

O guia de Dart

Fundamentos, prática, conceitos avançados
e tudo mais



Casa do
Código

| alura

JULIO BITENCOURT

Sumário

- ISBN
- Sobre o autor
- Sobre o livro
- Agradecimentos
- 1. Hello, Dart!
- 2. O básico
- 3. Benditos tipos
- 4. Explorando mais as funções e a Web
- 5. Cuidando dos erros
- 6. Entendendo as Libraries
- 7. Na prática - Packages
- 8. Oriente seus objetos
- 9. Generics<T> e as estruturas de dados
- 10. Concorrência
- 11. Na prática - Dart CLI
- 12. Stream é tão funcional...
- 13. Um pouco mais sobre streams
- 14. Um pouco mais sobre Isolates e Zones
- 15. Na prática - Gerando arquivos
- 16. Até mais, e obrigado pelos peixes!

ISBN

Impresso: 978-85-5519-298-2

Digital: 978-85-5519-299-9

Ilustração da capa: João Antunes (@antunesketchreal)

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

Sobre o autor



Autor.

Figura -2.1: Autor.

Bacharel em Ciência da Computação e pós-graduado em desenvolvimento mobile pela FIAP, atualmente no auge dos meus 26 anos, entrei de cabeça no mundo da programação há mais ou menos nove anos. Nele, trabalhei e me especializei principalmente no desenvolvimento de aplicações em Java (sim, Java sempre estará no meu coração) por quase toda a minha carreira. Atuei como Backend Developer em vários projetos com todo o stack JavaEE em conjunto com criações de diversas APIs REST e Soap.

Um eterno estudante, acredito que conhecimento nunca será demais e sempre é o momento certo para ser um aprendiz em algo. Por isso, mantendo um blog nas horas vagas, onde gosto de escrever sem amarras textos agradáveis, essencialmente voltados para tecnologias, que acredito que as pessoas gostariam de ler.

Entusiasta em desenvolvimento mobile, após anos flirtando com desenvolvimento Android, a partir do final de 2017 venho me especializando e trabalhando com Dart na criação de aplicações multiplataformas e nativas com o querido Flutter - SDK no qual tenho prazer em trabalhar hoje em dia em meu atual emprego como Desenvolvedor Mobile.

Posso ser facilmente encontrado em algum dos links abaixo:

- **Site e Blog:** juliobitencourt.com
- **Medium:** medium.com/@julio.henrique.b
- **Instagram:** [@juliobitencourt.dev](https://www.instagram.com/@juliobitencourt.dev)

Sobre o livro

Este livro é indicado para todos aqueles e todas aquelas que desejam expandir um pouco o seu conhecimento sobre Dart, ou até mesmo para quem não conhece nada da linguagem e quer se aventurar nesse novo mundo, tendo uma nova carta na manga em seu currículo para começar a desenvolver algo server side para web, desktop, mobile, IOT... São várias possibilidades!

O objetivo é que o livro sirva como um guia estruturado para introdução e aprofundamento na linguagem. Nele, abordarei desde os conceitos mais básicos envolvendo a sintaxe (como tipos, operadores, estruturas de repetição etc.) até *features* mais avançadas da linguagem (como programação assíncrona, generics etc.), sempre focando na parte teórica seguida de prática com exemplos esclarecedores.

Em algumas partes do livro, você encontrará uma seção denominada *Se liga aí*, ela vai conter alguma dica ou boa prática que não é necessariamente obrigatória no desenvolvimento, porém, com certeza facilitará ou tornará mais elegante a utilização de Dart, afinal, a gente sempre quer ter aquele código bonito de se ver no GitHub, não?

A seção *Na prática* estará presente durante todo o livro e sempre introduzirá um desafio com alguma aplicação prática do conteúdo sendo abordado, seguido da solução e discussão do resultado. Por fim, você será desafiada(o) nas seções denominadas *É com você*, com alguns problemas ou perguntas que servirão de extensão e aprimoramento dos conhecimentos apresentados.

Este livro não é uma abordagem de Dart focada para algum SDK específico, como a utilização do AngularDart para desenvolvimento web ou até mesmo o Flutter para criação de aplicações móveis. Contudo, como eles trabalham em cima do core do Dart, todos os fundamentos e conceitos aprendidos aqui serão essenciais e com certeza úteis para utilização de qualquer outro framework criado a partir da linguagem.

Embora não seja um requisito, o leitor ou leitora aproveitará melhor o livro caso tenha um conhecimento prévio em programação, mesmo que iniciante, afinal o foco do livro é introduzir e mostrar como os conceitos são abordados no universo específico de Dart.

Material atualizado e baseado na **versão 2.16.2** da linguagem. Também é possível acessar o website do livro em <https://dartguide.dev> para obter os códigos dos exemplos utilizados.

Agradecimentos

Agradeço a toda a minha família que direta ou indiretamente sempre me auxiliam em todas as etapas da minha vida, em especial minha mãe e pai que são a razão de tudo que faço e alcancei.

Agradeço à editora Casa do Código pela oportunidade de dar vida a este livro, principalmente a Vivian Matsui e Sabrina Barbosa que editaram e fizeram a revisão gramatical, sempre muito profissionais e com muita paciência nos momentos em que demorei entregar algum capítulo.

Agradeço aos colegas de trabalho e amigos do Instituto de Tecnologia do Senai, onde sempre tive liberdade e pude evoluir profissionalmente de forma exponencial.

Agradeço aos amigos João Damiani e Josiel Borges que contribuíram fazendo a revisão técnica, garantindo que tudo que está aqui descrito, de fato, funciona.

Sem todos vocês isso não seria possível. Obrigado.

CAPÍTULO 1

Hello, Dart!

Dart é uma linguagem de programação em ascensão, que vem ganhando destaque e um apoio maior da comunidade nos últimos anos. Por ser relativamente nova, comparada a outras linguagens que dominam o mercado há muito tempo, já nasceu com vários princípios e conceitos reaproveitados que deram certo em outras linguagens. Isso também torna a sua curva de aprendizado muito tranquila e de fácil adoção para programadores e programadoras já com experiência prévia em Java, C++ ou semelhantes.

Hoje, ela é considerada uma linguagem multiplataforma, uma vez que a partir dela e de seus frameworks conseguimos gerar aplicações nativas e de alta performance em server-side, na web (transpilando para JavaScript), em desktop (Windows, Linux, Mac e Chromebook), nos dispositivos mobile (Android, iOS, Fuchsia) e em IoT (internet das coisas).

Para conhecê-la melhor, neste capítulo, vamos:

- Conhecer e entender a história da linguagem;
- Descobrir os motivos para utilizá-la;
- Criar e executar o primeiro trecho de código.

1.1 Uma breve história

Nada melhor do que iniciar a nossa jornada sabendo exatamente onde estamos pisando. Por isso, ter o conhecimento da motivação e história por trás de uma tecnologia, embora não seja um fator determinante para estabelecer proficiência nela, nos dá uma bagagem maior para entender de onde ela veio e para onde ela vai.

Então esse é o nosso primeiro objetivo, estabelecer e compreender os principais acontecimentos que foram marcantes durante os anos, em todo o ecossistema de Dart, desde sua criação até os dias atuais. E é uma história longa, por isso vamos focar nos pontos principais de evolução da linguagem em si, e não em frameworks secundários. Tudo isso teve início lá em 2011.

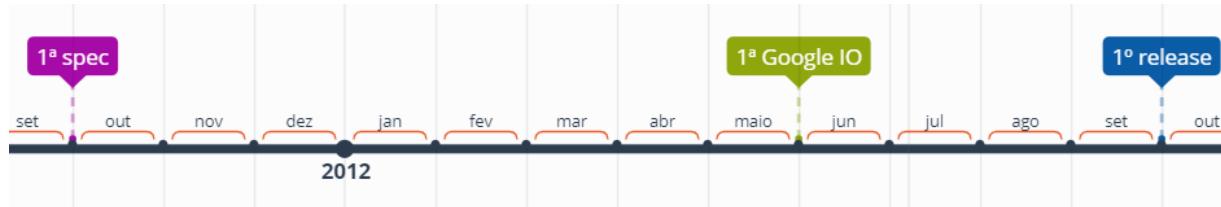


Figura 1.1: Timeline 2011-2012.

Dart é uma linguagem de programação criada e mantida pelo Google e teve sua primeira *spec* (documento técnico com todas as funcionalidades disponíveis em uma linguagem de programação) aberta ao público em outubro de 2011. A partir daí, após muito trabalho da equipe responsável, ela já foi apresentada ao mundo em uma *talk* feita pelos fundadores do projeto, Lars Bak e Kasper Lund, na conferência anual do Google I/O em junho de 2012.

Logo após um ano da primeira *spec*, teve o lançamento do primeiro *release* do SDK, apresentando uma versão que já trazia algumas ferramentas que utilizamos até hoje, como o pub (gerenciador de pacotes para a linguagem), além da criação de um editor próprio (conhecido apenas como Dart Editor baseado na estrutura do famoso editor Eclipse), que futuramente acabou sendo descontinuado.

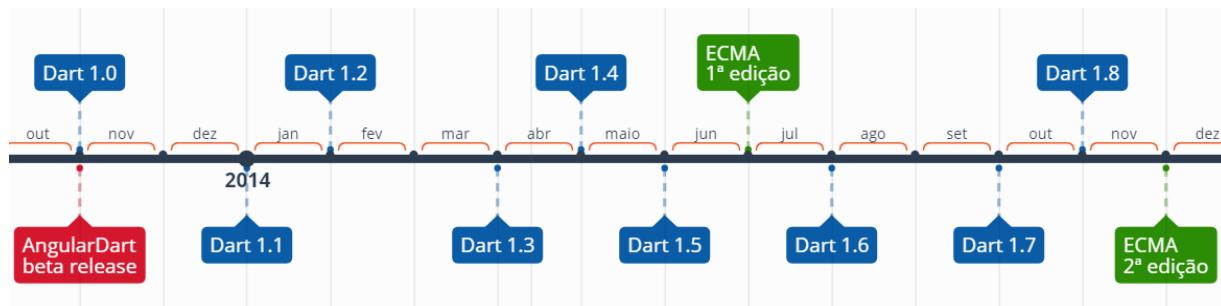


Figura 1.2: Timeline 2013-2014.

Foi só em outubro de 2013, após trabalhar na maturidade da linguagem com os *early adopters* (pessoas ou empresas que adotam tecnologias em fase inicial) e tornar o que antes era um protótipo de linguagem em algo pronto para produção, que foi lançada a versão estável 1.0 do SDK.

Com ela, acompanhavam, entre outras coisas, o editor oficial, o Dartium (uma versão do Chromium com a Dart Virtual Machine) e uma performance da VM que chegava a ser até 130% mais rápida do que a V8 (engine responsável por rodar o JavaScript no Chromium e outros projetos) na época. É importante ressaltar que Lars Bak também foi o fundador do V8, sendo um especialista em JavaScript.

Nesse mesmo mês do lançamento da 1.0, também pode ser destacado o anúncio da primeira versão beta do AngularDart, framework para construção de aplicações Web.

Já em 2014, os releases começaram a aparecer com uma maior frequência, afinal houve uma certa adoção da linguagem pela comunidade, na qual os responsáveis foram promovendo encontros e maneiras de receberem feedbacks para aperfeiçoar o ecossistema da linguagem ao longo do tempo. Entre novembro de 2013 e novembro de 2014, tivemos:

- **Dart 1.1:** o primeiro release após a versão 1.0 trouxe uma melhora de 25% em performance do JavaScript em relação à anterior, junto a uma maior expansão para o caminho de Dart no server-side.
- **Dart 1.2:** melhorias em performance e no editor, sendo possível debugar melhor as aplicações.
- **Dart 1.3:** melhorias significativas de performance de código assíncrono rodando na VM.
- **Dart 1.4:** a criação do Observatory, um conjunto de ferramentas para desenvolvimento que acompanham o SDK para medir a performance e analisar o comportamento das aplicações.
- **Dart 1.5:** melhorias voltadas principalmente para a execução das aplicações web no universo mobile.
- **Dart 1.6:** melhorias de segurança e a apresentação do carregamento sob demanda de *libs*.
- **Dart 1.7:** melhorias na utilização de Dart via linha de comando.

- **Dart 1.8:** mudanças pontuais nas *libs* do SDK e um experimento com a funcionalidade de *enums*.

Ainda nesse ano, em julho, a primeira especificação da linguagem foi aceita pelo ECMA, a associação internacional que padroniza especificações de sistemas de informação. No ano seguinte, em 2015, os releases diminuíram, mas não as novidades.

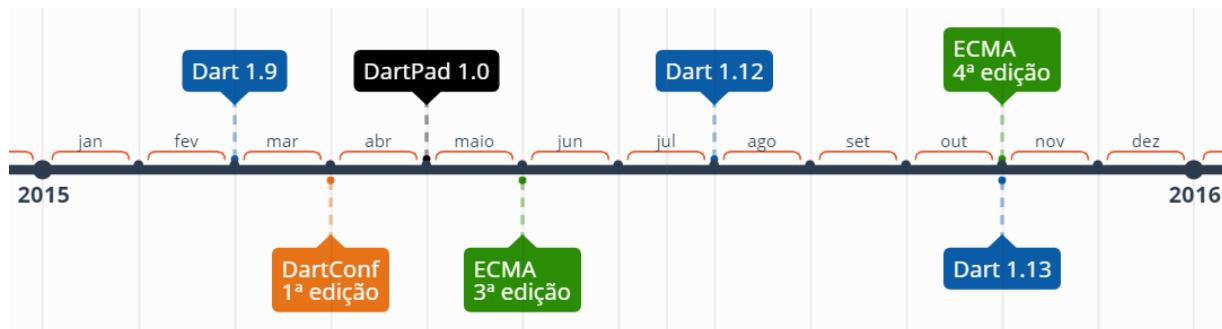


Figura 1.3: Timeline 2015.

Abril ficou marcado como o mês da primeira Dart Developer Summit, conferência realizada pelo Google para abordar assuntos específicos do universo da linguagem. Foi nela que o público conheceu o DartPad, uma ferramenta on-line, acessível facilmente através do link <https://dartpad.dev>, simples e perfeita para testes rápidos, pois ela apresenta os resultados de seus códigos em Dart no próprio navegador.

Um outro ponto interessante desta conferência foi uma palestra de apresentação do que até então era um framework experimental para execução de Dart em plataformas mobile, o denominado Sky. Um projeto que logo depois acabou sendo renomeado para Flutter e revolucionou este universo.

Os releases da linguagem no ano foram:

- **Dart 1.9:** trouxe o suporte para *enums*, mudanças na VM para execução de Isolates (forma em que Dart trabalha com threads) e os incríveis modificadores `async` e `await` para trabalhar com operações assíncronas acima da API de `Future`. Além dos métodos geradores com `sync*` e `async*`.

- **Dart 1.12:** sua principal novidade foi a criação dos operadores condicionais nulos (?? , ??= , ?), que facilitam a vida dos desenvolvedores e desenvolvedoras ao tratar possíveis referências a objetos nulos.
- **Dart 1.13:** teve melhorias na segurança de rede para aplicações server-side e na interoperabilidade do Dart com bibliotecas em JavaScript.

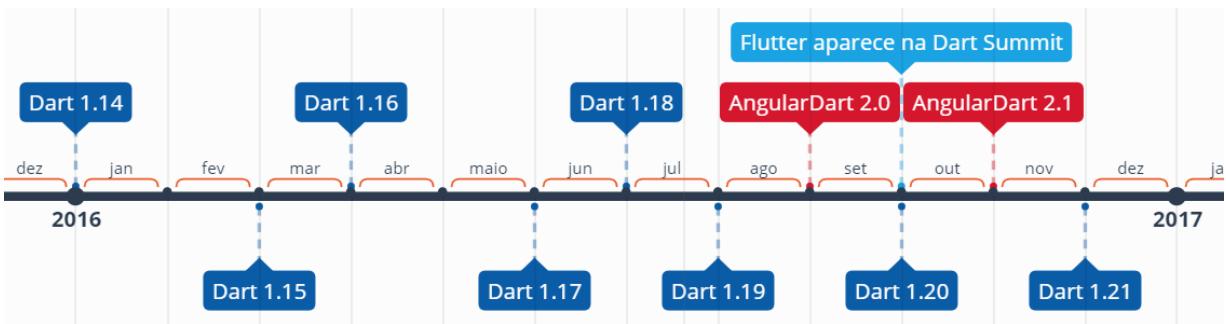


Figura 1.4: Timeline 2016.

Já 2016 se mostrou um ano mais movimentado com os releases e deu mais força a algo que viria se tornar uma das tecnologias de ponta para brigar no mercado de aplicações mobile. Na Dart Summit, em outubro, o projeto **Flutter**, que já havia perdido o nome de Sky a essa altura, foi oficialmente apresentado em *tech preview* como uma nova alternativa promissora para os desenvolvedores e desenvolvedoras que desejam criar apps nativos para as plataformas android e iOS. A partir desse momento, Dart estava completando um ciclo multiplataforma, podendo rodar perfeitamente em aplicações CLI, server-side, na web e mobile, comprovando ser uma linguagem madura o suficiente e completamente adaptável.

Sobre os releases da linguagem, podemos destacar:

- **Dart 1.14:** algumas mudanças na API de algumas *libs* padrões do SDK.
- **Dart 1.15:** atualização da versão do Dartium e melhorias no analisador do Dart utilizado pelas IDEs.
- **Dart 1.16:** atualização das APIs do html para refletirem a atualização do Dartium feita anteriormente, e melhorias de performance no Dartium.

- **Dart 1.17**: também trouxe melhorias de performance no analisador e Dartium.
- **Dart 1.18 e Dart 1.19**: focaram em melhorias e performance da linguagem para uma melhor experiência ao trabalhar com o Flutter.
- **Dart 1.20**: os *symbolic links* deixaram de ser suportados, além de haver uma melhora no *stacktrace* de erros de *test*.
- **Dart 1.21**: o primeiro release após a Dart Summit, veio com o suporte para sintaxe de *generics* em métodos.

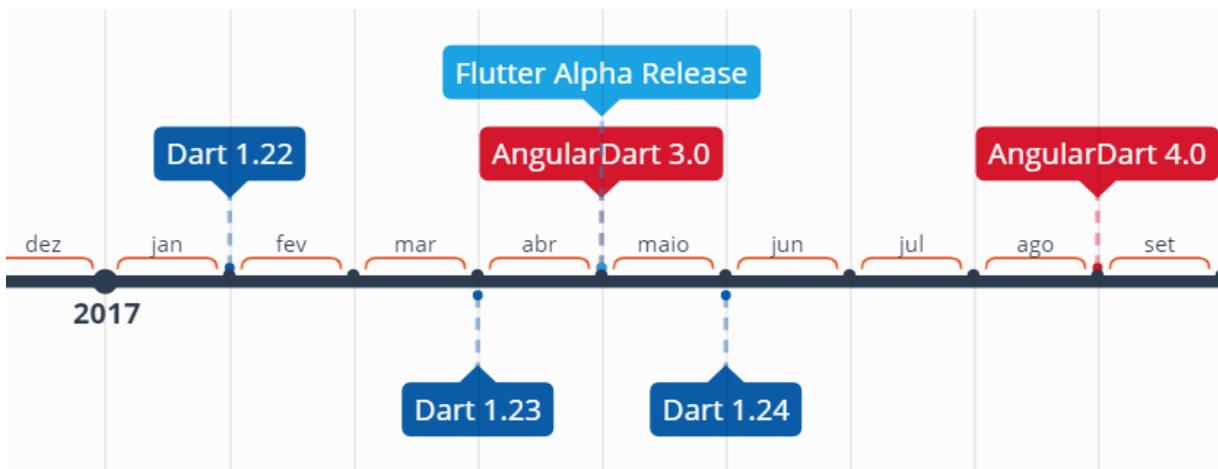


Figura 1.5: Timeline 2017.

No primeiro release de 2017, versão 1.22, além de melhorias na performance, o time de Dart trouxe mudanças para a sintaxe da linguagem, como a possibilidade de adição de mensagens diretamente nos *asserts* (funcionalidade para validação de código em desenvolvimento), a adição do modificador `covariant` ao trabalhar com heranças, e a criação do tipo `FutureOr<T>` para trabalhar com códigos assíncronos que podem em tempo de execução ser um `Future` ou `T`.

Por ter nascido com um foco para a Web e com bastante influências de JavaScript, Dart em sua versão inicial 1.xx era uma linguagem de tipagem dinâmica por padrão e fornecia um modo de tipagem opcional denominado `strong mode`, que podia ser habilitado ao executar as aplicações. O `strong mode` adicionava várias validações e inferências de tipos em tempo de compilação, que acabavam tornando a linguagem em uma tipagem estática, trazendo diversos benefícios que discutiremos ao longo do livro.

Por conta disso, muitas das modificações feitas na estrutura da linguagem nas últimas versões da 1.xx, incluindo a 1.23, foram realizadas justamente para um melhor suporte ao `strong mode` que se tornaria cada vez mais popular. Ainda mais com o Flutter, que em maio desse mesmo ano teve sua primeira versão do SDK, ainda em Alpha, liberada para a comunidade.

A versão 1.24 da linguagem, última do ano, trouxe um novo compilador para Web, denominado *dartdevc (Dart Development Compiler)*, que facilita os ciclos de desenvolvimento da aplicação uma vez que permite facilmente debugar um *web app* com o Chrome.

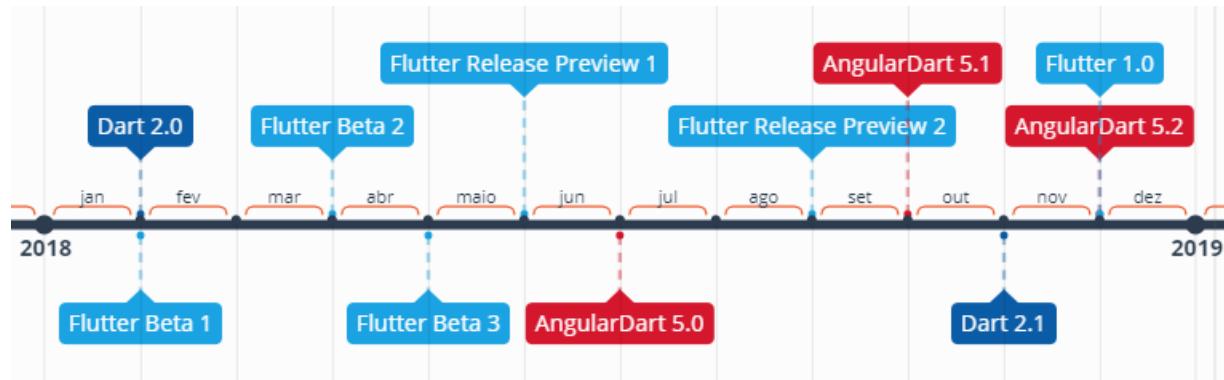


Figura 1.6: Timeline 2018.

O ano de 2018 foi, sem dúvidas, o ano de maiores avanços e crescimento de todo o ecossistema de Dart. Isso por conta da popularidade crescente do Flutter que consequentemente impulsionou a popularidade da linguagem.

Logo no início do ano fomos agraciados com a versão 2.0, muito mais madura e já trazendo o *strong mode* como padrão. Agora a tipagem da linguagem (forma com que ela lida com os tipos) passa a ser restrita e traz um avanço significante na análise e prevenção de erros pelo compilador.

A linguagem sofreu diversas mudanças com o foco para a performance das aplicações client-side (web e mobile). Foi removida a obrigatoriedade da utilização da keyword `new` na criação de objetos e a keyword `const` em contextos que já são constantes, com o intuito de facilitar a leitura e entendimento do código, deixando-o mais *clean*.

A segunda e última versão da linguagem no ano foi então a 2.1, que além de adicionar a inferência de valores do tipo `int` para `double` quando usados neste contexto, trouxe uma nova sintaxe para declaração de *mixins* (uma funcionalidade de relacionamento entre objetos da linguagem).

Com isso, Flutter teve finalmente sua primeira versão estável 1.0 já com suporte ao Dart 2.1, que foi lançada em um evento transmitido mundialmente, fechando em grande estilo o ano de 2018 e alimentando muito todo o *hype* em torno do novo framework focado na construção de UI (*User Interface*) multiplataformas.

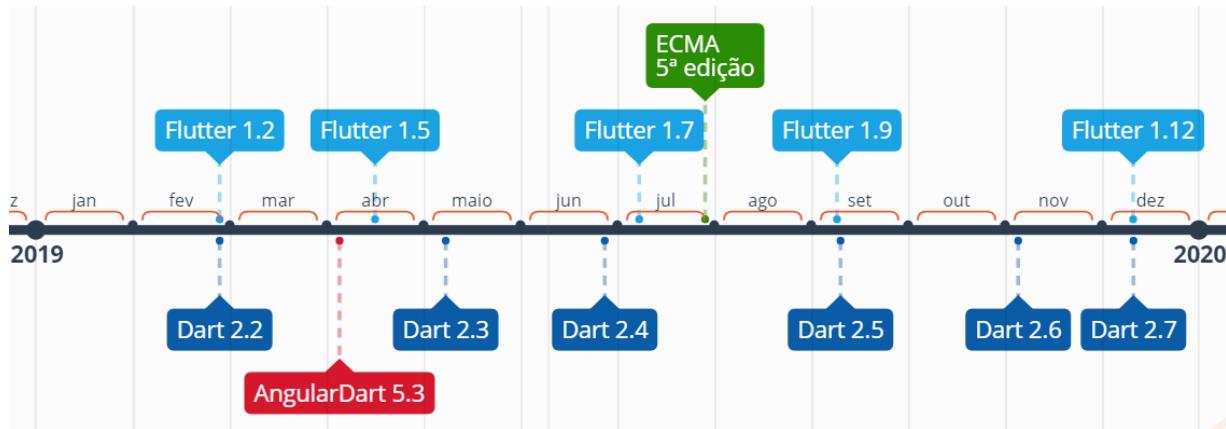


Figura 1.7: Timeline 2019.

O primeiro release da linguagem em 2019 trouxe a sua versão 2.2 com uma melhora significativa na performance de aplicações compiladas nativamente, junto com uma nova sintaxe para declaração de `Set` (estrutura de dados onde os valores não se repetem) de forma literal.

Logo em seguida, Dart 2.3 foi lançada com novas *features* (funcionalidades) focadas em uma melhor experiência na codificação de interfaces de usuário, principalmente no caso de uso do Flutter, que nessa altura com suas versões estáveis já tinha se tornado muito popular. Essa versão trouxe o operador de *spread* (que permite referenciar listas dentro de outras listas), e a possibilidade de utilizar *if* e *for* também dentro de listas. A versão 2.4 não trouxe alguma *feature* nova ou alteração muito significativa, além de pequenos ajustes e mudanças das APIs e ferramentas existentes.

Na versão 2.5 de setembro teve o *preview* de duas novas *features*. A primeira delas é a utilização de *machine learning* (aprendizado de máquina) no recurso de autocompletar código nas IDEs, o que permite prever os comandos de forma mais natural, agilizando a escrita do código. Já a segunda *feature* foi algo bastante solicitado pela comunidade, a nova API `dart:ffi` (*foreign function interface*), que permite a interoperabilidade entre Dart e códigos nativos em C de forma mais natural e direta.

Quando falamos que uma nova *feature* entrou em modo *preview*, significa que ela faz parte da versão atual do SDK, mas não foi lançada oficialmente. Então para os desenvolvedores utilizarem é necessário habilitá-la de alguma forma. Isso é feito para que a comunidade possa testar e dar *feedback* de uma funcionalidade que está em desenvolvimento.

Na versão 2.6 lançada em novembro, o `dart:ffi` já recebeu várias melhorias e passou para uma versão *beta*, deixando de ser apenas *preview*. Foi também nesta versão que foi lançado um novo compilador `dart2native`, que compila códigos Dart em binários executáveis e independentes (não precisam do SDK para rodar) para as plataformas Windows, Linux e macOS. Além de entrar em *preview* a nova *feature* da linguagem, *extension methods* ou métodos de extensão, que permite adicionar novos comportamentos para objetos existentes.

Essa mesma *feature* de *extension methods* deixou de ser apenas *preview* e foi lançada oficialmente na versão 2.7 da linguagem, que também trouxe um novo *package* oficial `characters` para manipular de forma segura caracteres de uma `String`. E foi também nesta versão que a nova *feature* de *null safety* entrou para *preview*, essa que foi a segunda maior mudança estrutural na linguagem, atrás apenas do avanço da versão 2.0, sua intenção era deixar Dart uma linguagem totalmente segura de possíveis erros com referências nulas.

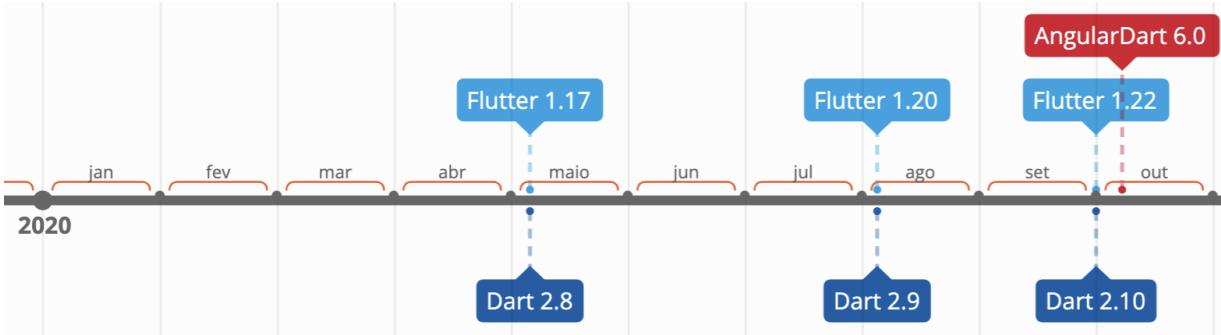


Figura 1.8: Timeline 2020

Chegando em 2020, a frequência de *releases* no ano diminuiu, mas tivemos algumas novidades. Para o Dart, ambas as versões 2.8 e 2.9 tiveram o lançamento focado em algumas mudanças que preparam as APIs da linguagem para a *feature* de *null safety*, que ainda estava em *preview*. Além disso, na 2.8 podemos destacar um novo comando para o `pub` avaliar as dependências das aplicações que estão defasadas. Enquanto na versão 2.9 foi introduzido um novo tipo `Never`, para ser declarado como tipo de retorno de funções que não possuem retorno (encerram o programa ou sempre lançam exceção).

Em outubro, com a 2.10 o foco do Dart foi trazer informações sobre a migração dos códigos existentes para a versão do SDK com null safety além de uma novidade para as ferramentas de linha de comando. Como veremos ao longo do livro, existem várias tarefas que conseguimos realizar com o SDK do Dart, e essas tarefas são separadas em diferentes ferramentas e comandos, como o `dart analyzer` para analisar o código ou `dartdoc` para gerar uma documentação. Nessa versão, iniciou-se a unificação dessas ferramentas de linha de comando, onde todas se tornarão acessíveis através do comando principal de acesso a VM: `dart`.



Figura 1.9: Timeline 2021

O ano de 2021 continuou seguindo o padrão de uma versão de Dart ser lançada em conjunto com uma versão de Flutter. E a atualização mais importante com certeza chegou logo em março, onde Flutter atingiu um novo *major release* devido as grandes mudanças originadas de Dart 2.12, que passou para *stable* as *features* de *null safety* e `dart:ffi`, representando com certeza um marco na evolução da linguagem que ao longo do livro entenderemos o porquê. As demais versões do ano trouxeram algumas melhorias e funcionalidades que podemos destacar:

- **Dart 2.13:** a nova funcionalidade de *type aliases* que permite adicionar apelidos a tipos existentes na linguagem.
- **Dart 2.14:** o suporte do SDK para execução nativa no novo chip da Apple e um novo operador *tripple shift* `>>>` para operações binárias.
- **Dart 2.15:** melhorias de performance na API de *isolates* e o suporte para utilizar *tear-offs* com construtores, que é basicamente poder utilizar construtores como ponteiros em chamadas de funções, veremos tudo isso ao longo do livro.

Por fim, 2022 começou com um release mais tímido da linguagem, a versão 2.16 veio sem muitas mudanças significativas e sim algumas melhorias gerais no ecossistema da linguagem, como a unificação dos comandos do SDK que teve a adição do `doc` e melhorias de UX em <http://pub.dev>.

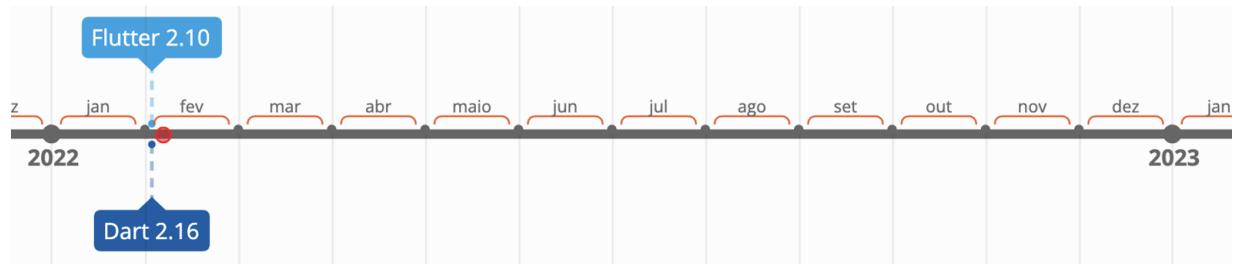


Figura 1.10: Timeline 2022

Uffa... com certeza foram muitos acontecimentos! Mas com relativamente poucas linhas conseguimos resumir aproximadamente 11 anos de história que ajudaram o ecossistema de Dart se tornar o que vamos estudar minuciosamente neste livro. E este não é o ponto final, afinal a linguagem,

assim como qualquer coisa no nosso universo de programação, está em constante evolução e novas histórias serão escritas.

E como agora já possuímos todo esse conhecimento histórico, vamos entender por que devemos, de fato aprender Dart.

