CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA

FACULDADE DE TECNOLOGIA DE CAMPINAS

CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GIACOMO VALENTIM SILVA MAGRI

GUIA TEÓRICO E PRÁTICO PARA INICIANTES EM PROGRAMAÇÃO FUNCIONAL COM SCALA

CAMPINAS/SP

2021

CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA

FACULDADE DE TECNOLOGIA DE CAMPINAS

CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GIACOMO VALENTIM SILVA MAGRI

GUIA TEÓRICO E PRÁTICO PARA INICIANTES EM PROGRAMAÇÃO FUNCIONAL COM SCALA

Trabalho de Graduação apresentado por Giacomo Valentim Silva Magri, como pré-requisito para a conclusão do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, da Faculdade de Tecnologia de Campinas, elaborado sob a orientação do Prof. Msc. Anderson Luiz Coan.

CAMPINAS/SP

2021

RESUMO E PALAVRAS-CHAVE

A linguagem Scala não possui grande popularidade e conta com poucos recursos em português. Visando aumentar a base teórica acerca da linguagem, foi desenvolvido um material em formato de e-book, contando com a experiência de desenvolvedores da área, para guiar um desenvolvedor iniciante com a linguagem, abordando tópicos de programação orientada a objetos, programação funcional, uso de funções popular e temas específicas de Scala.

Palavras-chave: Scala, Programação Funcional, Material Didático, Português.

ABSTRACT

Scala language doesn’t have great popularity and has few contents written in Portuguese. Looking to expand its theoretical basis, an e-book was built with the experience of Scala developers to guide a new developer with the language, approaching topics of object-oriented programming, functional programming, usage of popular functions, and specific topics of Scala.

Keywords: Scala, Functional Programming, Didactic Material, Portuguese.

LISTA DE TABELAS E QUADROS

Sumário

1. INTRODUÇÃO

A área de dados é bastante popular e visada por novos desenvolvedores que entram no mercado de trabalho, ao mesmo tempo, o conjunto de ferramentas utilizadas para desenvolver na área ainda não é bem estabelecido (MARQUESONE, 2016) e muitos desenvolvedores nem mesmo tiveram um contato inicial com muitas dessas tecnologias. Visto que existe uma grande busca por aprender linguagens, ferramentas e conceitos vinculados à Big Data, a proposta deste trabalho é oferecer suporte inicial para desenvolvedores que, pela primeira vez, entrarão em contato com Scala ou com o paradigma funcional.

1.1. CONTEXTUALIZAÇÃO

Scala é uma das linguagens utilizadas ao desenvolver com Spark[1], um framework Big Data criado e suportado pela Apache. Segundo o StackOverflow, em 2020 apenas 3.6% dos desenvolvedores entrevistados ao redor do mundo trabalhavam com Scala, porém é uma linguagem que possui uma comunidade com certa relevância e inúmeras documentações, cursos e artigos em inglês disponíveis na internet.

Mesmo com uma comunidade relativamente expressiva e inúmeros conteúdos publicados, no Brasil existe muito pouco material em português e segundo o Instituto de Pesquisa Data Popular, apenas 10% da população brasileira entre 18 e 24 anos possui algum conhecimento em inglês. A partir desses dados, é possível concluir que apenas uma parcela de desenvolvedores brasileiros conseguirá encontrar apoio na internet para começar sua jornada utilizando Scala como linguagem.

1.1.1. PROBLEMA DA PESQUISA

Scala não possui uma popularidade tão grande, ocupando o trigésimo quinto lugar no ranking de 2021[2] organizado pela Tiobe e ainda possui uma outra barreira para novos desenvolvedores, é uma linguagem multi-paradigma com foco no paradigma funcional. Esses desafios que novos desenvolvedores precisam superar ao começar a trabalhar com a linguagem podem se mostrar muito frustrantes, já que

[1] Site oficial do framework. Disponível em <https://spark.apache.org/>. Acessado em 01 jun. 2021
[2] Ranking de popularidade da Tiobe. Disponível em <https://www.tiobe.com/tiobe-index/>. Acessado em 08 nov. 2021.

a curva de aprendizado pode ser lenta e conhecer um novo paradigma é ainda mais difícil.

1.2. OBJETIVOS

Foi desenvolvido um material instrucional com foco em desenvolvedores iniciantes na linguagem, que apresenta e explica conceitos de programação funcional e estruturas da linguagem Scala totalmente em português. Neste trabalho, o objetivo é elaborar uma revisão bibliográfica do material gerado.

O material conta com teoria e prática abordando características do paradigma funcional, tópicos específicos da linguagem, estruturas orientadas a objetos em Scala e, através de exemplos, introduz conceitos e funções popularmente utilizadas por desenvolvedores adeptos ao paradigma funcional.

A proposta é ser um breve manual, um ponto de partida, para que novos desenvolvedores possam absorver conceitos e a partir disso explorar mais da linguagem e do próprio paradigma.

2. REVISÃO BIBLIOGRÁFICA

2.1. LINGUAGEM SCALA

Scala é uma linguagem com tipagem estática que funde orientação a objetos e programação funcional (ODERSKY, 2004). Scala possui interoperabilidade com Java e é capaz de rodar utilizando a Java Virtual Machine (JVM)[3] além de possuir a opção de ser interpretada em navegadores através do Scala.js[4] que transforma código Scala em código JavaScript altamente eficiente. Surgida em 2004 e possuindo Martin Odersky como figura principal por trás da linguagem, Scala foi evoluindo com o passar do tempo e em 2021 foi lançado Scala 3, uma nova versão que conta com inúmeras alterações em relação à sua predecessora.

A linguagem é utilizada em contexto de Big Data ao trabalhar com o framework Spark, também no desenvolvimento de aplicações web com o Play Framework[5], no design de APIs e na criação de serviços com arquitetura orientada a eventos utilizando o Akka[6].

Um dos paradigmas suportados por Scala é o paradigma funcional e mesmo que a linguagem ofereça diversas estruturas nativas para o desenvolvimento voltado ao paradigma, várias bibliotecas são desenvolvidas para oferecer suporte ao desenvolvedor. Dentre as bibliotecas disponíveis existem as que suportam estruturas funcionais ao lidar com banco de dados como o Slick[7] e outras que oferecem suporte ao desenvolvimento de soluções voltado a plataformas específicas, como o AckCord que oferece o necessário para a criação de bots para a plataforma Discord e ainda é uma ferramenta que está na lista de bibliotecas[8] que seguem os padrões da plataforma.

---

[3] Site oficial da linguagem Scala. Disponível em <https://www.scala-lang.org/>. Acessado em 01 jun. 2021.
[4] Site oficial do Scala.js. Disponível em <https://www.scala-js.org/>. Acessado em 01 jun. 2021.
[5] Site oficial do Play Framework. Disponível em <https://www.playframework.com/>. Acessado em 01 jun. 2021.
[6] Site oficial do framework Akka. Disponível em < https://akka.io/ />. Acessado em 01 jun. 2021.
[7] Site oficial da biblioteca Slick. Disponível em <https://scala-slick.org/>. Acessado em 01 jun. 2021.
[8] Bibliotecas que seguem o padrão do Discord. Disponível em <https://discord.com/developers/docs/topics/community-resources>. Acessado em 01 jun. 2021

2.2. NECESSIDADE DO USO DA LINGUAGEM

A linguagem, embora não esteja entre as mais populares do mercado, é utilizada por empresas como o Twitter e o Spotify e sua utilização aumenta ao passo que outros frameworks passam a disponibilizar integração com a linguagem e mais organizações adotam soluções Scala para seus projetos.

A área de dados utiliza a linguagem com o framework Spark. Existem artigos e instrutores que indicam o uso do framework com Scala como um substituto ao Pandas com Python. Entre as melhorias apontadas, estão performance, código mais seguro e legível, além de aproveitar uma estrutura específica do Spark, o Dataset, para permitir uma manipulação de dados rápido e com tipagem segura (CONNOR, 2020).

Além da utilização comercial, a linguagem pode ser utilizada em meio acadêmico para disciplinas de paradigma de programação, já que é uma linguagem multiparadigma e por seu suporte ao paradigma funcional, pode ajudar com a interdisciplinaridade de estudo nas áreas de cálculo e de programação, já que o paradigma funcional tem sua base no cálculo lambda e alguns conceitos matemáticos, como os Monads, são estruturas bem estabelecidas no contexto do paradigma (HEUNEN; JACOBS, 2006).

2.3. PROGRAMAÇÃO FUNCIONAL

O paradigma mais comum e voltado à criação de scripts, também é o paradigma que todo desenvolvedor tem seu primeiro contato, é o paradigma imperativo. Nele, cada linha de código escrita é executada em sequência, havendo desvios com chamadas de funções ou estruturas como o “goto”. O paradigma imperativo utiliza de estruturas de repetição, mudança de estado através de variáveis e estruturas condicionais para a estruturação do código.

Seguindo uma linha de pensamento que trata a computação não como uma sequência de instruções, mas como uma avaliação de funções matemáticas (JUNGHTON; GOULART, 2010), existe o paradigma funcional. No paradigma funcional, uma aplicação segue duas regras para sua criação: o código é feito utilizando apenas constantes, ou seja, não existem variáveis e mutabilidade; o código é escrito utilizando apenas funções puras (ALEXANDER, 2017). Uma função é considerada pura se: não possuir qualquer tipo de comunicação com o ambiente externo, como abrir um arquivo; não possuir efeitos colaterais, ou seja, uma função trata apenas de seu escopo interno e não deve modificar valores fora dela; lidar apenas com o que foi passado em sua assinatura, essa última afirmação complementa a afirmação anterior sobre não lidar com valores ou recursos que estejam fora do escopo da função, em outras palavras, se determinado valor não vier nos parâmetros daquela função, então não deve ser utilizado dentro dela.

Ao criar códigos utilizando programação funcional, por sua característica de evitar mutabilidade, estruturas de repetição que dependem do incremento ou decremento de variáveis não estão presentes, dessa forma, ao iterar sob um conjunto de dados, é preferível que seja utilizada a estratégia de recursão.

2.4. METODOLOGIA DE ENSINO E E-BOOK

A tecnologia e a virtualização do ensino são realidades cada vez mais comuns desde que o ensino à distância passou a ser popularizado. Conteúdos em vídeo, artigos em blog, fóruns educativos e e-books são disseminados em diferentes áreas do conhecimento, e áreas de exatas e humanas possuem inúmeras fontes de qualidade disponíveis na internet. A área de TI (Tecnologia da Informação) não é diferente, uma plataforma de cursos online, como a Udemy[9], conta com mais de 600 páginas de cursos[10] dentro da área, com diferentes tópicos e avaliações.

O formato de e-book para a criação do material instrucional se deve ao fato de que é um formato de fácil navegação, já que é possível redirecionar o leitor a partir dos títulos dos capítulos, além disso, é um formato que exige menos demanda de esforços para a sua produção se comparado ao formato de vídeo, que exige hardware, software e técnicas de edição para sua produção. Além disso, o e-book é um material que pode ser impresso, ter suas partes importantes grifadas (mesmo em visualização digital) e, por ser separado em pequenos tópicos, oferece uma leitura com maior liberdade de pausas entre assuntos.

