: 1500.0)
var hat: Asset? = Asset(name: "Cowboy Hat", value: 175.0)
var backpack: Asset? = Asset(name: "Blue Backpack", value: 45.0)

bob = nil
println("the bob variable is now \(bob)")

laptop = nil
hat = nil
backpack = nil

```

Você usou opcionais novamente para que pudesse definir as instâncias como nulas, o que acionaria os métodos deinit. Como esperado, todos os ativos são desalocados e não têm donos:

```

created Optional(Person(Bob))
Person(Bob) is being deallocated
Asset(Shiny Laptop, worth 1500.0, not owned by anyone) is being deallocated
Asset(Cowboy Hat, worth 175.0, not owned by anyone) is being deallocated
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
the bob variable is now nil

```

Pessoas podem possuir coisas; a sua classe de pessoas modelará essa qualidade com uma propriedade para os ativos. Volte para Person.swift e adicione uma propriedade e um método para pessoas obterem ativos:

```

import Foundation

class Person : Printable {
    let name: String
    var assets = [Asset]()

    var description: String {
        return "Person(\(name))"
    }

    init(name: String) {
        self.name = name
    }

    deinit {
        println("\(self) is being deallocated")
    }

    func takeOwnershipOfAsset(asset: Asset) {
        asset.owner = self
        assets.append(asset)
    }
}

```

Você adicionou `assets`, um array de `Assets` que a pessoa possui, e `takeOwnershipOfAsset`, um método para dar um ativo a uma pessoa. A posse de um ativo significa que a pessoa o adiciona ao array de `assets` e configura a propriedade `owner` do ativo para se referir de volta a ela. Em `main.swift`, dê alguns assets a Bob:

```

var laptop: Asset? = Asset(name: "Shiny Laptop")
var hat: Asset? = Asset(name: "Cowboy Hat")
var backpack: Asset? = Asset(name: "Blue Backpack")

bob?.takeOwnershipOfAsset(laptop!)
bob?.takeOwnershipOfAsset(hat!)

bob = nil

```

Compile e execute novamente. O resultado pode ser surpreendente:

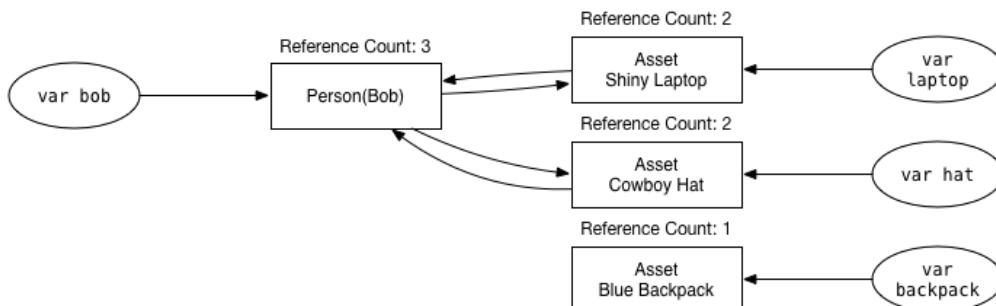
```

created Optional(Person(Bob))
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
the bob variable is now nil

```

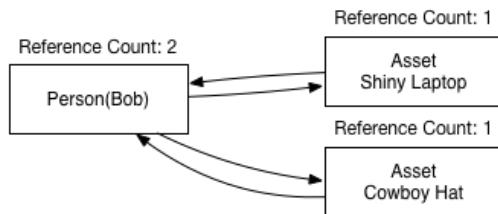
A única instância que está sendo desalocada agora é a mochila - a contagem de referência diminuiu até 0 quando você definiu `backpack = nil`. O laptop, o chapéu e o próprio Bob não estão mais sendo desalocados. Por que não? Antes de definir qualquer uma das variáveis como nula no `main.swift`, considere o seguinte diagrama de quem se refere a quem:

Figure 21.1 CyclicalAssets antes



Cada instância é rotulada com sua contagem de referência atual. A contagem de referência é exatamente o número de setas que apontam para a instância; isto é, o número de referências à instância. Depois de definir todas as variáveis em `main.swift` como nulas, essas referências somem, deixando o seguinte:

Figure 21.2 CyclicalAssets depois



Você criou dois *ciclos de referências fortes*, que é o termo usado quando duas instâncias têm referências fortes entre si. Bob tem uma referência ao laptop (por meio de sua propriedade `assets`) e o laptop tem uma referência a Bob (por meio de sua propriedade `owner`). O mesmo ciclo existe entre Bob e o chapéu. A memória para essas instâncias não pode mais ser alcançada - todas as variáveis que apontam para elas desapareceram -, mas a memória nunca será recuperada, porque cada instância tem uma contagem de referência superior a 0.

A solução para um ciclo de referências fortes é quebrar o ciclo. Você pode quebrar o ciclo no método `deinit` de `Person` executando um loop em cada ativo e configurando o dono como nulo. Contudo, a linguagem Swift tem uma palavra que causa o mesmo efeito automaticamente. Modifique `Asset` para tornar `owner` `property` uma *referência fraca* em vez de uma referência forte:

```

class Asset : Printable {
    let name: String
    let value: Double
    var owner: Person?
    weak var owner: Person?

    // ... rest of Asset remains unchanged
}
  
```

Uma referência fraca é aquela que não aumenta a contagem de referência da instância a qual se refere. Nesse caso, fazer com que `owner` seja uma referência fraca significa que, ao atribuir Bob como dono do laptop e do chapéu, a contagem de referência de Bob não aumentou. A única referência forte a Bob é a variável `bob` em `main.swift`. Quando você define a variável `bob` como nula, a contagem de referência em Bob diminui até 0, logo ela é desalocada. Quando Bob é desalocado, ele não contém mais uma referência forte a seus ativos, então a contagem de referência desses ativos também diminui até 0. Se você executar o programa novamente, verá que todos os objetos são desalocados:

```

created Optional(Person(Bob))
Person(Bob) is being deallocated
the bob variable is now nil
Asset(Shiny Laptop, worth 1500.0, not owned by anyone) is being deallocated
Asset(Cowboy Hat, worth 175.0, not owned by anyone) is being deallocated
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
  
```

O que acontecerá com uma referência fraca se a instância a qual ela se refere for desalocada? A referência fraca é definida como nula. É possível ver isso em ação editando `main.swift`:

```

bob?.takeOwnershipOfAsset(laptop!)
bob?.takeOwnershipOfAsset(hat!)

println("While Bob is alive, hat's owner is \(hat!.owner)")
bob = nil
println("the bob variable is now \(bob)")
println("After Bob is deallocated, hat's owner is \(hat!.owner)")
  
```

Referências fracas têm dois requisitos:

- As referências fracas sempre devem ser declaradas como `var`, não `let`.
- As referências fracas sempre devem ser declaradas como opcionais

Ambos os requisitos são resultado de as referências fracas serem alteradas para nulas se a instância a qual elas apontam é desalocada. Os únicos tipos que podem se tornar nulos são opcionais, então as referências devem ser opcionais. As instâncias declaradas com `let` não podem mudar, então as referências fracas devem ser declaradas com `var`.

Na maioria dos casos, os ciclos de referências fortes como os que você acabou de resolver são fáceis de evitar. `Person` é uma classe que possui ativos, então faz sentido que ela mantenha referências fortes aos ativos. `Asset` é uma classe possuída por uma `Person`; se ela quiser uma referência a seu próprio dono, essa referência deverá ser fraca. Afinal, uma pessoa possui um ativo, mas um ativo não possui uma pessoa!

Existe uma maneira muito mais sutil de criar ciclos de referência: a captura de `self` em um fechamento.

Ciclos de referência em fechamentos

Hora de adicionar uma classe de contador para acompanhar o patrimônio líquido da `Person`. Crie um novo arquivo em Swift chamado `Accountant.swift` e insira o seguinte:

```
import Foundation

class Accountant {
    typealias NetWorthChanged = (Double) -> ()

    var netWorthChangedHandler: NetWorthChanged? = nil
    var netWorth: Double = 0.0 {
        didSet {
            netWorthChangedHandler?(netWorth)
        }
    }

    func gainedNewAsset(asset: Asset) {
        netWorth += asset.value
    }
}
```

`Accountant` define uma typealias, `NetWorthChanged`, que é um fechamento que recebe um `Double` (o novo valor do patrimônio líquido) e não retorna nada. Possui duas propriedades: `netWorthChangedHandler`, que é um fechamento opcional para chamar quando houver mudança no patrimônio líquido, e `netWorth`, o patrimônio líquido atual de uma pessoa. `netWorth` tem um observador de propriedade `didSet` que chama o fechamento `netWorthChangedHandler` se for diferente de nulo. Finalmente, `gainedNewAsset` deve ser chamado para indicar ao contador que o valor de um novo ativo deve ser adicionado ao valor do patrimônio líquido.

Atualize `Person.swift` para que um contador monitore o patrimônio líquido:

```

import Foundation

class Person : Printable {
    let name: String
    let accountant = Accountant()
    var assets = [Asset]()

    var description: String {
        return "Person(\(name))"
    }

    init(name: String) {
        self.name = name

        accountant.netWorthChangedHandler = {
            netWorth in

            self.netWorthDidChange(netWorth)
            return
        }
    }

    deinit {
        println("\(self) is being deallocated")
    }

    func takeOwnershipOfAsset(asset: Asset) {
        asset.owner = self
        assets.append(asset)
        accountant.gainedNewAsset(asset)
    }

    func netWorthDidChange(netWorth: Double) {
        println("The net worth of \(self) is now \(netWorth)")
    }
}

```

Você adicionou uma propriedade `accountant` que tem como valor padrão um novo `Accountant`. `Person` tem uma referência forte a `Accountant`, o que é perfeitamente plausível. Em `init`, você definiu `netWorthChangedHandler` no contador para chamar o novo método `netWorthDidChange`, que apenas registra o novo patrimônio líquido da pessoa. Finalmente, você atualizou `takeOwnershipOfAsset` para notificar o contador sobre novos ativos. Compile e execute seu programa; você verá o seguinte:

```

created Optional(Person(Bob))
The net worth of Person(Bob) is now 1500.0
The net worth of Person(Bob) is now 1675.0
While Bob is alive, hat's owner is Optional(Person(Bob))
the bob variable is now nil
After Bob is deallocated, hat's owner is Optional(Person(Bob))
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated

```

Você recebeu mensagens de registro sobre a alteração do patrimônio líquido, então todo o código de contador adicionado parece estar funcionando corretamente. Contudo, parece que voltou a ocorrer vazamento de memória: Bob, o laptop e o chapéu não estão sendo desalocados. Por que essas instâncias não estão sendo removidas da memória?

Na verdade, o seu código novo inclui um ciclo de referências fortes não tão óbvio. `Person` tem uma referência forte a `Accountant`, mas `Accountant` não tem uma referência forte de volta a `Person`, pelo menos à primeira vista. Para ter uma ideia do que está acontecendo, tente modificar o método `init` de `Person` (isso provocará um erro de compilador):

```
init(name: String) {
    self.name = name

    accountant.netWorthChangedHandler = {
        netWorth in

        self.netWorthDidChange(netWorth)
        netWorthDidChange(netWorth)
        return
    }
}
```

Tente compilar seu programa agora. A mensagem de erro recebida diz: `Call to method 'netWorthDidChange' in closure requires explicit 'self.'` to make capture semantics explicit. O que é "semântica de captura" de um fechamento?

Um fechamento tem seu próprio escopo em sua definição. Por padrão, um fechamento aceita uma referência forte a qualquer variável usada dentro do escopo. `netWorthDidChange` é um método em `self`, portanto, chamá-lo só faria com que o fechamento recebesse uma referência forte a `self`. Isto explica por que está acontecendo vazamento de memória: `Accountant`, na verdade, tem uma referência forte de volta a `Person!` `netWorthChangedHandler` de `Accountant` contém uma referência forte a `Person`, que o possui por meio do `self` de `Person`.

Veja novamente a mensagem de erro: "to make capture semantics explicit". A linguagem Swift poderia permitir o uso de `self` implicitamente em fechamentos, mas isso também facilitaria a criação acidental de ciclos de referências fortes como aconteceu aqui. Em vez disso, essa linguagem exige que você seja *explícito* sobre o uso de `self`, impondo que você considere se um ciclo de referência é uma possibilidade.

Para mudar a semântica de captura de um fechamento para capturar referências de maneira fraca, você pode usar uma *lista de captura*. Modifique `Person.swift` para usar uma lista de captura ao criar o fechamento:

```
init(name: String) {
    self.name = name

    accountant.netWorthChangedHandler = {
        netWorth in
        [weak self] netWorth in

        netWorthDidChange(netWorth)
        self?.netWorthDidChange(netWorth)
        return
    }
}
```

A sintaxe da lista de captura é uma lista de variáveis dentro de colchetes (`[]`) imediatamente antes da lista de argumentos de fechamento. A lista de captura que você escreveu aqui manda a Swift capturar `self` de maneira fraca em vez de forte. Agora que o fechamento de `Accountant` não faz mais referência forte a `Person`, o ciclo de referências fortes está quebrado.

Perceba o uso de `self?` no corpo do fechamento. Como `self` é capturado de maneira fraca e todas as instâncias fracas devem ser opcionais, `self` é opcional dentro do fechamento. Execute seu programa novamente e confirme se todas as instâncias estão sendo desalocadas adequadamente:

```
created Optional(Person(Bob))
The net worth of Person(Bob) is now 1500.0
The net worth of Person(Bob) is now 1675.0
While Bob is alive, hat's owner is Optional(Person(Bob))
Person(Bob) is being deallocated
the bob variable is now nil
After Bob is deallocated, hat's owner is nil
Asset(Shiny Laptop, worth 1500.0, not owned by anyone) is being deallocated
Asset(Cowboy Hat, worth 175.0, not owned by anyone) is being deallocated
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
```

Conclusão

Você não precisa gerenciar a memória manualmente na linguagem Swift. A ARC faz a coisa certa na maior parte do tempo. Infelizmente, ela não resolve o problema de ciclos de referências fortes. Você aprendeu a pensar a respeito de relações de posse entre instâncias de classes. Você também aprendeu por que fechamentos exigem que você seja explícito sobre as referências a `self` e como usar a lista de captura para capturar `self` de maneira fraca em um fechamento.

Desafios

Desafio de bronze

A ideia de posse de ativos por uma `Person` está incompleta. `Person` tem uma maneira de aceitar a posse de um ativo, mas nenhuma maneira de desistir da posse de um ativo. Atualize `Person` para que uma instância possa abrir mão de um ativo. (Dica: Você provavelmente também precisará atualizar `Accountant` se quiser um valor de patrimônio líquido válido.)

Desafio de prata

Crie outra `Person` no `main.swift`. Imediatamente após dar a Bob a posse do laptop, tente dar à nova `Person` a posse do mesmo laptop. Agora ambas as pessoas possuem o laptop! Corrija esse bug.

22

Igualável e comparável

Boa parte da programação depende da comparação de valores. É importante ser capaz de saber se um valor é igual a outro valor. Além disso, muitas vezes, é importante ser capaz de saber *como* um valor se compara a outro. É menor ou maior que determinado valor? A linguagem Swift fornece dois protocolos para testar a igualdade e a comparabilidade: `Equatable` e `Comparable`.

Este capítulo mostrará como você pode fazer com que um tipo personalizado esteja em conformidade com esses protocolos. Isso envolverá a implementação de algumas funções que ensinarão às instâncias do seu tipo como se comparar com outras instâncias do mesmo tipo. Crie um novo playground chamado de `Comparison.playground` para começar.

Conformidade com igualável

Comece criando um novo tipo que ainda não esteja em conformidade com o protocolo `Equatable`.

```
struct Point {  
    let x: Int  
    let y: Int  
}
```

A struct acima define um tipo `Point`. `Point` usa suas propriedades `x` e `y` para descrever um local específico em um plano bidimensional.

No momento, `Point` não sabe como determinar se uma instância é igual à outra. Crie duas instâncias desse tipo para ver o que acontece quando você verifica se as duas são iguais.

```
struct Point {  
    let x: Int  
    let y: Int  
}  
let a = Point(x: 3, y: 4)  
let b = Point(x: 3, y: 4)
```

Você criou dois pontos novos, `a` e `b`, usando o inicializador memberwise fornecido pelo compilador. Observe que os pontos receberam os mesmos dados das propriedades `x` e `y`. Tente usar o operador `==` para testar se há igualdade entre esses dois pontos.

```
struct Point {  
    let x: Int  
    let y: Int  
}  
  
let a = Point(x: 3, y: 4)  
let b = Point(x: 3, y: 4)  
let abEqual = (a == b)
```

Você verá que essa verificação de igualdade não funciona. Na verdade, ela gera um erro do compilador. Esse erro é resultado do fato de que você ainda não ensinou à struct `Point` como testar se há igualdade entre duas instâncias.

Para isso, Point terá que estar em conformidade com o protocolo `Equatable`. Adicione o código abaixo à declaração de sua struct.