1.2 Mas por que Dart?

Essa pergunta é realmente inevitável quando ouvimos falar de algo novo. "Por quê?". É natural, é o que nos faz seres pensantes, estamos a todo momento avaliando as coisas ao nosso redor. E na tecnologia não poderia ser diferente: coisas novas vêm e vão com muita frequência. Porém, a verdade é que estar disposto a aprender uma linguagem nova já faz de você um programador ou uma programadora diferenciada, pois nesse nosso mundo não existe um jogador (linguagem) perfeito para cada posição (projeto). Cabe a nós, treinadores, escalarmos a melhor linguagem em cada projeto.

Então em vez de perguntar "Por que eu perderia tempo para aprender isso?", pergunte-se "Por que eu investiria mais tempo para aprender isso?", afinal, na pior das hipóteses, os seus conhecimentos em programação no geral ficarão ainda mais refinados. É conhecimento comum que, em quanto mais linguagens distintas você já programou, mais rápido você absorve o conhecimento de uma nova linguagem, pois elas compartilham muitas características e conceitos. Além de que o seu senso crítico computacional também é alavancado. Mas Dart vai muito além de uma hipótese, Dart hoje já é uma certeza, e nós vamos entender agora alguns pontos que farão você olhar para ela com olhos mais atentos.

Como Dart nasceu com um foco inicial na web, uma das principais razões de sua criação era a possibilidade de compilar fielmente para JavaScript, com melhor experiência de desenvolvimento, e embora isso realmente tenha acontecido, o estouro do JavaScript no mercado ofuscou por muito tempo qualquer outra tecnologia que se propusesse a fazer algo semelhante. E por mais que Dart fosse muito usada pelos times internos do Google na

construção de aplicações gigantes em usuários, o mercado em geral nunca pareceu dar a devida atenção para ela. Até que as coisas mudaram, quando entrou em cena o Flutter.

Falar de Dart sem citar Flutter ou falar de Flutter sem acabar citando Dart é praticamente inevitável. E é um fato que Flutter tem acendido os holofotes para a linguagem. A comunidade vem crescendo absurdamente, pois os desenvolvedores começaram a se dar conta de quão fácil, prático, prazeroso e produtivo é codificar em Dart, e consequentemente em Flutter.

As portas se abrem ainda mais com a possibilidade de trabalhar em multiplataformas, com aplicações server-side, web e mobile, tudo isso com uma performance nativa e excelente. Sem contar os indicativos de avanços do próprio SDK do Flutter para a web, desktop e no até então misterioso novo sistema operacional Fuchsia, que muitos dizem ser um possível substituto para o Android e que também utiliza Dart em seu core.

Desde sua criação, Dart nunca figurou como uma das linguagens mais populares e conhecidas dentre os rankings que geralmente encontramos pela internet. Mas o cenário quanto a popularidade tem mudado nos últimos tempos e Dart apresenta uma grande crescente. Para ter uma ideia, o GitHub em sua análise anual de *trendings*, conhecida como octoverse, divulgou que Dart foi a linguagem com maior crescimento em contribuidores entre 2018/2019, enquanto Flutter figurou como o terceiro repositório com mais contribuidores da plataforma. Também em 2019, na tradicional pesquisa feita pelo StackOverflow, Dart e Flutter apareceram entre as linguagens e frameworks mais amados, respectivamente.

Esses avanços do Flutter consequentemente fazem com que todo o ecossistema de Dart evolua junto, pois, como já vimos na sua história, a partir da versão 2.0 Dart se tornou realmente uma linguagem otimizada para client-side. E com o melhor dos dois mundos, Dart permite a compilação:

- **JIT (Just in Time)**: de forma inteligente, com a aplicação rodando, consegue identificar quais partes do código você alterou e recompilar sem que seja necessário um *rebuild* completo, perdendo o estado atual da aplicação. Com isso, garante o feedback das novas alterações quase

instantaneamente (em questão de milissegundos), elevando a produtividade e diminuindo o ciclo de desenvolvimento. Algo que o Flutter aproveita muito com o seu *hot reload*.

- **AOT (Ahead of Time)**: usado eficientemente por linguagens com tipagem estática, nas quais a compilação é feita anteriormente ao deploy e execução. Gera os binários que vão rodar de forma totalmente otimizada na VM ou Browser, sem a necessidade de aplicar validações de tipos em tempo de execução, garantindo uma ótima performance.

Dart já nasceu *open source* e abraça até hoje este mundo. Qualquer pessoa tem total liberdade para avaliar o comportamento interno de qualquer ferramenta do SDK, e até mesmo criar PRs (Pull Requests) para alterar ou corrigir um eventual problema. É diferente de algumas linguagens ou tecnologias proprietárias, em que existe todo um mistério quanto ao seu funcionamento interno.

O próprio repositório público de dependências localizado em <http://pub.dev> conta com mais de 10000 *packages* criados pela comunidade, que podem facilmente ser utilizados nas suas aplicações através do gerenciador de dependências `pub`. Isso, sem contar as *libraries* oficiais que já vêm com o próprio SDK, como as APIs de *streams* e programação assíncrona, que são excelentes e de fácil utilização.

Ecossistema

E ainda existe todo o *tooling* da linguagem, que fornece tudo o que você precisar: analisador de código para reportar erros ou avisos, formatador de código para manter a convenção e padrões da linguagem, geradores de códigos, gerador de documentação, plugins para as principais IDEs, um editor de códigos on-line sem instalação, um observatório para debug e análise de métricas da aplicação, gerenciador de dependências, compiladores para diferentes plataformas, entre outras coisas. Tudo isso para que você só tenha que se preocupar de fato em desenvolver suas aplicações.

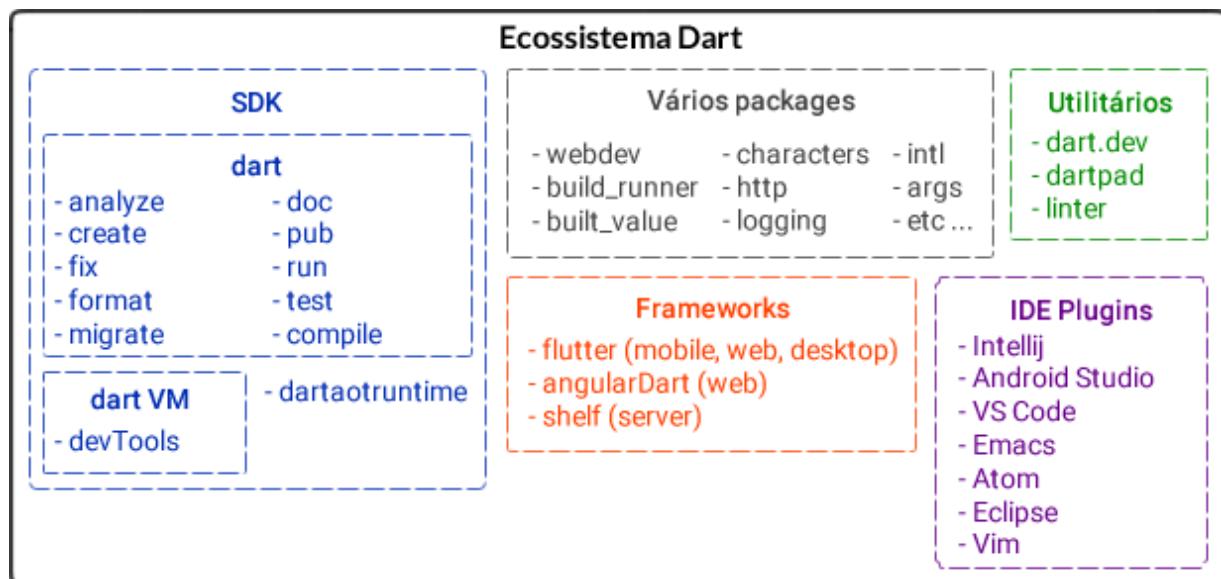


Figura 1.11: Ecossistema.

Afinal, quando instalamos o SDK na máquina, ele já vem acompanhado de algumas ferramentas:

- **Dart VM (Virtual Machine)**: a máquina virtual da linguagem, responsável pela execução do código. Possui internamente o *DevTools*, uma ferramenta para análise de performance e *debug* das aplicações.
- **dartaotruntime**: ferramenta para execução dos *snapshots AOT* gerador pelo `dart2native`.
- **dart**: ferramenta que unificou os comandos do SDK.

A partir da versão 2.10 da linguagem, conforme discutimos em sua história, foi dado início a uma unificação dos comandos do SDK, através do comando `dart`. Por isso, atualmente ele inclui:

- **analyze**: substituiu o antigo *dartanalyzer*, ferramenta de análise estática de código.
- **create**: substituiu o antigo *stagehand*, para criação de novos projetos através de templates.
- **fix**: substituiu o antigo *dartfix*, aplica correções automatizadas ao código.
- **format**: substituiu o antigo *dartfmt*, para formatação do código.

- **migrate**: ferramenta que auxilia na migração de código para novas versões do SDK.
- **doc**: substitui o antigo *dartdoc*, para gerar a documentação do código.
- **pub**: o gerenciador de dependências de Dart.
- **run**: comando para dar início a execução das aplicações.
- **test**: executa os testes existentes na aplicação.
- **compile**: ferramenta para compilar a aplicação em diferentes formatos, substitui os antigos compiladores *dart2js* e *dart2native*.

Você utilizará todas essas ferramentas direta ou indiretamente (através de uma IDE). É válido lembrar que, se você utiliza o Flutter, sua instalação já vem acompanhada do SDK de Dart embutido, não necessitando da instalação do SDK de Dart separadamente. Mas como o nosso foco aqui é o Dart, é recomendável a utilização e instalação de seu SDK independente.

Facilidade de aprendizado

E tem mais, sabe aquele sentimento de familiaridade com algo? Pois é, ao começar a aprender Dart, você pode se dar conta de que já conhece Dart! O intuito da equipe na criação da linguagem sempre foi a adoção em massa, ou ao menos a fácil adaptação tanto para quem vem do mundo das linguagens de script quanto para quem vem de linguagens estruturadas. E hoje em dia quem possui um conhecimento prévio principalmente em alguma linguagem com Orientação a Objetos, similares a Java ou C#, por exemplo, e quer aprender Dart, notará que a curva de aprendizado será quase linear, bastando aperfeiçoar alguns conceitos, que tudo estará tranquilo, pois a sintaxe já é muito familiar. Enquanto em um comparativo superficial a curva de aprendizagem para JavaScript seria muito mais extensa, apresentando um maior período para adaptação.

Eu sou programador Java há 20 anos e quero aprender..

Curva de aprendizado - conhecimento(y) * tempo(x)



Figura 1.12: Curva de aprendizado.

A própria documentação oficial de Dart, encontrada em <http://dart.dev/guides>, é considerada bastante completa e de fácil entendimento, abordando todos os aspectos da linguagem, com exemplos que com certeza também auxiliam no aprendizado.

Enfim, Dart é uma linguagem orientada a objetos com o benefício de ser tipada, possui conceitos de linguagem funcional em conjunto com um belo suporte a programação reativa! Vai ficar de fora dessa? *Come to the Dart side!*

1.3 Executando o código

Todos os trechos de códigos utilizados nos exemplos deste livro estão organizados por capítulo no repositório do GitHub (<https://github.com/JHBitencourt/dart-book>) criado especialmente para isso (deixa a famosa estrelinha lá para sabermos que você é leitor ou leitora). Eles podem facilmente ser copiados para execução, e embora eu encoraje você a utilizá-los, obviamente, não fique apenas no copia e cola. Estude o código, leia, entenda o que ele faz (eu vou ajudar nessa parte) e, por último, reescreva - isso é muito importante, mesmo. Quando você está de fato

digitando o código, seu cérebro com certeza absorve mais as informações, assim como se acostuma mais rápido com a sintaxe da linguagem. E não esqueça de fazer os seus próprios experimentos. Não tenha medo de erros, afinal eles são bons quando ocorrem só durante o desenvolvimento.

Dartpad

A maneira mais rápida e fácil de executar ou testar um código em Dart é utilizando o próprio DartPad, um editor web da linguagem, acessado apenas através do navegador em <http://dartpad.dev>.

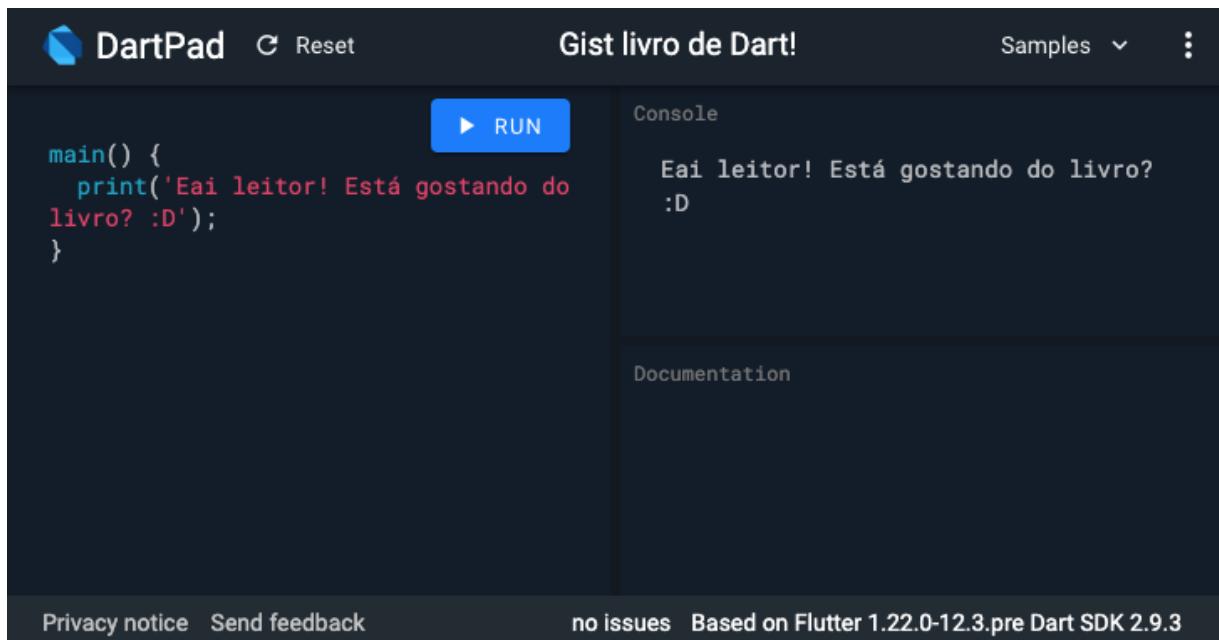


Figura 1.13: Editor DartPad.

Ele possui suporte para todas as *libraries* que acompanham o SDK de Dart, com exceção de `dart:io`, que não funciona na web. Inclusive foi implementado recentemente a possibilidade de importar alguns *packages* que executam diretamente no DartPad.

A partir da versão 1.12 do Flutter, com o Flutter Web entrando em *preview*, também foi habilitada a execução de exemplos em Flutter no próprio DartPad, tornando-o uma excelente alternativa para testes rápidos.

É possível ainda utilizar Gists do GitHub adicionando o id do Gist na URL `dartpad.dev/{idGist}`. Acesse, por exemplo:

<https://dartpad.dev/684ad5f8fab9894c89bb2767a2416f30>

Utilizando IDE

A segunda alternativa para a execução dos códigos em Dart é utilizando uma IDE (Ambiente de Desenvolvimento Integrado). É a forma mais profissional e que geralmente você utilizará no dia a dia para desenvolver as aplicações, pois ela aumenta muito a produtividade e facilita a integração com as funcionalidades do SDK.

Para isso ser possível, é necessário realizar a instalação do SDK de Dart na sua máquina. Por ser uma tarefa que pode sofrer pequenas mudanças com o tempo, dependendo do sistema operacional que você está utilizando, essa é uma tarefa que não será demonstrada aqui no livro. Mas é algo bastante simples e com um passo a passo definido em <https://dart.dev/get-dart>, onde é constantemente atualizado. Caso você ainda tenha alguma dificuldade ou dúvida nesse processo, acesse o mesmo repositório do GitHub do livro e cadastre uma *issue* com o seu problema.

Após isso basta escolher a sua IDE de preferência. Dart possui plugins que integram com diversas IDEs e editores de texto, entre os mais famosos e utilizados estão o IntelliJ, e caso você vá também trabalhar com Flutter, o Android Studio (baseado no IntelliJ) e o VSCode.

Uma terceira opção é rodar o código com a VM através da linha de comando, a qual eu te explico no próximo tópico.

1.4 Primeiro programa

O primeiro programa a gente nunca esquece, ainda mais se for o primeiro programa na primeira linguagem de programação sendo aprendida. Nada paga a experiência de ver pela primeira vez aquelas letras sendo impressas magicamente no console. E não poderíamos fazer diferente, até para evitar

possíveis maldições, vamos criar nosso programa responsável por printar o famoso `Hello World!` .

Mas vamos fazer diferente nesse primeiro exemplo. Vou apresentar a terceira forma de rodar um script em Dart, via linha de comando. Às vezes, trabalhar via linha de comando pode parecer contraprodutivo, mas faz parte do conhecimento do funcionamento de uma linguagem de programação, até porque o trabalho de uma IDE é apenas facilitar a execução desses comandos através de botões. Ao longo do livro, as execuções através de comandos sempre serão referenciadas, e ficará ao seu critério utilizar o terminal ou alguma IDE. De qualquer forma, independente de como você desejar executar, logicamente é necessária a instalação do SDK de Dart.

Com o SDK configurado, crie um arquivo denominado `main.dart`, os arquivos de código devem possuir a extensão `.dart`, e insira nele:

```
main() {  
  print('Hello World!');  
}
```

Esse trecho de código é muito simples, de fato, mas você já pode notar algumas semelhanças com outras linguagens. Criamos uma função `main()`, que possui um significado especial em Dart, pois é obrigatória e responsável por iniciar a execução de qualquer programa. Dentro dela estamos chamando outra função, `print()`, da *library* `dart:core` da linguagem, que recebe um objeto por parâmetro e simplesmente o imprime no console. No nosso caso, passamos uma `String`.

Agora abra a linha de comando do seu sistema operacional e navegue até o diretório em que está o arquivo criado. Nele, execute o comando `dart main.dart`. E... Lá está! Seu primeiro programa imprimindo `Hello World!` no console!

O que aconteceu foi que através do comando `dart` acessamos a VM e passamos como argumento um arquivo para ser executado. Experimente usar os comandos `dart --help` e `dart --verbose`. O primeiro lista uma ajuda para utilização dos comandos, enquanto o segundo mostra uma lista

completa dos comandos e opções disponíveis. Com o `dart --version`, por exemplo, obtemos algo como o seguinte:

```
Dart SDK version: 2.12.0-259.1.beta (beta) (Fri Jan 29 12:27:46  
2021 +0100) on "macos_x64"
```

O `dart --version` exibe a versão do SDK que estamos usando junto do sistema operacional. Agora altere o nome da função `main` e tente executar o arquivo novamente.

```
semMain() {  
    print('Hello World!');  
}  
  
//> Dart_LoadScriptFromKernel: The binary program does not contain  
'main'.
```

Sem a função `main`, a VM não consegue identificar de onde ela deve iniciar a execução do seu programa.

Antes da unificação dos comandos pelo SDK, utilizar apenas o comando `dart` acompanhado de um arquivo para executar era o padrão. Mas atualmente o recomendado é utilizar o subcomando `run`, então, `dart main.dart` é o equivalente a `dart run main.dart`, dando um maior significado para o que o comando faz.

Pronto, agora que estamos livres da maldição do `Hello World` e ainda aprendemos como executar arquivos em Dart diretamente através da linha de comando, podemos começar a nos aprofundar na linguagem.

Até aqui

Neste capítulo, acompanhamos toda a história de evolução do ecossistema de Dart, desde a sua criação até os dias atuais. Analisamos as motivações de escolha no aprendizado da linguagem, assim como os seus benefícios e crescimento recente impulsionado pelo sucesso de Flutter. Por fim, vimos que o DartPad é uma excelente opção para executar códigos e realizar testes rápidos, além de poder utilizar a VM em conjunto com uma IDE ou linha de comando, onde rodamos o primeiro programa escrito em Dart.

A seguir, iniciaremos os estudos vendo como Dart aborda o básico presente em toda linguagem de programação: os operadores e as estruturas de controle.

CAPÍTULO 2

O básico

A arte de programar e construir aplicações se resume ao ato de definir uma série de comandos que serão traduzidos para binários, de forma que uma máquina entenda e eventualmente execute fielmente este passo a passo de instruções. Essa é a primeira lição aprendida ao começar os estudos de algoritmos, e o papel de uma linguagem de programação de alto nível é justamente abstrair esse código binário e permitir a comunicação com as máquinas através de instruções lógicas que nós, humanos, entendemos facilmente.

Embora as linguagens de programação possam divergir bastante em funcionalidades e sintaxe, elas possuem em comum dois pilares que servem de base para a construção dessas instruções: os operadores e as estruturas de controle. Dart também segue essa convenção natural na utilização desses pilares, então pode ser que você já esteja familiarizado com todos esses conceitos.

Se esse for o caso e você acredita que já tem experiência o suficiente ou deseja realizar uma leitura mais rápida do livro, sinta-se à vontade para pular para o próximo capítulo. E eventualmente volte aqui se desejar relembrar algum comportamento ou opção disponível na linguagem. Com isso em mente, neste capítulo veremos:

- Quais são os operadores disponíveis em Dart.
- Como utilizar as estruturas de controle.

2.1 Operadores

Os operadores auxiliam na construção do que chamamos de *expressions* (expressões), trechos de código que possuem, modificam ou realizam alguma ação em um valor em tempo de execução. Por exemplo, o código a

`= b + c` é considerado uma *expression*, pois acessa as variáveis `b` e `c` somando seus valores com o operador de adição `+` ao mesmo tempo em que atribui o resultado da soma à variável `a` com o operador de atribuição `=`.

Dentre os operadores existentes, é comum separá-los em categorias de acordo com sua utilidade.

Operadores aritméticos

Frequentemente utilizados para realizar operações matemáticas.

Operador	Descrição	Exemplo
<code>+</code>	Adição / Concatenação	<code>40 + 2</code> / <code>'40' + '2'</code>
<code>-</code>	Subtração	<code>50 - 8</code>
<code>-expression</code>	Negação, inverte o sinal da operação	<code>-(-42)</code>
<code>*</code>	Multiplicação	<code>6 * 7</code>
<code>/</code>	Divisão	<code>11 / 2</code>
<code>~/</code>	Divisão com retorno da parte inteira	<code>11 ~/ 2</code>
<code>%</code>	Resto da divisão	<code>11 % 2</code>

Figura 2.1: Operadores aritméticos.

```
void main() {
    print(40 + 2); // > 42
    print('40' + '2'); // > 402
    print(50 - 8); // > 42
    print(-(-42)); // > 42
    print(6 * 7); // > 42
    print(11 / 2); // > 5.5
    print(11 ~/ 2); // > 5
    print(11 % 2); // > 1
}
```

É importante notar que alguns operadores podem ter um comportamento diferente de acordo com o tipo de objeto com o qual eles estão interagindo. O `+` no contexto de números realizará uma soma matemática, enquanto que se usado com *strings* concatenará os valores.

Operadores relacionais e de igualdade

Realizam comparações entre os valores de diferentes objetos.

Operador	Descrição	Exemplo
<code>==</code>	Equalidade	<code>42 == 42</code>
<code>!=</code>	Diferença	<code>42 != 42</code>
<code>> / >=</code>	Maior / Maior ou igual	<code>42 > 42</code> e <code>42 >= 42</code>
<code>< / <=</code>	Menor / Menor ou igual	<code>42 < 42</code> e <code>42 <= 42</code>

Figura 2.2: Operadores relacionais e de igualdade.

```
void main() {
    print(42 == 42); // > true
    print(42 != 42); // > false
    print(42 > 42); // > false
    print(42 >= 42); // > true
    print(42 < 42); // > false
    print(42 <= 42); // > true
}
```

O operador `==` diferente de outras linguagens, como o Java, valida o conteúdo do objeto e não a referência de memória.

Operadores lógicos

Operadores para construção de expressões booleanas.

Operador	Descrição	Exemplo
<code>&&</code>	AND	<code>true && true</code>
<code> </code>	OR	<code>true false</code>
<code>!expression</code>	Inversão valor lógico	<code>!false</code>

Figura 2.3: Operadores lógicos.

```
void main() {
    print(42 == 42 && 42 <= 10); // > false
    print((42 == 42 && 42 <= 10) || 42 != 42); // > false
```

```

    print(!(42 == 42 && 42 <= 10) || 42 != 10); // > true
}

```

Operadores de manipulação de bits

Realizam a manipulação de bits, que deve sempre ser feita com valores inteiros.

Operador	Descrição	Exemplo
&	AND	42 & 27
	OR	42 27
^	XOR	42 ^ 27
~	NOT	~42
<<	Deslocamento de bit para esquerda	42<<1
>>	Deslocamento de bit para direita	42>>1
>>>	Deslocamento de bit para direita sem sinal (unsigned)	-42>>>1

Figura 2.4: Operadores de manipulação de bits.

AND

O modificador `&` representa um operador AND em álgebra booleana. Por exemplo, resolvendo a *expression* `42 & 27` :

$$\begin{array}{r}
 & \text{00101010} & (42) \\
 \& \underline{\text{00011011}} & (27) \\
 & \text{00001010} & (10)
 \end{array}$$

Figura 2.5: Operador and (&).

Como a manipulação é em bits, estamos falando em nível de binário, então o 42 em binário é representado por `00101010`, e o 27 por `00011011`. Dessa forma, cada bit do primeiro número é confrontado com o bit da mesma casa no segundo número aplicando o operador AND. Com este operador, a operação só será 1 caso ambos os bits forem 1, resultando no valor binário `00001010`, que em decimal é 10.

```

void main() {
  var and = 42 & 27;
  print(42.toRadixString(2).padLeft(8, '0')); // > 00101010
  print(27.toRadixString(2).padLeft(8, '0')); // > 00011011
  print(and); // > 10
  print(and.toRadixString(2).padLeft(8, '0')); // > 00001010
}

```

Quando estudamos representações numéricas, aprendemos que um *radix* é a quantidade de dígitos distintos utilizados para representar os valores. O sistema decimal que utilizamos no dia a dia possui um *radix* de 10, pois os valores são representados em dígitos de 0 a 9. Assim como o sistema hexadecimal possui *radix* de 16, consequentemente o sistema binário possui *radix* 2, pois utiliza apenas zeros (0) e uns (1).

Outra coisa que veremos durante todo o livro é que em Dart tudo são objetos, até mesmo os números inteiros. Então o `toRadixString(2)` é um método de `int` que transforma o inteiro em sua representação binária em `String`, enquanto o método `padLeft(8, '0')` de `String` adiciona o caractere `0` na esquerda até que a string complete oito caracteres para padronizar a visualização do binário.

OR

O OR é representado pelo `|`, e nesse caso o resultado da operação será 1 caso um dos dois bits forem 1.

```

void main() {
  var or = 42 | 27;
  print(42.toRadixString(2).padLeft(8, '0')); // > 00101010
  print(27.toRadixString(2).padLeft(8, '0')); // > 00011011
  print(or); // > 59
  print(or.toRadixString(2).padLeft(8, '0')); // > 00111011
}

```

O resultado da operação é o binário `00111011`, cujo decimal é o 59.

XOR

Para o `^`, que é o XOR, o bit de resultado será 1 apenas quando ambos os bits comparados forem diferentes.

```
void main() {
    var xor = 42 ^ 27;
    print(42.toRadixString(2).padLeft(8, '0')); // > 00101010
    print(27.toRadixString(2).padLeft(8, '0')); // > 00011011
    print(xor); // > 49
    print(xor.toRadixString(2).padLeft(8, '0')); // > 00110001
}
```

O resultado da operação é o binário `00110001`, cujo decimal é o `49`.

NOT

O `-` representa o NOT, que basicamente inverte todos os bits do valor binário, incluindo o seu sinal.

```
void main() {
    var not = ~42;
    print(42.toRadixString(2).padLeft(8, '0')); // > 00101010
    print(not); // > -43
    print(not.toRadixString(2)); // > -101011
}
```

O resultado da operação é o binário `-101011`, cujo decimal é o `-43`. O resultado do NOT pode variar caso você teste no DartPad, isso se dá devido à forma com que JavaScript trata os valores numéricos, uma vez que o código em Dart é transpilado para JavaScript ao rodar na Web.

SHIFT direita

O operador `>>` realiza a operação de deslocamento de bits para a direita. Nessa operação, cada bit é deslocado `N` casas para a direita e o resultado é o equivalente a dividir o valor decimal `N` vezes por 2. Por exemplo, para deslocar os bits do valor `42` uma vez para a direita utilizamos a expression `42 >> 1`, e o resultado é o mesmo que `42 / 2`.

```
void main() {
    var shift = 42 >> 1;
```

```
print(42.toRadixString(2).padLeft(8, '0')) // > 00101010
print(shift); // > 21
print(shift.toRadixString(2).padLeft(8, '0')) // > 00010101
}
```

SHIFT esquerda

O funcionamento do `<<` é similar ao `>>`, com a diferença de que os bits são deslocados n casas para a esquerda. Dessa vez, o equivalente a multiplicar o valor decimal n vezes por 2. Por exemplo, para deslocar os bits do valor 42 duas vezes para a esquerda, utilizamos a *expression* `42 << 2`, e o resultado é o mesmo que `42 * 2 * 2`.

```
void main() {  
    var shift = 42 << 2;  
    print(42.toRadixString(2).padLeft(8, '0'));// > 00101010  
    print(shift); // > 168  
    print(shift.toRadixString(2).padLeft(8, '0'));// > 10101000  
}
```

SHIFT direita unsigned

Também faz deslocamento dos bits assim como o `>>`. A diferença é que o `>>` respeita e mantém o sinal do inteiro mesmo após o deslocamento, 42 `>> 1` resulta em 21 , assim como `-42 >> 1` resulta em -21 . O `>>>` entretanto é denominado **unsigned**, pois se comporta diferente para números negativos. Ele desloca o bit contendo o sinal e o substitui por 0 (bit que indica um número par), o que faz a operação sempre resultar em valores positivos.

De onde saíram todos esses uns (1)? Bom, o foco aqui não é explicar a fundo operações binárias, mas uma forma de representar números negativos em binário é chamada de "complemento de 2", onde todos os bits são invertidos e é somado 1 ao resultado. Como um inteiro em Dart usa 64 bits, todos os zeros(0) à esquerda se transformam em uns (1).

Operadores de atribuição

Atribuem o valor de uma *expression* a uma variável.

Operador	Descrição	Exemplo
=	Atribuição	a = 4
+=	Adição e atribuição	a += 5
-=	Subtração e atribuição	a -= 4;
*=	Multiplicação e atribuição	a *= 11
/=	Divisão e atribuição	a /= 5
~/=	Divisão com retorno da parte inteira e atribuição	a ~/= 2
%=	Resto da divisão e atribuição	a %= 2
&=	AND e atribuição	b &= 60
=	OR e atribuição	b = 42
^=	XOR e atribuição	b ^= 42
<<=	Deslocamento de bit para esquerda e atribuição	b <<= 4;
>>=	Deslocamento de bit para direita e atribuição	b >>= 1
>>>=	Deslocamento de bit para direita unsigned e atribuição	b >>>= 1

Figura 2.6: Operadores de atribuição.

Um simples = é o operador mais básico e utilizado para atribuição. As suas variações aritméticas funcionam de forma parecida, mas realizam a operação matemática antes de atribuir o valor.

```
void main() {
  num a = 4;
  print(a); // > 4
  a += 5;
  print(a); // > 9
  a -= 4;
```

```

print(a); // > 5
a %= 2;
print(a); // > 1
a *= 11;
print(a); // > 11
a /= 5;
print(a); // > 2.2
a ~/= 2;
print(a); // > 1

int b = 1;
b <= 4;
print(b); // > 16
b |= 42;
print(b); // > 58
b &= 60;
print(b); // > 56
b ^= 42;
print(b); // > 18
b >= 1;
print(b); // > 9
b >>= 1;
print(b); // > 4
}

```

Uma variável do tipo `num` pode receber tanto um inteiro quanto um *double*. Na *expression* `a += 5`, por exemplo, o operador `+=` soma o valor atual de `b` a 5 e depois atribui o resultado novamente à variável `b`. Os demais funcionam de forma semelhante.

Operadores de incremento e decremento

Adicionam ou diminuem o valor de uma variável numérica, são usados geralmente para controle de índice em *loops*.

Operador	Descrição	Exemplo
<code>++var</code>	Adiciona 1 antes de utilizar a variável	<code>var a = ++b</code>
<code>var++</code>	Adiciona 1 após utilizar a variável	<code>var a = b++</code>
<code>--var</code>	Diminui 1 antes de utilizar a variável	<code>var a = --b;</code>
<code>var--</code>	Diminui 1 após de utilizar a variável	<code>var a = b--</code>

Figura 2.7: Operadores de incremento e decremento.

Se o operador vem antes da variável (`++var` e `--var`), o valor é modificado antes que ela seja utilizada na operação.

```
void main() {
    var a = 0;
    var b = 1 + ++a; // 1 + 1
    print(a); // > 1
    print(b); // > 2

    var c = 0;
    var d = 1 + --c; // 1 + -1
    print(c); // > -1
    print(d); // > 0
}
```

Por outro lado, se o operador vem após a variável (`var++` e `var--`), o valor é modificado após ela ser utilizada na operação.

```
void main() {
    var a = 0;
    var b = 1 + a++; // 1 + 0
    print(a); // > 1
    print(b); // > 1

    var c = 0;
    var d = 1 + c--; // 1 + 0
    print(c); // > -1
    print(d); // > 1
}
```

Operadores de validação de tipos

Utilizados para validação e conversão de tipos em tempo de execução.

Operador	Descrição	Exemplo
as	Conversão de tipo	42 as num
is	Validação de tipo, true se for o tipo	42 is int
is!	Validação de tipo, true se não for o tipo	42 is! String

Figura 2.8: Operadores de validação de tipo.