Um material que consegue redirecionar o leitor através de links para vídeos, leituras extras e mais material de apoio, colabora com a aprendizagem do leitor e facilita a ampliação do conhecimento (CARGNELUTTI et al., 2018). Esses tópicos que dizem respeito à interatividade do material de estudo são facilmente aplicáveis no formato de e-book, já que através de links, o leitor pode ser redirecionado para um debate sobre um tema específico em um fórum online e ter contato com mais visões acerca do tema ou mesmo pode ser redirecionado para cursos gratuitos que ajudem a criar a base necessária para a absorção dos conceitos apresentados.

[9] Cursos de TI disponíveis em <https://www.udemy.com/courses/it-and-software/>. Acesso em 02 jun. 2021.
[10] Dado retirado do site oficial da Udemy ao pesquisar por cursos de TI.

2.5. RESULTADOS DA PESQUISA ACERCA DE TÓPICOS PARA O E-BOOK

Foi realizada uma pesquisa utilizando o Google Forms acerca de quais aspectos de Scala, programação orientada a objetos, programação funcional e alguns tópicos específicos ou avançados que seriam importantes para que uma pessoa pudesse começar a desenvolver com a linguagem. A pesquisa foi direcionada a desenvolvedores que programam ou já programaram em Scala profissionalmente e seu objetivo era nortear o desenvolvimento do sumário do e-book que foi desenvolvido.

A pesquisa contou com 17 respostas e revelou os seguintes resultados:

Tabela 1: Tópicos importantes (Orientação a Objetos)

Tabela 2: Tópicos importantes (Paradigma Funcional)

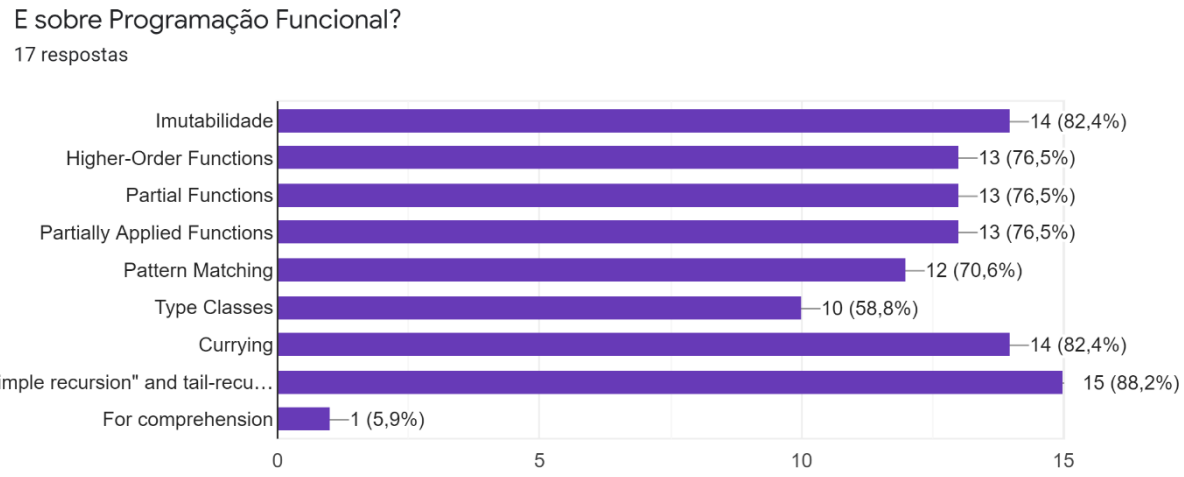


Tabela 3: Tópicos importantes específicos da linguagem

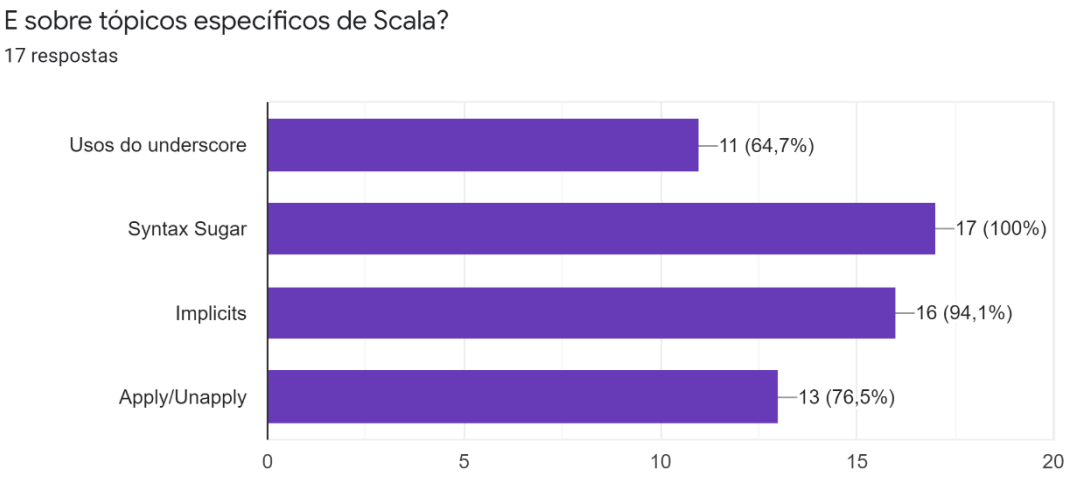


Tabela 4: Tópicos importantes considerados avançados

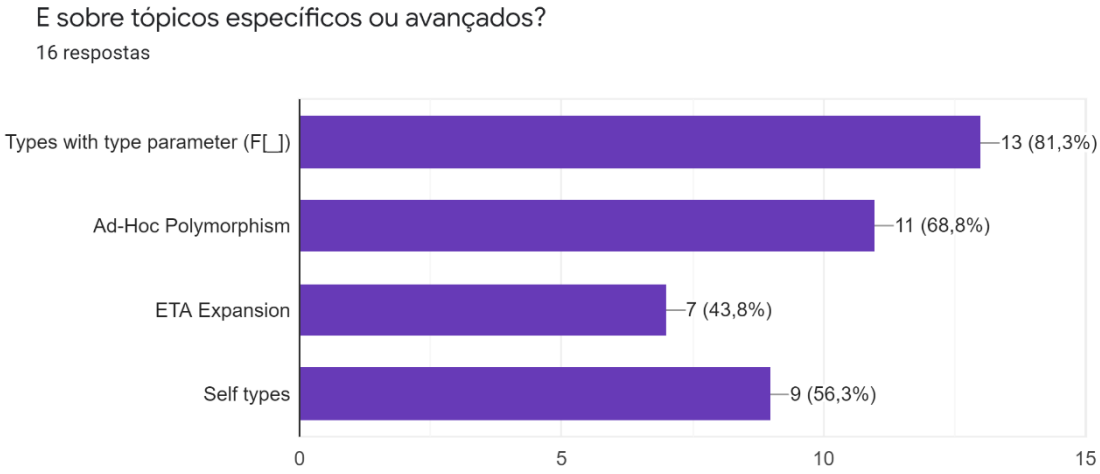
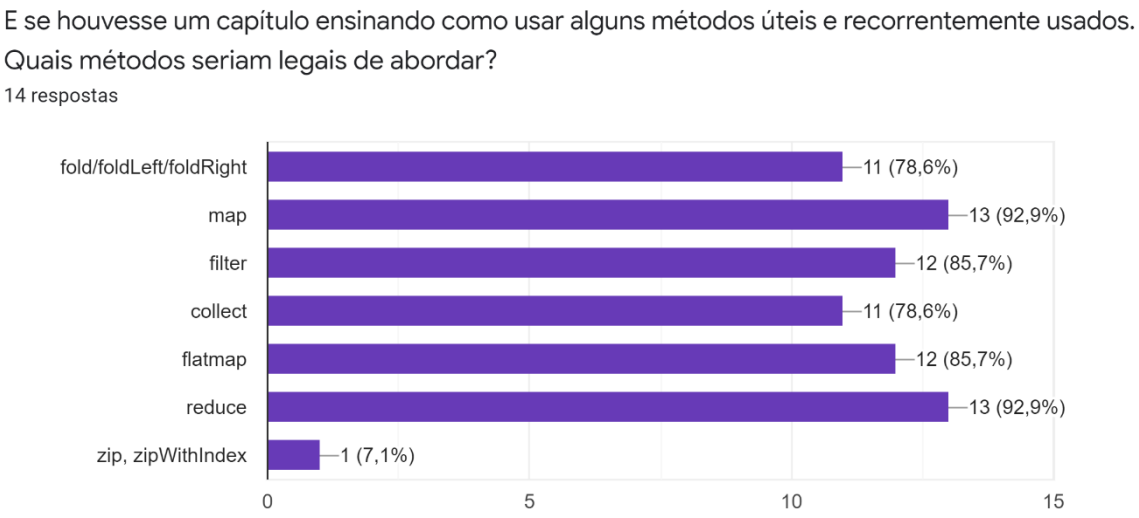


Tabela 5: Métodos úteis a serem abordados



Assim como os dados quantitativos são úteis, as sugestões deixadas na pesquisa também apontam bons assuntos que podem ajudar novos desenvolvedores a começar sua jornada desenvolvendo utilizando Scala. As sugestões incluem:

- Exemplos dos diferentes usos do underscore
- Diferenças entre programação funcional e imperativa
- Maneiras de reescrever códigos imperativos para o paradigma funcional
- Utilização de for-comprehension

A partir desses dados, pôde-se concluir que, entre os desenvolvedores entrevistados, os tópicos que incluem programação funcional e especificidades da linguagem são vistos como mais importantes a serem abordados durante o desenvolvimento do e-book.

Após analisar os resultados e entender melhor quais pontos outros desenvolvedores enxergam como importantes para que um novo desenvolvedor possa começar a utilizar Scala, o sumário do e-book pôde ser criado visando servir como um guia prático e teórico acerca dos assuntos mais votados e sugeridos da pesquisa.

2.6. DESENVOLVIMENTO DO E-BOOK

2.6.1. SUMÁRIO DO E-BOOK

Após a análise do resultado da pesquisa, os tópicos a serem abordados no e-book foram decididos na mesma separação de grupos que a pesquisa e no seguinte esquema.

1. Sintaxe da linguagem
2. Programação Orientada a Objetos
    2.1. Traits
    2.2. Classes
    2.3. Objects
    2.4. Case Classes
3. Programação Funcional
    3.1. Constantes VS Variáveis
    3.2. Funções como argumentos para outras funções
    3.3. Pattern Matching
    3.4. Recursão VS Loops
    3.5. Funções parciais e parcialmente aplicadas
    3.6. Currying
4. Tópicos avançados
    4.1. Types with type parameter
    4.2. Syntax Sugar
    4.3. Implicits
5. Funções populares
    5.1. Map e Flatmap
    5.2. Filter e Collect
    5.3. Reduce e Fold
6. Recomendações de leitura

Este sumário sofreu modificações ao longo do desenvolvimento do e-book.

O subtópico “Types with type parameter” foi removido da versão final por tratar de um assunto complexo demais para o escopo do e-book, a explicação e a aplicação do conceito exigem mais tempo de desenvolvimento com a linguagem,

além de mais contato com o uso de estruturas do paradigma funcional e ambos os requisitos não foram previamente solicitados do leitor no início do material.

Para facilitar o entendimento de certas estruturas “Syntax Sugar” (açúcar sintático) da linguagem, o tópico de Funções Populares foi adiantado, trocando de lugar com “Tópicos Avançados”. Essa alteração foi necessária para que funções abordadas no tópico de Funções Populares pudessem ser utilizadas ao explicar alguns Syntax Sugar, facilitando o entendimento já que o leitor pôde conhecer a aplicação das funções previamente e então entender como elas são reescritas utilizando Syntax Sugar.