```
struct Point {
    let x: Int
    let y: Int
}

struct Point: Equatable {
    let x: Int
    let y: Int
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)

let abEqual = (a == b)
```

Agora, você verá um novo erro; dessa vez, o erro está na linha onde a struct `Point` é declarada. O erro diz: Type '`Point`' does not conform to protocol '`Equatable`'. Para descobrir como estar em conformidade com o protocolo `Equatable`, abra a documentação clicando no item "Help" do menu e selecione "Documentation and API Reference".

A documentação informa que você precisa sobrecarregar o operador `==` no escopo global para que esteja em conformidade com esse protocolo. Por que você precisa que esteja em conformidade com o escopo geral? Se parar para pensar, esse requisito faz sentido porque `==` não é um método que pode ser chamado em um tipo específico. Em vez disso, `==` aparece entre e opera em dois destinos ao mesmo tempo: ele verifica se o valor à esquerda é igual ao valor à direita. Esse tipo de operador é chamado de *operador infix*.

Crie uma implementação de `==` que compare duas instâncias do tipo `Point` e teste se há igualdade.

```
struct Point: Equatable {
    let x: Int
    let y: Int
}

func ==(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y)
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)

let abEqual = (a == b)
```

Você acabou de escrever uma nova implementação do operador `==` (nota: um operador é apenas uma função com nome especial; por exemplo, `==`). Essa definição tem dois argumentos: um argumento `lhs` para o lado esquerdo e um `rhs` para o lado direito da verificação de igualdade. Ambos os argumentos devem ser do tipo `Point`.

A implementação da função é bastante simples. Ela compara os valores de `x` e `y` para ambas as instâncias do tipo `Point` passadas para os argumentos da função. Por último, a função retornará um `Bool`, indicando se as instâncias são iguais ou não.

Dê uma olhada na barra lateral de resultados do seu playground. Você notará que os erros sumiram e que o teste para ver se há igualdade entre os dois pontos `a` e `b` teve êxito. Os dois pontos são iguais porque os valores de `x` e `y` são iguais.

Mas não é só isso. A biblioteca padrão da linguagem Swift oferece uma implementação padrão da função `!=` que depende da definição de `==`. Esse recurso significa que, se seu tipo estiver em conformidade com `Equatable` ao implementar sua própria versão de `==`, ele também tem uma implementação da função `!=` que funciona.

Experimente adicionar o teste abaixo.

```
struct Point: Equatable {
    let x: Int
    let y: Int
}

func ==(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y)
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)

let abEqual = (a == b)
let abNotEqual = (a != b)
```

A barra lateral de resultados deve atualizar para mostrar que esse teste de desigualdade resulta em `false`. Em outras palavras, os dois pontos são *iguais*, o que significa que não são *desiguais*. Você pode considerar `false` como se fosse a resposta "Não!" para a seguinte pergunta: "Os pontos A e B são desiguais?".

Conformidade com comparável

Agora que o tipo `Point` está em conformidade com o protocolo `Equatable`, você pode estar interessado em formas de comparação com mais nuances. Por exemplo, talvez você esteja interessado em saber se um ponto é inferior a outro ponto. Essa funcionalidade é realizada com a conformidade com o protocolo `Comparable`.

Abra a documentação e pesquise por "Comparable" para determinar o que é necessário. Você descobrirá que precisa sobrestrar um operador: o operador infix `<`. Adicione o código a seguir a sua struct para que ela esteja em conformidade com `Comparable`.

```
struct Point: Equatable {
    let x: Int
    let y: Int
}
struct Point: Equatable, Comparable {
    let x: Int
    let y: Int
}

func ==(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y)
}

func <(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x < rhs.x) && (lhs.y < rhs.y)
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)

let abEqual = (a == b)
let abNotEqual = (a != b)
```

Você adicionou uma nova declaração ao `Point`, que diz que ele agora está em conformidade com o protocolo `Comparable`. Um pouco abaixo disso, você adicionou uma implementação do operador `<`. Essa implementação funciona de maneira similar à que você viu acima para a implementação de `==`. Ela verifica se o ponto passado à esquerda é inferior ao ponto passado à direita. Se os valores de `x` e `y` para o ponto à esquerda forem ambos menores que os valores à direita, a função retornará `true`. Caso contrário, a função retornará `false`, indicando que o ponto à esquerda é maior que o ponto à direita.

Crie dois pontos novos para testar essa função.