Em POO (Programação Orientada a Objetos), sabemos que os objetos se relacionam e uma variável pode assumir diferentes tipos em tempo de execução. Então o operador `as` permite que um objeto seja convertido para outro em tempo de execução, um processo que é conhecido como *typecast* (ou *cast* de tipos). Dada uma variável cujo tipo é `num`, podemos transformar o objeto que ela referencia em seu subtipo `int`:

```
void main() {
    num a = 42;
    print((a as int).bitLength); // > 6
}
```

A propriedade `bitLength`, que mostra a quantidade de bits do valor, não existe na classe `num`, está implementada apenas no objeto `int` e, para acessá-la, é necessário informar ao compilador que aquela variável é um inteiro - processo feito com o operador `as`. Não se preocupe, pois veremos muito mais sobre tipos e Orientação a Objetos ao longo do livro.

Entretanto, essa operação é válida apenas se os tipos possuem relação. Tentar forçar o *cast* de um objeto do tipo `double` para `int` com `(42.5 as int)`, por exemplo, fará com que o programa lance uma exceção em tempo de execução.

```
void main() {
    num a = 42.5;
    print((a as int).bitLength);
}
```

```
> Unhandled exception:  
> type 'double' is not a subtype of type 'int' in type cast
```

Afinal, `double` não é um subtipo de `int`. Por conta disso, caso você não tenha certeza de que o objeto é realmente de um determinado tipo, é possível verificar com o operador `is`.

```
void main() {  
    num a = 42.5;  
    if (a is int) {  
        print((a as int).bitLength);  
    }  
}
```

Se a `expression` objeto `is Tipo` resolve em `true`, é garantido que a `expression` objeto `as Tipo` executará sem qualquer problema. E como a validação feita no exemplo anterior resultou em `false`, ela impediu que a conversão fosse feita e a exceção lançada. Por fim, o `is!` é exatamente o contrário de `is`, `is!` valida se um determinado objeto não é de um tipo.

```
void main() {  
    num a = 42.5;  
    if (a is! int) {  
        print('Não é inteiro'); // > Não é inteiro  
    }  
}
```

Como veremos no capítulo sobre *libraries*, o modificador `as` também é utilizado para especificar prefixos de acesso.

Operadores gerais

Demais operadores presentes em Dart.

Operador	Descrição	Exemplo
.	Acesso a membros	<code>42.bitLength</code>
()	Chamada de função	<code>print(42)</code>
..	Operações em cascade	<code>StringBuffer()..write('oi')..write('olá')</code>
...	Operador <i>spread</i>	<code>var alfabeto = [...vogais, ...consoantes]</code>
<i>expressionA ? expressionB : expressionC</i>	Ternário	<code>var valor = b % 2 == 0 ? 'par' : 'ímpar'</code>
[]	Acesso a itens em listas e maps	<code>vogais[1]</code>

Figura 2.9: Operadores gerais.

Já estamos familiarizados com os operadores . e () , pois eles foram utilizados nos exemplos até aqui. O primeiro permite o acesso aos membros (propriedades e métodos) do objeto. Já o segundo é simplesmente um operador que indica a chamada para a execução de uma determinada função ou método.

Cascade: ..

O operador cascade facilita a construção de um código com uma interface mais fluída na linguagem. Ele permite realizar várias chamadas em forma de cascata para a mesma referência de objeto que a iniciou originalmente, como em `StringBuffer` , por exemplo, onde é possível criar uma `String` sob demanda. Em vez disso:

```
final frase = StringBuffer();
frase.write('Operação ');
frase.write('em ');
frase.write('cascade.');
```

Pode-se realizar várias chamadas em cascata para o mesmo objeto:

```
void main() {
    final frase = StringBuffer()
        ..write('Operação ')
        ..write('em ')
        ..write('cascade.');
```

```
    print(frase); // > Operação em cascade.  
}
```

Spread: ...

Uma *syntax sugar* que permite inserir valores de uma lista dentro de outra lista.

```
void main() {  
    final vogais = ['a', 'e', 'i'];  
    final consoantes = ['b', 'c', 'd'];  
    final alfabeto = [...vogais, ...consoantes];  
    print(alfabeto); // > [a, e, i, b, c, d]  
}
```

Ternário

Executa uma operação booleana condicional com três operandos. Em uma *expression* `a ? b : c`, onde `a` é uma expressão booleana, se `a` for `true`, o resultado do ternário será `b`, mas se for `false`, será `c`.

```
void main() {  
    int numero = 42;  
    print(numero % 2 == 0 ? 'par' : 'ímpar'); // > par  
}
```

Modifique o valor da variável para um número ímpar, e o ternário retornará o valor 'ímpar' .

Acesso a itens: []

Seu uso é muito simples, garante o acesso a valores de uma lista através do seu índice ou valores de um *map* através de sua chave.

```
void main() {  
    final map = {  
        'vogais': 'a,e,i,o,u',  
        'consoantes': 'b,c,d,...',  
    };  
    final vogais = ['a', 'e', 'i', 'o', 'u'];  
    print(vogais[0]); // > a
```

```

    print(vogais[4]); // > u
    print(map['vogais']); // > a,e,i,o,u
}

```

Listas e mapas serão melhor explorados mais à frente no livro.

Operadores de nulidade

Esses operadores existem exclusivamente com a finalidade de evitar que a pessoa desenvolvedora faça algo errado e acabe manipulando uma variável com uma referência nula, lançando uma exceção.

Com a chegada de *null-safety* na linguagem, a necessidade de utilização desses operadores diminuiu consideravelmente, uma vez que por padrão as variáveis não aceitam mais referências nulas, mas ainda assim são úteis em alguns casos.

Operador	Descrição	Exemplo
??	Ternário para valores nulos	<code>var a = b ?? 42</code>
?=	Atribuição caso nulo	<code>resposta ??= 42</code>
?.	Acesso a atributos caso não nulo	<code>resposta?.universo</code>
..	Operações em cascade caso não nulo	<code>resposta?..recalcular()</code>
?[]	Acesso a índices caso não nulo	<code>resposta?[42]</code>
expression!	Garantia de aviso ao compilador para objetos que podem ser nulos	<code>possivelNulo!.fazAlgo()</code>

Figura 2.10: Operadores de nulidade.

Ternário nulo: ??

Funciona de forma similar a um operador ternário, pois permite atribuir um valor a uma determinada variável se tal condição for nula. Dada a *expression* `a = b ?? c`, a variável `a` receberá o valor de `b` apenas se este não for nulo, caso contrário, recebe o valor de `c`. Sendo assim, escrever `a = b ?? c` é o equivalente a `a = b != null ? b : c`.

```
void main() {
    int? a = null;
    var resposta = a ?? 42;
    print(resposta); // > 42
}
```

Note a tipagem definida como `int?` com o modificador `?` no final. Em versões mais antigas do SDK, isso não é permitido, mas com as mudanças da linguagem de *null safety*, uma variável só poderá conter uma referência nula se ela for declarada explicitamente para tal, com o modificador `?`. Discutiremos sobre *null safety* no próximo capítulo sobre tipos.

Atribuição nulo: `??=`

Também um operador de atribuição, funciona similarmente ao anterior. Serve como atalho caso o valor seja atribuído na própria variável cuja nulidade deve ser verificada. Dada a *expression* `a ??= b`, a variável `a` receberá o valor de `b` apenas se ela for nula. Sendo assim, escrever `a ??= b` é o equivalente a `a = a ?? b`.

```
void main() {
    int? resposta = null;
    resposta ??= 42;
    print(resposta); // > 42
}
```

Acesso nulo: `?`

Operador que permite acesso a propriedades e métodos de um objeto de forma segura contra referências nulas. Pode facilmente ser encadeado em uma sequência de chamadas `a?.b?.c`. Ao acessar uma referência nula em tempo de execução, a *expression* resulta em `Null`, não lançando um erro.

```
void main() {
    int? resposta = null;
    print(resposta?.bitLength); // > null
}
```

Cascade nulo: ?..

Assim como o operador de acesso `.` possui o seu equivalente `?.` para validação de nulos, o operador em cascade `...` também possui o seu equivalente `?...`.

```
void main() {
    StringBuffer? frase = null;
    frase?..write('Operação ')
        ..write('em ')
        ..write('cascade.');
    print(frase); // > null
}
```

Caso o objeto de origem seja nulo, todas as chamadas em cascata retornarão `null`, evitando uma exceção.

Acesso a itens nulos: ?[]

Valida o acesso a valores de uma lista ou `map` quando a variável pode conter um valor nulo.

```
void main() {
    List<String>? vogais;
    print(vogais?[1]); // > null
}
```

Novamente, utilizamos o `?` ao final de `List<String>` para avisar ao compilador que esta variável pode ser nula, e de fato ela é, pois não inicializamos. Por conta disso, utilizando `?[]` para acessar seus valores evitamos uma possível exceção caso ela esteja nula.

expression!

O `!` ao fim de uma expressão é utilizado para converter uma variável possivelmente nula para o seu valor original, forçando o compilador a ignorar qualquer aviso de inconsistência. Veremos mais sobre ele ao abordar *null-safety*.

```
void main() {
    List<String>? vogais;
    print(vogais![1]);
}

> Unhandled exception:
> Null check operator used on a null value
```

Ao tentar executar `vogais[1]` diretamente, o código não vai compilar e nos avisará de que `vogais` está nula. Porém, utilizando `vogais![1]` estamos falando ao compilador: "Pode compilar que eu garanto que a variável não estará nula". Então o programa é compilado normalmente e, como a variável está nula, uma exceção é lançada.

Precedência de operadores

Ao ver uma expressão matemática `2+4*2` sabemos que o resultado é `10`, pois fomos ensinados que a multiplicação e a divisão possuem prioridade, então devem ser calculadas primeiro. O mesmo acontece com os operadores em Dart, existe uma convenção de prioridade na execução que influencia diretamente o resultado das *expressions*.

Descrição	Associatividade	Operadores
Unário postfix	16	e. e? e++ e-- e1[e2] e()
Unário prefix	15	-e !e ~e ++e --e await e
Multiplicativo	14	→ * / ~/ %
Aditivo	13	→ + -
Shift	12	→ << >> >>>
Bitwise AND	11	→ &
Bitwise XOR	10	→ ^
Bitwise OR	9	→
Relacional	8	< > <= >= as is is!
Equalidade	7	== !=
Lógico AND	6	→ &&
Lógico OR	5	→
Ternário nulo	4	→ ??
Condicional	3	← e1 ? e2 : e3
Cascade	2	→ ...
Atribuição	1	← = *= /= += -= &= ^= <<= >>= >>>= ??= ~/= = %=

Figura 2.11: Precedência de operadores. Tabela retirada da especificação da linguagem.

A ordem de prioridade é definida pelo valor da sua precedência nessa tabela, retirada da especificação da linguagem. Quanto maior a precedência, maior sua prioridade. Para os operadores que possuem a mesma precedência e cuja ordem de execução pode influenciar o resultado final, existe o conceito de associatividade.

A associatividade define em qual ordem as operações serão executadas. Por exemplo, a *expression*:

```
void main() {
    print(1+6/2*3-6); // > 4
}
```

Os operadores `/` e `*` são mais prioritários que `+` e `-`, logo devem executar primeiro. Mas entre a divisão e a multiplicação, qual vem primeiro tendo em vista que ambos têm a mesma ordem de precedência? Como a associatividade deles é da esquerda para direita, primeiro é executada a divisão: $1+3*3=6$. Depois a multiplicação: $1+9=10$. E, por fim, a adição e subtração, resultando em `4`.

2.2 Estruturas de controle

Na vida real, é comum definirmos uma ordem para a execução de atividades que, dependendo de algumas situações externas, podem influenciar no seu fluxo de execução. Ao fazer uma viagem de carro, por exemplo, nossa rota pode sofrer alterações. Enquanto o tráfego estiver livre é possível seguir em frente, se houver algum acidente, pegamos uma nova rua, se o trânsito estiver lento, o tempo estimado de chegada aumenta e assim por diante.

As linguagens de programação funcionam de forma similar. Nós, programadores e programadoras, criamos uma série de instruções que determinam a forma como os algoritmos serão executados caso alguma condição seja atingida. Para isso, são utilizadas as estruturas de controle, que, no geral, estão presentes e possuem muita similaridade entre as diferentes linguagens de programação. A seguir, vamos revisar as estruturas presentes em Dart.

if / else

Talvez seja a estrutura mais básica de controle existente. São os famosos "se" e "se não" em português, pois determinam a execução ou não de trechos de código através de uma *expression booleana*.

```
void main() {  
  var resposta = 20;  
  if (resposta != 42) {  
    print('A resposta está errada'); // > A resposta está errada
```

```

} else {
    print('A resposta está correta');
}
}

```

Embora a variável `resposta` seja um inteiro, a *expression* utilizada no `if` é booleana pois resulta em `true` ou `false`. Caso resulte em `true`, como no exemplo, o bloco de código do `if` é executado. Caso resulte em `false`, o bloco do `else` é executado, sendo que ter ou não um `else` é opcional. E ainda é possível combinar vários:

```

void main() {
    var imc = 25.1;
    if (imc < 18.5) {
        print('Abaixo do peso');
    } else if (imc >= 18.5 && imc < 24.9) {
        print('Peso normal ideal');
    } else if (imc >= 24.9 && imc < 29.9) {
        print('Sobrepeso'); // > Sobrepeso
    } else {
        print('Obesidade');
    }
}

```

Cada *expression* é validada na ordem natural em que são criadas, a primeira que resultar em `true` terá o seu trecho de código executado, e as demais serão ignoradas.

switch / case

Uma estrutura de controle que compara a igualdade de uma variável para a execução do código. Cada `case` utiliza o operador `==` para definir se ele será executado. Não há um limite de `case` e ainda é permitido definir um `default`, que é executado caso nenhum `case` execute.

```

void main() {
    var estacao = 'Verão';
    switch (estacao) {
        case 'Outono':
        case 'Verão':

```

```

        print('Tá calor'); // > Tá calor
        break;
    case 'Inverno':
        print('Tá frio');
        break;
    default:
        print('Tá bom..');
    }
}

```

Uma cláusula `case` vazia, como o `case 'Outono'`, funciona apenas como *fall through*, passando a execução para a próxima cláusula. Todo `case` que não está vazio deve terminar com algum modificador de encerramento. O `break` é um deles, que simplesmente encerra a execução do `switch`, mas existem outros:

```

void main() {
    var dia = 'Domingo';
    switch (dia) {
        segunda:
        case 'Segunda':
            print('Aff, já é segunda..');
            break;
        case 'Terça':
            print('Usando um feitiço do tempo..');
            continue sexta;
        case 'Quarta':
        case 'Quinta':
            throw 'Meio da semana';
        sexta:
        case 'Sexta':
            print('Sexxxxtou!');
            break;
        case 'Sábado':
            return;
        case 'Domingo':
            print('Aproveitando enquanto dá..');
            continue segunda;
        default:
            print('Esse dia não existe..');
    }
}

```

```
    }
}

> Aproveitando enquanto dá..
> Aff, já é segunda..
```

Esse exemplo já possui diversas situações. Altere os valores da variável dia para ver as diferentes saídas e perceba:

- No 'Sábado' , o case é finalizado com um return . Nesse caso, além de encerrar o switch , o fluxo da função atual também é encerrada.
- Na 'Quinta' é utilizado um throw , que também encerra o switch , só que lança uma exceção. Veremos mais sobre ele no capítulo sobre tratamento de erros.
- É possível definir apelidos para cada case , como feito para os cases de 'Segunda' e 'Sexta' . Esses apelidos são utilizados em conjunto com o continue , que, além de encerrar o case atual, continua a execução no case definido pelo apelido, funcionando como um redirecionamento.

No geral o switch é uma estrutura bastante utilizada com constantes ou enums . Mais exemplos serão abordados ao longo do livro.

while

Dando início às estruturas de *loop*, o while consiste em um bloco de código que se repetirá enquanto uma determinada condição for verdadeira.

```
void main() {
    var index = 0;
    while(index < 3) {
        print(index);
        index++;
    }
}
```

Esse código vai imprimir o valor do index a cada iteração do loop, sendo aqui 0 , 1 e 2 . O acréscimo do valor do index dentro do bloco de

código é o que vai determinar o ponto de parada do *loop* ao atingir o valor 3 .

É importante que em algum momento a *expression* resulte em `false` . Caso isso não aconteça, o `while` rodará eternamente, congelando a aplicação.

do while

Possui o mesmo conceito de um `while` normal, a diferença é que a *expression* é validada após a primeira execução do *loop*.

```
void main() {
    do {
        print('Executado'); // > Executado
    } while (false);
}
```

Por mais que a *expression* seja `false` , o bloco de código é executado pelo menos uma vez.

for

Com certeza o mais famoso entre as estruturas de repetição, `for` permite iterar sob um trecho de código N vezes. Repare na sua declaração, que é dividida em três partes:

```
void main() {
    for(var i = 0; i <= 10; i = i+2) {
        print(i); // > 0 2 4 6 8 10
    }
}
```

A primeira, `var i = 0` , é a inicialização de uma variável numérica qualquer. Por convenção, é comum nomeá-la com um simples `i` . Na segunda parte, é definida uma *expression* para o critério de parada, que é quando ela resultar em `false` . No exemplo, `i <= 10` indica que o `for` vai parar de iterar assim que o `i` deixar de ser menor ou igual a `10` . E no final, é especificado o critério de atualização da variável a cada *loop* .

Geralmente, é utilizado um simples `i++`, mas no exemplo a variável é acrescida de 2 em 2, fazendo com que o `for` itere seis vezes.

Ainda é possível utilizar dois comandos que podem influenciar nas iterações: `break` e `continue`. O primeiro simplesmente encerra a execução do *loop*, enquanto o segundo permite pular para a próxima iteração. Olhe só na prática:

```
void main() {
    for (var i = 0; i <= 10; i++) {
        if (i % 2 == 0) continue;
        if (i > 7) break;
        print(i); // > 1 3 5 7
    }
}
```

Um `for` com apelido

Você pode acreditar que esta é uma funcionalidade raramente utilizada, mas ainda assim é possível criar estruturas `for` com uma *label* de identificação. Bastante semelhante ao comportamento de um `case` no `switch`, pois também é utilizado em conjunto com o `continue` para redirecionar a execução.

```
void main() {
    forDeFora:
    for (var i = 0; i <= 2; i++) {
        print('forDeFora $i');
        for (var j = 0; j <= 2; j++) {
            if (j >= 1 || i == 1) continue forDeFora;
            print('forDeDentro $j');
        }
    }
}

> forDeFora 0
> forDeDentro 0
> forDeFora 1
> forDeFora 2
> forDeDentro 0
```

O apelido deve sempre estar anterior ao `for`, e repare como o `continue` pula para a próxima iteração do `forDeFora`, encerrando a execução do `for` que está dentro.

for in

Para os objetos que são também `Iterable`, como as listas (veremos a fundo o que é ser *iterable* no capítulo de listas), é possível utilizar um tipo especial de `for` com uma sintaxe um pouco diferente:

```
void main() {
    final vogais = ['a', 'e', 'i', 'o', 'u'];
    for(final vogal in vogais) {
        if(vogal == 'e') continue;
        if(vogal == 'u') break;
        print(vogal); // > a i o
    }
}
```

Ele simplesmente itera sob todos os valores da lista, um a um, seguindo sua ordem, sem a necessidade de definir uma variável numérica para ir acrescentando. O que é na verdade a principal diferença em relação a um `for` normal, nessa versão você não tem acesso ao índice do objeto na lista. Caso essa seja uma informação necessária, a saída é voltar a fazer um `for(var i = 0; i < vogais.length; i++)`.

assert

Durante o desenvolvimento também é possível utilizar um `assert` para quebrar o fluxo de execução do programa. Vamos entendê-lo melhor no capítulo de tratamento de erros, mas, basicamente, ele interrompe a execução do programa quando uma determinada *expression* é falsa.

```
void main() {
    final vogais = ['a', 'e', 'i', 'o', 'u', 'j'];
    assert(vogais.length == 5, 'Só deveriam existir 5 vogais!');
}
```

```
> Unhandled exception:  
> '....bin/main.dart': Failed assertion: line 3 pos 10:  
>   'vogais.length == 5': Só deveriam existir 5 vogais!
```

É comum ver *asserts* espalhados em algumas *libs* dizendo que naquele ponto do código determinada condição deve ser atendida para garantir o seu funcionamento. No exemplo, o código só rodaria normalmente se houvesse apenas cinco vogais na lista.

E quando falamos que é durante o desenvolvimento do programa, significa que, por padrão, quando o código está pronto para produção, os *asserts* são ignorados. Por isso, ao executar através do comando `run`, é necessário habilitar a validação dos *asserts*: `dart run --enable-asserts main.dart`.

2.3 Se liga aí

- Os operadores em Dart nada mais são do que simples métodos de uma classe. Com isso, nós podemos sobrescrevê-los e dar a eles o comportamento que quisermos. Já pensou em um operador `+` que, ao invés de somar, subtrai o valor? Então, é possível! Atente-se, pois veremos sobre isso no capítulo de orientação a objetos.

2.4 É com você

1 - Para praticar as estruturas de controle e operadores, crie um programa que calcule o IMC (Índice de Massa Corporal) a partir da altura (em metros), o peso atual (em quilogramas) e a idade de uma pessoa, considerando que existem diferentes tabelas de cálculo para crianças/adolescentes, adultos e idosos.

Até aqui

Este capítulo foi criado com o intuito de servir realmente como uma breve introdução ao funcionamento dos operadores e estruturas de controle em

Dart para você que está tendo a primeira experiência com a linguagem. Ele pode servir até mesmo como uma espécie de guia, caso você se esqueça de alguma sintaxe ou de como algo funciona, sinta-se à vontade para voltar aqui.

A partir de agora, daremos início à jornada de aprofundamento na linguagem. Aperte os cintos, pegue sua toalha e pule para o próximo capítulo.

Benditos tipos

Com certeza você já presenciou pessoalmente ou nas profundezas da internet a espécime não tão rara de ser humano, conhecida como desenvolvedores ou desenvolvedoras argumentando, em defesa de uma linguagem de programação, coisas como: "A minha linguagem é muito melhor que a sua porque possui uma tipagem forte e previne contra erros!", ou ainda: "A boa mesmo é a minha, que tem tipagem dinâmica e eu escrevo menos código verboso!". Essas discussões podem se estender por horas e horas.

Nessas mesmas discussões é muito comum ouvir palavras como "tipagem dinâmica", "estática", "forte", "fraca", "inferência de tipo", "soundness", entre outras, que definem a abordagem de uma determinada linguagem de programação em relação aos seus tipos. E de fato entender isso é uma etapa importante no aprendizado de qualquer linguagem, obviamente que sem discussões, afinal não existe certo ou errado, e sim abordagens diferentes. Por isso, neste capítulo, vamos:

- Entender conceitos atrelados aos sistemas de tipagem;
- Como Dart implementa esses conceitos;
- Aprender os tipos existentes em Dart;
- Entender a tão importante *null-safety*;
- Se aprofundar um pouco mais nas strings;
- Ver as diferentes maneiras de declarar uma variável.

3.1 Afinal, que raios é um tipo?

Toda linguagem de programação possui dois conceitos básicos definidos em sua criação, a **sintaxe** e a **semântica**. Esses conceitos são exatamente os mesmos para as linguagens naturais que nós humanos utilizamos (português, inglês etc.).

A sintaxe está relacionada à gramática da linguagem, qual a sequência correta de caracteres que determina uma sentença válida. Olhe só:

```
var valor = 'true';
while(valor) {
  print('oi');
  valor = 'false';
}
```

O conjunto de caracteres que formam os comandos e a ordem em que esses comandos estão organizados tornam a estrutura desse código sintaticamente correta. A variável está declarada corretamente, as sentenças são encerradas com um ;, o while pode conter uma variável na sua *expression* e um corpo delimitado por {}.

A semântica, entretanto, está diretamente ligada ao significado e ao comportamento dessas sentenças. As regras semânticas de um while são, por exemplo:

- A sua *expression* é validada:
 - Enquanto ela resulta em verdadeiro, o trecho de código em seu corpo é executado. Ao final, sua *expression* é validada novamente.
 - Caso resulte em falso, o trecho de código em seu corpo não é executado.

Tendo isso em mente, o código anterior é semanticamente correto? Bem, a resposta é "não". O compilador de Dart nem o executaria. E o motivo está na presença das palavras 'verdadeiro' e 'falso' nas regras do while, que estão diretamente associadas a um *booleano*.

Então, Dart entende semanticamente que a variável *valor*, que é uma *String*, nunca resultará em um *booleano*, pois esta foi uma regra definida em sua semântica. Note como *String* e *bool* (*booleano*) são tipos que estão

presentes na linguagem.

Com isso, podemos concluir que um tipo serve para dar semântica a uma determinada variável e as *expressions* em que ela está presente. Ao definir que uma variável é do tipo `int`, sabemos que ela vai representar valores numéricos, assim como uma variável do tipo `String` representará uma cadeia de caracteres.

Responda mentalmente qual seria o valor de `a` a seguir:

```
a = b + c;
```

Não tem como saber. A semântica desse trecho de código está diretamente associada ao tipo das variáveis e a como a linguagem as trata. Se `b` e `c` forem valores numéricos, como um `int`, o `a` pode ser o resultado da soma matemática de ambos. Se `b` e `c` forem respectivamente um `int` e uma `String`, em JavaScript, a seria os valores concatenados, enquanto em Dart o código nem compilaria.

E é exatamente por conta desse diferente comportamento entre as linguagens que existem diferentes abordagens quando falamos sobre sistemas de tipagem.

3.2 Sistemas de tipagem

A existência de tipos seria inútil se não houvesse algum mecanismo que certifica que suas regras semânticas sejam aplicadas. Esse é o papel do processo de *type checking*, que tem como principal objetivo fazer com que o programa se torne *type safe*, reduzindo ao máximo erros de tipos que possam ocorrer em sua execução. Existem dois tipos principais de *type checking* que se diferem em relação a quando são executados: *dynamic* (dinâmico) e *static* (estático).

No primeiro, *dynamic*, todas essas validações são feitas em tempo de execução. Assim, os tipos são associados a valores enquanto o programa executa, e não a variáveis. Então é muito comum que ocorram erros inesperados relacionados a tipos em situações não suportadas. Linguagens com essa abordagem são também denominadas de linguagens com tipagem dinâmica.

Já para o *static type system*, todo esse processo é feito antes, em tempo de compilação. O próprio compilador valida e se certifica de que não haverá erros de tipos ou conversão em tempo de execução. Se uma variável é definida como booleana, ela possuirá sempre esse mesmo tipo em toda execução do programa, ou até que ela seja removida da memória. O tipo está associado à variável, não havendo o risco de uma variável apontar para algum outro endereço de memória que contenha um outro tipo como valor.

Uma vez que várias dessas validações não precisam mais serem feitas em tempo de execução, os programas tendem a ser menores e geralmente apresentam um melhor desempenho. Porém, ter controle absoluto sob todas as operações de tipos antes mesmo de um programa executar não é uma tarefa trivial, algumas situações específicas podem demandar que um tipo seja definido apenas em tempo de execução (fazer um *downcast* por exemplo). Por isso, algumas linguagens de tipagem *static* podem também possuir algum comportamento de tipagem *dynamic*.

Coercion

Frequentemente ocorrem situações onde são misturados diferentes tipos em uma mesma operação. Como esse código em JavaScript:

```
var a = "Resposta: " + 42;
console.log(a); // > "Resposta: 42"
```

Embora a operação contenha uma string e um número inteiro, a *engine* de JavaScript implicitamente converte o 42 em string e concatena ambos em uma nova variável. Esse processo de conversão implícita também é chamado de coerção (*coercion*) implícita de tipos.

Dependendo do contexto em que um tipo é usado, ele pode implicitamente ser convertido para outro pela própria linguagem. E é por esse motivo que um dos mantras do JavaScript é: nunca use `==` para comparar igualdade de

variáveis (e sim o operador ===), caso contrário, podem retornar alguns resultados bizarros de coerção.

Por outro lado, essa coerção também pode ser explícita, sendo definido explicitamente na sintaxe da linguagem que o tipo deve ser convertido, seja porque desejamos, ou, no caso do Dart, porque o compilador nos obriga.

```
void main() {  
    var a = 'Resposta: ' + 42.toString();  
    print(a); // > Resposta: 42  
}
```

Esse programa só compila se ambos os tipos forem iguais, então foi necessário converter o 42 explicitamente para String usando o método `toString()`.

weak e strong

É aqui que as coisas começam a ficar um pouco desafiadoras. Muitos costumam associar de forma equivocada as definições de *strong type*(tipagem forte) com *static type* e *weak type*(tipagem fraca) com *dynamic type*, o que está errado, pois nada impede que uma linguagem seja *static* e *weak* ao mesmo tempo, ou *dynamic* e *strong*.

Mas a real é que não existe um consenso geral sobre esses conceitos que definem uma linguagem como tendo uma tipagem *weak* ou *strong*. O mais comum é em relação a quão rigorosa a linguagem é em relação as suas operações com tipos, principalmente se ela permite ou não coerção implícita. Se permitir, estaria mais relacionada a uma tipagem fraca.

3.3 E onde entra Dart?

Em suas versões 1.x, Dart possuía regras mais "frouxas" em relação a como seu sistema de tipagem funcionava. Pode-se dizer que ela permitia realizar operações dinâmicas por padrão, e as validações de tipo ocorriam muito mais em tempo de execução. Já nessa época, existia um modo de execução chamado "strong-mode" que os desenvolvedores poderiam optar por utilizar em seus programas. Este modo habilitava no compilador diversas validações e regras de tipos mais restritas, que tornavam a linguagem mais "segura".

Com o surgimento da versão 2.0, a linguagem passou por uma grande mudança e esse *strong mode* passou a ser o comportamento padrão. Nesse momento, Dart passou a ser declarada como *type safe* (com tipagem estática), combinando uma série de validações em tempo de compilação e execução para manter o *soundness* do programa - que é a garantia de que não ocorrerão erros relacionados à tipagem.

Sua aplicação nunca entrará em um estado errado quanto ao tipo de uma variável. Se você declarar explicitamente que uma variável é do tipo `bool` (booleano), ela sempre aceitará apenas valores desse tipo em tempo de execução. Uma dessas garantias é justamente o *dartanalyzer* (lembra do ecossistema Dart?), ferramenta que avalia seu código-fonte para encontrar possíveis erros de acordo com as especificações da linguagem. Considere o trecho:

```
bool erro = true;  
void main() {  
    erro = 'false';  
}
```

Vamos aplicar a análise de código nele. Claro que na vida real todo esse trabalho de análise já é feito para você automaticamente pelos *plugins* de Dart na sua IDE preferida, mas fazer isso manualmente auxilia muito no aprendizado. Crie um arquivo com o código acima e navegue pelo terminal até o diretório no qual ele se encontra. Agora execute a análise com o comando `dart analyze`:

```
error • A value of type 'String' can't be assigned to a variable of type 'bool' at main.dart:3:16
```

Um erro é apontado pelo analisador, avisando-nos do problema já em tempo de compilação. E a mensagem é muito clara: um tipo `String` não pode ser atribuído a um tipo `bool`. Agora este outro exemplo:

```
bool erro = true;  
void main() {
```

```
    erro as String;  
}
```

Ao executar o analisador, nenhum problema é encontrado! O programa está perfeito para ser executado. Ao rodar, entretanto, deparamo-nos com o erro: type 'bool' is not a subtype of type 'String' in type cast!

Acontece que definimos a variável `erro` estaticamente como booleana e, logo em seguida, com o operador `as`, estamos tentando convertê-la para `String` - pronto, estrago feito! Como já vimos, uma variável em Dart sempre deve manter o seu tipo, garantia mantida também pelos validadores em tempo de execução, que nos previnem de problemas como esse em que o analisador não consegue identificar.

Ter um sistema de tipagem "sound" traz diversos benefícios para Dart. Primeiro que existe uma segurança de que *bugs* relacionados a tipos não ocorrerão. Se o resultado de uma *expression* é uma string, sabemos que sempre resultará em uma string, o que também facilita muito a leitura e entendimento do código-fonte.

Outro ponto é a manutenibilidade. Basta modificar um trecho de código, que o analisador já avisará todos os locais que foram afetados pela mudança, prevenindo problemas. Sem contar que todas essas garantias em tempo de compilação fazem com que várias validações não precisem mais ser feitas em tempo de execução, o que torna o código final compilado AOT (Ahead-of-time) muito menor e mais eficiente.

Inferência

É claro que é muito comum as pessoas associarem uma linguagem de tipagem estática (e todos seus benefícios e restrições causadas pelo *soundness*) com uma linguagem de sintaxe "engessada", com muita verbosidade. Principalmente por experiência com outras linguagens do gênero, como Java, por exemplo, onde é necessário sair declarando os tipos por tudo que é lado.

Mas isso não é necessariamente verdade: até este momento do livro inclusive, já vimos diversos exemplos de códigos onde não definimos nenhum tipo para algumas variáveis. Sendo Dart uma linguagem de tipagem estática, como isso é possível?

A resposta é simples, inferência de tipo! Quando não informamos explicitamente o tipo de uma variável, o compilador tenta inferir ou "adivinar" qual seu tipo baseado no contexto em que a estamos utilizando. Por exemplo:

```
void main() {  
    int continentes = 6;  
    print(continentes.runtimeType); // > int  
  
    var planetas = 8;  
    print(planetas.runtimeType); // > int  
}
```

Enquanto definimos estaticamente que `continentes` é uma variável do tipo `int`, o compilador inferiu que `planetas` também é um inteiro baseado no valor que utilizamos para a inicialização, por mais que o tenhamos declarado com `var`. É claro que esse exemplo é algo simplista, sem qualquer diferença visualmente, mas imagine situações com tipos semelhantes a `Map<String, LinkedHashSet<MinhaClasse>>`.