O novo sumário do e-book ficou com o seguinte formato:

1. Introdução
2. Sintaxe da Linguagem
    2.1. Variáveis e Constantes
    2.2. Funções
    2.3. String Interpolation
3. Programação Orientada a Objetos
    3.1. Traits
    3.2. Classes
    3.3. Object
    3.4. Generics
4. Programação Funcional
    4.1. Case Classes
    4.2. Constantes VS Variáveis
    4.3. Funções como valores (e como parâmetro para outras funções)
    4.4. Pattern Matching
    4.5. Recursão VS Loops
    4.6. Funções parciais e Funções Parcialmente Aplicadas
    4.7. Currying
5. Funções Populares
    5.1. Map e FlatMap
    5.2. Filter e Collect
    5.3. Reduce e Fold
6. Tópicos Avançados
    6.1. Syntax Sugar

O tópico de Leituras foi escrito visando mostrar novas fontes de conhecimento acerca do tema para o leitor. Todas as recomendações foram feitas a partir de experiência própria e de outros desenvolvedores com as diversas fontes de estudo que a linguagem possui online.

2.6.2. MOTIVO DO FORMATO

O formato escolhido, o e-book, conta com a facilidade de organização do tema em capítulos e uma navegação mais objetiva já que o leitor pode, através do índice, ser redirecionado especificamente ao tópico que lhe interessa, de forma que dúvidas podem ser esclarecidas mais rapidamente. Outro ponto a ser destacado é a facilidade em consumir pequenas etapas do material escrito, como em um livro, o e-book pode ser facilmente lido aos poucos, permitindo que o leitor possa tomar intervalos entre os tópicos para retomar a leitura após realizar alguns testes com os conceitos apresentados, por exemplo.

Outro fator importante é a possibilidade de colaboração no projeto desenvolvido. O e-book pode ser atualizado, se disponibilizado em uma plataforma colaborativa, e outros desenvolvedores podem ajudar a sustentar o material conforme novas atualizações da linguagem, ou mesmo com a adição de novos assuntos úteis para programadores que inicializarão seu aprendizado em Scala ou no paradigma funcional.

2.6.3. FERRAMENTAS PARA DESENVOLVER O E-BOOK

O passo inicial da produção do texto é a coleta de informações que outros desenvolvedores julgam relevante abordar no e-book, para realizar a pesquisa e organizar os dados obtidos foi utilizado o Google Forms. A pesquisa foi realizada no ambiente corporativo e contou com desenvolvedores que trabalham ou já trabalharam com Scala.

O desenvolvimento do corpo do texto foi feito pelo Microsoft Word, utilizando como guias os livros Scala Cookbook e Functional Programming Simplified, ambos escritos por Alvin Alexander. Além dos cursos “Scala & Functional Programming Essentials” e “Advanced Scala and Functional Programming” disponíveis na Udemy e ministrados por Daniel Cîorcîlan.

Para a produção dos trechos de códigos utilizados como exemplos e exercício, a ferramenta escolhida foi o IntelliJ da JetBrains.

2.6.4. BIBLIOGRAFIA UTILIZADA

O e-book foi desenvolvido com base nos conhecimentos adquiridos em materiais online, como e-books, artigos e cursos, além da própria documentação oficial.

A documentação[11] oficial da linguagem foi utilizada para explicar as estruturas da linguagem, todo o corpo do texto foi escrito com base nessa documentação e cada tópico e subtópico pode ser encontrado individualmente, com exceção do tópico “Funções Populares” e dos subtópicos “Constantes VS Variáveis”, “Funções como valores (e como parâmetro para outras funções)” e “Syntax Sugar”.

O tópico “Funções Populares” foi escrito utilizando o e-book[12] de Alvin Alexander, “Functional Programming, Simplified” e o curso[13] Scala & Functional Programming Essentials do instrutor Daniel Ciocîrlan. Ambos os materiais abordam a utilização das funções que foram explicadas no tópico e os exemplos utilizados

---

[11] Documentação oficial. Disponível em <https://docs.scala-lang.org/tour/tour-of-scala.html>. Acessado em 10 out. 2021.

[12] Blog pessoal de Alvin Alexander. Disponível em <https://alvinalexander.com/scala/functional-programming-simplified-book/>. Acessado em 10 out. 2021

[13] Curso ministrado por “Rock The JVM”, pode ser encontrado na Udemy em: https://www.udemy.com/course/rock-the-jvm-scala-for-beginners/

inspiraram os exemplos escritos no material desenvolvido neste trabalho. Esses materiais também foram a fonte norteadora para os três subtópicos que não utilizaram a documentação da linguagem como fonte.

Os primeiros dois subtópicos possuem afirmações sobre o que é considerado como mais correto e o que é essencial no paradigma funcional, como essa é uma discussão teórica, a documentação oficial da linguagem não foi utilizada e os materiais citados anteriormente (livro e curso) que fundamentaram as afirmações feitas.

O subtópico “Syntax Sugar” foi fundamentado no curso[14] “Advanced Scala and Functional Programming” também do instrutor Daniel Ciocîrlan. O curso possui um tópico específico para falar de “Syntax Sugar” no Scala e o material gerado pela aula foi base para a explicação utilizada no e-book sobre como essas estruturas são utilizadas.

Artigos de blogs foram utilizados como material suplementar para a criação de exemplos que ilustrem o uso de certas estruturas, como no caso dos métodos Apply e Unapply que o artigo[15] “Scala pattern matching: apply the unapply” escrito para o site Medium pelo autor Linas Medžiūnas foi utilizado para fundamentar as afirmações e os exemplos utilizados. Para os tópicos sobre Implicits[16] e Pattern Matching[17], foram utilizadas contribuições da comunidade de desenvolvedores, além dos conteúdos em vídeo ministrados por Daniel Ciocîrlan citados anteriormente, para nortear as afirmações feitas no e-book.

---

[14] Curso ministrado por “Rock The JVM”, pode ser encontrado na Udemy em: https://www.udemy.com/course/advanced-scala/

[15] Artigo disponível em <https://medium.com/wix-engineering/scala-pattern-matching-apply-the-unapply-7237f8c30b41>. Acessado em 12 out. 2021.

[16] Discussão sobre Implicits. Disponível em < https://stackoverflow.com/questions/5598085/where-does-scala-look-for-implicits#:~:text=Numeric%20%2C%20not%20Integral%20.-,Implicit%20Scope%20of%20an%20Argument's%20Type,as%20per%20the%20rule%20above>. Acessado em 13 out. 2021.

[17] Discussão sobre Pattern Matching. Disponível em < https://stackoverflow.com/questions/2502354/what-is-pattern-matching-in-functional-languages>. Acessado em 20 out. 2021.

3. MATERIAIS E MÉTODOS

Para o desenvolvimento do e-book foram utilizados o Microsoft Word para edição, os trechos de código foram escritos utilizando a IDE da JetBrains, Intellij.

3.1. CONTATO COM E-BOOKS PARECIDOS

Os e-books de Alvin Alexander citados no desenvolvimento do material foram utilizados para entender a estrutura narrativa utilizada, como o conceito é apresentado e depois enriquecido com exemplos em código. Além disso, foi útil para entender como a informação ficaria disponível em seu formato final em pdf.

4. RESULTADOS E DISCUSSÃO

Embora houve um planejamento inicial de como seria a estrutura do material, houve uma revisão na ordem dos capítulos e algumas modificações nos tópicos abordados que só pôde acontecer conforme a escrita foi se desenvolvendo. Esse replanejamento precisou acontecer já que os assuntos se relacionavam de forma que era necessário criar uma base em um tópico específico, para que futuramente o exemplo dado servisse também para ilustrar outro conceito. Por exemplo, ao abordar o uso da função collect, foi necessário introduzir inicialmente o que era uma função parcial, para que esse conceito pudesse ser resgatado ao explicar como utilizar a função collect.

O resultado, a produção final, foi focada em trazer tópicos importantes para o dia a dia no desenvolvimento com a linguagem, buscando equilibrar o que era importante mostrar de estrutura da linguagem e técnicas frequentemente utilizadas.

Existem discussões[18] na comunidade de programação sobre a dificuldade em aprender o paradigma funcional, assim como relatos de experiências frustrantes[19] com o caminho de aprendizagem. Portanto o desafio foi abordar ambos os temas sem que o e-book se tornasse específico demais em relação às funcionalidades do Scala e evitar também o caráter de tutorial que o texto poderia tomar ao abordar o uso de funções e técnicas de programação funcional. Dessa forma, o e-book poderia ser consumido por desenvolvedores iniciantes na linguagem sem que se sentissem intimidados por conceitos muito complexos.

Embora o conteúdo do texto tenha sido bem fundamento e norteado, a forma com que os tópicos foram abordados, a linguagem utilizada e a estrutura dos exemplos, foram todos feitos sem uma base teórica definida, seguindo apenas o padrão observado em outros materiais semelhantes e utilizando uma linguagem informal com o intuito de manter a leitura mais leve.

---

[18] Discussão acerca do texto de Charles Scalfani em seu blog pessoal. Disponível em < https://www.reddit.com/r/functionalprogramming/comments/e3y4pf/why_is_learning_functional_programming_so_damned/>. Acessado em 05 nov. 2021.

[19] Texto disponível no blog pessoal de Charles Scalfani. Disponível em <https://cscalfani.medium.com/why-is-learning-functional-programming-so-damned-hard-bfd00202a7d1>. Acessado em 05 nov. 2021.

5. CONSIDERAÇÕES FINAIS

A escrita do e-book seguiu o planejamento proposto, o material foi embasado em teoria sobre como desenvolver um material instrucional e contou com uma pesquisa entre desenvolvedores para que atingisse um nível de maturidade coerente com o que o mercado espera, sendo compatível com o que um desenvolvedor busca em um material para guiá-lo nos primeiros passos com a linguagem Scala. A pesquisa trouxe dados de desenvolvedores que trabalham com Scala para contribuir com a elaboração do sumário do e-book.

Embora a produção tenha seguido todo o planejamento, seria ideal a inclusão de uma sessão no cronograma apenas para a coleta de opinião de desenvolvedores após a leitura do material finalizado, para que uma nova pesquisa fosse realizada com o intuito de identificar se realmente os envolvidos sentiram que o e-book cumpriu com sua proposta.

6. REFERÊNCIAS BIBLIOGRÁFICAS

ALEXANDER, Alvin. **Functional Programming Simplified**. 2017.

CONNOR, Chloe. **Stop using Pandas and start using Spark with Scala**. Disponível em <https://towardsdatascience.com/stop-using-pandas-and-start-using-spark-with-scala-f7364077c2e0>. Acessado em 02 jun. 2021.

CARGNELUTTI, Camila et al. **E-book na educação a distância**: A construção de livros didáticos digitais em uma equipe multidisciplinar. Disponível em <https://esud2018.ufrn.br/wp-content/uploads/187247_1_ok.pdf>. Acessado em 02 jun. 2021.

HEUNEN, C; JACOBS, B. **Arrows, like Monads, are Monoids**. Disponível em <https://www.sciencedirect.com/science/article/pii/S1571066106001666>. Acessado em 01 jun. 2021.