```
struct Point: Equatable, Comparable {
    let x: Int
    let y: Int
}

func ==(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y)
}

func <(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x < rhs.x) && (lhs.y < rhs.y)
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)

let abEqual = (a == b)
let abNotEqual = (a != b)
let c = Point(x: 2, y: 6)
let d = Point(x: 3, y: 7)

let cdEqual = (c == b)
let cLessThanD = (c < d)
```

Perceba que esses dois pontos recebem valores diferentes para `x` e `y`. É necessário também verificar se `c` e `d` são iguais, o que retorna `false`; os dois pontos não são iguais. Por último, a sobrecarga do operador `<` é realizada para determinar se `c` é inferior a `d`. Nesse caso, a comparação avalia como `true`. O ponto `c` é inferior ao ponto `d` porque os valores de `x` e `y` são menores que o valor de `d`.

Assim como a conformidade com o protocolo `Equatable`, a implementação de uma função, na verdade, oferece muito mais funcionalidades. Isso é devido ao fato de que a biblioteca padrão da linguagem Swift define os operadores `>`, `>=` e `<=` com base nos operadores `<` e `==`. De forma correspondente, se o seu tipo estiver em conformidade com `Comparable`, ele receberá as implementações desses operadores gratuitamente.

Teste essa funcionalidade adicionando uma série de novas comparações.

```
struct Point: Equatable, Comparable {
    let x: Int
    let y: Int
}

func ==(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y)
}

func <(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x < rhs.x) && (lhs.y < rhs.y)
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)

let abEqual = (a == b)
let abNotEqual = (a != b)
let c = Point(x: 2, y: 6)
let d = Point(x: 3, y: 7)

let cdEqual = (c == b)
let cLessThanD = (c < d)

let cLessThanOrEqualToD = (c <= d)
let cGreaterThanOrEqualToD = (c > d)
let cGreaterThanOrEqualToD = (c >= d)
```

Essas três últimas comparações verificam se:

- a) c é inferior ou igual a d,
- b) c é superior a d,
- c) c é superior ou igual a d.

Como esperado, essas comparações avaliam como `true`, `false` e `false` respectivamente.

Herança do comparável

`Comparable` é, na verdade, herdeiro de `Equatable`. É possível adivinhar o que essa herança implica. Para estar em conformidade com o protocolo `Comparable`, você também deve estar em conformidade com o protocolo `Equatable`.

Como `Equatable` exige o operador `==`, qualquer conformidade com `Comparable` deve fornecer uma implementação dessa função. Observe que essa relação entre os dois protocolos indica que o tipo não precisa declarar explicitamente a conformidade com `Equatable`. Remova a declaração explícita de conformidade com `Equatable` de sua struct `Point`.

```
struct Point: Equatable, Comparable {
    let x: Int
    let y: Int
}

func ==(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x == rhs.x) && (lhs.y == rhs.y)
}

func <(lhs: Point, rhs: Point) -> Bool {
    return (lhs.x < rhs.x) && (lhs.y < rhs.y)
}