Inferir um tipo é uma ferramenta inteligente do compilador para deixar a leitura do código mais leve e clara, para casos onde os tipos são óbvios ou indiferentes para o leitor. Isso significa que não devemos usar tipos e deixar o compilador se virar? Não é bem assim, existem situações em que o compilador não possui informações suficientes e pode inferir um tipo indesejado. Por isso, existem as boas práticas.

É importante entender que a inferência está presente em outras áreas da linguagem, como em parâmetros e retorno de funções. Analise este exemplo:

```
void main() {  
    var somaInt = soma(1, 2);  
    var somaString = soma('1', '2');  
    print('$somaInt: ${somaInt.runtimeType}'); // > 3: int
```

```
    print('$somaString: ${somaString.runtimeType}'); // > 12: String
}
soma(a, b) => a + b;
```

Nele não foi definido qual o tipo de retorno da função `soma`, nem seus tipos aceitos por parâmetro, *spoiler alert*: isso não é uma boa prática. E mesmo assim conseguimos somar valores inteiros e concatenar strings utilizando a mesma função. Inclusive, o próprio retorno foi inferido para seus tipos correspondentes, como dá para ver no resultado impresso do programa.

Mas pense comigo, se uma variável deve sempre possuir um tipo, como raios nossos parâmetros foram de `int` para `String` de uma execução da função para a outra? Vamos entender.

Static ou dynamic? O melhor dos dois mundos

Quando o compilador não possui informações suficientes para inferir um tipo em um determinado contexto, ele acaba tipando a variável com um tipo especial chamado *dynamic*. Por conta disso, a função `soma` foi entendida como:

```
dynamic soma(dynamic a, dynamic b) => a + b;
```

Isso significa que em tempo de execução essas variáveis podem aceitar e retornar qualquer tipo. Aí neste momento, você como bom conhecedor de Orientação a Objetos diria que nesta situação bastaria tipar com `Object`, afinal ele é conhecido como pai de todos os tipos (na verdade este 'pai' em Dart seria o `Object?`, já veremos isso).

```
Object soma(Object a, Object b) => a + b;
```

O retorno até funcionaria, porque `int` e `String` também são `Object` e a inferência ocorreria normal. Porém, o *analyzer* não deixará o código compilar, avisando-nos de que o operador `+` não existe no `Object`, o que é verdade. Isso impossibilita que a função funcione para o seu propósito inicial de somar inteiros e strings com o mesmo código.

É exatamente esse o ponto especial do *dynamic*. Embora considerado um tipo, ele é mais conhecido como um mecanismo que desabilita todas as validações de tipos e habilita a própria tipagem *dynamic* para a variável (lembra da tipagem *static* X *dynamic*?). Por isso, o programa passa a aceitar qualquer coisa por parâmetro.

Obviamente, ao utilizar esse mecanismo, estamos abdicando momentaneamente dos benefícios e seguranças que a tipagem *static* nos fornece. Ao executar o exemplo a seguir, é lançado um `NoSuchMethodError`:

```
void main() {
  dynamic objeto = 42;
  objeto.metodoQueNemExiste(); // > NoSuchMethodError
}
```

Embora seja claro para nós que um `int` não vai possuir um método chamado `metodoQueNemExiste()`, o erro só acontece em tempo de execução, afinal desabilitamos as validações de tipo para essa variável ao definir explicitamente que ela é *dynamic*. É o mesmo que falar para o compilador: "Confie em mim, que eu sei o que estou fazendo", e é aí onde mora o perigo.

É por isso que não é muito comum utilizarmos esse recurso. E se necessário, não deve ser utilizado implicitamente (através de inferência como na função `soma`), pois pode gerar dúvidas na leitura do código. Se você deseja ter esse tipo de comportamento, deve tipar a variável explicitamente, assim outros desenvolvedores entenderão que as validações de tipo foram desativadas com consciênciia.

Agora, se você quer apenas fazer com que determinada variável aceite qualquer tipo, sem desabilitar os mecanismos de tipagem, é recomendada a utilização de `Object` ou ainda `Object?`, cujas diferenças discutiremos em breve. Mas antes, vamos esclarecer de uma vez a seguinte pergunta.

3.4 Quais os tipos existentes em Dart?

Assim como a maioria das linguagens, Dart possui suporte a alguns tipos predefinidos que auxiliam na resolução de diferentes problemas. Porém, não há um conceito tão claro sobre tipos primitivos como em Java, por exemplo; aqui, todos os tipos são também objetos. Alguns abordaremos diretamente neste capítulo, enquanto os outros serão abordados durante o livro em capítulos mais pertinentes. São eles:

- Strings e Runes: são as representações para trabalhar na criação de texto e manipulação de caracteres Unicode.
- num, int e double: valores numéricos.
- bool: valor lógico booleano.
- Iterable, List, Set e Map: representações das estruturas de dados da linguagem, veremos mais no capítulo sobre *generics*.
- Symbol: modificadores que são imunes ao processo de minificação.
- Null: representação de valores nulos.

Além desses tipos mais comuns, Dart traz tipos específicos que cumprem alguns papéis especiais na linguagem:

- Object: em Orientação a Objetos, representa a classe pai na hierarquia. É a superclasse de todas as outras classes com exceção do Null.
- dynamic: utilizado para desabilitar a tipagem estática.
- Future e Stream: utilizados para programar de forma assíncrona, veremos mais em seus capítulos.
- void: indica um valor que não deve ser utilizado, tipicamente usado em retorno de funções e métodos. Teremos mais sobre ele no capítulo de funções.
- Never: um tipo especial que indica que uma função nunca encerrará normalmente. Teremos mais sobre ele no capítulo de funções.
- Function: tipo que representa as funções da linguagem. Teremos mais sobre ele no capítulo de funções.

É possível identificar em tempo de execução qual é o tipo de uma determinada variável através da propriedade runtimeType presente em Object e consequentemente em todos os demais tipos existentes.

Os números

Existem dois tipos de números em Dart:

1. **int**: representação de valores inteiros até 64 bits, indo de -2^{63} a 2^{63} , com a diferença de que para a web esse valor não segue essa regra de tamanho, e sim a mesma de JavaScript.
2. **double**: representação de valores em ponto flutuante, ou simplesmente valores quebrados, em até 64 bits.

Ambos os tipos estendem de num, que possui as implementações básicas de um número, assim como os operadores comuns entre ambos, como o +, -, /, *, entre outros. Com algumas exceções, os operadores de bit (binário), por exemplo, são implementados apenas no int, uma vez que não faz sentido estarem presentes no double. Exemplificando na prática, podemos criar números como a seguir.

```
void main() {  
  int a = 5;  
  double b = 8.2;  
  double c = 1;  
  
  num x = (a * b) + c;  
  print('Resultado: \$x'); // > Resultado: 42  
  print('a: \${a.runtimeType}'); // > a: int  
  print('b: \${b.runtimeType}'); // > b: double  
  print('x: \${x.runtimeType}'); // > x: double  
}
```

Repare na propriedade runtimeType sendo acessada para validar o tipo de x, que, embora declarado como num, é um double em tempo de execução. Após a versão 2.1, inteiros podem ser utilizados no contexto de double e são convertidos automaticamente para ele, conforme a variável c. O contrário, entretanto, não é permitido.

Booleanos

Os valores booleanos são apenas os literais `true` e `false`, utilizados através do tipo `bool`. Não existe conversão de valores como `0` e `1` ou as strings '`true`' e '`false`', como já vimos no exemplo do `dartanalyzer`. Na prática:

```
void main() {
    bool verdadeiro = true;
    bool falso = false;
    bool primeiroMaior = 'Esse é muito maior'.length > 'Esse é grande'.length;
    print('Primeiro é maior? ${primeiroMaior && (verdadeiro || falso)}');
    // > Primeiro é maior? true
    print('verdadeiro: ${verdadeiro.runtimeType}');
    // > verdadeiro: bool
}
```

Symbols

Podem ser facilmente associados como strings imunes à minificação, principalmente para o Dart web. Quando compilado para JavaScript, nosso código está sujeito a esse processo de otimização para redução do tamanho do código-fonte, através de remoção de caracteres desnecessários para a execução do programa original. Os *symbols* garantem que um determinado modificador (ou símbolo) se manterá sempre o mesmo.

São criados através do `#` ou do construtor `Symbol()`. É provável que você nunca precise utilizar um *symbol* diretamente, mas é bom saber que ele existe.

```
void main() {
    var mod = #modificador;
    print(#modificador); // > Symbol("modificador")
    print(mod); // > Symbol("modificador")
    print('mod: ${mod.runtimeType}'); // > mod: Symbol
}
```

3.5 Bem-vinda, null safety!

Dentre as várias funcionalidades e características que Dart foi ganhando ao longo dos anos, tiveram duas versões que são consideradas um marco na linguagem (pelo menos até o momento de escrita deste livro). Isso porque trouxeram mudanças radicais para o comportamento da linguagem, principalmente em seu sistema de tipagem.

A primeira delas, como já sabemos, foi o surgimento da versão 2.0 que introduziu o *strong mode* como padrão, contribuindo para as restrições de tipagem *static* e *soundness* da linguagem. A segunda e mais recente foi sua versão 2.12, que trouxe uma versão *stable* da tão aguardada *null safety*.

Por que então esta foi de fato uma grande mudança? Bem, você provavelmente já se deparou com alguma *exception* ao executar o código ocasionada por valores nulos. O famoso `NullPointerException` em Java já tirou o sono de muita gente, ou o `NoSuchMethodError` equivalente em Dart. Antes da *null safety*, por exemplo, código assim era totalmente permitido:

```
void main() {
    int valor;
    print(valor.isEven);
}
```

Bastava rodar e ver o erro estourando no console ao tentar executar `valor.isEven`. Isso porque a nulidade em Dart também é um objeto representado pelo tipo `Null`, e toda variável não inicializada recebia por padrão o valor `null`. Só que a classe `Null` não possui uma propriedade denominada `isEven` (afinal, isso é do `int`), e essa é a razão de estourar um `NoSuchMethodError` para os erros de referências nulas.

Com a *null safety*, este mesmo código nem compila mais. Quando falamos que uma linguagem é *null safe*, significa que ela por padrão não aceita mais valores nulos trafegando pelo seu código-fonte, eliminando pela raiz os problemas causados pelas referências nulas. E Dart fez exatamente isso, agora as variáveis passaram a não aceitar mais valores nulos por padrão, e o compilador se encarregará de realizar diversas validações estaticamente para garantir isso.

Tudo isso introduziu o que chamamos de *sound null safety*. Todo o *soundness* que já existia na linguagem para prevenir problemas de tipagens se expandiu para evitar também problemas de nulidade. Essa "simples", porém grande, mudança acarretou uma série de outras modificações e melhorias na linguagem, a começar pelos novos *nullable types*.

Nullable types

Antes de chegarmos lá, precisamos entender a hierarquia dos tipos presentes em Dart. O tipo `Object`, comum em linguagens orientadas a objeto, era considerado o tipo mais alto na hierarquia, tornando ele um *top type* ou supertipo para todos os outros tipos existentes. Na outra ponta, estava o tipo `Null`, encarado como um *bottom type* ou subtipo de todos os demais.

Era justamente por estar lá embaixo que o valor nulo podia fluir livremente por toda a cadeia de tipos da linguagem, podendo ser associado a qualquer variável de qualquer tipo. Para tornar a linguagem *null safe* foi necessário então modificar essa estrutura hierárquica, removendo o `Null` do *bottom type*.

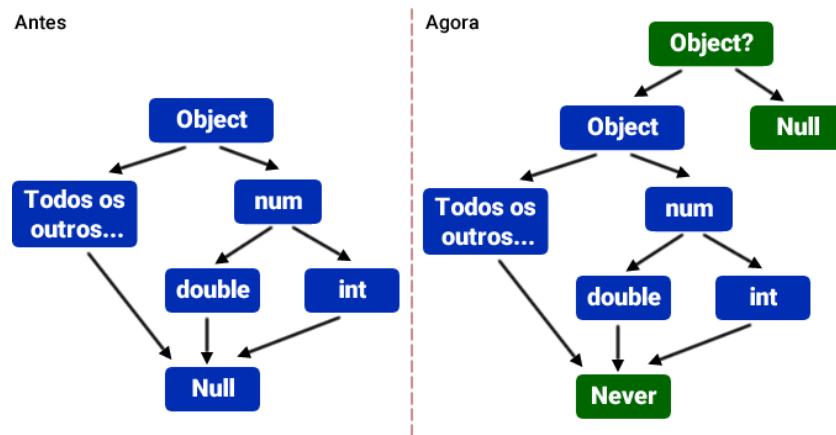


Figura 3.1: Hierarquia de tipos.

Para substituí-lo no papel de *bottom type* foi introduzido o novo tipo `Never`, que ao mesmo tempo em que é um subtipo de todos os outros tipos, ele possui nenhum valor, pois uma expressão com `Never` indica que ela nunca finalizará com sucesso. Veremos como usá-lo no próximo capítulo.

Mas só essa mudança já faz com que todos os tipos existentes deixem de aceitar valores nulos, com exceção do próprio tipo `Null`. Mas o foco não é banir completamente o valor `null` da linguagem, se fosse, era só remover o tipo `Null` por completo. O problema está quando valores nulos aparecem em locais inesperados, causando erro.

A ideia de "ausência de valor" ainda é e vai continuar sendo útil em várias situações. Então ainda havia trabalho a ser feito. E por isso foram introduzidos os *nullable types*!

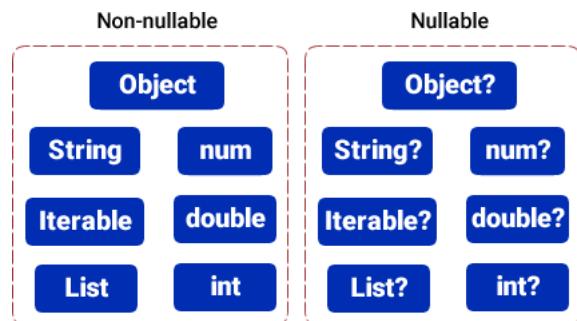


Figura 3.2: Tipos non-nullable e nullable.

Um tipo normal qualquer pode se tornar *nullable* apenas adicionando o sinal de interrogação ? ao seu final. É como se todos os tipos existentes fossem "duplicados", pois além de sua versão normal, contam com uma nova versão *nullable*. Mas o que isso significa na prática afinal?

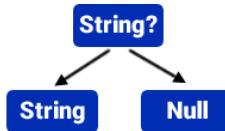


Figura 3.3: String nullable.

Bom, um tipo *nullable* é a união de seu tipo normal com o tipo `Null`. Um tipo `String?`, por exemplo, é a união de `String` e `Null` e aceita valores de ambos os tipos, então também é correto falar que `String?` é um supertipo de `String` e `Null`. E é exatamente por isso que agora o tipo "pai" de toda hierarquia de tipos é o `Object?`.

```

void main() {
    printNome('Douglas', 'Adams');
    printNome('Douglas', null);
}
void printNome(String nome, String? sobrenome) {
    print('$nome $sobrenome');
}
> Douglas Adams
> Douglas null
  
```

Perceba como o parâmetro `sobrenome`, que está tipado com `String?`, aceita ambos os valores.

Um tipo presente na família dos *non-nullable* vai sempre lhe permitir acessar todas as suas propriedades e métodos, mas nunca aceitará um valor nulo, enquanto um tipo da família dos *nullable* pode conter nulos, mas você vai conseguir acessar apenas as propriedades presentes em `Null`, que são seus métodos `toString()` e `hashCode()` e o operador `==`.

```

void valorImpar(int numero) {
    print(numero.isEven);
}
void valorImparNullable(int? numero) {
    print(numero.isEven);
}
  
```

Na primeira função `valorImpar()` do código anterior, o compilador tem a certeza de que nunca haverá um valor nulo sendo recebido por parâmetro, por isso é seguro e permitido acessar a propriedade `isEven` do inteiro. Já a segunda função apresenta um erro ao tentar executar este mesmo código.

Sabemos que `int?` permite receber um null, então deixar essa variável acessar algo inexistente em `Null` quebra a regra principal, que é garantir a segurança contra esses erros. É como se o compilador estivesse falando: "Já que você explicitamente está me informando que quer poder receber nulos com um *nullable*, então eu exijo que você valide essa situação para eu poder compilar o código com segurança!".

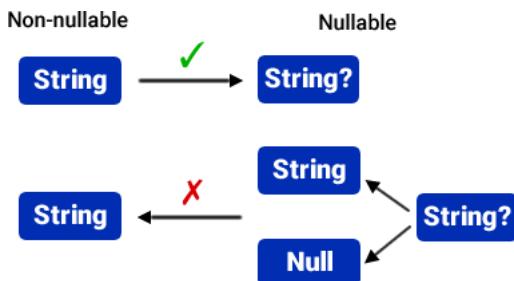


Figura 3.4: Conversão entre tipos nullable e non-nullable.

Mover valores *non-nullable* para o lado *nullable* é seguro, já o contrário não, pois isso permitiria valores nulos vazarem para o lado *non-nullable*. Mas felizmente existem algumas formas de fazer esse caminho inverso, transformando valores *nullable* em *non-nullable* para que eles possam ser utilizados.

Type promotion

No capítulo 2, nós vimos como as estruturas de controle determinam o fluxo de execução do nosso código. E durante o processo de compilação, o compilador executa uma série de análises para tentar otimizar ao máximo a execução, uma dessas análises é conhecida justamente como *control-flow analysis* (análise de controle de fluxo).

Durante esse processo, todos os fluxos de execução do código são mapeados em uma espécie de grafo. E é por causa disso que as IDEs conseguem inclusive indicar trechos de código que nunca serão executados e podem ser removidos. Um outro grande benefício dessa análise é denominado *type promotion*.

Ele acontece quando um compilador identifica que, a partir de um ponto nessa árvore de fluxos previamente mapeada, um determinado tipo pode ser promovido a outro tipo, por exemplo:

```
bool textoGrande(Object objeto) {
    if (objeto is String) {
        return objeto.length > 120;
    } else {
        return false;
    }
}
```

Nessa situação hipotética, o compilador sabe que o trecho dentro do `if` só será executado quando o `objeto` passado na função for de fato uma `String`, então ele promove o mesmo para o tipo `String`. Dessa forma, é possível acessar a propriedade `length`, que só está presente em uma `string`, sem a necessidade de fazer um *cast* explícito (com o operador `as`).

Com a *null-safety*, essa *flow analysis* se tornou ainda mais poderosa e foi estendida para a análise de nulidade dos objetos. Essa é inclusive a maneira mais comum de passar os objetos do lado *nullable* para *non-nullable* e utilizá-los com segurança. Olhe só esta função que separa as letras de um nome:

```
List<String> letrasNome(String nome, String? sobrenome) {
    var letras = nome.split('');
    if (sobrenome != null) {
        letras.addAll(sobrenome.split(''));
    }
    return letras;
}
```

A ideia é exatamente a mesma, só que agora em vez do operador `is` é utilizada a `expression != null`. Como o compilador sabe que para entrar neste bloco de código o `sobrenome` não é nulo, então seu tipo é promovido de *nullable* para o *non-nullable*, permitindo acessar o método `split()` da `string`. Essa mesma função também pode ser escrita como:

```
List<String> letrasNome(String nome, String? sobrenome) {
    var letras = nome.split('');
    if (sobrenome == null) return letras;
    letras.addAll(sobrenome.split(''));
    return letras;
}
```

Afinal, o *flow analysis* é inteligente o suficiente para também entender que, se `sobrenome` é nulo, a função é encerrada e, a partir dessa validação, o tipo também pode ser promovido para *non-nullable*.

O operador bang!

Mas nem tudo são flores com o *flow-analysis*, repare neste exemplo:

```

String? nome;
void main() {
    nome = 'Julio';
    if (nome != null)
        print(nome.length); // Erro de compilação
}

```

Claramente estamos atribuindo um valor para `nome` e também validando com `!= null` antes de utilizar a variável, mas ainda assim o compilador continua não permitindo acessar a propriedade de `String`. Onde está o *type promotion* nessa hora?

Bom, o *flow-analysis* funciona muito bem com variáveis que pertencem ao escopo da função que está sendo analisada, o que não é o caso do exemplo, pois `nome` é uma variável global declarada fora do escopo de `main()`. Esse caso pode parecer trivial, mas controlar o estado de variáveis de toda a aplicação é extremamente complexo e até inviável.

Por isso, o compilador simplesmente assumirá que essas variáveis globais (e as propriedades de classes) poderão sempre conter `null`, forçando-nos a convertê-las explicitamente para o lado *non-null* da força. E um simples operador de *cast* é suficiente para realizar a tarefa.

```
print((nome as String).length);
```

Como esse *casting* acabaria se tornando uma tarefa frequente, a *null-safety* introduziu um novo operador que funciona como um atalho para esta operação. O `!` (exclamação), também conhecido como *bang*.

```
print(nome!.length);
```

Ele simplesmente força a conversão para o tipo *non-null*. É como se avisássemos ao compilador: "Eu sei que essa variável pode ser nula, mas eu garanto que neste momento ela não é, então pode promovê-la para o tipo *non-null*!". E assim é feito, com um único porém (assim como o `as`): se por algum motivo você estiver errado e a variável for nula, o *bang* lançará uma exceção!

Ainda existem outros aspectos e conceitos que vieram junto com a *null-safety* para a linguagem, que veremos adiante no livro. O interessante é entender o quanto importante foi essa melhoria para a linguagem, afinal não é um simples "não vamos mais aceitar variáveis nulas por *default*". Vários aspectos da linguagem evoluíram como resultado dessa mudança.

Com *sound null-safety*, a linguagem se torna mais segura. A não ser que você explicitamente indique que quer um valor nulo e assuma o risco de tratar essas situações, os erros relacionados a referências nulas foram abolidos. O poder fica na mão da pessoa que desenvolve.

Além disso, sabendo que não haverá valores nulos por padrão, o compilador não precisa mais realizar uma série de validações no código final de execução, então ocorre uma otimização no tamanho do código e consequentemente em sua eficiência. Uma evolução incrível!

3.6 Se liga aí

Algumas boas práticas ao trabalhar com os tipos:

- Quando não definido, o tipo de uma variável nem sempre é óbvio, e isso pode deixar o código um pouco confuso tanto na leitura quanto no seu uso, afinal, a tipagem acaba servindo de uma documentação implícita de como seu código deve ser usado (pense em você criando uma nova *lib* para outras pessoas desenvolvedoras). Portanto, procure utilizar tipagem estática para dar um maior sentido nesses casos.

ruim	bom
remover(arquivo, caminho) bool remover(File arquivo, String caminho)	remover(arquivo, caminho) bool remover(File arquivo, String caminho)

Olhando para o primeiro método, o que é esse arquivo que precisamos passar? É o nome de um arquivo? E o caminho é qual objeto? A função retorna algo? Todos esses questionamentos são evitados e facilmente respondidos ao definir os tipos estaticamente na segunda declaração.

- Quando estamos tipando os membros públicos de nossas classes, estamos auxiliando os usuários do nosso código, assim como tipando os membros privados auxiliamos os mantenedores dele. Mas existem os casos óbvios em que não há uma necessidade de tipagem. Não há uma receita de bolo a ser seguida em relação ao que é ou não considerado evidente, depende especialmente do contexto e clareza do código em si. Na dúvida, defina o tipo.

ruim	bom
<pre>const String pathDownload = '../Downloads'; const pathDownload = '../Downloads';</pre>	
<ul style="list-style-type: none">• Para as variáveis dentro de funções, cujos escopos tendem a ser pequenos (não esqueça que funções pequenas são reflexo de um código mais conciso), é interessante não definir o tipo quando estamos inicializando, evitando uma verbose desnecessária e facilitando o entendimento da funcionalidade.	
<pre>// ruim void adicionarCache(String pessoa, Telefone telefone) { Map<String, List<Telefone>> cache = Map<String, List<Telefone>>(); for(MapEntry<String, List<Telefone>> entry in cache.entries) { if(entry.key == pessoa) { cache[pessoa].add(telefone); } } } // bom void adicionarCache(String pessoa, Telefone telefone) { var cache = Map<String, List<Telefone>>(); for(final entry in cache.entries) { if(entry.key == pessoa) { cache[pessoa].add(telefone); } } }</pre>	

- Por outro lado, caso não ocorra uma inicialização imediata onde vamos usufruir da inferência do tipo pelo compilador, o tipo em sua declaração deve ser definido. Com isso, evitamos obter de forma indesejada o `dynamic`.

```
List<Pessoa> promover(Empresa empresa) {
    //var pessoas; // ruim
    List<Pessoa> pessoas; // bom
    if(empresa.lucro > 5000) {
        pessoas = empresa.desenvolvedores;
    } else {
        pessoas = empresa.designers;
    }
    return pessoas;
}
```

- Não utilize declarações de tipos generic redundantes, mas procure sempre definir o tipo estaticamente.

ruim	bom
<pre>List<String> emails = List<String>(); List<String> emails = List();</pre>	
<pre>var emails = List();</pre>	<pre>var emails = List<String>();</pre>

3.7 Strings

As famosas *strings* com certeza estão entre os primeiros recursos aprendidos quando estudamos uma linguagem nova, afinal manipulação de texto é uma tarefa muito comum em programação, já que ainda não lidamos apenas

com máquinas. Então é justo nos aprofundarmos mais neste recurso em Dart. Elas são basicamente um conjunto de códigos *Unicode*:

```
void main() {
  //String('Erro'); // Sem construtor default
  var a = '42'; // Forma literal

  print(a); // > 42
  print(a.codeUnits); // > [52, 50]
  print(String.fromCharCode(52)+String.fromCharCode(50)); // > 42
}
```

Diferente dos objetos que criamos, elas não possuem um construtor *default* definido, então elas são inicializadas com o que chamamos de "forma literal". Com o *codeUnits*, método da classe *String*, obtemos acesso à lista de códigos UTF-16 utilizados, nesse caso o 52 (referente ao 4) e 50 (referente ao 2). Com isso, existe a possibilidade de reconstrução da string original, por exemplo, através do construtor *fromCharCode()*.

As Strings em Dart utilizam UTF-16, isto é, representações em 16 bits (ou 2 bytes), que em Unicode vão de 0000 a FFFF e são definidas no plano 0 (classificação dos caracteres Unicode). Levando isso em consideração, como faríamos então se quiséssemos expressar um símbolo como a saudação do Spock?



Figura 3.5: Saudação Vulcania.

Esse símbolo é representado em *Unicode* como **U+1F596**, não constando dentro da margem hexadecimal do plano 0, e sim presente apenas no plano 1. Para sua representação seria necessário trabalhar com 32-bit, mas já vimos que Dart só trabalha com códigos em 16-bit. É aí que entra o que chamamos de *surrogate pairs* (pares substitutos), onde um código Unicode de 32-bit é quebrado em 2 códigos de 16-bit para a sua representação. Vamos ver um exemplo:

```
void main() {
  var spock = '\u{1f596}';

  print(spock); // Imprime o símbolo
  print(spock.length); // > 2
  print(spock.codeUnits); // > [55357, 56726]
  print(spock.runes); // > (128406)
  print(String.fromCharCode(55357) + String.fromCharCode(56726));
  // Imprime o símbolo
}
```

Primeiro, instanciamos uma *String* com uma sintaxe diferente para a representação do Unicode \u + {hexadecimal}. Então utilizamos o método *length* para verificar o tamanho desta *String* e obtemos 2 como resultado, embora, ao imprimir, o resultado seja apenas um símbolo. Ao investigar um pouco mais e olhar o resultado do *codeUnits*, obtemos 2 códigos, e é isso! Dart utilizou 2 códigos 16-bit (*surrogate pair*) para representar um Unicode de 32-bit!

Já o método *runes* retorna uma lista de objetos *Rune*, sendo essa a implementação que Dart usa para representar um código completo em 32-bit. Por último, é possível construir a *String* a partir dos códigos individuais, recriando o símbolo original.

Outras formas de criar uma *String* são:

```
void main() {
  print("Aspas duplas com 'aspas simples'");
  // > Aspas duplas com 'aspas simples'
  print('Aspas simples com "aspas duplas"');
  // > Aspas simples com "aspas duplas"
  print('''Mais de
  uma linha''');
```

```
    print(r'Unicode = \u{1f596}'); // > Unicode = \u{1f596}
}
```

Além de criá-las com aspas simples ou duplas, podemos usar três aspas para a criação de strings multilinhas ou utilizar o `r` para definir uma string *raw*. Dessa forma tudo o que está dentro das aspas é interpretado literalmente, ignorando os caracteres ou sintaxes especiais.

Interpolação de Strings

As strings são imutáveis, o que significa que cada vez que alteramos uma string estamos na verdade criando uma nova. No decorrer da criação de uma aplicação, com certeza a `String` é uma das classes mais utilizadas e, além de ser muito chato ter que ficar montando frases através da concatenação, podemos adotar a opção de interpolar strings presentes no Dart (e geralmente nas linguagens de script). Isso nada mais é do que utilizar o operador `$` para conseguir acessar variáveis de dentro da própria string, como no código:

```
var resposta = 42;

void main() {
  print('''A resposta para a vida
  o universo
  e tudo mais é: $resposta'''');
  print('42 * 42 = ${resposta * resposta}'); // > 42 * 42 = 1764
  print('Variável: = \'$resposta\'''); // > Variável: = $resposta
  print(r'Variável: = $resposta'); // > Variável: = $resposta
}
```

Estamos acessando a variável `resposta` de dentro da string. Para realizar operações é necessário usar o `$` em conjunto com a *expression* dentro de chaves `${}`. Note que, caso escapemos o operador `$` com o auxílio da barra invertida `\` ou coloquemos *raw* dentro de uma string, o conteúdo é impresso de forma que a ação do operador é ignorada.

Otimização

Embora sempre sejam criadas novas variáveis `String` por elas serem imutáveis, Dart é inteligente o suficiente para reutilizar as variáveis quando elas possuem os mesmos códigos UTF-16 em sequência, por exemplo:

```
var elonUm = 'Elon Musk';
var elonDois = 'Elon Musk';
var jeff = 'Jeff Bezos';

void main() {
  print(elonUm == jeff); // > false
  print(identical(elonUm, jeff)); // > false
  print(elonUm == elonDois); // > true
  print(identical(elonUm, elonDois)); // > true
}
```

O operador `==` realiza a comparação dos valores dos objetos. No caso das strings, essa comparação é feita código por código. Já o `identical()` é uma função do `dart:core` que valida se as variáveis apontam para a mesma referência de memória. Levando isso em consideração, as três primeiras comparações do nosso exemplo funcionam conforme o esperado, porém, ao utilizarmos as variáveis `elonUm` e `elonDois` com o `identical()` obtemos `true` como retorno, indicando que, por mais que sejam duas variáveis diferentes, ambas apontam para o mesmo endereço de memória em vez de criar um endereço diferente para cada uma.

Sob demanda

Podemos criar strings concatenadas em Dart com o operador `+` ou simplesmente com uma string ao lado da outra. Mas será que concatenar é a forma mais performática?

```
void main() {
  print('Dart' + ' é ' + 'incrível!'); // > Dart é incrível!
```

```

print('Dart' ' é ' 'incrível!'); // > Dart é incrível!
var buffer = StringBuffer();
buffer.write('Dart é');
buffer.write(' incrível!');
buffer.writeAll(['E esse', ' livro ', 'também!']);
print(buffer); // > Dart é incrível!E esse livro também!
}

```

Uma melhor alternativa para a criação de strings sob demanda de forma eficiente, quando há a necessidade de concatenar várias, é a utilização do `StringBuffer`, classe que permite unir diversos objetos através dos seus métodos. No nosso exemplo, o objeto `String` só é criado de fato ao utilizarmos o `print()`, que consequentemente chama o `toString()` do `StringBuffer`, que é quando a `String` realmente é criada.

Muito mais

Passamos por alguns pontos principais ao trabalhar com string em Dart, mas ainda existem muitas operações que podem ser realizadas. Validação de conteúdo vazio, adição de *padding* baseado no tamanho da string, métodos para troca de caracteres, divisão da string em partes e validações de *matching* são alguns dos exemplos. Consulte a documentação da classe `String`, lá você encontrará inúmeros outros métodos e funções, então dedique um tempo explorando-os e crie seus próprios exemplos! A seguir, alguns desses exemplos para servir de inspiração:

```

String e = 'Douglas Adams';
void main() {
  print('Vazio: ${e.isEmpty}'); // > Vazio: false
  print("Adicionando à esquerda '>': ${e.padLeft(15, '>')}");
  // > Adicionando à esquerda '>': >>Douglas Adams
  print("Adicionando à direita '<': ${e.padRight(16, '<')}");
  // > Adicionando à direita '<': Douglas Adams<<
  print("Mudar todos 'a' para 'e': ${e.replaceAll('a', 'e')}");
  // > Mudar todos 'a' para 'e': Douglies Adems
  print("Mudar primeiro 'a' para 'e': ${e.replaceFirst('a', 'e')}");
  // > Mudar primeiro 'a' para 'e': Dougles Adams
  print("Dividindo: ${e.split(" ")[0]}"); // > Dividindo: Douglas
  print("Dividindo: ${e.split(" ")[1]}"); // > Dividindo: Adams
  print('''Matching:
    Contém 'gl': ${e.contains("gl")}
    Termina com 'Adams': ${e.endsWith("Adams")}
    Começa com 'D': ${e.startsWith("d".toUpperCase())}'''');
}

```

3.8 Se liga aí

Ao trabalhar com strings, leve em consideração essas boas práticas:

- Dependendo do tamanho da string é necessário quebrá-la em algumas linhas. Ao fazer isso, não utilize o `+` para concatenação. Por mais que estejamos acostumados a utilizá-lo em outras linguagens de programação, dê preferência a simplesmente utilizar strings adjacentes em Dart.

ruim	bom
<code>var a = 'Se' + ' liga ' + 'aí';</code>	<code>var a = 'Se' ' liga ' 'aí';</code>

- Ao construir strings com valores dinâmicos, utilize interpolação em vez de concatenação. É uma *feature* muito boa e permite uma legibilidade maior do código.

ruim	bom
<code>'A resposta é: ' + resposta + '!' A resposta é: \$resposta!'</code>	

- Ao utilizar interpolação de strings, utilize as chaves apenas quando necessário, ou seja, ao realizar alguma operação ou acessar valores e métodos do objeto.

```

ruim           bom
'A resposta é: ${resposta}!' 'A resposta é: $resposta!'