Instituto de Pesquisa Data Popular. **Demandas de Aprendizagem de Inglês no Brasil**. 2014. Disponível em <https://www.britishcouncil.org.br/sites/default/files/demandas_de_aprendizagempesquisacompleta.pdf>. Acessado em 20 mai. 2021.

JUNGTHON, G; GOULART, C. **Paradigmas de Programação**. Faculdade de Informática de Taquara (FIT), *[S. l.]*, p. 8, 2010. Disponível em: <https://fit.faccat.br/~guto/artigos/Artigo_Paradigmas_de_Programacao.pdf>. Acessado em 20 mai. 2021.

MARQUESONE, Rosangela. **Big Data:** Técnicas e tecnologias para extração de valor dos dados. Dezembro de 2016. Disponível em < https://play.google.com/store/books/details?id=cbWlDQAAQBAJ&rdid=book-cbWlDQAAQBAJ&rdot=1&source=gbs_vpt_read&pcampaignid=books_booksearch_viewport>. Acessado em 27 mai. 2021.

ODERSKY, Martin et al. **An Overview of the Scala Programming Language**. Disponível em <https://www.researchgate.net/profile/Gilles-Dubochet/publication/37434683_An_Overview_of_the_Scala_Programming_Language_2_Edition/links/53e9052f0cf2dc24b3c7e91b/An-Overview-of-the-Scala-Programming-Language-2-Edition.pdf>. Acessado em 01 jun. 2021.

StackOverflow. **Most popular Technologies 2020**. Disponível em <https://insights.stackoverflow.com/survey/2020>. Acessado em 28 mai. 2021.

Wikipedia. **Açúcar Sintático**. Disponível em <https://pt.wikipedia.org/wiki/A%C3%A7%C3%BAcar_sint%C3%A1tico>. Acessado em 31 mai. 2021.

7. APÊNDICE

CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA

FACULDADE DE TECNOLOGIA DE CAMPINAS

CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GIACOMO VALENTIM SILVA MAGRI

GUIA TEÓRICO E PRÁTICO PARA INICIANTES EM PROGRAMAÇÃO FUNCIONAL COM SCALA

CAMPINAS/SP

2022

LISTA DE TABELAS E QUADROS

Nenhuma entrada de índice de ilustrações foi encontrada.

1. Sumário

2. Introdução

Scala é uma linguagem um tanto intimidadora para novos desenvolvedores por ser uma linguagem com foco em programação funcional.

Esse paradigma muitas vezes pode parecer difícil de entender durante a faculdade e para desenvolvedores que já estão acostumados com o paradigma orientado a objetos ou com o imperativo, a dificuldade principal é em entender como “pensar diferente” e enxergar as soluções com caráter mais orientado aos princípios do paradigma funcional.

Gostaria de enfatizar que esse material é escrito visando tornar mais fácil a entrada de novos desenvolvedores no “mundo do Scala”, ou seja, é um guia para iniciantes e não um guia para desenvolvedores que já estão acostumados com o paradigma funcional ou que esperam um aprofundamento muito grande em como desenvolver totalmente orientado ao paradigma. Experiência com Java ou Kotlin contribui para facilitar a absorção do conteúdo já que são linguagens da JVM e compartilham algumas estruturas, no entanto, não é obrigatório para o entendimento deste material. A ideia é apresentar a linguagem e seus aspectos do paradigma orientado a objetos e do funcional, além de falar sobre funções populares que vêm do paradigma funcional.

Não abordarei detalhes sobre IDE nem apresentarei passo a passo de como instalar ferramentas, este material serve para esclarecer sobre a linguagem.

Sinta-se livre para utilizar ferramentas online para compilar seu código, como o repl.it ou IDEs robustas como o Intellij.

Você notará que ao apresentar algum termo, sua primeira aparição será em *itálico* e depois o termo será usado normalmente. Além disso, para não manter uma formalidade tão rígida, chamarei algumas estruturas por nomes equivalentes em português, por exemplo “classe” ao invés de “class”.

Veremos em dado momento que existe uma diferença entre método e função no Scala. No entanto, existem tantas semelhanças entre ambas as estruturas que por vezes chamarei um método de função pela conveniência do termo.

3. Sintaxe da Linguagem

Variáveis e Constantes

```
object Main extends App {

  println("Hello World!")

}
```

Este é o “Hello World” em Scala. Lindo, né?

Repare que não precisamos de ponto e vírgula no fim da linha!

```
val umaString = "Uma string!"
```

Essa outra linha nos mostra como definir uma constante através da palavra reservada *val*. Por ser uma constante, não podemos reatribuir o valor de “umaString”, seu valor será sempre “Uma String!”. Para definirmos valores mutáveis nós utilizamos a palavra reservada *var*.

```
var nome = "Bob Esponja"
```

Como estamos utilizando uma variável, “nome” pode receber outro valor, mas lembre-se, Scala é uma linguagem “fortemente tipada”, portanto podemos apenas atribuir outra String para a variável nome.

A tipagem dos valores demonstrados acima não está sendo feito explicitamente por nós, mas o compilador é esperto o bastante para saber que definimos nossa constate e nossa variável como strings, portanto não podemos reatribuir um valor de tipo diferente!

Para tiparmos explicitamente nossa variável basta seguirmos o exemplo abaixo.

```
var nome: String = "Bob Esponja"
```

Aqui temos alguns exemplos dos tipos mais comuns que utilizamos

```
val string: String = "String"
val int: Int = 10
val double: Double = 10.00
val boolean: Boolean = true
```

Funções

Para definirmos métodos ou funções em Scala, podemos fazer de duas formas.

```
def function(palavra: String): String = {
  palavra
}

val anotherFunction = (palavra: String) => {
  palavra
}
```

A primeira forma nós chamamos de “método” e definimos através da palavra reservada *def*, logo em seguida colocamos seu nome, a lista de parâmetros entre parênteses e então tipamos seu retorno (assim como não fomos obrigados a tipar

nossa variável “nome”, também não precisamos tipar o método, o compilador fará isso por nós!) e por fim colocamos o corpo do método entre chaves.

A segunda forma é uma função e definimos ela de uma forma um pouco diferente, definimos o nome que a função terá e então colocamos entre parênteses os parâmetros que ela recebe. Logo em seguida fazemos “uma setinha” utilizando o igual (=) e o símbolo de maior (>) para então definir o corpo de nossa função entre chaves assim como fizemos no método.

Você deve ter notado que essa função tem a palavrinha reservada que usamos para definir uma constante anteriormente, o que estamos fazendo aqui é atribuir uma função à uma constante e isso é um dos princípios da programação funcional “funções são tratadas como valores”.

A forma que definimos a função é chamada por vários nomes, Lambda, Arrow Function, Função Anônima. Pode parecer estranho à primeira vista, mas é bastante comum e com o tempo essa notação fica mais natural.

Note também que na função e no método, nós não usamos a palavra reservada *return*. Em Scala, a última linha de um bloco ou corpo de função que define o valor retornado por ele.

As estruturas que definimos anteriormente recebem um parâmetro do tipo String e retornam esse parâmetro, portanto, ambas as estruturas recebem e retornam o mesmo tipo. Além disso, o corpo que definimos para cada estrutura é o mesmo (simplesmente retorna o parâmetro recebido), portanto podemos considerar que as estruturas definidas são semelhantes, seja utilizando a palavra def ou criando uma função e atribuindo a uma val, os parâmetros, tipo de retorno e corpo é o mesmo para ambos os casos!

Caso você tenha se perguntado se o compilador foi esperto o suficiente para dizer o tipo da nossa “anotherFunction”, a resposta é: Sim!

Aqui está a mesma estrutura, mas deixando explícito o tipo.

```
val anotherFunction: String => String = (palavra: String) => {
  palavra
}
```

Como podemos observar, da mesma forma que a setinha separa os parâmetros do corpo da função, ela também separa o tipo dos parâmetros (lado esquerdo da seta) do tipo do retorno da função (lado direito da seta). Se refletirmos sobre essa notação, ela é confusa em primeiro momento, mas nos dá uma noção bastante rápida do que temos definido, uma função que recebe uma String e retorna também uma String.

Já que agora estamos deixando explícito o tipo do parâmetro recebido pela nossa função, podemos omitir a tipagem que fizemos no trecho:

```
(palavra: String) => {
  palavra
}
```

Removendo o tipo String do nosso parâmetro “palavra”, podemos reescrever a mesma função dessa forma:

```
val anotherFunction: String => String = palavra => {
  palavra
}
```

Aqui está um exemplo de como ficaria uma função e um método que recebem mais de um parâmetro

```
def function(nome: String, idade: Int): String = {
  nome + idade.toString
}

val anotherFunction: (String, Int) => String = (nome, idade) => {
  nome + idade.toString
}
```

O corpo de ambas as funções simplesmente concatena o valor de “nome” com String que representa a idade (fazemos isso ao utilizar o método *.toString*)

E se a nossa função não retornasse nada? Digamos que simplesmente vamos receber os parâmetros e printar eles para o usuário, qual seria o tipo da nossa função/método?

```
def function(nome: String, idade: Int): Unit = {
  println(nome + " tem " + idade + " anos")
}

val anotherFunction: (String, Int) => Unit = (nome, idade) => {
  println(nome + " tem " + idade + " anos")
}
```

Unit. Esse tipo é usado quando precisamos dizer que o tipo retornado ali é “nada”, o equivalente ao *void* do Java.

Podemos definir uma val/var como unit?

```
val nada: Unit = println("Nada aconteceu")
```

Podemos!

String Interpolation

Podemos lidar com concatenação de strings em Scala utilizando o operador de soma entre strings:

```
def falarNome(nome: String): String = {
  "meu nome é: " + nome
}
```

Porém existe outra estrutura mais charmosa de utilizar, *String Interpolation* ou interpolação de String, através do método *s* (sim, é um método):

```
def falarNome(nome: String): String = {
  s"meu nome é: $nome"
}
```

basta acrescentarmos um cifrão ($) na frente de nossa variável e pronto!

Caso fôssemos utilizar um método ou atributo durante a interpolação, basta utilizarmos chaves em volta da chamada:

```
def falarIdade(idade: Int): String = {
  s"minha idade é ${idade.toString}"
}
```

4. Programação Orientada a Objetos

Traits

Em orientação a objetos existe o conceito de Interface, um lugar que definimos o contrato que será implementado por uma classe ou que servirá de base para a definição de outra interface, em Scala, essa funcionalidade é chamada de *Trait*.

Uma trait é definida da seguinte forma

```
trait simpleTrait
```

pode ser utilizada na definição de uma classe com a palavra reservada *extends*

```
class simpleClass extends simpleTrait
```

pode ser utilizada na definição de outra trait

```
trait anotherTrait extends simpleTrait
```

e quando implementamos várias traits em uma mesma classe ou trait, precisamos utilizar a palavra reservada *with* para as traits posteriores à primeira:

```
trait simpleTrait
trait aTrait
trait bigTrait extends aTrait with simpleTrait
```

Além disso, para melhorar o encapsulamento e evitar que outras partes do código implementem uma determinada trait, podemos utilizar a palavra reservada *sealed* antes da definição da trait

```
sealed trait simpleTrait
```

Uma trait definida dessa forma só pode ser estendida por classes ou traits dentro do escopo em que ela foi definida (dentro do mesmo arquivo ou do mesmo objeto, por exemplo).