...
```

Você deve notar que o playground funciona da mesma forma que antes.

Uma última observação em relação ao estilo. Embora seja extremamente explícito declarar a conformidade tanto com `Equatable` como com `Comparable`, não é necessário fazer isso. Se seu tipo estiver em conformidade com `Comparable`, ele também deve estar em conformidade com `Equatable`. Essa questão é um detalhe listado na documentação, o que faz com que seja uma consequência esperada da conformidade com `Comparable`. A adição da conformidade explícita com `Equatable` não adiciona tanta informação assim. Por outro lado, como não está listado na documentação, talvez faça sentido ter um tipo que esteja explicitamente em conformidade com todos os protocolos envolvidos ao estar em conformidade com um protocolo personalizado herdeiro de outro protocolo.

Conclusão

Este capítulo introduziu a conformidade com dois protocolos fornecidos pelo sistema: `Equatable` e `Comparable`. `Equatable` é usado para determinar se duas instâncias de determinado tipo são iguais. `Comparable` é usado para obter uma comparação com nuances entre duas instâncias de um tipo. Por exemplo, é útil determinar se uma instância é inferior ou superior a outra. É bom se acostumar com esses protocolos; você vai usá-los com frequência.

Desafios

Desafio de bronze

É possível que você queira adicionar dois pontos juntos no futuro. A adição de dois pontos juntos deve retornar um novo `Point` que adiciona os valores de x e y dos pontos dados respectivamente. Torne possível adicionar dois pontos juntos sobrecarregando o operador `+` para a struct `Point`.

Desafio de ouro

Crie uma nova classe Person. Ela deve ter duas propriedades: name e age. Por questões de conveniência, crie um inicializador que forneça argumentos para ambas as propriedades.

Em seguida, crie duas instâncias novas da classe Person. Atribua essas instâncias a duas constantes chamadas de p1 e p2. Também crie um array chamado de people para conter essas instâncias, coloque-as dentro dele.

Ocasionalmente, você precisará encontrar o índice de uma instância de um tipo personalizado dentro de um array. Tente fazer isso com `find(:, :)`. O primeiro argumento espera a coleção que você vai procurar, e o segundo argumento é o valor cujo índice você quer encontrar. Use a função para encontrar o índice de `p1` dentro do array `people`.

Você receberá um erro. Reserve um momento para entender o erro e solucione-o. Você deve conseguir atribuir o resultado de `find(: , :)` a uma constante chamada de `p1Index`. O valor deve ser igual a 0.

Para os mais curiosos: Operadores personalizados

A linguagem Swift permite que o desenvolvedor crie seus próprios operadores personalizados. Esse recurso significa que você pode criar seu próprio operador para indicar que uma instância do tipo Person se casou com outra instância. Digamos, por exemplo, que você queira criar `+++` para casar uma instância com outra.

Crie uma nova classe Person assim:

```
class Person: Equatable {  
    var name: String  
    weak var spouse: Person?  
  
    init(name: String, spouse: Person?) {  
        self.name = name  
        self.spouse = spouse  
    }  
}
```

A classe tem duas propriedades: uma para um nome e outra para um cônjuge. Ela também tem um inicializador que atribuirá valores a essas propriedades. Observe que a propriedade `spouse` é uma opcional para indicar que uma pessoa pode não ter necessariamente um cônjuge.

Em seguida, crie duas instâncias dessa classe.

```
class Person: Equatable {
    var name: String
    weak var spouse: Person?
    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }
}
let matt = Person(name: "Matt", spouse: nil)
let drew = Person(name: "Drew", spouse: nil)
```

Agora, declare seu novo operador infix. Ele deve ser declarado em escopo global. Também defina como a função do novo operador vai funcionar.

```

class Person: Equatable {
    var name: String
    weak var spouse: Person?

    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }
}

let matt = Person(name: "Matt", spouse: nil)
let drew = Person(name: "Drew", spouse: nil)

infix operator +++ {}

func +++(lhs: Person, rhs: Person) {
    lhs.spouse = rhs
    rhs.spouse = lhs
}

```

O novo operador, `+++`, será usado para casar duas instâncias da classe `Person`. Como operador infix, ele será usado entre duas instâncias. Sendo assim, a implementação de `+++` simplesmente atribuirá cada instância à propriedade `spouse` da outra.

Exercite esse novo operador da seguinte maneira:

```

class Person: Equatable {
    var name: String
    weak var spouse: Person?

    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }
}

let matt = Person(name: "Matt", spouse: nil)
let drew = Person(name: "Drew", spouse: nil)