```

3.9 Na prática - Palíndromo

Até aqui só fizemos alguns valores aparecerem na tela, mas agora vamos construir um identificador de palíndromos! Mas você lembra o que são palíndromos? São aquelas palavras ou frases em que a sequência de caracteres é a mesma da esquerda para a direta e da direita para a esquerda, como "reviver", "arara" ou "A sacada da casa". Se você ler de trás para frente, vai ser igual.

Então a ideia é que, ao executar nosso programa, ele peça para inserir uma frase e aguarde o usuário digitar. Após isso, valida se é ou não um palíndromo e mostra o resultado, exatamente assim:

----- Palíndromo ----- > Informe sua frase: A base do teto desaba abasedotetodesaba -> abasedotetodesaba É um palíndromo!! o/ -----	----- Palíndromo ----- > Informe sua frase: Bazinga bazinga -> agnizab Não é um palíndromo!!
--	--

Figura 3.6: Identificador de palíndromos.

Assim como o programa é simples, ao final da construção você verá que a solução também é. Inclusive este é um bom ponto para você parar a leitura, tentar implementar e depois voltar para comparar o resultado.

O primeiro passo quando nos deparamos com um problema é parar e pensar na lógica para a sua solução, antes mesmo de querer codificar algo. Essa é a etapa mais importante. É quando a gente pega um papel e sai rabiscando um fluxo de ações necessárias. Uma possível solução então é:

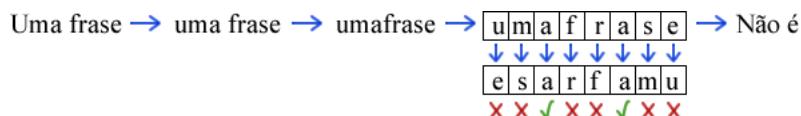


Figura 3.7: Fluxo do programa.

1. Ler uma palavra qualquer de entrada.
2. Transformar todos os caracteres para sua grafia minúscula (ou maiúscula).
3. Fazer um tratamento para remoção dos espaços caso seja uma frase.
4. Inverter as posições dos caracteres.
5. Comparar caractere por caractere. Sem a remoção dos espaços essa comparação após inverter não iria funcionar.
6. Todos são iguais? É um palíndromo! Caso contrário, não é.

Por ser um programa relativamente curto, já vamos ver o código seguido da explicação:

```

import 'dart:io';
void main() {
  print('----- Palíndromo -----');
  print('> Informe sua frase:');
  var frase = stdin.readLineSync()?.replaceAll(' ', '').toLowerCase();
  var fraseContrario = frase?.split('').reversed.join();
  print('\n$frase -> $fraseContrario \n');

  if (frase == fraseContrario) {
    print('É um palíndromo!! o/');
  } else {
    print('Não é um palíndromo!!');
  }
}

```

```
    print('-----');
}
```

Começamos delimitando o início e o fim de nosso programa com a ajuda da função `print()` onde imprimimos um título e as linhas finais. Em seguida, solicitamos que o usuário informe a frase a ser avaliada. A partir daí começam a aparecer coisas com as quais ainda não estamos acostumados, mas são de fácil entendimento. Vamos lá.

O `stdin` (*Standard input*) é um objeto que representa o padrão de entrada de dados da plataforma. Para usá-lo, precisamos importar o package `dart:io`, que é responsável pelas implementações do SDK de *input/output*. Ao chamar o método `readLineSync()`, o programa vai ficar aguardando de forma síncrona a leitura de uma linha digitada pelo usuário.

O retorno desse método será uma `String?` com o conteúdo inserido pelo usuário, e já aproveitamos para chamar nesse mesmo retorno um outro método `replaceAll()` logo na sequência. Com esse método, conseguimos substituir caracteres dentro de nossa string através de um padrão, e é exatamente o que precisamos para remoção de espaços, substituindo esse padrão " " (espaço) por uma string vazia.

Mais uma vez, o retorno é uma nova `String?`. Note que isso é um comportamento frequente nos métodos dessa classe. Aproveitando isso, chamamos na strings de retorno o método `toLowerCase()`, que basicamente transforma todos os caracteres para seus respectivos em forma minúscula. Assim concluímos o tratamento do nosso dado de entrada guardando esse resultado na variável `frase`.

Para que seja possível invertermos as posições dos caracteres na string, é necessário primeiramente quebrá-la em partes, e é com o método `split()` que conseguimos este resultado. Novamente, passando um padrão como parâmetro, no nosso caso uma string vazia ' ', essa quebra é feita em todos os caracteres, retornando uma lista de strings/caracteres. Em seguida, chamamos o método `reversed` presente na classe `List`, que vai inverter a ordem dessa lista, consequentemente invertendo a ordem dos caracteres da nossa string original.

Dessa vez, o retorno desse método é um `Iterable` ou "pai das listas", não se preocupe pois vamos nos aprofundar nisso mais para a frente no livro. O que interessa agora para nós é que esse `Iterable` possui um método `join()`, que vai nos ajudar a unir novamente todos os caracteres em uma única `String` respeitando a ordem. O resultado é salvo na variável `fraseContrario`.

Agora ficou fácil, é só compararmos a frase original com a frase ao contrário. O `==` vai comparar cada código UTF-16 individualmente e, caso sejam iguais, é um palíndromo! Com isso, o primeiro desafio está concluído, mas existe um ponto-chave nessa implementação que ficará para sua análise na seção **É com você** a seguir.

É com você

1 - Em relação ao nosso programa de palíndromos, ele funciona com todas as frases? Há algum problema com frases contendo acentos ou caracteres especiais? Se sim, por quê?

2 - Ainda relacionado ao programa de palíndromos, qual é o possível problema que pode ocorrer ao utilizar métodos como `toUpperCase()` e `toLowerCase()` da classe `String?` Todos os caracteres possuem um equivalente maiúsculo e minúsculo?

3 - Conforme as duas últimas perguntas, deu para ver que manipular strings em algumas situações pode acabar sendo complexo. Um outro problema é quando utilizamos *surrogate pairs* como em emojis. Existe um package oficial nomeado "characters" que auxilia nessa tarefa, pesquise sobre ele.

4 - Criar strings concatenadas através do operador `+` tem muita diferença performática do `StringBuffer`? Construa um programa de benchmarking que avalie o tempo entre os dois métodos com 9999 strings, e depois 99999 strings. Mudou muito? Dica: utilize um loop e a classe `Stopwatch` para a contagem de tempo.

5 - Implemente um programa que faça validação da máscara de um CPF. Através de uma string como "999.999.999-99" por exemplo, informe se é uma máscara válida ou não. Para isso, utilize regex (RegExp) em Dart.

3.10 Variando um pouco

Dart fornece algumas opções para criação de variáveis durante nossa codificação, algumas das quais já utilizamos ao longo dos exemplos. E embora possam ter algum comportamento diferente dependendo da sintaxe, todas possuem a mesma premissa: guardar a referência de um objeto qualquer que está em memória.

A maneira mais simples para criar uma variável é utilizar a palavra reservada `var`, onde a definição do tipo ocorrerá através de inferência. Outra forma é declarar explicitamente o tipo, como `int a = 1;` ou `int? a;` caso exista a necessidade de utilizar valores nulos. Vale sempre lembrar que, se o tipo for *non-nullable*, é obrigatório que a variável seja inicializada.

É perfeitamente possível variáveis distintas apontarem para objetos iguais em memória. Mas conforme aprendemos sobre o sistema de tipagem em Dart, uma vez que seu tipo é definido, é expressamente proibido que seja alterado:

```
var pi = 3.14;
double y = pi;
var x = 'Arquimedes';
String? matematico;

void main() {
  //x = y; // Erro, x já possui o tipo String.
  matematico = x;
  print(matematico); // > Arquimedes
}
```

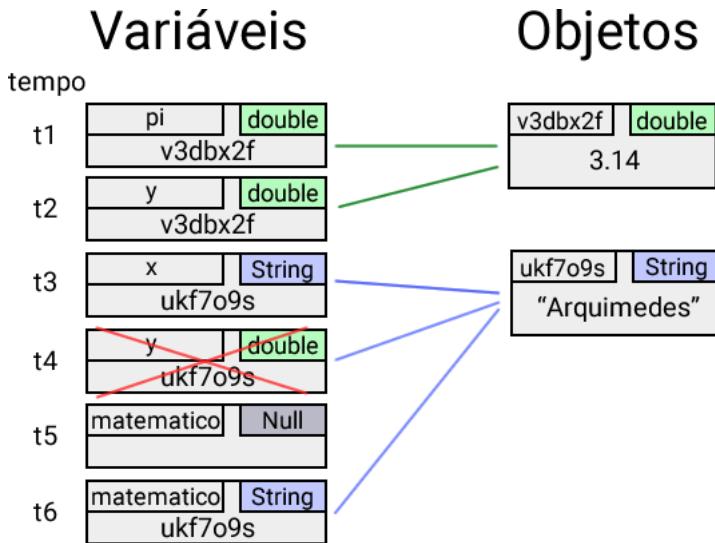


Figura 3.8: Variáveis apontam para referências de memória que contém objetos.

A imagem procura ilustrar a alocação de valores nas variáveis, e o tipo em que elas são declaradas representa um papel importante nesse processo, pois sempre se apontará para referências de objetos que sejam do mesmo tipo. Claro que existem situações especiais como os tipos *nullables*, onde uma variável `String?` aceitará `Null` ou `String` (dois tipos em vez de um), mas nunca um tipo diferente destes.

Seguindo as regras de boas práticas na definição de tipos, também existe a possibilidade de criação de uma variável diretamente com o `dynamic`, como `dynamic pi;`. Neste caso, o desenvolvedor está explicitamente indicando no código que deseja utilizar uma tipagem dinâmica, desabilitando a checagem de tipos.

E o tal do static?

Assim como outras linguagens orientadas a objetos, Dart possui o mesmo conceito de variáveis *static*, que são compartilhadas entre as instâncias de uma mesma classe.

Imagine que precisamos controlar a quantidade de livros que são adicionados ao carrinho de um e-commerce, e cada criação de um novo objeto `Livro()` indica uma adição de livro no carrinho de compras. Isso pode ser facilmente solucionado com a utilização do modificador `static` em um membro de uma classe (para efeitos didáticos, obviamente):

```
//static var global = 'String global'; // Erro
class Livro {
    static intinstancias = 0;
    Livro(){instancias++;}
    String autor = 'Douglas Adams';
}

void main() {
    print('Instâncias: ${Livro.instancias}');
    Livro();
    Livro();
    print('Autor: ${Livro().autor}');
    print('Instâncias: ${Livro.instancias}');
}

> Instâncias: 0
> Autor: Douglas Adams
> Instâncias: 3
```

Por estarem associadas diretamente ao conceito de uma classe, não é possível ter uma variável *static* global, por exemplo, e nem faria sentido. Afinal, com o modificador `static`, estamos definindo que essa determinada variável pertence agora à classe, e não a cada instância do objeto.

Sendo assim, ao criar uma variável `instancias` para contar a quantidade de objetos `Livro` criados, adicionando 1 sempre que o construtor da classe é chamado, mantemos o controle da quantidade criada entre todas as instâncias do objeto. Note que a variável é acessada diretamente através do nome da classe `Livro.instancias`, enquanto o autor que pertence a cada objeto necessita de uma instância da classe para acesso `Livro().autor`.

O `static` deve ser utilizado em conjunto com uma das palavras reservadas para a criação de variáveis, `var`, `dynamic`, `final` ou `const`. Falando em `final` e `const`...

final x const

As variáveis, como o próprio nome indica, podem ter seus valores alterados a qualquer momento, apontando para a referência de um outro objeto em memória. Mas acontece que existem determinados valores que nunca vão mudar, como o caso do `pi` que usamos anteriormente, afinal ele é um valor matemático constante. E é exatamente essa a palavra-chave! Constantes.

Em alguns casos, precisamos de uma variável que, uma vez que tenha seu valor definido, seja proibida de ter qualquer alteração, afinal, não queremos que uma nova pessoa mexa no nosso código e acabe reinventando as leis matemáticas ao alterar esses valores universais.

A utilização de constantes está profundamente relacionada ao conceito de imutabilidade, sendo esse um dos pilares de Programação Funcional. Em Dart elas podem ser criadas através do modificador `final` ou `const`. Ambos possuem esse mesmo propósito, mas com uma abordagem e conceito um pouco diferente na prática.

```
// const double pi = 3.14;
const pi = 3.14;
// final double circunferencia = 2 * pi;
final circunferencia = 2 * pi;
```

A declaração de uma constante pode ou não ser definida junto com o seu tipo, e, uma vez declarada, a variável precisa obrigatoriamente ser inicializada. Para entender a primeira diferença entre as declarações, vamos fazer um exercício básico com o código anterior.

Definimos que `pi` é uma const de valor `3.14`, já a `circunferencia` possui o valor de 2 vezes o `pi`, havendo essa dependência com o valor anterior. A ideia é representar a fórmula de cálculo de circunferência, que em sua forma completa seria `2*pi*r`, como `r` (raio) é um valor variável, foi omitido da fórmula. Ao fazer uma simples alteração, invertendo a declaração delas, o que você acha que acontecerá?

```
final pi = 3.14;
const circunferencia = 2 * pi;
```

Se você apostou no compilador nos avisando de alguma burrada sendo feita e não deixando o código compilar... Acertou na mosca! Acontece que uma const é uma constante criada em tempo de compilação, antes da execução do código. Já a variável `final` é criada durante a execução, na primeira vez em que for utilizada. Por conta disso, toda constante deve ser inicializada com um valor cuja definição seja possível em tempo de compilação, o que não acontece ao depender do `final pi`.

Toda variável `const` é implicitamente `final`, sendo essa uma verdade unilateral, pois o contrário não é verdadeiro. Além de modificar variáveis, `const` também pode modificar valores, como a criação de uma lista de valores constantes:

```
var valoresConstantes = const [1,2,3];
```

E a verdade é que qualquer objeto pode ter um valor constante, que pode ser definido através da criação de um construtor `const`. E esse objeto criado é *canonicalizado*, o que significa que, independente de quantas vezes esse valor constante for chamado no código, sempre vai existir apenas um único objeto na memória.

```
void main() {
    var a = getValorCanonicalizado();
    var b = getValorCanonicalizado();
    print('A e B são os mesmos objetos? ${identical(a, b)}');
    // > 'A e B são os mesmos objetos? true'
}
List getValorCanonicalizado() => const [4, 5, 6];
```

Uma última diferença, mas também muito importante é representada no código a seguir:

```
final listFinal = [1, 2, 3];
const listConst = [1, 2, 3];
void main() {
    listFinal[0] = 4; // é permitido
    listConst[0] = 4; // erro de execução
}
```

Uma lista sendo `final` não permite que essa variável aponte para uma outra lista, porém é possível alterar ou adicionar novos valores nela. Entretanto, ao inicializar um objeto como `const`, todo o objeto em cascata também se torna imutável, então não é possível a alteração desses valores, ocorrendo um erro de execução se houver.

late

Por último, temos o `late`, um modificador relativamente novo pois foi introduzido junto com a *null-safety*. Ocasionalmente nos deparamos com situações similares a este código:

```
class Pizza {
    int pedacos;
    void media() {
        pedacos = 8;
    }
    void grande() {
        pedacos = 16;
    }
    String servir() => '$pedacos pedaços servidos!';
}
```

Nesse exemplo, `pedacos` é uma propriedade de `Pizza` que em algum momento será inicializada através dos métodos `media()` ou `grande()`. Se você se recordar o que discutimos durante o capítulo, vai reparar que `pedacos`

também é *non-nullable*, e por não ser inicializada no momento de sua criação, o código não compila!

O compilador não tem como saber, ou melhor, garantir que esta variável estará inicializada no momento que o método `servir()` for chamado. Poderíamos declarar a variável como `int?` e tudo estaria perfeitamente correto. Mas isso faria com que uma pizza pudesse ficar com a ausência de pedaços, valor nulo, e não é isso o que queremos. Para esses casos, foi introduzido o novo modificador, onde é possível substituir para `late int pedacos;`. Ele serve de aviso para o compilador de que nós garantiremos que antes de utilizar essa variável ela será inicializada, e se não for, uma exceção pode ser lançada. Com ele, o código compila normalmente e pode ser utilizado.

```
void main() {  
    final pizza = Pizza();  
    pizza.media();  
    print(pizza.servir()); // > 8 pedaços servidos!  
}
```

3.11 Se liga aí

Algumas boas práticas ao criar as variáveis:

- Não initialize uma variável *nullable* com o valor `null` explicitamente. Como já sabemos, Dart já possui esse comportamento por padrão, o que acaba criando uma redundância.

ruim	bom
<code>String? x = null; String? x;</code>	

- Nomeie as variáveis no padrão *lowerCamelCase*, incluindo constantes, e não utilize prefixos como os famosos `m` ou `k` (também conhecidos por 'notação húngara').

ruim	bom
<code>var mes_pagamento; const MES_NATAL = 'Dezembro'; final mTempoExecucao = Duration();</code>	<code>var mesPagamento; const mesNatal = 'Dezembro'; final tempoExecucao = Duration();</code>

- Existe a possibilidade de combinar o modificador `late` com o `final`, nesse caso, quando a variável for inicializada, ela não poderá receber outro valor.

3.12 É com você

1 - Construa um programa que receba um valor de raio e calcule a circunferência do círculo com a fórmula `c = 2*pi*r`. Utilize para isso a constante presente em '`dart:math`'.

2- Embora sintaticamente seja permitido pelo compilador declarar uma variável com `final` ou `const` em conjunto do `dynamic`, por exemplo, `final dynamic a = 'dinâmica?'`, por que não faz sentido algum a utilização de `dynamic` neste caso?

Até aqui

Ufa! O caminho foi longo. No decorrer do capítulo, passamos por diversos conceitos essenciais para o entendimento da linguagem, e, com certeza, agora você conta com uma base muito mais sólida de como Dart implementa seu sistema de tipagem.

Entendemos quais são os tipos existentes e todas as mudanças e melhorias que a *null-safety* trouxe para o ecossistema de Dart. Aprofundamo-nos mais nas strings, que é sem dúvida o tipo mais utilizado entre os existentes, entendendo como esses textos são armazenados em conjunto com as *runes*. Implementamos um

identificador de palíndromos para exercitar o que foi aprendido e, por fim, exploramos os diferentes modificadores e tipos para criação de variáveis.

Agora, o mundo das funções é logo ali, no próximo capítulo.

CAPÍTULO 4

Explorando mais as funções e a Web

Quem nunca implementou uma função, ou sub-rotina, ou subprograma, ou procedure, ou *closure*, ou método, ou ... Ahhh! São tantas nomenclaturas diferentes para algo que em sua essência representam os mesmos conceitos. É normal até causar confusão.

Neste capítulo, vamos explorar esses conceitos que, independente do nome, cumprem um papel essencial na construção dos nossos programas e vamos obviamente analisar suas peculiaridades em Dart. Então:

- Entenderemos o que são as funções, métodos e closures;
- Entenderemos a finalidade dos tipos `void` e `Never` ;
- Veremos até onde vai o escopo de uma função;
- Exploraremos `dart:web` em uma aplicação de exemplo;
- Aprenderemos os diferentes tipos de parâmetros em uma função;
- Conheceremos os *enums* e *typedef*.

4.1 Funções! Métodos! Closures?

As funções, embora sejam conhecidas por diversos termos, estão sempre presentes e são um dos conceitos mais utilizados em programação. Mas é obrigatório criar funções em Dart? A resposta curta é "sim". A não ser que você esteja muito mal-humorado em um dia e deseje criar um programa longo, com todas as instruções seguidas e ordenadas para execução direto de dentro da função `main()`, que já vimos ao longo dos exemplos. Opa, espera aí, **função `main()`**. É, você é obrigado a utilizar funções.

A verdade é que, em sua pura definição, funções são blocos de código que podem ser executados **n** vezes a partir de outros blocos de código. Por exemplo, você tem um programa e quer que, ao clicar no botão X da sua página inicial, ele seja fechado. Para isso, você chamaria uma função

`exit()`. Só que agora você adicionou mais três páginas que também devem fechar o programa, então bastaria fazer a chamada em todas elas para a mesma função `exit()`.

É exatamente esse o papel de uma função, permitir a modularização da nossa aplicação, garantindo um grande reaproveitamento de código ao dividir as responsabilidades. Na prática, quanto menor for a função, mais especializada em realizar apenas uma determinada ação ela vai ser, e consequentemente mais reutilizável em todos os lugares.

Em Dart, uma função pode ser definida em qualquer lugar sem estar ligada necessariamente a uma classe. Ela tem a seguinte anatomia:

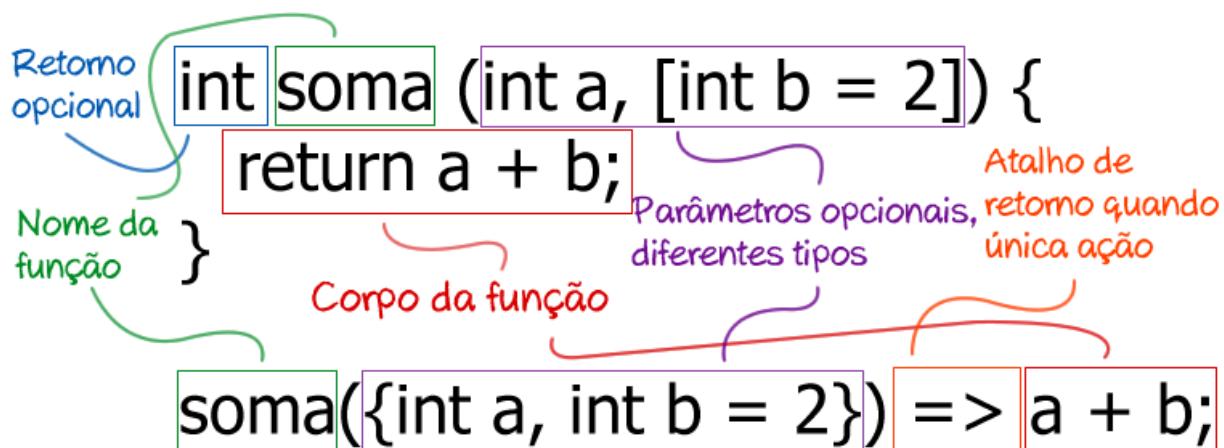


Figura 4.1: Anatomia de uma função em Dart.

Na imagem, conseguimos perceber as diversas possibilidades ao se criar uma função. E para começar a explorá-las, vamos resolver um famoso exemplo matemático, o cálculo factorial.

Em matemática, o factorial de um número é representado por $n!$, onde n é um número qualquer natural, e o resultado é o valor do produto de todos os números inteiros menores ou iguais a n . Simplificando, o factorial de 4, por exemplo, seria $4! = 4 * 3 * 2 * 1 = 24$, obedecendo à regra especial de que $0! = 1$.

Pense um pouco sobre o cálculo, já é possível identificar um padrão, não? Se pegarmos esse valor n e fizermos um loop entre os valores menores que

ele, nosso problema está resolvido:

```
int factorial(int numero) {  
    if(numero == 0) return 1;  
    var resultado = 1;  
    for(var i = 1; i <= numero; i++) {  
        resultado *= i;  
    }  
    return resultado;  
}
```

Com uma simples função de **nome** `factorial` recebemos por **parâmetro** um inteiro e ao mesmo tempo já definimos que ela vai **retornar** também um inteiro, utilizando os principais recursos de uma função. Note que validamos já no início a regra especial, se for 0 já retornamos o 1, descartando todo o resto, pois o `return` faz com que a função pare a execução.

Após isso, um `for` iniciando em 1 (afinal, qualquer multiplicação por 0 resultaria em 0) é utilizado, onde a cada iteração o valor da variável `resultado` é multiplicado por `i` e atribuído a ele mesmo. Quando `i` for maior que `número`, o `for` para a execução e retornamos nosso fatorial calculado. Experimente executá-la com diferentes valores, o retorno será o fatorial do número passado por parâmetro inicialmente.

Ainda que nossa `factorial()` seja muito básica, não é o mais simples que uma função pode ser, vamos chegar lá em breve. Mas note que com funções podemos receber variáveis por parâmetro, declarar novas variáveis dentro dela ou até mesmo dentro de operações como o loop, e essas variáveis possuem o que chamamos de escopo, geralmente delimitado pelas chaves `{}`. Como Dart possui um escopo léxico, esses mesmos modificadores podem ser acessados apenas no escopo em que estão declarados. Considere um arquivo `main.dart` que contenha apenas o seguinte trecho de código:

```
var escopoGlobal = 'global';  
void main() {  
    var escopoMain = 'escopoMain';  
    a() {  
        var escopoDentro = 'escopoA';  
    }  
}
```

```

print('a: $escopoGlobal - $escopoMain - $escopoDentro');
b() {
  var escopoDentro = 'escopoB';
  print('b: $escopoGlobal - $escopoMain - $escopoDentro');
}
b();
}
c() {
  a();
}
c();
print('main: $escopoGlobal - $escopoMain');
}

> a: global - escopoMain - escopoA
> b: global - escopoMain - escopoB
> main: global - escopoMain

```

- Variáveis como `escopoGlobal` e até a função `main()` do exemplo são consideradas como no escopo global, estão no primeiro nível de acesso do arquivo.
- Variáveis, como `escopoMain` e `escopoDentro`, são consideradas locais e possuem o escopo apenas da função dentro da qual foram declaradas.
- O mesmo ocorre com funções, `b()` é acessível apenas dentro do escopo de `a()`.
- Por outro lado, `c()` está no mesmo nível que `a()`, por isso consegue chamá-la.
- `b()` declara um modificador `escopoDentro` de mesmo nome que o existente em `a()`. Ao referenciar esse modificador, Dart vai lexicalmente escolher o mais próximo do seu nível chamado, escolhendo no caso do exemplo o que imprime '`escopoB`' .

Realmente analisar essas chamadas e códigos em que existem funções dentro de funções pode parecer complexo e assustar a princípio, mas é algo simples e comum em Dart, assim como em outras linguagens. Afinal, essas são as famosas **closures!** Em uma explicação simplista, uma função é chamada de closure quando ela possui acesso ao escopo de uma função

externa. No código anterior, as funções `a()`, `b()` e `c()` que foram declaradas dentro de uma outra função podem ser classificadas como closure. Mais adiante visualizaremos uma peculiaridade dessas funções, mas antes é importante aprender outro conceito da linguagem.

Funções como objetos

Até o momento, escrevemos vários trechos de código que demonstram a liberdade que possuímos para transitar nossos objetos por todo nosso código através das variáveis, utilizadas nos parâmetros ou retornos das funções. E isso tudo ganha um poder ainda maior quando descobrimos que uma simples função também é um objeto em Dart, do tipo `Function`! É totalmente válido guardar uma função em uma variável:

```
var ola = (String nome) {  
  print('Olá $nome');  
};
```

A variável `ola` aponta para a referência de uma função. Essa mesma função agora pode ser executada a qualquer momento a partir da variável:

```
void main() {  
  ola('Julio'); // > Olá Julio  
  ola('${ola.runtimeType}'); // > Olá (String) => Null  
}
```

Ao utilizar os parênteses na variável, estamos indicando que a função será executada. Normalmente é possível passar também os parâmetros; nesse caso, é uma `String` que, quando a função for executada, vai ser impressa no console. Ao verificar o `runtimeType` dessa variável, o resultado é a impressão dos parâmetros (`String`) junto ao seu tipo de retorno `Null`.

Em nenhum momento da função foi definido o seu retorno, mas obrigatoriamente toda função possui um retorno. Caso não retornemos algo, o compilador vai acrescentar implicitamente um `return null;` ao final da execução da nossa função. A mesma função anterior poderia ser reescrita em apenas uma linha com:

```
Function ola = (String nome) => print('Olá $nome');
void main() {
  ola('${ola.runtimeType}'); // > Olá (String) => void
}
```

Afinal, funções que possuem apenas uma operação podem ser escritas com a ajuda do `=>`, que serve como um atalho para o `{return ;}`. Essa sintaxe também é conhecida como *fat arrow*, e nesse caso, utilizando o `=>`, estamos retornando algo explicitamente, por isso a impressão do `runtimeType` mostra o retorno de tipo `void`. Mas se retornamos a função `print()`, por que é do tipo `void`? Justamente por isto, se você acessar a implementação da função `print()` no SDK através da sua IDE, notará que ela possui um retorno definido explicitamente como `void`.

```
// Assinatura da função print do pacote dart.core
void print(Object? object){}
```

Retorno de funções e o void

Um retorno do tipo `void` pode parecer muito estranho se você vem de uma linguagem como o Java, por exemplo. Por lá, quando um método é anotado com retorno `void`, significa que ele nunca retornará algo, enquanto aqui em Dart, não é bem assim.

Para começar, toda função em Dart tem um retorno, mesmo que você não declare explicitamente. Essas funções são totalmente válidas:

```
retornoNulo(){}
retornoNuloDois() { return; }
retornoString() {
  return 'String';
}
```

Para esses casos, o tipo retornado será inferido pelo compilador baseado no que você retornar dentro da função. Se você não retornar algo, como as funções `retornoNulo()` e `retornoNuloDois()`, o compilador retorna o valor `null` implicitamente.

Mas também não é porque é possível, que você deve fazer. É boa prática **sempre** declarar o tipo de retorno de uma função ou método. Por isso, em casos onde o retorno não é importante para nós, declaramos como *void*. Quando isso acontece, a função pode ter três tipos de retorno:

```
void semRetorno() {}
void retornoVazio() { return; }
void retornoDynamic() {
    dynamic dinamico = 'dynamic';
    return dinamico;
}
void retornoFuncao() {
    return print('retornoFuncao');
}
```

1. Quando não informado, implicitamente o compilador continua retornando **nulo**, como as duas primeiras funções acima.
2. Um objeto tipado como **dynamic**.
3. Uma **função** com retorno também *void*, como `print()`.

O "x da questão" é que, independente do que é retornado em uma função *void*, esse retorno não pode ser utilizado, e é esse o seu propósito: indicar para quem quer que esteja chamando esta função que o seu retorno deve ser ignorado. Por isso a função a seguir nem compilará:

```
void main() {
    var objeto = retornoDynamic();
    print(objeto);
}
```

O *void* em Dart é na verdade também um tipo e a partir da versão 2 da linguagem ele foi generalizado, o que significa que pode ser utilizado em outros locais também. Um bom exemplo é no retorno de funções assíncronas com `Future<void>`, que estudaremos em capítulos futuros.

O novato: Never

No capítulo 3, vimos como funciona a hierarquia de tipos da linguagem e como o `Never` foi introduzido junto com a *null-safety*, sendo o novo

bottom type desta hierarquia. Na prática, um tipo `Never` não tem nenhum valor pois uma expressão que resulte em `Never` indica que nunca será finalizada com sucesso, ela abortará de alguma forma ou lançará uma exceção. Um exemplo prático no SDK é a função `exit()` presente em `dart:io`.

```
import 'dart:io';
void main() {
  print('Antes de encerrar');
  exit(0);
  print('Após encerrar');
}

> Antes de encerrar
> Process finished with exit code 0
```

Ao chamar essa função, o processo do programa sempre é encerrado na VM com o código informado, por isso a assinatura da função é `Never exit(int code)`.

Para que usar o `Never`, afinal? A verdade é que raramente você vai precisar utilizá-lo de fato, mas ele fornece uma informação semântica muito válida, tanto para os desenvolvedores quanto para o próprio compilador, que apenas olhando para a assinatura da função sabem que ela nunca terá um retorno normal.

Segue um código de exemplo retirado da própria documentação da linguagem:

```
class Point {
  late final double x, y;
  @override
  bool operator ==(Object other) {
    if (other is! Point) wrongType('Point', other);
    return x == other.x && y == other.y;
  }
}
Never wrongType(String type, Object value) {
  throw ArgumentError('Expected $type, but was
```

```
    ${value.runtimeType}.');
}
```

Ainda teremos um capítulo especificamente para falar sobre exceções e outro para objetos. Mas o operador `==` é o famoso `equals` de Orientação a Objetos, que compara se dois objetos são iguais. No exemplo, caso o objeto informado não seja um `Point`, a função `wrongType()`, que sempre lança uma exceção, é chamada.

Como ela retorna `Never`, o compilador sabe que a execução será encerrada naquele momento. Por isso, no resto do método `equals`, ele consegue promover o objeto `other` corretamente para o tipo `Point`, melhorando o *flow analysis* do código.

Escopo, ao infinito e além

Já estamos craques e sabemos que funções possuem escopos, closures acessam o escopo externo, e ao mesmo tempo existe a possibilidade de as utilizarmos como variáveis e jogarmos para lá e para cá no nosso código. Pensando em tudo isso, o que aconteceria se executássemos fora de seu escopo de criação uma closure que referencia uma variável do escopo externo? Complicou? O código deixa mais fácil:

```
Function criaClosure() {
  var resposta = 42;
  return () {
    resposta++;
    print(resposta);
  };
}
```

Nossa função `criaClosure()` define em seu corpo uma variável `resposta`, e fora dela nenhuma outra função consegue enxergar essa variável, a não ser uma nova função declarada ali dentro. Essa mesma função simplesmente adiciona 1 ao valor da `resposta` e imprime o novo valor. Mas nesse momento não estamos executando esta função interna ainda, e sim utilizando-a como retorno de `criaClosure()`. O retorno então é uma `Function`.

Além disso, você notou algo de diferente nessa função retornada? A falta de um nome! Até eles são opcionais e isso é uma característica das funções denominadas anônimas. Essas funções anônimas, também conhecidas por expressões *lambda*, são muito utilizadas em operações em que você não precise chamar essa função pelo nome posteriormente no código.