Classes

Vimos no tópico sobre traits como definir uma classe em Scala, no entanto não vimos como definir classes com atributos em seu construtor e nem como instanciar nossas classes.

```
class Pessoa(nome: String, idade: Int)
val aluno: Pessoa = new Pessoa("Carlito", 65)
```

Os membros definidos dentro dos parênteses serão atributos privados e constantes (val) dentro da classe Pessoa. Para definirmos o membro do construtor como público, basta adicionar “val” na frente do nome do atributo:

```
class Pessoa(val nome: String, medoSecreto: String)
```

No exemplo acima, o atributo “nome” é público, mas “medoSecreto” é um atributo privado da classe.

Vamos passar rapidamente pelos modificadores de acesso disponíveis na linguagem:

“public” define um membro que pode ser acessado de fora da classe

“private” define um membro que não pode ser acessado de fora da classe

“protected” define um membro que não pode ser acessado de fora da classe, mas pode ser acessado por herdeiros da classe, por exemplo:

```
class Componente {
  protected val detalhes: String = "detalhes complexos"
}

class Motor extends Componente {
  def trabalhar: String = s"trabalha usando '$detalhes'"
}
```

caso “detalhes” fosse um membro privado da classe Componente, a classe Motor, mesmo estendendo as funcionalidades de Componente, não poderia acessá-lo. Como definimos esse membro como “protected”, Motor pode inclui-la em seus atributos, como acontece no método trabalhar (sim, podemos declarar métodos de uma linha sem o uso de chaves).

Além disso, podemos utilizar a palavra “final” para definir uma trait ou classe que não pode ser estendida por outras.

Object

Scala possui uma estrutura chamada de *Object*, um *Singleton* que contém membros internos (val, def, outras classes ou traits) com seus próprios modificadores de acesso. Um object pode definir diversas funções em seu corpo que poderão ser utilizadas em outro escopo através da palavra *import* para importar os detalhes do object para o escopo, ou acessando-os diretamente pelo nome do object definido:

```
object Main extends App {

  object Utils {
    def intToString(int: Int): String = int.toString
  }

  Utils.intToString(10)
}
```

Dessa forma, definimos o método intToString dentro do object Utils e então o invocamos de fora do object ao escrever a última linha:

```
Utils.intToString(10)
```

Podemos optar por importar o conteúdo de Utils para dentro da Main, por exemplo:

```
object Main extends App {

  object Utils {
    def intToString(int: Int): String = int.toString
  }

  import Utils._
  intToString(10)
}
```

Um object possui dois métodos muito úteis: *apply* e *unapply*. Vamos começar falando do primeiro, o método apply.

Apply é um método definido dentro de um object, pode receber parâmetros n parâmetros e devolver qualquer tipo.

```
object Pessoa {
  def apply(): String = "Eu sou uma pessoa agora!"
}
```

No exemplo acima nosso método apply gera uma nova String contendo o valor “Eu sou uma pessoa agora!”. A diferença do apply para um método comum é que ele pode ser invocado da seguinte forma:

```
Pessoa()
```

Basta adicionarmos parêntesis após o nome do object, como se estivéssemos invocando um método, e pronto!

Esse comportamento sozinho não parece ser grande coisa, já que qualquer outro método faria exatamente a mesma coisa, no entanto, o Object pode agir “em cooperação” com outra Class que possui o mesmo nome.

Por exemplo, vamos repensar a Class Pessoa:

```
class Pessoa(val nome: String, val idade: Int)
```

e vamos definir um Object com o mesmo nome e possuindo um método apply implementado

```
class Pessoa(val nome: String, val idade: Int)
object Pessoa {
  def apply(nome: String): Pessoa = new Pessoa(nome, 18)
}
```

no exemplo, ao invocarmos o método apply do Object Pessoa, criaremos uma instância da Class Pessoa que recebe apenas o nome como parâmetro e utiliza idade como 18 por padrão!

```
val aluno: Pessoa = Pessoa("Carlito")
```

Observe como criamos uma val “aluno” do tipo Pessoa invocando o método apply do Object Pessoa. Dessa forma, além de criar métodos construtores para nossa Class Pessoa, ainda conseguimos criar instâncias no nosso código sem precisarmos usar a palavra *new*!

Esse comportamento é um Syntax Sugar do Scala, mas abordaremos esse tema em um tópico específico mais para frente.

Quando criamos essa estrutura de Object e Class com o mesmo nome, dizemos que temos um *Companion Object* para uma Class. O Companion Object de uma classe pode compartilhar seus atributos, mesmo privados:

```
class Pessoa(val nome: String, val idade: Int) {
  def sentidoDaVida: Int = Pessoa.numeroMagico
}
object Pessoa {
  private val numeroMagico = 42
```

```
  def apply(nome: String): Pessoa = new Pessoa(nome, 18)
}
```

Mesmo definindo “numeroMagico” como um atributo privado, podemos acessá-lo de dentro da nossa classe Pessoa como fizemos na implementação do método “sentidoDaVida”.

O método unapply faz o caminho inverso do método apply, aqui a ideia é definir um “comportamento destrutivo”, nós vamos receber uma instância e “destruí-la” em peças menores, como seus atributos. Por exemplo, nossa Class Pessoa tem nome e idade definidos no construtor padrão, e um novo método construtor definido no Companion Object através do método apply. O método unapply se encaixará aqui para fazer a decomposição de Pessoa em um outro tipo de dados, uma tupla de String e Int que contêm os valores de nome e idade de seu construtor.

```
class Pessoa(val nome: String, val idade: Int) {
  def sentidoDaVida: Int = Pessoa.numeroMagico
}
object Pessoa {
  private val numeroMagico = 42
  def apply(nome: String): Pessoa = new Pessoa(nome, 18)
  def unapply(pessoa: Pessoa): (String, Int) = (pessoa.nome, pessoa.idade)
}
```

E nosso código ficará assim. O método unapply recebe uma instância de Pessoa e retorna uma tupla contendo os valores que definimos. Em primeiro momento parece bem inútil, mas ao utilizarmos o conceito de *Pattern Matching*, nosso código terá muito mais utilidade!

Vamos dar uma olhadinha em como podemos deixar esse código mais útil, mesmo que as coisas não façam muito sentido agora. Você pode ignorar esse trecho caso queira ou pode ver um pouquinho de mágica agora.

Primeiro vamos reimplementar o código do nosso método unapply para que ele retorne uma Option. Uma Option é um tipo de dado que usamos para representar a possibilidade de um valor existir ou não.

A estrutura de um Option é a seguinte: caso haja um valor interno, esse valor será Some(valor), caso não haja nada, o Option será um None.

```
def unapply(pessoa: Pessoa): Option[(String, Int)] = {
  if (pessoa != null) Some(pessoa.nome, pessoa.idade)
  else None
}
```

O que fazemos aqui é checar se a instância que recebemos no parâmetro é diferente de null, caso seja, retornamos a tupla de valores encapsuladas numa instância de Some. Caso nosso parâmetro seja null, retornaremos None para representar o valor vazio.

Ok, mas o que tem demais?

Bom, agora podemos utilizar Pattern Matching em uma instância de Person para pegarmos o nome e a idade dela:

```
val aluno: Pessoa = Pessoa("Carlito")

aluno match {
```

```
  case Pessoa(nome, idade) => println(s"Meu nome é $nome e eu tenho $idade
anos!")
  case _ => println("algo deu errado...")
}
```

Nessa expressão checamos se podemos reescrever “aluno” como Pessoa(nome, idade), mas para isso precisamos implementar um método unapply que retorne um tipo que implementa o método .get() (Option possui esse método). Os valores “nome” e “idade” são String e Int pois definimos esse comportamento no método unapply, poderíamos definir outro comportamento para retornar apenas a idade, por exemplo, então teríamos um retorno do tipo Option[Int].

Pattern Matching é um conceito que será abordado futuramente na sessão sobre programação funcional, lá teremos exemplos e entenderemos para que essa expressão serve, além de entendermos seu **real poder.**

Generics

Uma feature comum em linguagens é a capacidade de definir uma classe ou método que, ao invés de ser de um tipo específico, possui um tipo genérico em seus membros ou no parâmetro do método.

Isso é diferente de dizer que estamos recebendo um parâmetro do tipo Any, já que o tipo genérico pode ter restrições. Podemos dizer que um tipo genérico só pode ser um subtipo de outro específico ou supertipo de outro. Esse conceito é chamado de Variance, mas não abordaremos isso aqui por ser meio complexo para o escopo desse material.

Falando de tipos genéricos, vamos ver um exemplo de como declarar uma classe que possui um tipo genérico como parâmetro. Vamos definir uma classe Caixa que pode guardar qualquer tipo dentro dela:

```
class Caixa[T](val elemento: T)
```

Repare que definimos após o nome da classe, entre colchetes, o tipo “T”. Se fôssemos usar mais de um tipo genérico dentro de nossa classe, bastaria lista-lo por vírgula dentro dos colchetes. Nossa classe tem em seu construtor o parâmetro “elemento” que é do tipo que definimos, T.

Para instanciarmos a classe, basta fazer:

```
new Caixa[Int](10)
```

O tipo pode ser omitido e o compilador (na maioria das vezes) conseguirá inferi-lo:

```
new Caixa(10)
```

O funcionamento é o mesmo.

Podemos utilizar tipos genéricos em métodos também:

```
def printaQualquerCoisa[S](parametro: S): Unit = println(parametro)
```

Da mesma forma que fizemos na classe, aqui definimos um tipo S e o parâmetro do método será desse tipo. Para invocarmos o método, basta chamá-lo, omitindo ou deixando o tipo explícito na chamada:

```
printaQualquerCoisa(10)
printaQualquerCoisa[Int](11)
```

E se quiséssemos printar nossa classe Caixa?

Também podemos!

```
printaQualquerCoisa(new Caixa(10))
printaQualquerCoisa[Caixa[Int]](new Caixa(11))
```

5. Programação Funcional

Nessa sessão abordaremos tópicos básicos de programação funcional e sua implementação em Scala. Você entenderá o motivo de usarmos constantes ao invés de variáveis, recursividade ao invés de loops, funções como valores e como parâmetro de outras funções. Também veremos por aqui o que é Pattern Matching e um pouco de coisas estranhas como Funções Parcialmente Aplicadas, Funções Parciais e Currying.

Case Classes

Essa estrutura é bastante simples e muito usada em Scala, sua estrutura lembra muito o de uma classe comum, com a diferença que não precisamos utilizar a palavra new ao criar instâncias de Case Classes

```
case class Carro(numeroRodas: Int)

val carroEstranho = Carro(3)
```

Aqui definimos uma Case Class Carro que logo abaixo é instanciada sem o uso de new, utilizando uma notação idêntica ao dos nossos queridos métodos apply. Se você está se perguntando como isso acontece, é pela natureza da Case Class. Sempre que declaramos uma, o compilador faz mágica para nós e cria também um Object com os métodos apply/unapply e vários outros métodos úteis sem precisarmos escrever uma linha de código a mais. *Mágico*.

Constantes VS Variáveis

Programação funcional é um paradigma que trata o código, e todo o programa, como uma composição de funções e valores constantes.

Como na matemática, se declaramos que um valor x é igual a 1, esse valor não será reavaliado e sempre será 1. Se quisermos definir um outro valor, devemos utilizar outro identificador, por exemplo, y = x + 1. Assim como na matemática, a ideia é criarmos um código com valores constantes ao invés de variáveis. Essa abordagem permite criarmos código com menos complexidade para de debug já que um valor definido inicialmente no fluxo se manterá o mesmo até o fim.