infix operator +++ {}

func +++(lhs: Person, rhs: Person) {
    lhs.spouse = rhs
    rhs.spouse = lhs
}

matt +++ drew
matt.spouse?.name
drew.spouse?.name

```

O código `matt +++ drew` serve para casar as duas instâncias. Você pode verificar se esse processo funcionou examinando a barra lateral de resultados do playground.

Embora esse operador "funcione" e não seja muito difícil determinar o que está acontecendo quando damos uma olhada, no geral, é uma boa ideia evitar declarar seus próprios operadores personalizados. Recomenda-se apenas criar operadores personalizados para seus próprios tipos quando o operador puder ser reconhecido por alguém que for ler o seu código. Isso normalmente significa que seus próprios operadores personalizados devem se restringir a operadores matemáticos conhecidos pela maioria das pessoas. De fato, a linguagem Swift permite que você use uma coleção bem definida de símbolos matemáticos para criar operadores personalizados. Por exemplo, você não pode refatorar o operador `+++` para que seja o Emoji do "rosto mandando um beijo" (isto é, U+1F61A).

O pensamento por trás dessa recomendação é resultado do medo de que alguém, ao revisar o seu código no futuro, possa não saber exatamente o que você quer dizer com `+++`. Afinal, nesse caso, isso é bastante ambíguo. Além disso, não é como se esse operador realizasse algo de maneira mais elegante ou eficiente do que `marry(_:_)` conseguiria sozinho.

Por exemplo, uma função `marry(_:_)` pode ficar parecida com o seguinte:

```
class Person: Equatable {
    var name: String
    weak var spouse: Person?

    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }

    func marry(spouse: Person) {
        self.spouse = spouse
        spouse.spouse = self
    }
}

let matt = Person(name: "Matt", spouse: nil)
let drew = Person(name: "Drew", spouse: nil)

infix operator +++ {}

func +++(lhs: Person, rhs: Person) {
    lhs.spouse = rhs
    rhs.spouse = lhs
}

matt +++ drew
matt.marry(drew)
matt.spouse?.name
drew.spouse?.name
```

O código `matt.marry(drew)` é muito mais legível. Está bastante claro o que ele está fazendo. Essas qualidades fazem com que, no futuro, seja mais fácil realizar a manutenção desse código.

Index

A

Anulação da inicialização, 159
assert, 113

C

Classe
 Herança, 124
Classes
 super, 125
Comentários, 7
Comparável, 203
Composição de protocolos, 174
Controle de acesso, 145
Correspondência de faixa, 36
Correspondência de padrão, 111
Currying de função, 131

E

Editor assistente, 9, 43, 45

F

Fechamentos
 Nomes de argumento abreviados, 96
 Sintaxe de fechamento final, 97
Ferramenta de linha de comando, 7
Funções
 Parâmetros variadic, 85

I

Identidade, 129
Igualável, 203
Igualdade, 129
Inferência de tipo, 47
Inicialização
 Conveniência, 156
 Delegação, 152
 Designada, 155
 Memberwise, 150
 Vazia, 150
Inicializadores suscetíveis a falhas, 160
Instrução de transferência de controle
 fallthrough, 36
Instruções de transferência de controle, 49, 50
 continuar, 49
 interromper, 49, 50
Int, 14
Interface de Programação de Aplicativos (API), 8
Interpolação de strings, 19, 37

L

Ligaçāo de opcionais, 91
Ligaçāo de valores, 37

M

Método, 109
Métodos
 Métodos de instânciā, 122
 Mutáveis, 122
 Métodos de tipo, 130

N

Nomes de parâmetros explícitos, 85
Nomes de parâmetros externos abreviados, 85

O

onde, 37
Opcionais
 desempacotamento forçado, 60
 Encadeamento de opcionais, 125
 Ligaçāo de opcionais, 124, 125
 Tipos de opcionais, 59
Opcionais>
 ligação opcional, 60
 opcionais desempacotadas implicitamente, 61
Opcional
 Tipo de retorno, 90
Operadores
 Infix, 204
Operadores lógicos, 21

P

Parâmetros in-out, 87
Playground, 6
Barra lateral de resultados, 70
Barra lateral de resultados>, 7
Editor de código, 7
Histórico de valores, 43
Linha do tempo, 43
Olhada rápida, 70
Playhead, 45
precondition, 113
Propriedades
 Leitura/gravação, 137
 Observadores de propriedades, 142
Propriedade computada, 141
Propriedades armazenadas, 137
Propriedades de tipos, 143
 Somente leitura, 137
Propriedades armazenadas
 Lentas, 138

S

Sintaxe de expressão de fechamento, 95

String

 Equivalência canônica, 56

Subíndice, 71

T

Tipos aninhados, 138

Tipos de coleção, 14

Tipos de função, 91

Tipos de referência, 127

Tipos de valor, 127

Tupla, 38

tupla, 88

 com nome, 89

typealias, 177

U

Unicode, 54

 Escalares de Unicode, 54

V

Valores padrão de parâmetros., 86

X

Xcode

 instalar, 5