Ao utilizar a `criaClosure()` :

```
void main() {  
    var somaImprime = criaClosure();  
    somaImprime();  
    somaImprime();  
}
```

Salvamos a função anônima retornada em uma variável e, a partir deste ponto, estamos em um escopo totalmente diferente daquele em que ela foi criada, então poderíamos passar essa variável para qualquer outro lugar. Em seguida, nós a executamos duas vezes. Consegue imaginar o que é impresso? Sabendo que a variável iniciou como `42`, é impresso `43 43` ou `43 44`?

O x da questão é que nossa função anônima referencia uma variável `resposta` de seu escopo externo e, mesmo após termos já finalizado a execução de `criaClosure`, Dart manteve o estado de seu escopo de variáveis para ser utilizado em qualquer outro lugar em que essa nova função seja executada. E a cada nova execução, é referenciada a mesma variável `resposta`, e por conta disso o que é impresso é `43 44`.

Aproveitando o tear-off

É comum lidarmos com situações onde utilizamos uma função anônima como um parâmetro para uma outra função. Um exemplo clássico é o `forEach`.

```
void main() {  
    final vogais = ['a', 'e', 'i', 'o', 'u'];  
    vogais.forEach((e) {  
        print(e);  
    });  
}
```

```
});  
}
```

Nele é passada uma função que será executada para cada elemento da lista. Olhando sua assinatura percebemos que ele aceita uma função qualquer que receba um parâmetro (o elemento da lista):

```
void forEach(void f(E element))
```

Como é um método da lista, o `E` representa o tipo de objeto que a lista possui, que nesse caso seria uma `String`. A função `print` por outro lado tem esta assinatura:

```
void print(Object? object)
```

E também recebe um único parâmetro, só que um `Object?`. E como um `Object?` está no topo da hierarquia de tipos, ele acaba sendo um supertipo de todos os outros que possivelmente estão presentes na lista. Então como existe um *match* nas assinaturas das funções, podemos utilizar o que Dart chama de *tear-off*:

```
void main() {  
    final vogais = ['a', 'e', 'i', 'o', 'u'];  
    vogais.forEach(print);  
}
```

É possível só passar a referência para a função, que Dart se encarrega de criar uma *closure* por baixo dos panos e chamar a função com o parâmetro. Isso serve para qualquer situação onde as assinaturas das funções sejam iguais ou, como no exemplo, um supertipo.

4.2 Na prática - Dart Web

Sempre fica mais fácil associar o conhecimento colocando a mão na massa, então nada mais justo do que começarmos um novo projeto. A ideia é recriarmos um famoso jogo à medida que vamos vendo alguns outros conceitos com funções. E o melhor de tudo é que dessa vez não vai ser

apenas via linha de comandos, você vai ser apresentado ao outro lado de Dart, onde interagimos com a web!

Mas primeiro, vamos começar entendendo o desafio. Com certeza você já jogou ou ao menos conhece o famoso Pedra, Papel e Tesoura! Ele geralmente dispensa apresentações e consiste basicamente em 2 jogadores que a cada rodada escolhem uma das três opções. Seguindo as regras de qual opção vence qual, é então definido o vencedor ou vencedora ou se ocorreu um empate.

Porém, seria muito simplista recriarmos apenas esse jogo com poucas regras, por isso vamos implementar a sua expansão conhecida como Pedra, Papel, Tesoura, Lagarto e Spock! Ela ficou bastante conhecida após ser comentada pela série de televisão The Big Bang Theory, adicionando 2 novas opções de escolha, aumentando para 10 as regras entre as combinações.

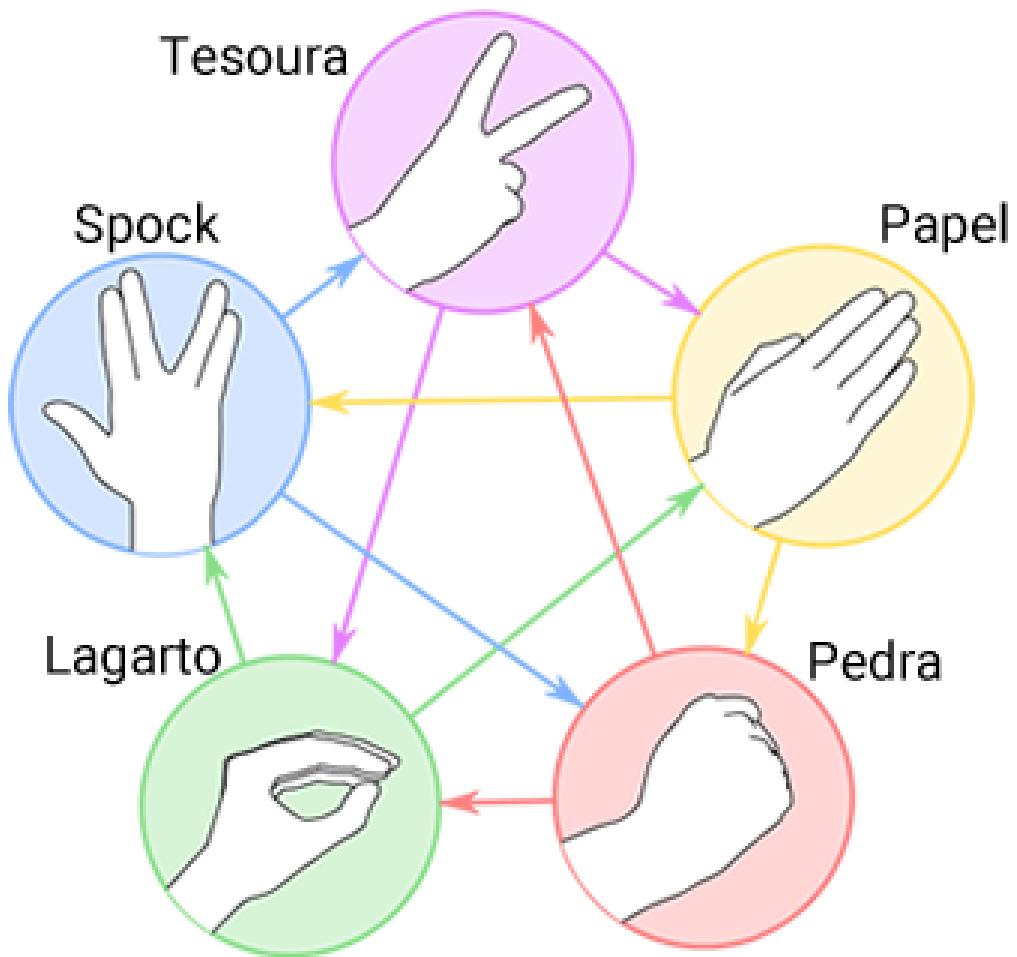


Figura 4.2: Jogo Pedra, Papel, Tesoura, Lagarto e Spock.

As regras então podem ser traduzidas como:

Vencedor	Ação	Perdedor
Tesoura	corta	Papel
Tesoura	decapita	Lagarto
Papel	cobre	Pedra
Papel	refuta	Spock
Pedra	esmaga	Lagarto
Pedra	quebra	Tesoura

Vencedor	Ação	Perdedor
Lagarto	envenena	Spock
Lagarto	come	Papel
Spock	esmaga	Tesoura
Spock	vaporiza	Pedra

Para o caso em que os dois jogadores escolheram a mesma opção, a rodada será empatada.

Criando o projeto

Para ser possível a criação de aplicações web com Dart é necessária a instalação do `webdev`, um pacote com um conjunto de ferramentas utilitárias que permitem, além da criação, a disponibilização e testes dessas aplicações. Para isso, abra o seu terminal e insira o comando a seguir. Note que é necessário ter o SDK do Dart configurado previamente.

```
dart pub global activate webdev
```

O `pub` é uma ferramenta presente no SDK com uma série de comandos utilizados para o gerenciamento de *packages* (pacotes) e dependências da linguagem. Ao digitar `pub --help` no terminal você verá uma lista com os comandos disponíveis. Um deles é justamente o `global`. Com a instalação de um *package*, conseguimos rodar seus scripts apenas no diretório do projeto em que ele é utilizado, porém, com o comando `global` é possível referenciar os *packages* globais. Então, em conjunto com o `activate`, o `webdev` é ativo globalmente, tornando possível a execução desses scripts em qualquer diretório.

Confirme a instalação rodando `dart pub global list`, que deve listar o `webdev` como um dos *packages* ativos globalmente. Esses *packages* utilizados pelo `pub` são geralmente disponibilizados através do diretório on-line: www.pub.dev. Experimente acessar e pesquisar pelo `webdev`, cuja instalação acabamos de fazer.

Agora, para a criação do projeto, existe a possibilidade de utilizar um template previamente definido na linguagem. Por exemplo, ao executar `dart create --help`, veremos uma lista de *templates* de projetos disponíveis, dentre os quais estão:

- `console-full` : um exemplo de aplicação via linha de comando.
- `package-simple` : um ponto inicial para construção de *libs* para Dart.
- `web-simple` : uma aplicação web utilizando apenas as *libs* do core do Dart.

Para o nosso jogo, utilizaremos o template `web-simple`. Navegue pelo terminal até o diretório em que você deseja criar o projeto e execute:

```
dart create -t web-simple jogo --no-pub
```

Onde `-t web-simple` representa o template; `jogo` é o diretório onde o projeto é criado; e com o `--no-pub` estamos apenas pedindo para o Dart ainda não baixar as dependências do pub. O resultado será a informação de que oito arquivos foram criados, sendo eles:

- **.gitignore**: arquivo de configuração para ignorar alguns padrões de nomenclaturas de arquivos ao versionar o projeto com o git.
- **README.md e CHangelog.md**: são arquivos padrões em projetos versionados. O primeiro é usado para uma descrição do projeto, e o segundo, para manter o registro das mudanças nas diferentes versões.
- **analysis_options.yaml**: contém regras de análise do código utilizadas pelo *dartanalyzer*, que podem ser customizadas.
- **pubspec.yaml**: arquivo padrão em projetos Dart, contém algumas configurações e a lista com todos os pacotes de dependências necessárias, gerenciados pelo `pub`.
- **web/styles.css**: arquivo com as configurações do CSS da aplicação.
- **web/index.html**: contém o código HTML utilizado na aplicação.
- **web/main.dart**: contém o código Dart utilizado, responsável por interagir com o HTML.

O `pubspec.yaml` lista todas as *libs/packages* que o projeto contém de dependência, e para ser possível rodar a aplicação é necessário fazer o download delas para nosso ambiente local. Para isso, navegue até o novo diretório `jogo` (é nele que usaremos todos os comandos a partir de agora) e execute `dart pub get`.

Esse comando fará o download de todas as dependências encontradas no diretório on-line. Após a execução com sucesso, você poderá ver que foram criados dois novos arquivos: `.packages` e

`.dart_tool/package_config.json`. O primeiro representa a forma antiga de gerenciar as dependências e está depreciado, pode ser inclusive que no momento da leitura deste livro ele já não exista mais.

Já o `package_config.json` é o novo substituto, ele mantém o registro local de todas as dependências necessárias e baixadas. Um exemplo de linhas desse arquivo está a seguir.

```
{
  "configVersion": 2,
  "packages": [
    {
      "name": "_fe_analyzer_shared",
      "rootUri": "file:///Users/jhbitencourt/.pub-
cache/hosted/pub.dartlang.org/_fe_analyzer_shared-22.0.0",
      "packageUri": "lib/",
      "languageVersion": "2.12"
    }
  ],
  "generated": "2021-07-21T02:11:02.292441Z",
  "generator": "pub",
  "generatorVersion": "2.13.1"
}
```

Agora sim é possível rodar a aplicação. Para isso, utilize:

```
webdev serve
```

Algumas informações vão aparecer no log, e dentre elas estará '[INFO] Serving web on http://127.0.0.1:8080', indicando que o servidor está

rodando e escutando por padrão na porta 8080. Para rodar em alguma outra porta, utilize `webdev serve web:8082` , por exemplo.

Agora abra um navegador e entre com o endereço `localhost:8080` , ou a porta escolhida. Você deve ver o seguinte:



Jogo Pedra, Papel, Tesoura, Lagarto e Spock.

Figura 4.3: Jogo Pedra, Papel, Tesoura, Lagarto e Spock.

Uma simples página HTML com um texto imprimindo *Your Dart app is running*. Nossa primeira aplicação web em Dart está rodando com sucesso! É hora de entender que mágica foi essa.

Dart e a web

Como vimos no início do livro, Dart originalmente nasceu com a ideia de ser uma linguagem para a web, e de fato hoje em dia existem diversos pacotes e frameworks para esse propósito, como o AngularDart e até o próprio Flutter. Mas como nosso foco é o aprendizado de todos os aspectos específicos de Dart, nada mais justo que aprendermos essa interação com a web em seu modo mais puro, utilizando apenas o *package dart:html*.

Sabemos que as páginas web são criadas através da linguagem de marcação de texto HTML, porém um simples arquivo com tags HTML produz apenas informações estáticas que são interpretadas pelo navegador. Para que as páginas ganhem vida, respondendo a ações do usuário através de eventos, como um clique, o navegador utiliza o famoso DOM (*Document Object Model*).

O DOM funciona como uma interface da leitura de arquivos HTML feita pelo browser, pois ao interpretar esse arquivo HTML é criada uma representação em formato de árvore de *nodes* (nós). Por exemplo, o arquivo `index.html` gerado pelo template:

```
<html>
<head>
  <meta charset="utf-8">
  <title>jogo</title>
  <link rel="stylesheet" href="styles.css">

  <script defer src="main.dart.js"></script>
</head>
<body>
  <div id="output"></div>
</body>
</html>
```

Foram ocultadas algumas tags `<meta/>` do código para facilitar a ilustração, mas funcionam da mesma forma. A representação do DOM seria:

 DOM index.html

Figura 4.4: DOM index.html

Todo elemento do DOM possui diretamente um filho ou um pai na árvore, o elemento `document` representa o nó inicial e permite o acesso a todos os outros nós. A partir daí, cada tag HTML é um elemento da árvore que pode possuir atributos ou um texto.

O DOM mantém o registro e permite a alteração dos elementos da árvore, e é exatamente isso que o pacote `dart:html` faz, permitir a interação do Dart com o HTML através da manipulação do DOM, adicionando, removendo e alterando os elementos. Perceba que em nenhum momento o `index.html` do nosso projeto insere o texto *Your Dart app is running* mostrado na página, então de onde ele vem? Existe uma tag `script` que está referenciando um arquivo de nome `main.dart.js`. Isso garante a ligação do arquivo `main.dart` com a nossa página. Esse arquivo contém:

```
import 'dart:html';
void main() {
    querySelector('#output')?.text = 'Your Dart app is running.';
}
```

Dentro do `main` é feita uma chamada para a função `querySelector` do pacote `dart:html`. Esta é sua implementação oficial:

```
Element? querySelector(String selectors) =>
document.querySelector(selectors);
```

Ela é responsável por acessar a árvore através do `document` e encontrar todos os elementos (tags) que possuem o mesmo identificador que passamos, no caso o `#output`. O `#` representa o atributo `id`, mesma nomenclatura famosa utilizada no `css`. Ao olhar no HTML, a tag `<div id="output"></div>` possui o mesmo `id`, e é a referência a este nó que será retornada pela função.

Em Dart todos os elementos do HTML são representados como um objeto `Element`, no caso da `div`, teremos como retorno da função um `DivElement`. Através desse objeto, é possível manipular seus atributos e, consequentemente, o DOM. Ao alterar o `text`, o HTML será também

atualizado para `<div id="output">Your Dart app is running.</div>`, resultando no texto impresso na tela.

Iniciando o jogo e métodos

Chegou a hora de começar a manipular os arquivos do projeto para dar vida ao nosso jogo. Para isso, acesse e remova todo o conteúdo dos arquivos `index.html`, `styles.css` e `main.dart`. Após isso, no diretório do projeto, crie a estrutura de pastas `lib/src/` e um arquivo `partida.dart` dentro. E é por ele que vamos iniciar.

Existem cinco escolhas que o jogador pode fazer em cada partida, que podem muito bem ser definidas da seguinte forma.

```
//Arquivo partida.dart
const pedra = 'Pedra';
const papel = 'Papel';
const tesoura = 'Tesoura';
const lagarto = 'Lagarto';
const spock = 'Spock';

const opcoes = [pedra, papel, tesoura, lagarto, spock];
```

Agora essas escolhas estão representadas por cinco constantes e uma lista, também constante. Além das opções, outra informação muito importante para uma partida são justamente as regras, então nosso próximo passo é encontrar uma maneira de definir e armazená-las no código.

Já vimos que '**Tesoura corta Papel**' é uma regra, assim como '**Tesoura decapita Lagarto**' é outra. Existe um padrão nessa definição, onde uma opção vencedora (Tesoura) tem mais de uma ação (corta/decapita) contra as opções perdedoras (Papel/Lagarto). De que forma podemos guardar isso em Dart?

 Regras organizadas em um Map.

Figura 4.5: Regras organizadas em um Map.

Um `Map`! Sim, essa é uma das estruturas de dados existentes que armazena registros no formato chave/valor. Então podemos ter um `Map` contendo como **chave** a opção vencedora e **valor** sendo um outro `Map`, de **chave** igual à ação e **valor** correspondente à opção perdedora. Todas essas definições podem perfeitamente serem armazenadas em uma classe

`Partida`:

```
//Arquivo partida.dart
class Partida {
    final regras = <String, Map<String, String>>{};
```

Um map pode ser criado com a sintaxe `{}`, que chamamos de forma literal. Dentro dos sinais de menor e maior `< >`, estão os tipos aceitos como chave e valor. Exploraremos mais o funcionamento dessa sintaxe conhecida como *generics* em capítulos futuros. Com o map criado, está na hora de populá-lo com as regras, então vamos criar uma função para isso.

```
class Partida {
    //...código omitido
    void criarRegra(String vencedor,
        {required String acao, required String perdedor}) {
        if (!regras.containsKey(vencedor)) {
            regras[vencedor] = {};
        }
        regras[vencedor]![acao] = perdedor;
    }
}
```

A verdade é que esse `criarRegra` não é exatamente uma função, e sim, um método! Isso porque ele está associado diretamente a uma classe. Além disso, a diferença dos métodos para as funções é que eles possuem acesso ao `this`, que é uma referência à instância daquele objeto sendo executado no momento.

Ainda no método, o seu objetivo é basicamente preencher o map com as regras. Sempre verificamos se ele já contém a chave `vencedor` para

inicializar o seu valor, que também é um map, para poder então registrar nele a regra contendo a chave `acao` e valor `perdedor`. Note que ao acessar uma chave com o operador `[]`, caso ela não exista pode ser retornado um valor nulo. Por isso, precisamos utilizar o operador `bang`, já que temos certeza de que ela existirá neste momento.

Ainda existe algo novo a ser explorado neste método, a declaração dos parâmetros.

4.3 Parâmetros

Os parâmetros em Dart podem ser classificados de acordo com essa simples tabela:



Tipos de parâmetros.

Figura 4.6: Tipos de parâmetros.

Posicional e obrigatório

No primeiro quadrante, estão os posicionais e obrigatórios, são eles que viemos utilizando ao longo de todos os exemplos do livro.

```
void temperaturaEm(String cidade, int? ano, int? mes, int dia) {  
    print('$cidade $dia/$mes/$ano');  
}  
void main() {  
    temperaturaEm('Floripa', null, 12, 1); // > Floripa 1/12/null  
}
```

Ao chamar a função, é obrigatório que todos sejam informados respeitando a mesma ordem de declaração. Mesmo para parâmetros *nullable* o valor `null` deve ser informado explicitamente caso não tenha outro valor.

Posicional e opcional

No segundo quadrante, temos os posicionais e opcionais. Embora sua ordem continue sendo importante, eles aceitam a atribuição de valores *default*, o que torna a sua declaração opcional. São definidos entre colchetes `[]` e podem ser utilizados junto a parâmetros normais:

```
void temperaturaEm(String cidade, [int? ano = 2020, int? mes, int  
dia = 01]) {  
    print('$cidade $dia/$mes/$ano');  
}  
void main() {  
    temperaturaEm('Floripa'); // > Floripa 1/null/2020  
    temperaturaEm('Floripa', null); // > Floripa 1/null/null  
    temperaturaEm('Floripa', 2021, 1); // > Floripa 1/1/2021  
    temperaturaEm('Floripa', 2021, 1, 2); // > Floripa 2/1/2021  
}
```

Ao chamar a execução da função, não é obrigatório passá-los. Para os tipos *non-nullable*, como o `dia` é obrigatório informar um valor *default*. Já para os *nullable* como `ano` e `mes`, é opcional e caso não informado o *default* é `null`.

Nomeado e opcional

No quarto quadrante (sim, pulamos o terceiro), estão os considerados nomeados e opcionais. Definidos entre chaves `{ }`, eles também permitem valores *default*.

```
void temperaturaEm(String cidade, {int? ano = 2020, int dia = 01,
int? mes}) {
    print('$cidade $dia/$mes/$ano');
}

void main() {
    temperaturaEm('Floripa', ano: 2021); // > Floripa 1/null/2021
    temperaturaEm('Floripa', mes: 1); // > Floripa 1/1/2020
    temperaturaEm('Floripa', mes: 1, dia: 2, ano: null); // > Floripa
2/1/null
}
```

Justamente por serem nomeados, o nome do parâmetro deve ser declarado ao chamar a função e, por conta disso, a ordem da definição dos parâmetros não é importante e pode ser alterada.

Nomeado e obrigatório

Anterior a *null-safety* existia esse "buraco" nos tipos de parâmetros da linguagem, pois não havia uma forma de obrigar que um parâmetro nomeado fosse declarado ao chamar uma função. Então foi decidido incluir o terceiro quadrante criando um parâmetro nomeado e obrigatório, que precisa ser declarado com o novo modificador `required`:

```
void temperaturaEm(String cidade,
    {required int dia, required int? mes, int? ano = 2020}) {
    print('$cidade $dia/$mes/$ano');
}

void main() {
    temperaturaEm('Floripa', dia: 1, mes: null);
    // > Floripa 1/null/2020
    temperaturaEm('Floripa', mes: 1, dia: 12);
    // > Floripa 12/1/2020
    temperaturaEm('Floripa', mes: 1, dia: 2, ano: 2021);
```

```
// > Floripa 2/1/2021  
}
```

Quando definido como `required`, o parâmetro não pode ter um valor `default` declarado e ele passa a ser obrigatório. A ordem, entretanto, continua sendo opcional. No método `criarRegra` do nosso jogo, utilizamos 1 parâmetro posicional/obrigatório e 2 nomeados/obrigatórios, e o motivo para isso está logo a seguir.

Declarando as regras do jogo

Voltando para a criação do jogo, vá até o arquivo `main.dart` e adicione:

```
import 'package:jogo/src/partida.dart';  
void main() {  
    final partida = configurarPartida();  
}
```

Importamos o `partida.dart` criado anteriormente, e já podemos definir a função `configurarPartida()`:

```
Partida configurarPartida() {  
    return Partida()..criarRegra(tesoura, acao: 'corta', perdedor:  
papel)  
        ..criarRegra(tesoura, acao: 'decapita', perdedor: lagarto)  
        ..criarRegra(papel, acao: 'cobre', perdedor: pedra)  
        ..criarRegra(papel, acao: 'refuta', perdedor: spock)  
        ..criarRegra(pedra, acao: 'esmaga', perdedor: lagarto)  
        ..criarRegra(pedra, acao: 'quebra', perdedor: tesoura)  
        ..criarRegra(lagarto, acao: 'envenena', perdedor: spock)  
        ..criarRegra(lagarto, acao: 'come', perdedor: papel)  
        ..criarRegra(spock, acao: 'esmaga', perdedor: tesoura)  
        ..criarRegra(spock, acao: 'vaporiza', perdedor: pedra);  
}
```

Essa função é a responsável por definir todas as regras de uma partida. Com a utilização dos parâmetros nomeados, não ficou mais fácil a leitura do código? Parece que estamos lendo um texto que descreve as regras.

Uma novidade é o operador em cascata . . . , ele permite fazer uma sequência de chamadas a atributos e métodos de um mesmo objeto. Funciona como um *syntax sugar* da linguagem e se parece muito com o padrão de design *builder*, a cada chamada ele retorna a referência para o próprio objeto, que permite encadear uma nova chamada. Então ao finalizar a execução desta função, teremos a referência para um objeto Partida já com as regras do jogo definidas.

Manipulando o DOM do jogo

Vamos dar início à manipulação do DOM para construir o nosso jogo no browser. A ideia é que o resultado fique parecido com o planejado a seguir.



Regras organizadas em um Map.

Figura 4.7: Regras organizadas em um Map.

Para facilitar a criação, é possível dividir o protótipo acima em áreas. Ao acessar a página, o jogador será apresentado ao título e à área de opcoes para escolha. Ao escolher, a área de resultado é renderizada dando a opção de jogar novamente. Defina essas áreas então no `index.html`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Pedra, Papel, Tesoura, Lagarto e Spock</title>
    <link rel="stylesheet" href="styles.css">
    <script defer src="main.dart.js"></script>
</head>
<body>
    <h2>Vamos jogar!</h2>
    <h4>Escolha um:</h4>
    <div id="opcoes"></div>
    <div id="resultado"></div>
</body>
</html>
```

No header definimos o título, a referência ao arquivo de css e nosso `main.dart`, que funcionará como o script da página. Já no body, como os títulos serão fixos, eles já podem estar diretamente no arquivo. Por outro lado, as áreas do desenho anterior são dinâmicas (serão manipuladas pelo código em Dart), então apenas fizemos uma demarcação definindo duas divs para referenciar posteriormente no código.

Se o servidor estiver rodando, basta salvar o arquivo e atualizar a página do navegador, isso porque o webdev fica observando as modificações para aplicar o resultado. Ao executar, temos:



Regras organizadas em um Map.

Figura 4.8: Regras organizadas em um Map.

Você não verá exatamente o mesmo título estilizado e alinhado ao centro, para isso é necessário atualizar o arquivo `styles.css`. Como esse não é o objetivo do livro, e não é obrigatório estilizar a página, você pode encontrar esse arquivo no GitHub do livro.

Para que possamos manipular o conteúdo da nossa página, o código em Dart precisa saber da existência dessas `divs` que definimos no HTML, então volte no `main.dart` e insira:

```
import 'dart:html';
import 'package:jogo/src/partida.dart';

lateDivElement divOpcoes;
lateDivElement divResultado;

void main() {
    inicializarReferencias();
    final partida = configurarPartida();
}
void inicializarReferencias() {
    divOpcoes = querySelector('#opcoes') asDivElement;
    divResultado = querySelector('#resultado') asDivElement;
}
```

O `late` avisa ao compilador que essas variáveis podem ser *non-nullable* e serão inicializadas antes de usarmos. E é com o `querySelector` que realizamos essa tarefa, informando o id das `divs` do HTML. Assim é criado uma referência para os elementos do DOM e guardado nas variáveis de tipo `DivElement`. Precisamos utilizar o operador `as` neste caso, pois o `querySelector` retorna um *nullable* `Element?`.

Com isso, já é possível adicionar as opções de escolha na tela. Então crie uma pasta `images` dentro da pasta `web`. Agora faça o download das imagens do repositório do livro no GitHub e insira nesta pasta, ficando assim:



Estrutura da pasta de imagens.

Figura 4.9: Estrutura da pasta de imagens.

Pronto, já podemos manipular o DOM para inserir as opções:

```
void main() {  
    // ...código omitido  
    for (final opcao in opcoes) {  
        divOpcoes.append(  
            ImageButtonInputElement()  
                ..className = 'opcao'  
                ..src = 'images/$opcao.png'  
                ..height = 120,  
        );  
    }  
}
```

Com um loop `for` na constante com a lista de `opcoes` criadas anteriormente, podemos criar os elementos de imagem e inserir os mesmos na `div` através do método `append` da `divOpcoes`, que serve justamente para adicionar novas tags HTML ao DOM.

Ao final serão adicionados cinco objetos do tipo

`ImageButtonInputElement`, que representa um `input` de tipo `image` em HTML. Com o operador em cascata, podemos alterar os atributos desse elemento, como definir o nome da classe `css`, qual o `src/caminho` da imagem (que baixamos) e o `height/tamanho` dela. Na prática, Dart vai adicionar à página o seguinte código em HTML para cada opção:

```
<input type="image" src="images/Papel.png" height="120"  
class="opcao">
```

E a seguir o resultado visual na página.



Imagen das opções adicionadas ao jogo.

Figura 4.10: Imagem das opções adicionadas ao jogo.

4.4 Enums

Para a próxima área, é necessário pensar em uma representação para o resultado do jogo. Sabemos que existem apenas três situações em que o jogo poderá acabar, a vitória, a derrota ou o empate. Esse é um caso típico onde é possível representar o estado de um objeto através da utilização de um `enum`.

```
enum ResultadoType { empate, vitoria, derrota }

void main() {
    print(ResultadoType.values);
    // > [ResultadoType.empate, ResultadoType.vitoria,
    ResultadoType.derrota]
    print(ResultadoType.empate); // > ResultadoType.empate
    print(ResultadoType.empate.name); // > empate
}
```

Com a palavra-chave `enum`, criamos esse novo tipo especial. Os enums em Dart funcionam de forma parecida com simples constantes, isso porque eles são o mais simples possível. Não permitem atribuir parâmetros aos atributos, como existente em outras linguagens. Possuem por padrão apenas o método `values` cujo retorno é uma lista com os valores existentes, e podem ser acessados diretamente através de seu nome

`ResultadoType.empate`.

A vantagem da utilização de um `enum` é a sua representação em um tipo específico e centralização dos valores desse tipo. Então aproveite e crie o arquivo `resultado.dart` também no diretório `lib/src` e adicione:

```
enum ResultadoType { empate, vitoria, derrota }
class Resultado {
    Resultado(this.resultadoType, this.resumo);
    final ResultadoType resultadoType;
```

```
    final String resumo;  
}
```

A classe `Resultado` vai de fato guardar o resultado da partida, sendo ele o `ResultadoType` em conjunto com o `resumo`. Este último vai conter a informação do que aconteceu para o jogador, como 'Tesoura corta Papel', por exemplo.

Definindo o vencedor da partida

Com as regras do jogo definidas e a representação do resultado criada, ficou faltando a lógica da partida para definir quem ganhou. Para isso, temos que comparar as 2 opções, a primeira é a escolha feita pelo jogador humano que vai interagir com o nosso jogo, clicando em uma das opções na tela. E a segunda opção tem que ser a escolhida pelo adversário, que nesse caso é o próprio jogo, simulando a escolha do computador.



Escolha aleatória feita pelo jogo.

Figura 4.11: Escolha aleatória feita pelo jogo.

Todas as opções já estão definidas em uma lista, então basta que o jogo tenha a capacidade de escolher randomicamente uma delas! Volte em `partida.dart` e crie o método `escolherPc()` dentro da classe `Partida`:

```
import 'dart:math';
//... código omitido
String escolherPc() {
    final index = Random().nextInt(5);
    return opcoes[index];
}
```

Com a ajuda do `Random` do pacote `dart:math` é possível gerar um valor inteiro randômico de 0 a 4 (o 5 é não inclusivo), sendo esse a representação do índice de um dos itens da lista. Pronto, nosso jogo já é inteligente o suficiente para escolher sua jogada! Quanto a decidir o vencedor, no mesmo arquivo crie o método `iniciar` da `Partida`.

```
//... código omitido
Resultado iniciar({required String humano}) {
    final pc = escolherPc();
    if (humano == pc) {
        return Resultado(ResultadoType.empate, '$humano empata com
$pc');
    }
}
```

Nosso método possui um parâmetro nomeado que representa o nome da opção selecionada pelo jogador humano. Logo em seu início já usamos o `escolherPc()` para definir a escolha do computador. A primeira possibilidade do resultado, e também a mais fácil, é o empate. Então se as duas opções forem iguais, já retornamos um `Resultado` com `ResultadoType.empate`, junto da informação das escolhas.

A segunda possibilidade é a de vitória. Dentro do mesmo método:

```
//... código omitido
if (regras[humano]!.containsValue(pc)) {
    final entry = regras[humano]!.entries.firstWhere((e) => e.value
```

```
== pc);
    return Resultado(
        ResultadoType.vitoria, '$humano ${entry.key} ${entry.value}');
}
```

Para validar uma vitória é necessário utilizar o Map de regras e relembrar como ele foi construído. Todas as opções vencedoras estão como `key` desse Map, e o `value` corresponde às opções perdedoras.

 Escolha aleatória feita pelo jogo.

Figura 4.12: Escolha aleatória feita pelo jogo.

Então com `regras[humano]` estamos acessando no Map de `regras` o valor cuja chave é a opção do `humano`, como esse retorno pode ser *nullable* utilizamos o `!`, pois sabemos que sempre haverá um registro para a chave. A resposta é também um Map com as opções perdedoras, e se esse Map contém como `value` a escolha do `pc`, significa que o jogador `humano` ganhou a partida.

Todo item de um Map é conhecido como *entry* e é representado por um objeto `MapEntry<key, value>`. Eles são acessíveis através da lista de `entries`, onde estamos filtrando e pegando o primeiro item que tenha o `value` igual à opção do `pc`. Com isso, é possível montar o Resultado com `ResultadoType.vitoria` para retorno.

A terceira e última possibilidade é a derrota.

```
//... código omitido
final entry = regras[pc]!.entries.firstWhere((e) => e.value ==
humano);
return Resultado(ResultadoType.derrota, '$pc ${entry.key}
${entry.value}');
```

Se chegar a este ponto do método, já sabemos que o jogador foi derrotado e que o `pc` ganhou. A lógica utilizada para montar o resultado é exatamente igual à anterior, só que dessa vez as opções são invertidas. Então o Resultado com `ResultadoType.derrota` é retornado. Assim finalizamos todas as modificações do arquivo `partida.dart`.

4.5 Typedef

Antes de finalizarmos o jogo, é importante conhecermos o conceito de `typedef`.