Outra vantagem do uso de constantes ao invés de variáveis é quando devemos nos preocupar com execução multi-thread. Utilizando constantes, nunca haverá o risco de uma thread alterar um valor previamente definido e atrapalhar a execução de outra thread que usaria aquele dado.

Funções como valores (e como parâmetro para outras funções)

Outro aspecto de programação funcional que está intimamente ligada à matemática é a forma com que funções são usadas.

Esse aspecto é conhecido como *Higher Order Function* e é amplamente utilizado por outras linguagens.

Em programação funcional vemos funções como valores, ou seja, uma função pode ser atribuída a uma constante ou variável (como vimos anteriormente) e seu uso no código será o mesmo que o de uma constante ou variável. Podemos então dizer que da mesma forma que uma função aceita um int como parâmetro:

```
def somaUm(n: Int): Int = n + 1
```

Também podemos aceitar no parâmetro uma outra função:

```
def executaFuncao(f: Unit => Int): Int = f()
```

E não tem problema recebermos uma função e um "parâmetro comum" juntos:

```
def executa(fun: Int => Int, numero: Int): Int = fun(numero)
```

No método definido acima, passamos uma função e um parâmetro inteiro "numero" e então chamamos a função com "numero" sendo seu parâmetro. Essa capacidade de receber uma função como parâmetro é bastante explorada em programação funcional e diversos métodos utilizam, como *filter, map, fold* que veremos no detalhe mais pra frente.

Pattern Matching

Pattern Matching é uma estrutura condicional com certas vantagens, podemos desconstruir certas estruturas (como vimos anteriormente), fazer asserção de tipo e tratar o valor recebido como o tipo da asserção tudo em uma linha de código.

Estruturas condicionais como switch-case e if-else são populares em várias linguagens, no entanto, não possuem o poder que um Pattern Matching possui.

```
val valor = 4

valor match {
  case 10 => println("Ok!")
  case 11 => println("Estranho...")
  case 1 | 2 | 3 => println("Baixo...")
  case _ => println("algo errado!")
}
```

Aqui estamos utilizando Pattern Matching para verificar o valor de "valor", a estrutura é bastante semelhante ao switch-case. Colocamos a dado que queremos fazer comparações e logo à frente a palavra "match", então, dentro das chaves criamos vários casos diferentes utilizando a palavra case.

A primeira e a segunda linha são asserções simples, apenas checa se o valor é 10 ou 11 e direciona o fluxo para cada caso. A terceira linha faz uma asserção múltipla, checando se o valor é 1 ou 2 ou 3. Já na última linha utilizamos um "_" para definir

um caso default, ou seja, qualquer outro caso que não seja os casos definidos acima. O uso de underscore nessa estrutura é um Syntax Sugar do Scala e falaremos disso no futuro. Podemos substituir o underscore por um nome qualquer se quisermos

```
val valor = 4

valor match {
  case 10 => println("Ok!")
  case 11 => println("Estranho...")
  case 1 | 2 | 3 => println("Baixo...")
  case outroValor => println("algo errado!")
}
```

outroValor é agora o nome que utilizamos para o último caso do nosso Pattern Matching, nomeando ou utilizando o underscore, o funcionamento é o mesmo ainda.

É importante se atentar para qual posição devemos utilizar esse caso default, já que ele literalmente abrange qualquer caso, ou seja, caso escrevêssemos ele na primeira linha, todos os casos abaixo seriam ignorados, já que o caso default não faz checagem nenhuma.

```
val dado: Any = 10

dado match {
  case valor: Int => println("Inteiro")
  case valor: Double => println("Double")
  case valor: String => println("String")
  case valor => println("outro tipo")
}
```

Aqui definimos “dado” como tipo Any, o que significa que o valor que ele guarda pode ser de qualquer tipo. Logo abaixo checamos qual o tipo dessa nossa constante. O que acontece aqui é que o compilador checa se o valor interno de “dado” faz sentido com cada tipo que colocamos para a checagem. Como definimos o valor como 10, a checagem da primeira linha será um sucesso já que 10 é um inteiro.

Quando escrevemos “case valor: String”, estamos dizendo que o compilador tentará checar se o que está contido em “dado” pode ser do tipo String e se sim, criamos uma constante chamada “valor” para guardar essa String. Poderíamos obter o mesmo resultado com a seguinte estrutura:

```
if (dado.isInstanceOf[String]) {
  val valor = dado.asInstanceOf[String]
  println("String")
}
```

Primeiro checamos o tipo e então criamos uma constante para guardar aquele valor no tipo que queremos (String), para aí então executarmos algum código com esse casting que fizemos. Tá vendo como Pattern Matching simplifica muito a vida?

Também podemos utilizar a desconstrução de classes através do método unapply que vimos anteriormente:

```
case class Pessoa(nome: String, idade: Int)

val pessoa = Pessoa("Carlito", 18)

pessoa match {
  case p if p.idade > 20 => println(s"Tenho mais de vinte anos!")
  case Pessoa(nome, idade) => println(s"Meu nome é $nome e tenho $idade
anos!")
}
```

No primeiro case estamos fazendo uma coisa diferente, estamos utilizando um *Guard* para checar se a pessoa que estamos verificando tem idade maior que vinte anos (o poder do Pattern Matching não acaba nunca!). Quando fazemos essa checagem, não precisamos utilizar parêntesis para definir o boolean usado no if.

O segundo case é uma desconstrução da nossa case class em duas variáveis: nome e idade. No print estamos mostrando o nome e a idade da pessoa sem precisar escrever pessoa.nome e pessoa.idade graças à essa desconstrução. Nós extraímos os valores que queríamos e colocamos ambos em constantes com os nomes “nome” e “idade” para utilizar no bloco de código após a seta.

Podemos ter mais de uma linha no bloco a ser executado por um case de um Pattern Matching:

```
pessoa match {
  case p if p.idade > 20 => println(s"Tenho mais de vinte anos!")
  case Pessoa(nome, idade) =>
    println(s"Meu nome é $nome e tenho $idade anos!")
    println("Podemos ter mais de uma linha no bloco!")
    val exemplo = "exemplo"
    println(s"print de $exemplo")
}
```

Aqui temos até uma atribuição de uma nova constante dentro do bloco a ser executado no nosso segundo case, o uso de chaves para definir esse bloco é totalmente opcional, mesmo com múltiplas linhas!

Recursão VS Loops

Falamos de coisas estranhas que o paradigma funcional trás, como o uso restrito de constantes e de funções como valores e como parâmetro em outras funções. O último tópico que abordamos também é um conceito (em primeiro momento bem bizarro) muito poderoso de programação funcional. Agora, discutiremos como esse paradigma nos faz abandonar outra estrutura comumente utilizada, os loops.

Quando estruturamos um loop, uma variável terá seu estado alterado até que uma condição booleana seja atendida. Certo?

Como precisamos de uma variável alterando seu valor para que nosso código se repita, isso já não se encaixa no que vimos sobre programação funcional, já que deveríamos utilizar constantes, e como o próprio nome diz, seu valor não é mutável.

Para lidar com a necessidade de repetição de um determinado bloco de código, podemos utilizar recursão para solucionar nosso problema.

Existem inúmeros exemplos de como evitar laços de repetição utilizando recursividade na internet e algumas leituras sobre isso serão citadas na área final deste material.

Para ilustrar um pouco melhor o uso de recursividade em Scala e adicionar ainda mais um motivo para sempre utilizar Pattern Matching, vamos imaginar a seguinte situação: queremos somar todos os elementos de uma lista de inteiros de forma recursiva. Como fazer isso?

Bom, é importante saber que uma lista pode ser decomposta por Pattern Matching em uma head e uma tail. A head é o primeiro elemento da lista e a tail é toda a lista após o primeiro elemento. Outro ponto importante é que se uma lista estiver vazia, ela pode ser representada pela palavrinha *Nil*.

Observe como podemos estruturar essa função:

```
@tailrec
def soma(list: List[Int], somaAcumulada: Int = 0): Int = list match {
  case Nil => somaAcumulada
  case head :: tail => soma(tail, somaAcumulada + head)
}
```

Nós recebemos a lista e um valor, por padrão 0, para a soma que será acumulada pela recursão. No corpo da função nós utilizamos Pattern Matching para verificar o conteúdo da lista.

O primeiro caso checa se temos uma lista vazia e se for esse o caso, retornaremos o valor acumulado na execução.

No segundo caso nós desconstruímos a lista em head e tail. Essa estranha notação "::" é implementada como uma Case Class no Scala, portanto essa linha pode ser reescrita como ::(head, tail). Essa estrutura é um Syntax Sugar do Scala e será abordada em outro capítulo. Seguindo o fluxo, após nossa desconstrução, chamamos a função novamente utilizando a tail como a próxima lista e somando nossa head ao "somaAcumulada" e passando o resultado como o segundo parâmetro.

Repare que nesse exemplo nós utilizamos uma *annotation* no nosso método. @tailrec sinaliza ao compilador que esse é um método *Tail Recursive* (utiliza Recursão de Cauda) e deve ser otimizado. O que é Recursão de Cauda? Bom, como pode notar no método, nós chamamos a função novamente na última instrução do nosso código. Não existe nada após essa chamada, de forma que, ao executar o programa, essa recursão não precisa "guardar memória da chamada anterior" e isso otimiza a performance da nossa recursão.

Funções parciais e Funções Parcialmente Aplicadas

Embora os nomes sejam parecidos, essas estruturas são diferentes e não devem ser confundidas. Uma função parcial (*Partial Function*) é uma função que tem como

implementação uma lógica que cobre apenas uma parcela de casos para um determinado problema.

Confuso demais, né? Pois é, meio bizarro mesmo.

Vamos imaginar a seguinte implementação de uma função:

- Se recebermos uma idade menor que 16 anos, a pessoa é criança.

- Se a idade for maior de 18, não é criança.

Nesse cenário, caso a idade seja 16, 17 ou 18 anos, nossa função não funciona, ela simplesmente deveria lançar uma exceção ou algo do tipo. Para checarmos se a função funcionaria para a idade 16, teríamos que colocá-la em um try-catch e então fazer a checagem dessa forma. É aí que entram as partial functions.

```
val partialFunction: PartialFunction[Int, String] = {
  case idade if idade < 16 => "Criança"
  case idade if idade > 18 => "Não Criança"
}
```

Aqui definimos uma função tipo PartialFunction e passamos dois tipos para ela. O primeiro tipo, Int, é o tipo que essa função recebe de parâmetro, é a entrada da função. O segundo tipo é o output, a saída da função, no nosso caso são as strings que dizem se é criança ou não.

Se chamarmos essa função, passando 16 no parâmetro, como no exemplo abaixo:

```
println(partialFunction(16))
```

o que conseguiremos é uma bela exceção sendo lançada, um “MatchError” sinalizando que o parâmetro não é suportado pela função.

E qual a graça disso? Como isso é melhor que só definir a função normalmente e lançar uma exceção manualmente?

Nossa função parcial nos permite checar se um determinado parâmetro é suportado ou não pela função, sem um try-catch:

```
println(partialFunction.isDefinedAt(16))
```

Esse método retorna um boolean dizendo se a função retorna um valor válido para o parâmetro passado. Nesse caso, ela retornaria false.

Outra funcionalidade que já vem implementada é a possibilidade de aplicarmos um parâmetro para a função e um outro parâmetro que é uma função para tratar os casos que não foram contemplados na função parcial:

```
val partialFunction: PartialFunction[Int, String] = {
  case idade if idade < 16 => "Criança"
  case idade if idade > 18 => "Não Criança"
}

val trataResto: Int => String = i => "Alguma coisa"

println(partialFunction.applyOrElse(16, trataResto))
```

Aqui definimos a função “trataResto” para receber um Int i e independente do que seja, retorne “Alguma coisa”. Dessa forma, quando utilizamos applyOrElse e passamos 16 e a função “trataResto”, conseguimos aplicar nosso parâmetro e deixar

no segundo parâmetro a função que tratará os casos que não cobrimos anteriormente.

Veremos funções parciais novamente no método *collect* futuramente.

Vamos falar um pouco mais das funções parcialmente aplicadas.

Imagine que temos uma função que soma dois valores:

```
val soma = (a: Int, b: Int) => a + b
```

Normalmente, ao chamarmos a função soma, passaríamos dois parâmetros para que ela retorne algum valor. No entanto, podemos omitir um dos dois parâmetros, digamos que vamos passar apenas o primeiro parâmetro da função:

```
soma(5, _)
```

Utilizamos o "_" para omitir o segundo parâmetro. Consegue imaginar o que essa função gera? Normalmente passamos Int e Int e ela retorna Int, mas nesse caso passamos apenas um dos dois Int dos parâmetros, ou seja, ainda falta o outro parâmetro, o outro Int. Dessa forma, o valor resultante da chamada acima é uma nova função, uma que recebe Int e retorna Int!

```
val soma5: Int => Int = soma(5, _)
```

Agora nossa função precisa de mais um parâmetro e o 5 que passamos anteriormente "está guardado lá dentro", pronto para ser somado ao parâmetro que for passado. Por exemplo:

```
println(soma5(2))
```

O resultado aqui será 7!

Currying

O conceito de *Currying* introduz um poder a mais para as funções que escrevemos. Normalmente criamos funções com múltiplos parâmetros todos agrupados entre parêntesis, todos "na mesma caixinha", não é? Currying é a técnica que permite que uma segunda lista de parâmetros seja adicionada à nossa função através de uma sintaxe extremamente simples e intuititiva:

```
def somaDoisNumeros(primeiro: Int)(segundo: Int): Int = primeiro + segundo
```

Apenas precisamos colocar o novo grupo de parâmetros após o primeiro e pronto!

Para chamar a função é igualmente simples:

```
print(somaDoisNumeros(10)(11))
```

Apesar de parecer uma funcionalidade pouco relevante, a capacidade de escrever funções utilizando currying é bastante explorada em libs da linguagem, inclusive em métodos muito comuns e poderosos que utilizamos em estruturas como a nossa tão querida *List*.

6. Funções Populares

Os métodos que aparecem nessa seção não são específicos para o tipo List, mas abordaremos sua aplicação apenas nesse tipo para não estender muito os exemplos. A ideia é que possamos entender o funcionamento desses métodos e depois aplica-los a outras estruturas, já que a lógica de uso não muda.

Map e FlatMap

De forma simplificada, podemos definir essas duas funções como funções que transformam elementos, popularmente conhecidas por seu uso dentro de coleções como List ou Seq.

Pensando em listas, vamos ver como é a definição de um map em uma List[Int]:

```
map[B](f: Int => B): List[B]
```

podemos notar que é uma função que recebe em seu parâmetro uma outra função, uma função que recebe um elemento Int e o transforma em um tipo B. Por exemplo, podemos utilizar map em uma lista de Int e transformá-la em uma lista de String:

```
List(1, 2, 3).map(n => n.toString)
```

De forma genérica, dizemos que a função map recebe uma função que transforma um tipo A em um tipo B, o tipo A é o tipo da List que estamos trabalhando, nesse caso é uma List[Int], então nosso tipo A é o tipo Int e o tipo B é o tipo que queremos como resultado ao aplicar essa função à cada elemento da nossa List.

```
map[B](f: A => B): List[B]
```

No exemplo, utilizamos a função toString para transformar cada Int dentro de nossa List em uma String.

Outro exemplo de uso da função map:

```
List(1, 2, 3).map(n => List(n))
```

Cada número da nossa função passa a ser uma nova lista interna dentro da nossa lista maior, logo, o resultado dessa função é uma List[List[Int]] (lista de lista de inteiros).

Vamos continuar falando de listas que possuem listas e comentar sobre flatMap.

Imagine que temos uma lista com dois elementos, cada um é uma lista de inteiros, o primeiro possui números pares e o segundo números ímpares:

```
val listOfLists = List(List(2, 4, 6, 8), List(1, 3, 5, 7, 9))
```

Se quisermos duplicar o valor de cada elemento interno de cada lista, precisamos escrever dois maps:

```
List(List(2, 4, 6, 8), List(1, 3, 5, 7, 9))
  .map(list => list.map(n => n * 2))
```

E se quiséssemos retirar essas listas de dentro? Torná-las uma grande List[Int] ao invés de uma List[List[Int]].

Podemos utilizar o método flatMap:

```
List(List(2, 4, 6, 8), List(1, 3, 5, 7, 9))
  .flatMap(list => list.map(n => n * 2))
```

Dessa forma, nossa lista mais externa passa a ser a única lista existente, já que o método flatMap tem a funcionalidade de tornar nossa estrutura aninhada de listas em uma única lista.

Utilizando uma simples List[Int] para nosso exemplo, a notação de flatMap é:

```
flatMap[B](f: Int => List[B]): List[B]
```

Nosso flatMap recebe como parâmetro uma função que transforma um elemento Int em uma nova lista de outro tipo. Pensando de forma genérica nos tipos envolvidos, flatMap transforma um tipo A em um tipo List[B] e retorna o tipo List[B]).

```
flatMap[B](f: A => List[B]): List[B]
```

Você verá diversas estruturas que suportam os métodos map e flatMap para transformação de dados, estruturas muito úteis e populares como Option, Either e Try são alguns exemplos, não falarei delas nesse material, mas na seção de leituras deixarei textos para futuros estudos.

Filter e Collect

Os métodos filter e collect possuem uma aplicação até que parecida, com eles conseguimos fazer filtragem de elementos.

Vamos começar falando de filter. Sua assinatura em uma List[Int] é:

```
filter(f: Int => Boolean): List[Int]
```

Para cada elemento da nossa lista, a função que passamos no parâmetro é aplicada e pode retornar true ou false, esse retorno determina se esse elemento continuará ou não na lista.

Imagine que queremos filtrar uma lista para deixar apenas os números pares nela:

```
List(1, 2, 3, 4, 5).filter(n => n % 2 == 0)
```

Ou que queremos manter apenas os números maiores que 10:

```
List(12, 1, 199, 20).filter(n => n > 10)
```

Seu uso é extremamente simples. A notação pensando em elementos genéricos é:

```
filter(f: A => Boolean): List[A]
```

Já o método collect é bem mais poderoso, mas também faz uma filtragem.

Com collect podemos passar uma função parcial por parâmetro que será aplicada aos elementos da lista. Caso o elemento não esteja no domínio da função parcial, ele será descartado, senão, será mantido no resultado final.

Essa função, por receber uma função parcial como parâmetro, pode fazer mais que apenas filtrar a lista, por exemplo:

```
List(11, 10, 9)
  .collect {
    case n if n < 10 => n * 2
    case n if n > 10 => n + 1
  }
```

Nesse caso, nossa função parcial não é aplicada ao número 10, que simplesmente deixará de fazer parte da lista final, enquanto os outros elementos serão transformados de acordo com a lógica para cada “case”.

Reduce e Fold

Imagine que queremos pegar toda uma lista e transformá-la em um único elemento, como quando precisamos somar todos os elementos de uma lista, por exemplo.

Para isso, podemos utilizar as funções reduce e fold.

Vamos olhar a assinatura da função reduce:

```
reduce[A1 >: A](op: (A1, A1) => A1): A1
```

A1 é um supertipo de A. O parâmetro op é uma função que recebe dois parâmetros do tipo A1 e retorna o tipo A1. O resultado de um reduce é o tipo A1.

Vamos deixar isso menos abstrato com um exemplo:

```
List(11, 10, 9)
  .reduce((a, b) => a + b)
```

Temos aqui uma lista de inteiros e passamos para o reduce uma função de soma entre dois inteiros, então A1 é do tipo Int aqui.

A função será aplicada para cada elemento da lista e o resultado dela ficará guardado no primeiro parâmetro dessa função, no caso, o parâmetro “a”. Ao final da operação, para esse exemplo, teremos a soma de todos os elementos da lista.

Um outro exemplo:

```
List(11, 10, 9)
  .reduce((a, b) => if (a < b) b else a)
```

Aqui, utilizamos a função reduce para encontrar o maior item da nossa lista. Já que o resultado da operação fica armazenado em “a”, basta checar seu valor em relação à “b” e conseguiremos encontrar o maior elemento.

Fold é uma função que atinge os mesmos resultados que reduce, com a diferença de que é uma função construída por Currying, portanto possui dois grupos de parâmetros que precisamos preencher:

```
fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1
```

O parâmetro “z” é nossa seed, nosso valor inicial para começar a função. É o primeiro valor que o primeiro parâmetro da função op receberá no início da execução (chamamos esse primeiro parâmetro de “a” anteriormente).

Veja um exemplo de soma:

```
List(1, 2, 3)
  .fold(0)((a, b) => a + b)
```

Como comentei, “a” receberá 0 como seu valor inicial e “b” receberá a head da lista.

Dessa forma, podemos utilizar fold em uma lista vazia sem uma exception ser lançada, já que o primeiro parâmetro (“z”) é o retorno de fold aplicado à uma lista vazia.

7. Tópicos Avançados

Gostaria de introduzir essa etapa do material explicando um pouco melhor o título. Não abordarei conteúdo realmente avançado de programação funcional, mas comentarei um pouco mais a fundo sobre alguns Syntax Sugar que usamos e sobre a facilidade que os Implicits trazem na linguagem.

Syntax Sugar

Funções e o método Apply

Já vimos várias vezes essas estruturas ao longo do material e todas as vezes foi dito que veríamos elas mais detalhadamente no futuro. Bom, o futuro é agora.

Vamos começar da primeira vez que definimos uma Case Class ou um Companion Object que disponibilizasse implementação da função apply.

As vezes que vimos nossa classe ser instanciada sendo chamada sem a palavra **new** na frente era pela capacidade que o compilador tem de “reescrever” a estrutura para uma chamada da função apply que fica dentro do Object.

Mas o que não ficou claro é que estávamos usando essa Syntax Sugar a todo momento quando invocávamos uma função. Por exemplo:

```
val somaUm: Int => Int = i => i + 1
print(somaUm(1))
```

Quando chamamos nossa função dentro do print, na verdade estamos fazendo:

```
print(somaUm.apply(1))
```

Doido, né?

Funções que declaramos em Scala são instâncias anônimas da Trait Function que contém um método apply que deve ser implementado para a criação da trait. Podemos reescrever a declaração da nossa função para:

```
val somaUm = new Function[Int, Int] {
  override def apply(v1: Int): Int = v1 + 1
}
```

Os tipos que passamos como parâmetro para “Function” representam o tipo que recebemos na função e o tipo que ela retornará, consecutivamente. No método apply nós definimos qual será o corpo da função.