```
typedef Operacao = Object Function(double a, double b);
```

Um *typedef* funciona como um alias (ou apelido) para um outro tipo, como `operacao` acima, que representa uma função que recebe dois parâmetros `double` e retorna um `object`. Na prática, as variáveis declaradas com este novo apelido só poderão referenciar funções que possuem a mesma assinatura. Considere no exemplo a seguir:

```
double somar(double a, double b) {
    return a + b;
}
String subtrair(double a, double b) {
    return (a - b).toString();
}
Object calcular(double a, double b, Operacao operacao) {
    return operacao(a, b);
}
void main() {
    print(calcular(22, 20, somar)); // > 42
    print(calcular(22, 20, subtrair)); // > 2
}
```

A função `calcular` recebe dois valores `double` por parâmetro, seguidos de uma variável do *typedef* `operacao`. Esse último parâmetro vai aceitar qualquer função que respeite a assinatura `Object Function(double a, double b)`, por isso conseguimos executar o `calcular` passando por parâmetro as funções `somar` e `subtrair`, obtendo resultados distintos. E mesmo que `somar` e `subtrair` possuam retornos diferentes, lembre-se de que ambos os tipos também são aceitos como `Object`.

O uso do *typedef* é recomendado para situações em que desejamos atribuir um maior significado para um tipo ou função específica. Em nosso jogo, por exemplo, usaremos um *typedef* já existente no SDK para implementar o `click` nos botões.

4.6 Adicionando interação ao jogo

Voltando ao `main.dart`, onde havíamos adicionado no método `main()` a definição dos elementos `ImageButtonInputElement`, que são nossas imagens, precisamos agora fazer com que elas sejam clicáveis, para que o jogador consiga escolher uma das opções.

Esses objetos do tipo `Element` fornecem listeners que podemos acompanhar para saber quando um determinado evento ocorreu, por exemplo, o clique do *mouse*. Então no mesmo objeto `ImageButtonInputElement()` já criado, adicione um novo parâmetro `onClick` via cascade:

```
import 'package:jogo/src/resultado.dart';
bool jogando = true;
late Resultado resultado;
void main() {
// ...código omitido
    ImageButtonInputElement()
        ..onClick.listen(
            (MouseEvent e) {
                if (jogando) {
                    resultado = partida.iniciar(humano: opcao);
                    jogando = false;
                    mostrarResultado(resultado);
                }
            },
        )
// ...código omitido
}
```

O `onClick`, assim como vários outros atributos existentes desse objeto que começam com `on`, fornece uma `Stream` de dados que podemos sobrescrever através do método `listen()`. Vamos entender `streams` mais para a frente, mas basta saber agora que estamos passando uma função anônima para o `listen` que será executada sempre que esse evento for disparado, no nosso caso, o clique do *mouse*.

A parte interessante é que essa função passada, que também pode ser chamada de `callback`, deve respeitar uma assinatura, que é a assinatura do `typedef EventListener`.

```
typedef EventListener(Event event);
```

Estamos passando essa função anônima que recebe por parâmetro um `MouseEvent`, que também é um `Event`. Dentro dela, utilizamos uma variável booleana `jogando` para controlar o estado do jogo e, caso seja `true`, vamos chamar o método para iniciar a partida, obtendo um `Resultado` de retorno.

Esse mesmo resultado precisa ser apresentado para o usuário na tela. Lá atrás, quando criamos o `index.html`, o dividimos em áreas, a `<div id="opcoes"/>` já foi usada, então nos restou a `<div id="resultado"/>`. Esse vai ser o papel da função `mostrarResultado()` incluída também em `main.dart`.

```
void mostrarResultado(Resultado resultado) {
    String mensagem;
    String classeCss;

    switch (resultado.resultadoType) {
        case ResultadoType.empate:
            classeCss = 'empatou';
            mensagem = 'Empatou..';
            break;
        case ResultadoType.vitoria:
            classeCss = 'venceu';
            mensagem = 'Você ganhou :D';
            break;
        case ResultadoType.derrota:
            classeCss = 'perdeu';
            mensagem = 'Você perdeu :/';
            break;
    }

    divResultado.append(
        SpanElement()
            ..className = classeCss
            ..text = mensagem,
    );
}
```

Primeiro, através de um `switch case` definimos a mensagem e classe do css a serem utilizados, baseado no `enum resultadoType`. A variável `divResultado` mantém a referência para a div, então podemos acrescentar elementos ao DOM. O `SpanElement` representa a tag ``, que conterá a mensagem e classe do css definidas.

```
void mostrarResultado(Resultado resultado) {
    // ...código omitido
    adicionarEspaco();
    divResultado.append(SpanElement()..text = resultado.resumo);
    adicionarEspaco();
    divResultado.append(
        ButtonElement()
            ..text = 'Jogar novamente!'
            ..onClick.listen(jogarNovamente),
    );
}
void adicionarEspaco() {
    divResultado.append(BRElement());
    divResultado.append(BRElement());
}
void jogarNovamente(MouseEvent e) {
    jogando = true;
    divResultado.children.clear();
}
```

O restante do método segue a mesma lógica já apresentada, adicionando novos elementos ao DOM, como o `BRElement()`, que representa uma simples quebra de linha `
`. Com destaque para o `ButtonElement`, o novo botão adicionado, que permite ao usuário a opção de jogar novamente.

Repare que no `listen` desse botão definimos uma função que, ao clicar, limpará todo o conteúdo da `div` de resultado com `divResultado.children.clear()`, fazendo com que os elementos dessa `div` sumam da tela e representando um novo início de jogo. Assim chegamos ao fim da implementação. Divirta-se desafiando o computador.



Resultado final do jogo pronto.

Figura 4.13: Resultado final do jogo pronto.

4.7 Se liga aí

Algumas boas práticas dos assuntos que conhecemos neste capítulo:

- Por motivos de compatibilidade com versões anteriores, Dart ainda permite o uso de : na definição de valores para parâmetros opcionais, mas o correto atualmente é a utilização de = . Também não defina explicitamente um valor opcional como null para variáveis *nullable*, pois isso é implícito.

ruim	bom
<pre>void criarEvento({int mes : 1, int? dia : null})</pre>	<pre>void criarEvento({int mes = 1, int? dia})</pre>

- Caso um dos parâmetros de sua função seja *nullable* e possa ter o valor passado como nulo, defina-o como um parâmetro opcional, em vez de forçar o usuário do seu código a passar null explicitamente.

Ruim	Bom
<pre>enviarEmail('Cobrança', null, null)</pre>	<pre>enviarEmail('Cobrança')</pre>

- Em algumas situações em que é passado por parâmetro uma função, em vez de criar uma closure e dentro desta executar a função, passe diretamente a referência para a função.

```
// ruim  
onClick.listen(  
    (MouseEvent e) {  
        processarEvento(e);  
    },  
);  
  
// bom  
onClick.listen(processarEvento);
```

- Ao criar uma closure, na maioria das vezes não precisamos definir um nome, sendo assim uma função anônima. Mas existem alguns casos em que é necessário nomeá-la, então atribua um nome diretamente em vez de atribuir a uma variável.

```
// ruim
void main() {
    var minhaClosure = () {};
    minhaClosure();
}
```

```
// bom
void main() {
    void minhaClosure() {};
    minhaClosure();
}
```

- Nomeie os enums com *CamelCase* e seus valores com o mesmo padrão das constantes, em *lowerCamelCase*.

ruim	bom
enum meses {JANEIRO, FEVEREIRO};	enum Meses {janeiro, fevereiro};

4.8 É com você

1 - Em programação, recursividade é quando uma função acaba chamando a si mesma durante sua execução. Transforme a função de fatorial criada no capítulo para realizar o cálculo de forma recursiva.

2 - Recursão é o mesmo que closure? Você consegue imaginar uma função utilizando recursão e closure ao mesmo tempo? Experimente fazer isso com o cálculo de fatorial.

3 - Fique sabendo que até mesmo classes podem ser "tratadas" como funções. Pesquise sobre *Callable Classes*.

4 - Um enum em Dart é o mais simples possível, com isso não é possível associar valores (variáveis) para cada constante presente nele, diferente de outras linguagens. Como faríamos para simular esse tipo de comportamento? Tente simular com o enum `ResultadoType` de forma que ele já guarde a informação da classe de css e mensagem para o usuário.

Até aqui

Mais um capítulo com vários conceitos chegando ao fim. Estamos cada vez mais eficientes na linguagem e agora, mais do que nunca, entendendo tudo sobre escopo, tipos e retornos de funções e métodos! Além é claro, de entender a semântica por trás dos tipos `void` e `never`.

Ao criarmos o jogo web também, por mais simples que tenha sido, colocamos um pouco desses conceitos em prática e conseguimos aprender sobre os *enums* e *typedefs*. Tente adicionar novas funcionalidades e até mesmo recriá-lo em Flutter, afinal por que não? Aproveite esta pausa no fim do capítulo, ou já vá direto ao próximo, pois agora entraremos em um assunto que, dependendo de quando aparece no código, pode causar uma grande dor de cabeça.

CAPÍTULO 5

Cuidando dos erros

Sem vilões não há heróis e assim também é na programação. Apesar de sempre tratarmos nosso código criado como o salvador de todos os problemas, ele sempre enfrentará alguma exceção no meio do caminho. E é o nosso papel estar ciente disso e prepará-lo para lidar com esses erros da melhor forma possível.

Dart, assim como qualquer outra linguagem de programação, fornece meios para capturarmos eventuais exceções e reagirmos de forma adequada para cada situação. Mas tem algo curioso que a princípio pode gerar uma certa dúvida para quem vem de outras linguagens: os criadores de Dart optaram por fazer uma distinção entre dois tipos de problemas que um desenvolvedor pode enfrentar. Então, neste capítulo, vamos:

- Entender a diferença entre `Error` e `Exception` ;
- Aprender a lançar e capturar as diversas exceções;
- Descobrir a funcionalidade do `assert` .

5.1 Error versus Exception

Existem duas classes principais ao trabalharmos com erros em Dart, a `Exception` e a `Error` . Pode parecer confuso, afinal estamos acostumados a pensar que toda exceção é também um erro, mas na prática existem propósitos diferentes para cada uma dessas classes.

Um `Error` representa erros de utilização do código em si, aqueles que em teoria só devem ocorrer durante o desenvolvimento e que o programa deve corrigir imediatamente, pois o código está se comportando (sendo usado) de forma incorreta. Um *error* deve sempre parar a execução do programa e, na maioria das vezes, são causados pela má utilização de alguma biblioteca ou

da própria linguagem. Alguns exemplos desses tipos de erros que podemos encontrar são:

- `CastError` : ocorre ao tentar realizar uma operação de *cast* inválida, como forçar o *cast* de um inteiro para um booleano, por exemplo.
- `ArgumentError` : ocorre ao passar um parâmetro inválido para alguma função.
- `NoSuchMethodError` : ocorre ao chamar um método não existente em um determinado objeto, como chamar um método qualquer `a.metodoNaoExistente()` em uma variável tipada como `dynamic`.

No geral, é possível notar um padrão e, ao se deparar com um *error*, o desenvolvedor já sabe que o seu código está utilizando algo de forma errada. Por exemplo, a seguinte função:

```
void main() {  
    var lista = <int>[1, 2];  
    for(var i = 0; i<= 2; i++) {  
        print('Atribuindo valor $i no índice $i');  
        lista[i] = i;  
    }  
}
```

Que, ao ser executada, imprime:

```
> Atribuindo valor 0 no índice 0  
> Atribuindo valor 1 no índice 1  
> Atribuindo valor 2 no índice 2  
> Unhandled exception:  
> RangeError (index): Invalid value: Not in range 0..1, inclusive:  
2  
> #0      List._setIndexed (dart:core-patch/array.dart:19:64)  
> #1      List.[]=(dart:core-patch/array.dart:16:5)  
> #2      main (file:///C:/Users/julio/work/dart-  
book//pt_Br/05_capitulo/01_error/main.dart:5:10)
```

Analisando tanto o código quanto o que foi impresso, você consegue identificar o problema? O fato é que ocorre um *error* e, ao olharmos a mensagem, ela indica o nome `RangeError`, o que já é um indício de que estamos fazendo algo errado.

Para auxiliar na solução, além da mensagem descritiva `Invalid value: Not in range 0..1, inclusive: 2`, Dart imprime também todo o *stacktrace* do erro indicando onde ele ocorreu e por onde ele passou, como pode ser notado na linha `#2`, que indica o caminho completo do nosso arquivo no sistema operacional. Ao final desse caminho, repare como são informados dois números juntos ao nome do arquivo: `main.dart:5:10`. Eles representam o local exato do erro dentro desse arquivo, ou seja, na linha 5 e coluna 10 do `main.dart`.

O que acontece é que foi criada uma lista de inteiros com dois objetos dentro da lista usando `<int>[1, 2]`. E posteriormente tentamos acessar o índice 2 desta lista, o que não existe, uma vez que os índices dela vão apenas de 0 a 1 (tamanho fixo de 2). Portanto, o contrato de uso da API de Dart foi quebrado pelo desenvolvedor, representando um erro que deve ser corrigido imediatamente.

Por outro lado, uma `Exception` indica uma situação inesperada mesmo que o código esteja implementado corretamente, e não necessariamente é algo que precisa ser corrigido imediatamente. Elas devem ser capturadas e tratadas de acordo com as regras pertinentes. São situações que quem programa sabe que podem ocorrer e o programa saberá se recuperar apropriadamente. Dois exemplos de *exception*:

- `TimeoutException` : ocorre ao expirar o tempo para realizar uma determinada ação, como fazer uma chamada assíncrona para algum serviço com um tempo de resposta predefinido, caso ultrapasse o tempo, a exceção é lançada.
- `FileSystemException` : pode ocorrer ao realizar operações de IO, como manipular arquivos do sistema operacional corrompidos ou inacessíveis.

Ou um terceiro exemplo, supondo que você esteja desenvolvendo um sistema via linha de comandos, onde o usuário informa uma data em formato de texto e é necessário realizar o *parse* dela para o tipo `DateTime` :

```
import 'dart:io';
void main() {
```

```
final dataUsuario = stdin.readLineSync();
DateTime.parse(dataUsuario!);
}
```

O `stdin` da `lib dart:io` é utilizado para manipular o *standard input* ou padrão de entrada de dados do sistema operacional, e com o seu método `readLineSync()` é possível fazer a leitura de forma síncrona do que o usuário digita na linha de comando. Com isso, se for executado o código acima e informado o valor `01/01/2021`, por exemplo, que para nós brasileiros é uma data válida, é impresso:

```
> Unhandled exception:
> FormatException: Invalid date format
> 01/01/2021
> #0      DateTime.parse (dart:core/date_time.dart:336:7)
> #1      main (file:///C:/Users/julio/work/dart-
book/pt_Br/05_capitulo/02_exceptions/main.dart:4:12)
```

Uma `FormatException` foi lançada, indicando que o formato de data utilizado ao tentar realizar o `parse` é inválido, afinal o padrão internacional aceito pelo método é `ano/mes/dia` ou `2020-01-01`. Sendo assim, exceções podem e vão ocorrer durante o uso de algum sistema, devendo ser tratadas de forma que ele possa se recuperar adequadamente, como pedir para o usuário inserir no padrão aceito, por exemplo.

5.2 Lançando exceções

Quando os desenvolvedores implementaram a API de `DateTime`, eles definiram um padrão de valores aceitos no método `parse()`, que caracterizam uma data como válida. Para manter a integridade dessa API é necessário avisar de alguma forma quem a chamar, de que um determinado valor é inválido. Confira um trecho da documentação desse método:

```
/// Constructs a new [DateTime] instance based
/// on [formattedString].
/// Throws a [FormatException] if the input string
```

```
//> cannot be parsed. ...
static DateTime parse(String formattedString) {
```

Na terceira linha está explícito que será lançada uma exceção `FormatException` caso não seja possível efetuar o *parse* desse valor. Eventualmente lidamos com situações parecidas onde precisamos lançar uma exceção de acordo com a lógica do nosso código, para manter sua integridade. Por isso, o comando `throw` pode ser utilizado em qualquer local para disparar uma exceção:

```
void voar() {
    throw Exception('Você não tem asas!');
}
void main() {
    voar();
}

> Unhandled exception:
> Exception: Você não tem asas!
> #0      ...stacktrace
```

O objeto `Exception` recebe uma mensagem descritiva por parâmetro que é impressa junto do *stacktrace* no console para auxiliar a análise do problema. Só que em Dart qualquer objeto não nulo pode ser lançado como uma exceção, até uma simples `String`:

```
void ligarCarro() => throw 'Sem gasolina!';
void main() {
    ligarCarro();
}

> Unhandled exception:
> Sem gasolina!
> #0      ...stacktrace
```

E assim, como qualquer exceção não tratada, o programa é encerrado com o log de erro no console, onde o `toString()` do objeto é utilizado como mensagem descritiva.

Suas próprias exceptions

Embora seja possível disparar qualquer objeto como uma exceção, esta geralmente não é uma boa prática. Discutimos no início do capítulo justamente o significado de existirem exceções e erros, e a semântica que eles trazem no entendimento dos problemas com o código. Por isso, é muito mais comum e esperado utilizar esses próprios objetos para lançar uma exceção customizada. Por exemplo, para customizar uma exceção para que ela represente um problema do seu próprio projeto, basta implementar a interface da classe principal `Exception`.

```
class SemGasolinaException implements Exception {  
    final String mensagem;  
    const SemGasolinaException(this.mensagem);  
  
    String toString() => 'SemGasolinaException: $mensagem';  
}  
void ligarCarro() =>  
    throw SemGasolinaException('Carro sem gasolina..');  
void main() {  
    ligarCarro();  
}  
  
> Unhandled exception:  
> SemGasolinaException: Carro sem gasolina..  
> #0      ...stacktrace
```

Mais à frente nos aprofundaremos nos objetos e nos relacionamentos entre eles, incluindo interfaces e o significado de *implements*. Mas basicamente `SemGasolinaException` se torna também uma `Exception`, pois implementa sua interface. É muito comum uma *exception* ter um atributo de mensagem que será impresso no *console* explicando o motivo do problema.

5.3 Seus próprios errors? Lance um existente ou use assert!

Assim como é possível criar exceções customizadas, também é possível criar um *error* customizado estendendo da classe `Error`. Veja a implementação de `AssertionError`, que faz parte do SDK, por exemplo:

```
class AssertionError extends Error {
    final Object? message;
    AssertionError([this.message]);

    String toString() {
        if (message != null) {
            return "Assertion failed: ${Error.safeToString(message)}";
        }
        return "Assertion failed";
    }
}
```

Acontece que, na grande maioria dos casos em que é preciso lançar algum *error*, você poderá utilizar os já existentes na linguagem, pois eles cobrem quase todas as necessidades, principalmente o `StateError` ou o `AssertionError`.

Um `StateError` é um tipo de erro mais genérico que é utilizado em diversas situações pelo SDK, e representa um estado de erro do objeto ao tentar realizar alguma operação que no momento é inválida, como tentar acessar as propriedades `first` e `last` de uma lista vazia, por exemplo. Lançar um *error* faz muito mais sentido quando estamos criando trechos de códigos ou bibliotecas e queremos nos certificar de que o programador que for utilizá-lo faça de forma correta, e uma das opções para isso é o `assert()`, que dispara uma `AssertionError`.

Utilizando assert

O `assert` é um mecanismo de validação de expressões booleanas que interrompe a execução do código caso sua condição seja falsa. Pode ser considerado também uma ferramenta de *debug*, pois só funciona durante a execução do código em desenvolvimento, o que significa que todos os `asserts` são ignorados quando o programa está rodando em produção. Sua utilização é muito simples:

Mensagem descritiva opcional
assert(id != null, 'O id não pode ser nulo.');
Expressão booleana

Figura 5.1: Estrutura do assert.

Basta a expressão booleana resultar em falso, que será lançado um `AssertionError` com a mensagem informada.

```
void main() {  
    int? id;  
    assert(id != null, 'O id não pode ser nulo.');//  
}  
  
> Unhandled exception:  
> 'file:///C:/Users/julio/work/dart-  
book/pt_Br/05_capítulo/05_assert/main.dart': Failed > assertion:  
line 3 pos 10: 'id != null' : O id não pode ser nulo.  
> #0      _AssertionError._doThrowNew ...  
> #1      ...stacktrace
```

A expressão pode ser lida como: se o `id` for diferente de `null` lance um `AssertionError` com a mensagem '`O id não pode ser nulo`'. Com isso, ao executar o programa via linha de comando, é necessário habilitar os `asserts` passando a *flag* `dart run --enable-asserts <arquivo.dart>`. Caso seja executado através de alguma IDE, geralmente elas os habilitam por padrão.

Agora imagine que você criou um código com diversos cálculos matemáticos e quer disponibilizar para outras pessoas e, dentre eles, está o cálculo de uma circunferência através do raio de um círculo.

```
import 'dart:math';  
double calcularCircunferencia(double raio) {  
    assert(raio >= 0, 'O raio deve ser positivo.');//  
    return 2 * pi * raio;  
}
```

Como bem sabemos, não faz sentido calcular a circunferência de um círculo cujo raio seja negativo, então com o *assert* garantimos que esse parâmetro tem que ser positivo. Dessa forma uma pessoa desavisada que utilizar essa função com o parâmetro errado vai receber um aviso com o erro. Isso vai funcionar perfeitamente enquanto o programa está sendo desenvolvido, mas é válido lembrar que, quando compilado para rodar em produção, os *asserts* serão ignorados e a função estará suscetível a receber um parâmetro inválido. Então, se você quer garantir essa validação em produção, o *error* deve ser lançado manualmente:

```
import 'dart:math';
double calcularCircunferencia(double raio) {
    if (raio < 0)
        throw AssertionError('O raio deve ser positivo.');
    return 2 * pi * raio;
}
```

Note que a validação booleana muda e fica exatamente o oposto da anterior, uma vez que com *assert()* estamos validando o sucesso (como deve ser), enquanto com o *throw* validamos o erro (como não deve ser). Ambos também podem ser utilizados para validar dados de uma classe direto no seu construtor:

```
class Motorista {
    final String nome;
    final int idade;

    const Motorista(this.nome, this.idade)
        : assert(nome != '', 'O nome não pode ser vazio'),
        assert(idade >= 18, 'O motorista deve ser maior de idade');
}

class Motorista {
    final String nome;
    final int idade;

    Motorista(this.nome, this.idade) {
        if (nome.isEmpty)
            throw AssertionError('O nome não pode ser vazio');
```

```
    if (idade < 18)
        throw AssertionError('O motorista deve ser maior de idade');
}
}
```

A diferença é que com `assert` é possível criar as validações em um construtor `const`, o que não é possível com uma cláusula `throw`, pois esse não é um valor constante em tempo de compilação. Veremos mais sobre construtores nos próximos capítulos.

De forma resumida, se você quer criar validações para programadores que funcionem apenas durante o desenvolvimento ou validar parâmetros de classes enquanto pode aproveitar o uso de um construtor `const`, use `assert`. Agora se você deseja que essas validações garantam a utilização do código tanto em desenvolvimento quanto em produção, lance os `errors` com o `throw`.

5.4 Capturando exceções

Todas as exceções em Dart são *unchecked*, ou seja, você não é obrigado a declarar qual exceção um método pode lançar e nem mesmo tratar qualquer exceção que ocorra. Mas obviamente todo código é suscetível a exceções e um programa que não as trate adequadamente não vai fornecer uma boa experiência de usuário e nem mesmo passar confiança para quem quer que esteja utilizando.

Felizmente, somos bons programadores e programadoras e, com a ajuda de um simples bloco `try/catch`, conseguimos capturar as exceções. Ele possui a seguinte estrutura:

```
try {  
    // Código que está no escopo do bloco try  
} on String {  
    // Cláusula on pode definir qualquer objeto  
    // específico para ser capturado  
} on TimeoutException catch(e) {  
    // Junto com a cláusula on é possível definir  
    // um bloco catch para obter o objeto do erro  
} catch(e, s) {  
    // Um bloco catch sem on captura qualquer objeto  
    // lançado. Seu primeiro parâmetro é o objeto do  
    // erro e o segundo (opcional) é o stacktrace  
} finally {  
    // O bloco finally é opcional e executa sempre.  
    // independente de ocorrer erro ou não no try  
}
```

Figura 5.2: Anatomia de um bloco try/catch.

O nome `try/catch` é uma convenção, sendo que esta é uma funcionalidade presente na maioria das linguagens de programação, cujo objetivo é tentar executar algo (`try`) e capturar (`catch`) os eventuais problemas. Mas na prática, o bloco `try` deve estar acompanhado de pelo menos uma das cláusulas: `catch` , `on` ou `finally` .

O funcionamento do `catch` é bem explícito:

```

void comer() => throw Exception('Acabou a comida..');
void main() {
  try {
    comer();
    print('Não vai chegar aqui..');
  } catch(e, s) {
    print('Exceção capturada: $e');
    print('Stacktrace: $s');
  }
}

> Exceção capturada: Exception: Acabou a comida..
> Stacktrace: #0      comer...

```

Com ele, qualquer exceção **síncrona** lançada pelo código que está dentro do bloco `try` é capturada. E sim, erros assíncronos não podem ser capturados por blocos `try/catch`, mas isso não é problema para agora, é apenas para instanciar um alerta no seu cérebro pois ao longo do livro veremos como cuidar de problemas assíncronos.

O bloco de código dentro do `try` encerra o seu fluxo de execução imediatamente e o `catch` então recebe por padrão o objeto da exceção capturada (que por convenção é nomeado `e`), sendo que o tipo desse `e` vai depender exatamente de qual objeto foi lançado. O seu segundo parâmetro `s` é opcional e é um `Stacktrace`, com detalhes da árvore de execução dos métodos por onde o erro se propagou.

Especificando o tipo com `on`

Um bloco de código pode e geralmente lança diferentes exceções que por si só exigem diferentes comportamentos de resposta. Uma

`FormatoInvalidException`, por exemplo, faz sentido que seja tratada para informar o usuário para inserir um novo formato adequado para um campo, enquanto um `TimeoutException` pode exigir uma outra estratégia de recuperação, como tentar a mesma requisição dentro de alguns minutos. A cláusula `on` é quem permite especificar um determinado tipo de objeto para capturar:

```

void main() {
    try {
        DateTime.parse('01/01/2022');
    } on FormatException catch (e) {
        print('O formato deve ser ano-mes-dia.. $e');
    } catch (e, s) {
        print('Exceção capturada: $e');
        print('Stacktrace: $s');
    }
}
> O formato deve ser ano/mes/dia.. FormatException: Invalid date
format

```

Assim, exceções do tipo `FormatException` passam a ter um comportamento diferente em relação às demais, que serão capturadas normalmente pelo `catch` geral. Mas cuidado, em algum momento você poderá se deparar com esta cláusula `on Exception catch(e)` e achar que ela é geral e que vai capturar todas as exceções, só que não. Por exemplo:

```

void comer() => throw 'Acabou a comida..';
void main() {
    try {
        comer();
    } on FormatException catch (e) {
        print('O formato deve ser ano-mes-dia.. $e');
    } on Exception catch (e, s) {
        print('Exceção capturada: $e');
        print('Stacktrace: $s');
    }
}
> Unhandled exception:
> Acabou a comida..
> #0      comer...

```

O fato de ter uma cláusula `on Exception` pode induzir ao erro já que a tendência é assimilarmos que, por esta ser a classe pai, tudo será capturado. Mas lembre-se de que com o `throw` é possível lançar qualquer tipo de objeto, e o objeto `String` não é uma `Exception`. Sendo assim, para

capturar todos os objetos lançados, é possível apenas com o `catch` sem cláusula `on`, ou com `on Object`, já que valores nulos não podem ser lançados.

O `on` também pode ser utilizado com qualquer objeto, então usar `on String` para capturar este erro em específico seria bem válido.

Garantindo execução com `finally`

Por fim, a última cláusula `finally` é opcional e muito simples de entender. É um trecho de código que sempre vai executar, independente de o bloco de código principal dentro do `try` ter executado ou não, e pode funcionar como uma medida de segurança.

Imagine que estamos executando diversas tarefas dentro do bloco de código e para isso precisamos alocar alguns recursos, abrir alguma conexão com um banco de dados ou até mesmo uma *stream* de dados externa. E como bem sabemos é sempre boa prática com esses tipos de tarefas liberar os recursos após o seu uso (encerrar as conexões) para evitar qualquer problema de memória. Só que nada impede de nesse meio tempo ocorrer alguma exceção que encerre a execução padrão da sequência de código, não executando a liberação dos recursos. Imagine o cenário lúdico:

```
import 'dart:async';
void main() {
  try {
    abrirConexao();
    buscarDados();
    fecharConexao();
  } catch(e, s) {
    print('Exceção capturada: $e');
    print('Stacktrace: $s');
  }
}
void abrirConexao() => print('Conexão aberta..');
void buscarDados() => throw TimeoutException('Rede lenta..');
void fecharConexao() => print('Conexão fechada..');
```

```
Conexão aberta..  
Exceção capturada: TimeoutException: Rede lenta..  
Stacktrace: #0 ...
```

Supondo que uma conexão com algum banco de dados tenha sido aberta, da forma como foi codificada acima, qualquer erro que venha a acontecer após isso impede a liberação desse recurso, no caso, o fechamento da conexão. É algo que pode ser evitado com o uso do `finally`:

```
void main() {  
    try {  
        abrirConexao();  
        buscarDados();  
    } catch (e, s) {  
        print('Exceção capturada: $e');  
        print('Stacktrace: $s');  
    } finally {  
        fecharConexao();  
    }  
}
```

```
Conexão aberta..  
Exceção capturada: TimeoutException: Rede lenta..  
Stacktrace: #0 ...  
Conexão fechada..
```

Independentemente de qualquer erro que interrompa a execução normal do código, o `finally` sempre será executado. Assim, concluímos o entendimento de como prevenir erros síncronos na aplicação e podemos partir para um novo assunto: as *libraries*.

5.5 Se liga aí

- Por convenção, mantenha a nomenclatura padrão ao nomear os parâmetros de um `catch`, e sem tipá-los.

ruim	bom
-------------	------------

ruim	bom
<code>catch(error, stacktrace)</code>	<code>catch(e, s)</code>
<code>catch(Exception e, Stacktrace s)</code>	<code>catch(e, s)</code>

- Um `Error` pode mas não deve ser capturado, então não crie código para tratar um `error` no `try/catch`. Em vez disso, corrija a causa raiz do erro sempre que ele ocorrer.

5.6 É com você

1 - Em algumas situações dentro de uma árvore de métodos onde há um `try/catch` você pode desejar que um determinado tipo de exceção não seja capturado neste momento e, sim relançado para que seja capturado por algum outro método dessa árvore. Em Dart, isso pode ser feito com o comando `rethrow`. Pesquise e implemente um exemplo de uso.

Até aqui

Desvendamos as exceções que agora já não são mais aquele problema que causava medo ao ver no terminal. Aprendemos as diferenças semânticas entre um `error` e uma `exception`, algo específico do design de Dart, e que como bons desenvolvedores e desenvolvedoras devemos conhecer.

Por fim, descobrimos que um `assert` é uma ferramenta interessante para ser utilizada em nosso código, prevenindo possíveis situações de erro que outros desenvolvedores possam enfrentar em nosso código. Algo bastante comum na criação de bibliotecas, que inclusive são o assunto do próximo capítulo. Vamos lá!

CAPÍTULO 6

Entendendo as Libraries

Quando desenvolvemos uma aplicação (entenda por aplicação qualquer script que possua uma funcionalidade), passamos por diversos problemas pelos quais outras pessoas também já passaram e já resolveram. Afinal as aplicações possuem diversas características em comum, como imprimir mensagens na tela, salvar dados em um banco, criar arquivos, gerar relatórios, se comunicar com outros sistemas etc.

Imagine que você precisa agora implementar um relatório em PDF para o seu sistema. Por ser uma tarefa comum, provavelmente alguém que já implementou isso, abstraiu essa funcionalidade e extraiu todo o código necessário para se criar um PDF em uma biblioteca, de forma que você possa reutilizar. Daí vem aquela famosa frase que todo programador e programadora já ouviram: "Não vá reinventar a roda", ou seja, não precisa solucionar um problema já solucionado.

Sendo assim, uma biblioteca, ou *library*, em inglês, ou simplesmente *lib*, é um conjunto de funções, métodos, classes e tudo que uma determinada linguagem de programação permite criar. Funciona como um mecanismo utilizado para facilitar a modularização e reaproveitamento de código, sendo algo presente em todas as linguagens de programação, embora elas possam divergir um pouco no conceito dependendo da nomenclatura (bibliotecas, pacotes, módulos etc.).

Em Dart, esses conceitos de modularização, reaproveitamento e compartilhamento de código são divididos em *libraries* (bibliotecas) e *packages* (pacotes). Então abordaremos neste capítulo:

- O conceito de *library* em Dart;
- A utilização de diferentes *libraries*;
- O funcionamento da privacidade;
- Como separar uma *library* em diferentes arquivos.

6.1 Criando uma library

Toda aplicação durante o desenvolvimento ou até mesmo após sua publicação utiliza alguma espécie de *logger* para captura de informações que facilitem na depuração de algum problema. Em muitos casos são espalhados vários `print()` pelos códigos para algumas validações rápidas, e não é incomum esquecer de removê-los. Mas quando se precisa de algo mais profissional, com alguns recursos extras, sempre acabamos utilizando alguma implementação de *logger* mais robusta.

Nossa tarefa agora é criar uma *library* simples que servirá para exemplificação e ajudará no *log* de aplicações. Então indo direto ao ponto comece criando um arquivo `log.dart`. E pronto, a *library* já está criada, e não é brincadeira.

O ato de criar um arquivo em Dart é o mesmo que criar uma *library*, pois todo arquivo é considerado o escopo de uma *library*. Então é prática comum ter várias mini *libraries* na construção de uma aplicação modularizada. Entretanto, mesmo que opcional, uma *library* pode conter um *namespace* para identificação, que por convenção deve seguir o padrão `minuscuso_com_underscore`. Não necessariamente esse *namespace* precisa ser igual ao nome do arquivo, mas, se informado, o *namespace* deve ser a primeira sentença presente no arquivo.

```
library logger;  
//Implementação
```

Para início, podemos definir que nosso *logger* tratará de três níveis de *logs*: *info*, *warning* e *error*, que vão imprimir as mensagens de acordo com seu nível.