Podemos reescrever essa mesma definição da seguinte forma:

```
object somaUm extends Function1[Int, Int] {
  override def apply(v1: Int): Int = v1 + 1
}

println(somaUm(1))
```

Embora essas estruturas podem ser reescritas sem a utilização do Syntax Sugar, o ideal é esquecermos que elas existem dessa forma feia e nos apegarmos mais à nossa notação limpa, objetiva e elegante que utiliza setinha:

```
val somaUm: Int => Int = i => i + 1
```

A Trait Function vai de Function0 a Function22, sendo Function0 uma função que não recebe parâmetros e Function22 uma que recebe 22 parâmetros. Em ambos os casos precisamos disponibilizar, na última posição, o tipo que representa o retorno daquela função:

```
val fazNada = new Function0[Unit] {
  override def apply(): Unit = println("nada")
}
```

Repare como mesmo Function0 sendo uma função que não recebe parâmetros (podemos notar pelo método apply que não possui parâmetros) ainda precisamos especificar o tipo de retorno, que nesse caso, é Unit.

Underscore

Veremos diversas vezes códigos Scala que utilizam essa funcionalidade para “enxugar” a escrita de algumas estruturas. O underscore é odiado por uns e amado por outros, já que pode deixar o código bem menor, mas também pode dificultar mais a legibilidade.

Como vimos no uso do método map, podemos multiplicar uma lista de inteiros por dois da seguinte forma:

```
val lista = List(1, 2, 3)

lista.map(numero => numero * 2)
```

Como utilizamos apenas um parâmetro nessa função lambda que definimos e esse único parâmetro é utilizado apenas uma vez no corpo da função, podemos economizar uns caracteres reescrevendo a função dessa forma:

```
lista.map(_ * 2)
```

A função está muito menor, mas meio estranha. Essa estrutura é totalmente válida e significa exatamente a mesma função que escrevemos anteriormente.

Podemos também acessar elementos de uma classe utilizando a sintaxe do underscore:

```
case class Pessoa(nome: String)

val pessoas = List(Pessoa("Alice"), Pessoa("Bob"))

val nomes = pessoas.map(_.nome)
```

Essa estrutura compacta trás certa clareza na leitura, já que o atributo “nome”, que é o que queremos retornar de cada pessoa, fica mais evidenciado com o uso do underscore para “diminuir” a função.

Podemos pensar que o underscore torna a leitura mais direta, já que não tem mais a “redundância” do parâmetro antes e depois da seta.

Outro caso em que usamos o underscore é no uso de Pattern Matching, quando precisamos apenas verificar qual o tipo que estamos recebendo, por exemplo:

```
def qualTipo(x: Any): String = x match {
  case _: String => "string"
  case _: Int => "int"
  case _: Double => "double"
  case _: Boolean => "boolean"
}
```

Ou usando um exemplo que explora mais a capacidade do Pattern Matching

```
trait Pessoa
trait PJ extends Pessoa
trait PF extends Pessoa

def qualSubTipo(pessoa: Pessoa): String = pessoa match {
  case _: PF => "pessoa física"
  case _: PJ => "pessoa jurídica"
}
```

Em Pattern Matching também podemos escolher utilizar o underscore quando vamos desconstruir um objeto, mas nem todas as informações que ele possui serão relevantes para nós:

```
case class Pessoa(nome: String, idade: Int, altura: Double, solteiro:
Boolean)

val bob = Pessoa("Bob", 12, 1.90, true)

bob match {
  case Pessoa(_, idade, altura, _) if idade < 14 & altura > 1.8 =>
println("diferenciado!")
}
```

Nesse caso, nossa Pessoa possui diversos atributos que virão juntos ao desconstruir o tipo, para evitar declarações desnecessárias, podemos simplesmente omitir outros atributos com o underscore.

A última estrutura que vimos e era um Syntax Sugar também foi dentro de um Pattern Matching, quando utilizamos a notação:

```
case head :: tail =>
```

que poderia ser reescrita como:

```
case ::(head, tail) =>
```

Isso é totalmente estranho e seu uso é bem situacional (minha opinião), mas é um Syntax Sugar que cai muito bem quando estamos fazendo desconstrução de listas.

Vamos ver como funciona.

Primeiro definimos uma Case Class que possui dois membros em seu construtor.

```
sealed trait Possuidor

case class DonoDe(pessoa: String, cachorro: String) extends Possuidor
case class FilhoteDe(gatinho: String, gatao: String) extends Possuidor
```

Defini uma Trait duas Case Class só para fazer mais sentido no exemplo.

Agora precisamos desconstruir esses carinhas em um Pattern Matching e ver a mágica:

```
def mostraRelacao(possuidor: Possuidor): Unit = possuidor match {
  case pessoa DonoDe cachorro => println(s"A pessoa $pessoa é dona do
cachorro $cachorro")
  case gatinho FilhoteDe gatao => println(s"O gatinho $gatinho é filhote do
gato $gatao")
}
```

Aí está. Podemos desconstruir nosso parâmetro “Possuidor” e checar se é uma pessoa dona de um cachorro ou um gato filhote de um outro gato. Como na desconstrução da lista, temos parâmetro, classe e parâmetro (head, ::, tail) da mesma forma aqui nesses “cases” do Pattern Matching (pessoa DonoDe cachorro) (gatinho FilhoteDe gatão).

Um Syntax Sugar do Scala realmente bonito e que faz com que nosso código fique bem limpo é a capacidade de omitir o “.(parametro)” ao chamar funções que possuem apenas um parâmetro:

```
val valores = List(1, 2, 3)
val menoresQue10: Int => Boolean = i => i < 10

valores filter menoresQue10
```

Definimos uma lista e uma função que recebe um único parâmetro Int e retorna um Boolean (chamado de Predicate em Java), essa função é o único parâmetro da função filter, dessa forma, podemos invocar a função sem a sintaxe comum:

```
valores.filter(menoresQue10)
```

Implicits

Vamos começar direto com um código que faz a gente entender como Implicits são incríveis:

```
val diaQuente = true
val ventiladorLigado = false

if (diaQuente butNot ventiladorLigado) {
  // código para ligar ventilador
}
```

Essa estrutura “butNot” não é padrão do Scala, é uma classe escrita por nós!

Como estamos “recriando” uma checagem entre dois argumentos Boolean? Bom, isso é possível através de uma *Implicit Class* (classe implicita) que definimos:

```
implicit class BetterBoolean(boolean: Boolean) {

  def and(another: Boolean): Boolean = boolean && another
  def or(another: Boolean): Boolean = boolean || another
  def butNot(another: Boolean): Boolean = boolean && !another
}
```

Uma classe implícita é definida pela palavra “implicit” antes da definição da classe e obrigatoriamente recebe apenas um parâmetro em seu construtor. Esse único parâmetro é o tipo que será alvo de uma *Implicit Conversion* (conversão implícita) pelo compilador. No nosso caso, nossa classe implícita recebe um Boolean e

transforma ele em um BetterBoolean que possui implementado métodos “and”, “or” e “butNot” para facilitar a legibilidade da relação entre dois valores Boolean.

Graças a um Syntax Sugar que já vimos, podemos invocar esses métodos omitindo o ponto e os parênteses, possibilitando um resultado bem bonito de ler, como mostrado no início do capítulo.

Mais um exemplo de uso da classe que montamos:

```
val somAlto = true
val estouDormindo = true

if (somAlto and estouDormindo) {
  // lógica para acordar bravo
}
```

Vamos dar uma olhadinha em mais código, mas dessa vez em uma estrutura do Scala, o Range:

```
val range1a10 = 1 to 10
```

Definimos um Range.Inclusive, ou seja, um range que vai de 1 a 10, inclusive o último. Isso graças ao método “to” disponível na classe RichInt. Como conseguimos acessar esse método se RichInt não é uma classe implícita, mas apenas uma classe comum? Bom, existem conversões implícitas através de métodos. Vamos reescrever nossa classe BetterBoolean utilizando conversão implícita por método:

```
class BetterBoolean(boolean: Boolean) {

  def and(another: Boolean): Boolean = boolean && another
  def or(another: Boolean): Boolean = boolean || another
  def butNot(another: Boolean): Boolean = boolean && !another
}

implicit def toBetterBoolean(boolean: Boolean): BetterBoolean =
  new BetterBoolean(boolean)

val compilou = true
val rodou = true

if (compilou and rodou) {
  // bão demais!
}
```

Lembra de como Currying nos permite criar dois grupos de parâmetros para nossas funções? Bom, podemos utilizar essa técnica ao declarar parâmetros implícitos para nossos métodos:

```
class Pessoa
class Lobisomem {
  def uivar: Unit = println("UUUUUU")
}

class Noite(val luaCheia: Boolean)

def transformacao(noite: Noite, pessoa: Pessoa)(implicit transformation:
Pessoa => Lobisomem): Unit = {
  if (noite.luaCheia) {
    pessoa.uivar
  } else {
    println("noitinha boa pra um Scala...")
```

```
  }
}
```

Para preencher esse parâmetro implícito, basta termos definido um método implícito que satisfaça a tipagem que definimos na assinatura do nosso método, no caso, algum método que recebe Pessoa e devolva Lobisomem. Esse método extra pode ser passado explicitamente no segundo grupo de parâmetro ao chamar a função ou podemos apenas omiti-lo (que é o mais comum), ambas formas resultam no mesmo fim. Por exemplo:

```
implicit def pessoaLobisomem(pessoa: Pessoa): Lobisomem = new Lobisomem
val noite = new Noite(true)
val pessoa = new Pessoa
transformacao(noite, pessoa)
transformacao(noite, pessoa)(pessoaLobisomem)
```

Fora os fru-frus para enfeitar o exemplo, o que estamos fazendo aqui é usar Currying para receber uma conversão implícita por parâmetro entre os tipos Pessoa e Lobisomem. O uso dessa técnica aqui é obrigatório, pois temos algumas regras no uso de Implicits:

- Métodos que recebem parâmetro implícito devem separá-lo dos parâmetros comuns.

- Classes implícitas só podem ter um parâmetro em seu método construtor.

- Classes implícitas não podem ser “top level”, devem estar dentro de um Object ou de outra classe.

O escopo da busca de conversões implícitas é:

- Escopo local

- Escopo importado

- Companion Objects dos tipos relacionados com a conversão

8. Leituras

Para continuar seus estudos com a linguagem ou com o paradigma existem diversas fontes disponíveis.

No Brasil, possuímos o grupo “Scaladores”, um grupo de desenvolvedores com foco em compartilhar conhecimento sobre Scala e Programação Funcional. Você pode encontrar eles no Slack (scaladores.slack.com).

Fora do Brasil possuímos diversas fontes de aprendizado.

Existem ótimos e-books para aprender mais sobre Scala, como o “Functional Programming, Simplified” do Alvin Alexander ou o “Scala Cookbook” do mesmo autor. Além disso, o autor desses e-books possui um blog.

Para aulas em vídeo temos o excelente instrutor Daniel Ciocîrlan, com o “Rock The JVM”, um site e um canal na Udemy com ótimos cursos sobre Scala e libs muito populares como Cats ou frameworks grandes como o Akka e o Spark.

Temos material oficial de Scala no site da linguagem e na Coursera um curso ministrado por alguns instrutores, entre eles a figura central da linguagem, Martin Odersky.

Para exercitar o aprendizado, além de criar projetinhos próprios aplicando as estruturas e técnicas aprendidas, você também pode encontrar exercícios online.