```
library logger;  
  
void info(Object object) => print('[INFO] $object');  
void warning(Object object) => print('[WARNING] $object');  
void error(Object object) => print('[ERROR] $object');
```

É comum falarmos que toda *library* disponibiliza teoricamente uma API de funcionalidades para ser utilizada por outras *libraries*. Nesse caso, estamos disponibilizando uma API com três funções de alto nível para serem acessadas de fora do escopo de nossa *lib*, desacoplando funcionalidades.

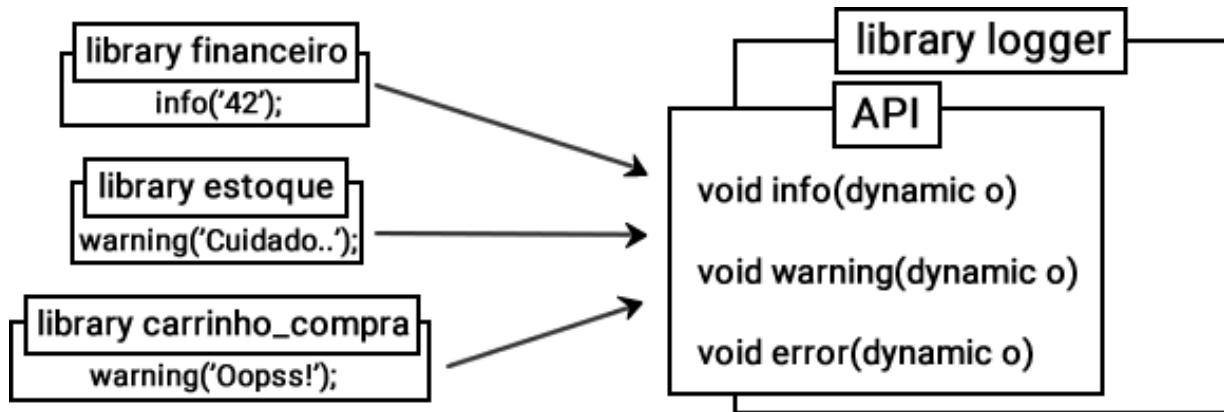


Figura 6.1: API teórica da library.

6.2 Utilizando outras libraries

Uma *library* pode importar e utilizar recursos de outras *libraries* através do comando `import`, que inclusive até este ponto do livro já utilizamos bastante com os scripts criados, afinal, como já bem sabemos, todo código em Dart é obrigatoriamente organizado em *libraries*. Mas se você prestou atenção em alguns exemplos ao longo do livro, não importamos nada. Por exemplo, neste código do capítulo 4 quando estávamos explorando as funções:

```
Function ola = (String nome) => print('Olá $nome');
void main() {
  ola('${ola.runtimeType}');
}
```

Sem qualquer `import`, conseguimos utilizar os tipos `Function` e `String` além da função `print`. Isso porque eles fazem parte da *library* `dart:core`, que é importada por padrão em qualquer script em Dart. Ela contém as implementações que são consideradas principais para o SDK, como os tipos

estudados no capítulo 3, as *exceptions* e *errors* padrões, as *collections*, além de outros recursos que são julgados como parte do *core* da linguagem.

Mas além de `dart:core`, o SDK vem com uma série de outras *libraries* disponíveis para utilização, algumas são multiplataformas, outras são feitas especificamente para Dart nativo ou web. A seguir, veja as *libs* presentes no SDK:

Multiplataformas:

- **dart:core:** lib principal, importada por padrão em qualquer script.
- **dart:collection:** possui classes que complementam o suporte a *collections*, além do que está presente em `dart:core`.
- **dart:async:** dá suporte para programação assíncrona.
- **dart:developer:** possui APIs para comunicação com ferramentas de desenvolvimento como o Dart *debugger* ou *inspector*.
- **dart:convert:** possui conversores (*encoders* e *decoders*) de dados em diferentes tipos, como JSON ou UTF-8.
- **dart:typed_data:** possui classes utilitárias para trabalhar com listas de tamanho fixo, como a `Uint8List`, que representa 8-bit, e é muito utilizada para trabalhar com *bytes*.
- **dart:math:** dá suporte a operações matemáticas e valores constantes.

Plataforma web:

- **dart:html:** dá suporte para utilização dos recursos e elementos presentes no HTML. Utilizamos esta *lib* na construção do jogo "Pedra, Papel, Tesoura, Lagarto e Spock".
- **dart:js_util:** contém algumas funcionalidades úteis que não estão presentes em `dart:html`.
- **dart:web_gl:** contém recursos para programação 3D no browser.
- **dart:web_audio:** contém recursos para trabalhar com áudio no browser.
- **dart:indexed_db:** dá suporte para armazenar dados com chave/valor no *client side*, com suporte a indexes.

Plataformas nativas:

- **dart:io**: possui operações I/O, manipulação de arquivos, protocolo HTTP e sockets.
- **dart:isolate**: dá suporte para concorrência com *isolates*, as *threads* de Dart.
- **dart:mirrors**: dá suporte básico para utilização de *reflections* em Dart.

O primeiro tipo de *import* existente é então quando precisamos utilizar algum recurso dessas *libs* do SDK que não estejam em `dart:core`. Vamos modificar nossa *library* para imprimir os *logs* através do *standard output*, que costuma ser o padrão de escrita de dados em aplicações nativas e está presente em `dart:io`.

```
library logger;
import 'dart:io';

void info(Object object) => stdout.writeln('[INFO] $object');
void warning(Object object) => stdout.writeln('[WARNING] $object');
void error(Object object) => stdout.writeln('[ERROR] $object');
```

Todo comando de `import` deve estar no início do arquivo antes de qualquer código, e após a diretiva da `library` (se houver). O *namespace* `dart:` indica que a *library* está presente no SDK, o que as difere de uma *library* criada fora dele.

O objeto `stdout` garante o acesso ao *standard output* presente no sistema operacional em que o código estiver rodando, para que possamos imprimir os logs. E para testar nossa *library* chegamos ao segundo tipo de *import*: utilizando caminho relativo. Considere que esta é a nossa estrutura atual de diretórios:

```
+-- lib
|   main.dart
\-- log
    |   logger.dart
    \-- exemplo
        outra_lib.dart
```

Para utilizar a *library* dentro de `main.dart` bastaria definir no `import` o caminho relativo `log/log.dart`:

```
import 'log/log.dart';

void main() {
  error('Este é um erro');
  warning('Este é um warning');
  info('Esta é uma info');
}

> [ERROR] Este é um erro
> [WARNING] Este é um warning
> [INFO] Esta é uma info
```

Isso garante que dentro de `main.dart` toda a API de `log.dart` está agora acessível. Utilizamos o `main.dart` para exemplificar e conseguir executar a *lib*, mas qualquer outro arquivo dentro do diretório `lib` poderia importar `log.dart` através de um caminho relativo. Se importássemos em `outra_lib.dart`, de acordo com o diretório anterior, seria: `import '../log.dart';`.

Sendo assim, toda *library* pode importar N outras *libraries*, incluindo de forma cíclica, onde uma *library* `a` importa uma `b`, e a `b` também importa `a`, sem qualquer problema. E desde que esses arquivos estejam dentro do diretório `lib`, ou seja, *libraries* que fazem parte do seu próprio projeto, faz parte das boas práticas que elas sejam importadas utilizando este padrão de caminho relativo.

O terceiro e último tipo de *import* é feito através da URI do *package*, como `import 'package:http/http.dart'`, por exemplo, onde referenciamos uma *library* `http.dart` de um pacote externo denominado `http`. Ele é utilizado para referenciar *libraries* externas e discutiremos isso no próximo capítulo.

Criando um alias

Ao importar uma *library*, é possível especificar um alias ou apelido utilizado para referenciar os recursos dela, o que é feito com o comando `as` :

```
import 'log/log.dart' as logger;
```

Assim, todos os recursos de `log.dart` são acessíveis apenas referenciando através deste alias `logger` :

```
void main() {  
    logger.error('Este é um erro');  
    logger.warning('Este é um warning');  
    logger.info('Esta é uma info');  
}
```

Isso se dá porque diferentes *libraries* podem ter recursos com nomes iguais, o que causaria um conflito caso ambas fossem importadas em um mesmo arquivo, resultando em um erro. Nesse caso, uma das duas precisaria obrigatoriamente de um alias.

Personalizando os recursos importados

O comando `import` também pode ser utilizado em conjunto com os comandos `show` e `hide`. Eles permitem personalizar os recursos que são importados de uma *library*. Por exemplo, se dentre todos os recursos disponíveis em `log.dart` você quer importar apenas a função `error()`, então use `show` :

```
import 'log/log.dart' show error;
```

É possível ainda combinar com `as` e listar múltiplos recursos, como disponibilizar apenas as funções `error()` e `warning()` .

```
import 'log/log.dart' as logger show error, warning;
```

Já o modificador `hide` faz o contrário do `show` , permitindo esconder recursos ao importar.

```
import 'log/log.dart' hide info;
```

No exemplo estamos importando todos os recursos, exceto a função `info()`.

6.3 Privacidade em libraries

Atualmente estamos diferenciando os tipos de *logs* da nossa *library* apenas com o nome entre colchetes. Mas seria interessante também poder imprimir no console cada tipo com uma cor diferente, assim eles se destacam mais e ficam mais fáceis de identificar. Felizmente, existem os chamados códigos ANSI, onde conseguimos modificar propriedades do terminal, como a cor de impressão do texto.

Através de Dart conseguimos alterar a cor de impressão do terminal com o comando `\x1b[31m`, onde `31` é um número de identificação para uma cor, sendo este o vermelho. Isso nos possibilita alterar as funções para:

```
void error(Object object) {
    stdout.writeln('\x1b[31m[ERROR] $object\x1b[m');
}
```

Onde, para cada chamada a `error()`, o comando no início `\x1b[31m` alterará a cor para vermelho, imprimirá a mensagem com `[ERROR]` `$object`, e no final o comando `\x1b[m` sem o código numérico resetará o terminal para a sua cor padrão. Execute essa função para ver o resultado das mensagens imprimindo em vermelho.

Para não ter que repetir a mesma coisa para as funções `warning()` e `info()`, e também para organizar melhor o código da nossa *library*, vamos refatorar essa funcionalidade.

```
library logger;
import 'dart:io';

enum Cores { verde, vermelho, azul }
const resetarCor = '\x1b[m';
const ansiCores = {
    Cores.vermelho: '\x1b[31m',
    Cores.verde: '\x1b[32m',
    Cores.azul: '\x1b[34m'}
```

```
    Cores.verde: '\x1b[32m',
    Cores.azul: '\x1b[36m',
};

void info(Object object) => log(Cores.verde, '[INFO] $object');
void warning(Object object) => log(Cores.azul, '[WARNING]
$object');
void error(Object object) => log(Cores.vermelho, '[ERROR]
$object');

void log(Cores cor, Object object) {
    stdout.writeln('${ansiCores[cor]}$object$resetarCor');
}
```

Desta forma, separamos melhor as responsabilidades e deixamos o código mais organizado. A nova função `log` é o que de fato vai imprimir os valores no terminal e modificar a cor do texto de acordo com o seu parâmetro. As chamadas para as funções principais que já existiam continuam as mesmas, então os usuários desta `library` podem continuar chamando `info()` ou `error()` como anteriormente.

Porém, note que adicionamos mais informações para a API da `library`. Agora os usuários possuem acesso à nova função `log()`, ao `enum` de cores, e às constantes de código ANSI. E se pararmos para analisar o objetivo da nossa `lib`, que é apenas imprimir *logs* em diferentes níveis, essas novas informações não precisam estar publicamente na sua API disponibilizada.

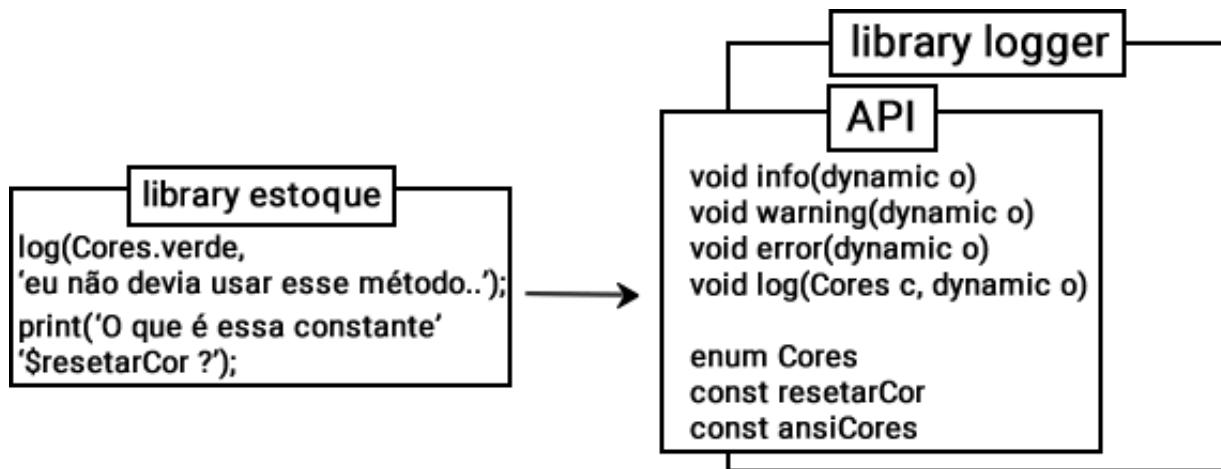


Figura 6.2: API com informações desnecessárias.

Em outras palavras, para quem vai usar esta *lib* não interessa como ela vai fazer para modificar a cor do terminal, muito menos que existem esses códigos ANSI. O que é necessário saber é que: se chamar a função `error()`, o objeto informado será impresso no *log* em vermelho. Toda a informação de como alterar a cor impressa no terminal é privada e pertence apenas ao escopo da nossa *lib*.

E é neste momento que entra o conceito de privacidade em Dart (privacidade em si é um conceito de Orientação a Objetos e encapsulamento, assunto para os próximos capítulos). Mas, diferente de outras linguagens, como o Java, onde a privacidade é em nível de classe, a privacidade em Dart é em nível de *library*, significando que todo código dentro de uma *library* é acessível dentro dela mesma.

Para indicar que um recurso não deve ser visível fora da *library*, ele pode ser anotado com um `_` no início da sua nomenclatura, por exemplo, não queremos que os usuários tenham acesso ao nosso *enum* de cores, às constantes com os códigos ANSI, e nem à função `log()`:

```

enum _Cores { verde, vermelho, azul }
const _resetarCor = '\x1b[m';
const _ansiCores = {
    _Cores.vermelho: '\x1b[31m',
    _Cores.verde: '\x1b[32m',
    _Cores.azul: '\x1b[36m',
}

```

```

};

void _log(_Cores cor, Object object) {
    stdout.writeln('${ansiCores[cor]}$object$resetarCor');
}

```

Agora os recursos se tornaram privados e não são mais acessíveis externamente pela API da nossa *library*. A tentativa de usá-los externamente vai resultar em um erro de compilação.

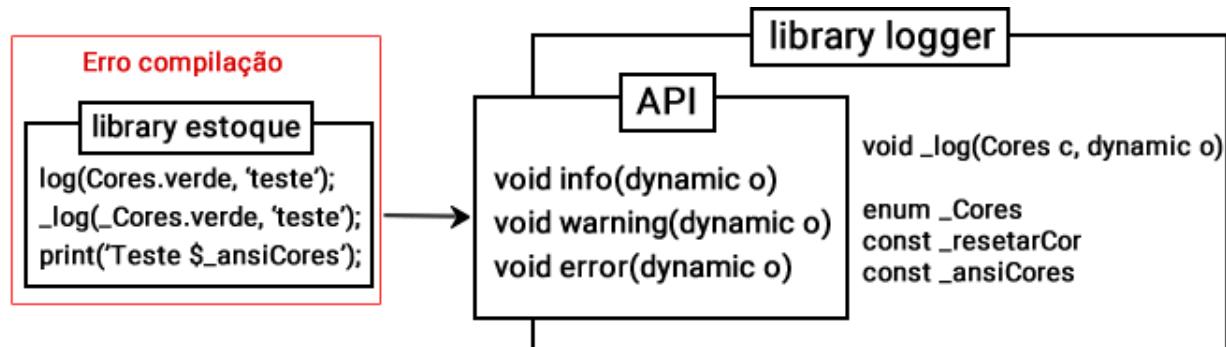


Figura 6.3: API com recursos privados.

Eles continuam acessíveis normalmente dentro da própria *library*, precisando alterar apenas suas chamadas para conterem o `_`. Já o acesso externo (em outros arquivos) desses recursos não é mais possível.

E não são somente os recursos de alto nível do arquivo que podem ser definidos como privados, membros de classes ou até mesmo uma classe também podem ser privados, como veremos agora na implementação da próxima funcionalidade para nossa *lib*.

Geralmente, as *libs* de *logger* existentes costumam permitir definir quais os níveis de *logs* que serão capturados e impressos dependendo do modo em que a aplicação está rodando. Por exemplo, durante o desenvolvimento da aplicação é interessante imprimir todos os níveis, ao contrário de quando a aplicação está em produção onde é comum imprimir apenas os *logs* mais críticos, como de erro.

Sendo assim, quanto mais crítico o nível do *log* menos níveis são impressos. Se rodarmos a aplicação com o nível `warning` habilitado, serão impressos todos os *logs* de `warning` e `error`. Se definirmos como `info`,

todos são impressos, e como `error` apenas *logs* de `error` serão impressos. Para isso, precisamos encapsular as funções em uma classe:

```
enum Nivel { info, warning, error }

class Logger {
  const Logger({required this.nivel});
  final Nivel nivel;

  void info(Object object) {
    if (_habilitado(Nivel.info))
      _log(_Cores.verde, '[INFO] $object');
  }

  void warning(Object object) {
    if (_habilitado(Nivel.warning))
      _log(_Cores.azul, '[WARNING] $object');
  }

  void error(Object object) {
    if (_habilitado(Nivel.error))
      _log(_Cores.vermelho, '[ERROR] $object');
  }

  bool _habilitado(Nivel nivelHabilitado) =>
    nivelHabilitado.index >= nivel.index;
}
```

É através da classe `Logger` agora que os usuários da *library* vão imprimir os *logs*. Ela recebe um enum `Nivel` pelo construtor que vai definir quais os níveis habilitados para impressão. E é com o novo método `_habilitado()` do código anterior, que é validado se um determinado nível será ou não impresso - note que este é um método privado, acessível apenas dentro da nossa *library*. Para utilizar esta nova classe, basta instanciá-la com o nível desejado:

```
void main() {
  const logger = Logger(nivel: Nivel.warning);
  logger.error('Este é um erro');
  logger.warning('Este é um warning');
```

```
    logger.info('Esta é uma info');
}
```

Ao utilizar o `Nivel.warning`, apenas as mensagens de `error` e `warning` serão impressas pelo `logger`, desabilitando a impressão de `info`. Uma outra funcionalidade interessante para nossa *library* é permitir a customização da impressão de todas as mensagens através de uma espécie de `Printer` (impressora), então adicione esta nova classe ao mesmo arquivo `log.dart`:

```
class Printer {
  const Printer({this.inicio = ' ', this.fim = ''});
  final String inicio;
  final String fim;

  void _log(_Cores cor, Object object) {
    stdout.writeln(
      '${_ansiCores[cor]}'
      '$inicio$object$fim'
      '_resetarCor',
    );
  }
}
```

A função `_log()` que já existia tornou-se um método dentro de `Printer`, afinal agora esta é a responsável na nossa API pela impressão das mensagens. No construtor, ela pode receber dois parâmetros opcionais para customizar o início e fim de cada mensagem, que podem servir para formatação ou identificação das mensagens em meio a todos os outros *logs* que a aplicação possa imprimir.

Agora podemos fazer com que o `Logger` use um `Printer`:

```
class Logger {
  const Logger({
    required this.nivel,
    this.printer = const Printer(),
  });

  final Printer printer;
```

```

final Nivel nivel;

void info(Object object) {
    if (_habilitado(Nivel.info))
        printer._log(Cores.verde, '[INFO] $object');
}
//demais funções
}

```

Caso não seja informado um `Printer` customizado, criamos um padrão sem informar os parâmetros de início e término das mensagens. E alteramos as funções para chamarem o método `_log()` do `Printer` com `printer._log()`. Este é um ponto interessante a ser notado, que pode confundir desenvolvedores e desenvolvedoras com um *background* em linguagens onde a privacidade é um conceito da classe. Dentro de `Logger`, estamos acessando um método privado de `Printer`, ficando claro que para Dart a privacidade está presente no escopo de *library*, e não de classes. Para customizar a impressão então, basta informar um `Printer`:

```

void main() {
    const logger = Logger(
        nivel: Nivel.error,
        printer: Printer(inicio: 'Customizado: {', fim: '}'),
    );
    logger.error('Este é um erro');
}

> Customizado: {[ERROR] Este é um erro}

```

6.4 Separando libraries em arquivos

Sabemos que o escopo de uma *library* é o seu próprio arquivo `.dart`, fazendo com que tudo o que esteja dentro dele forme seu conjunto de recursos disponíveis. Entretanto, também sabemos que este conjunto de recursos (classes, métodos, tipos, funções, enums etc.) pode crescer muito e deixar o arquivo extenso. E quanto maior, mais complexo e difícil de manter.

E é por isso que Dart possui as diretivas `part` e `part of` para a separação de uma mesma *library* em diferentes arquivos. Na nossa *library*, por exemplo, possuímos o objeto `Printer`, que é o responsável por imprimir os *logs* em algum local, que por padrão é o terminal. Futuramente poderíamos também criar outros tipos de *printers* que fossem especialistas em imprimir em arquivos de texto, em servidores etc.

Então, para uma melhor manutenibilidade do código podemos extrair esta classe para um arquivo `printer.dart`:

```
part of 'log.dart';

class Printer {
    // Implementação classe
}
```

`part of 'log.dart'`; indica que esse arquivo faz parte da *library* do arquivo `log.dart`. Já em `log.dart` devemos adicionar a diretiva `part`:

```
library logger;
import 'dart:io';

part 'printer.dart';
// Demais recursos
```

E `part 'printer.dart'`; indica que o arquivo `printer.dart` faz parte da *library* deste arquivo.

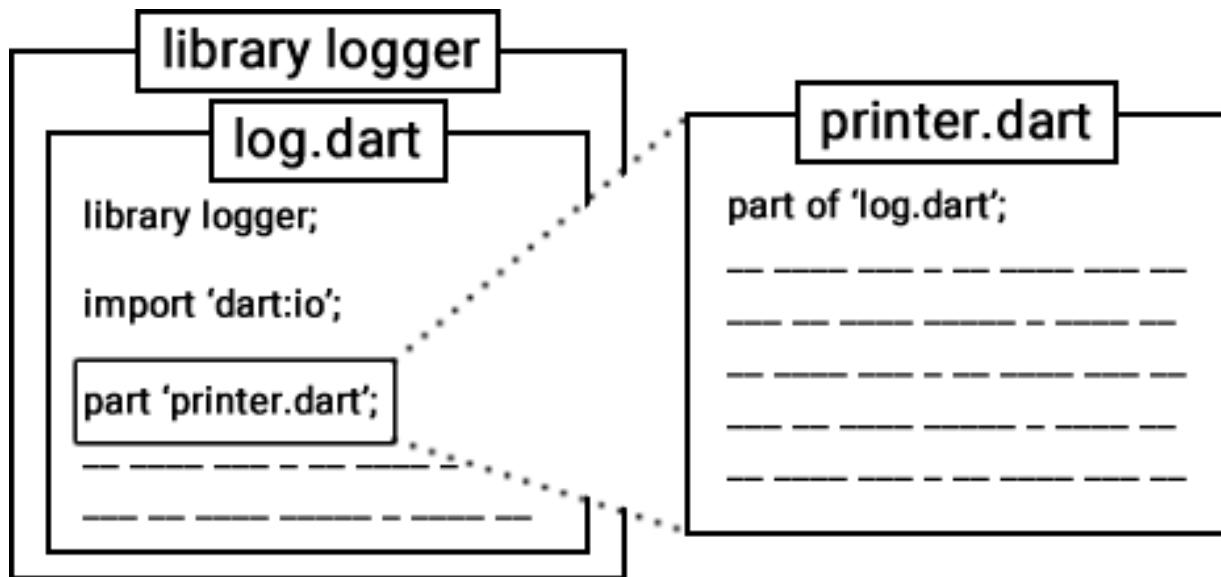


Figura 6.4: Library separada em arquivos diferentes.

Ambos os arquivos ficam conectados de forma que possam utilizar os recursos declarados nos dois, incluindo recursos privados ao acesso externo da *library*. Na prática, é como se todo o código de `printer.dart` ainda continuasse dentro de `log.dart`.

Um arquivo pode fazer parte de (`part of`) apenas uma única *library*, o que torna o compartilhamento de arquivos entre *libraries* proibido. Da mesma forma, esse arquivo que já faz parte de outra *library* (como o `printer.dart`) não pode definir outros arquivos que fazem parte (`part of`) de `printer.dart`, e nem definir qualquer outro tipo de *import*. Todos os *imports* necessários precisam estar declarados no arquivo original da *library*, no nosso caso o `log.dart`.

Esse arquivo original, por outro lado, pode definir vários outros arquivos que fazem parte dele com o comando `part`:

```

library logger;
import 'dart:io';

part 'printer.dart';
part 'outro_arquivo.dart';
part 'mais_um.dart';
// Demais recursos

```

Um bom exemplo são as *libraries* que fazem parte do SDK. Elas se dividem em vários arquivos menores, por exemplo, este trecho do código de `dart:core` :

```
library dart.core;
// Imports
part "annotations.dart";
part "bigint.dart";
part "bool.dart";
part "comparable.dart";
part "date_time.dart";
// Demais arquivos
```

Devo separar todas libraries em arquivos?

Conhecer essa possibilidade de utilizar o `part of` e ver que as próprias *libraries* do SDK utilizam faz parecer ser uma boa ideia sair separando o escopo das *libraries* em vários arquivos. Mas não é bem assim, as boas práticas no desenvolvimento de aplicações em Dart recomendam o contrário.

Quando estamos desenvolvendo uma aplicação, é natural termos diversos arquivos e consequentemente diversas *libraries*, e o ideal é agrupar recursos semelhantes na mesma *library*, mas sempre dando preferência na divisão de responsabilidades e criação de mini *libraries* (arquivos diferentes).

Agrupar arquivos em uma única *library* é um recurso muito bom especificamente para as *libraries* do SDK, e é por isso que elas usam, mas não se aplica muito no nosso desenvolvimento do dia a dia. Um outro caso de uso específico onde isso é válido e bastante usado é nos geradores de código.

Esse não é um assunto que será abordado agora, mas basicamente existem formas de criar scripts que geram código em Dart através de alguns templates. Alguns *packages* famosos que utilizam esse recurso são o `json_serializable`, que gera código para serialização e deserialização de JSON, e o `mobx`, utilizado para gerenciamento de estados da aplicação. O importante para o momento é que você já possui todo o conhecimento

necessário para controlar suas próprias *libraries* e relacionar umas com as outras da melhor maneira que o seu projeto necessite, e seguindo as boas práticas. O que nos leva ao próximo assunto, totalmente correlacionado ao conceito das *libraries*: os *packages*.

6.5 Se liga aí

- Respeite a ordem das diretivas de *import* definidas nas boas práticas da linguagem. Utilizar uma IDE auxilia essa organização de forma automática, que deve ser:
 1. O namespace da *library*.
 2. *Import* das *libraries* do SDK.
 3. *Import* com *package* completo.
 4. *Import* com caminho relativo.
 5. *Libraries* exportadas.

```
library logger;

import 'dart:html';
import 'package:exemplo/exemplo.dart';

import 'src/printer.dart';
import 'src/log.dart';

export 'src/log.dart';
```

- Procure manter na API pública da sua *library* apenas o que é pertinente e deva ser acessível externamente.

6.6 É com você

1 - Dart possui uma opção de carregamento *lazy* de *libraries* disponível apenas para a web ao compilar com `dart2js`. Pesquise como isso pode ser

feito.

2 - Utilize sua IDE e navegue pelas libs nativas de Dart, você vai encontrar diversas informações e começar a se familiarizar com a forma de organização delas. São bons exemplos porque em um mesmo arquivo você vai encontrar diversas funções, diferentes classes, types e tudo o que for pertinente para a *library*, incluindo recursos privados.

Até aqui

Você já tinha o conhecimento de que todo arquivo em Dart era de fato considerado uma *library*? Isso é algo que até mesmo pessoas que programam com a linguagem comumente não sabem. Isso é algo essencial de se conhecer pois influencia diretamente no funcionamento do conceito de privacidade, que é diferente de linguagens como Java, onde a privacidade é por classes.

Também vimos quais *libraries* vêm por padrão junto com o SDK, acessíveis através do namespace `dart:`, e como elas podem ser organizadas em diferentes arquivos. A seguir, veremos como essas mesmas *libraries* podem ser organizadas e compartilhadas entre diferentes projetos através dos *packages*.

CAPÍTULO 7

Na prática - Packages

Uma vez que a nossa *library* está pronta e nós nos preocupamos com reaproveitamento de código, seria muito interessante poder reutilizá-la em diferentes aplicações sem ter que copiar esses arquivos para todos os projetos. Esse é um conceito dentro do ecossistema de Dart conhecido como um *package*.

É muito comum as pessoas acabarem confundindo e acharem que uma *library* e um *package* são a mesma coisa, justamente porque essa nomenclatura pode até ser igual em outras linguagens, mas aqui em Dart existe essa distinção. Por isso, durante este capítulo, vamos:

- Ver o que são os packages;
- Criar e utilizar um package na prática;
- Entender a importância de um *linter*;
- Importar libraries de forma dinâmica;
- Aprender a criar uma documentação padrão.

7.1 Os packages

Um package é justamente um conjunto de N *libraries* que ficam disponíveis para serem importadas em qualquer projeto. É uma forma de você empacotar todos os seus arquivos e conseguir reutilizá-los, ou até mesmo compartilhar para que outras pessoas desenvolvedoras também possam utilizá-los. Esse é justamente o propósito do website <http://pub.dev>, que funciona como um diretório oficial dos packages desenvolvidos pela comunidade.

O que vai determinar se esse conjunto de *libraries* é um package é a existência de um arquivo denominado `pubspec.yaml`. Esse é o principal responsável pelo controle de dependências, gerenciamento de versões e

definição de vários metadados para o projeto. A estrutura mais básica de um package seria:

```
\---meu_package
  |  pubspec.yaml
  \---lib
        meu_package.dart
```

Existem dois tipos de *packages*:

- Application package : um *package* sem intenção de ser usado como *library* por outros projetos. Existe apenas para ser executado como uma aplicação independente. Pode ter normalmente dependências de outros *packages*, mas nenhum outro terá este como dependência. O nosso projeto do jogo do capítulo 4 se encaixa nesta categoria.
- Library package : é o oposto do *application package*. A criação deste package parte do princípio de que ele será utilizado como dependência por outras aplicações.

O arquivo `pubspec` utilizado segue o padrão de formatação YAML e por conta disso a indentação das diretivas é extremamente importante para que as coisas funcionem. E é através dessas diretivas que algumas coisas são configuradas, por exemplo, este `pubspec.yaml` do *package* `http` :

```
name: http
version: 0.13.3
homepage: https://github.com/dart-lang/http
description: A composable, multi-platform, Future-based API for
HTTP requests.

environment:
  sdk: '>=2.12.0 <3.0.0'

dependencies:
  async: ^2.5.0
  http_parser: ^4.0.0
  meta: ^1.3.0
  path: ^1.8.0
  pedantic: ^1.10.0
```

```
dev_dependencies:  
  fake_async: ^1.2.0  
  shelf: ^1.1.0  
  test: ^1.16.0
```

Nessa configuração:

- **name**: define o nome do package, aquele que utilizamos para referenciar as libraries no `import`, como `import package:http/http.dart``.
- **version**: apresenta a versão do package. Caso você deseje publicar o package no <http://pub.dev>, este preenchimento é obrigatório.
- **homepage**: para os packages publicados, é normal informar uma URL de homepage do projeto.
- **description**: uma descrição sucinta do package.
- **environment**: todo package possui uma dependência direta com o próprio SDK de Dart, e é através deste campo que são definidas as versões de Dart que o package suporta. No exemplo, é suportado entre as versões `2.12.0` e `3.0.0`.
- **dependencies**: contém as dependências deste package, ou seja, todos os outros packages que são necessários para a criação e execução deste.
- **dev_dependencies**: dependências que são necessárias apenas para a fase de desenvolvimento e testes do package, não acompanham o *build* final. O uso mais comum é o próprio package `test` para criação de testes unitários.

Existem diversas outras diretivas e regras associadas ao arquivo que fogem do escopo de detalhamento neste momento. Consulte a documentação para conhecer as possibilidades.

7.2 Criando o package de logger

Agora que possuímos a teoria, podemos partir para a prática e transformar nossa library do capítulo anterior em um package. Novamente podemos

utilizar o próprio SDK para criar um projeto baseado em um template, então navegue até o diretório desejado e execute:

```
dart create -t package-simple logger && cd logger
```

Dessa vez o template utilizado é o `package-simple`, que cria a estrutura e os arquivos básicos de um package. Como informamos no comando para criar a estrutura e os arquivos dentro de um diretório `logger`, concatenamos o comando `&& cd logger` para que, ao terminar a criação, o terminal acesse este novo diretório.

Por padrão, o `dart create` já executa internamente o comando `pub get` para a atualização das dependências. Por isso, temos os seguintes diretórios e arquivos criados:

```
\---logger
|   .gitignore
|   .packages
|   analysis_options.yaml
|   CHANGELOG.md
|   pubspec.lock
|   pubspec.yaml
|   README.md
+---.dart_tool
|       package_config.json
+---example
|       logger_example.dart
+---lib
|   |   logger.dart
|   \---src
|       logger_base.dart
\---test
    logger_test.dart
```

Esse é o conjunto básico de diretórios e arquivos que um package em Dart geralmente possui, onde:

- **package_config.json** e **.packages**: são arquivos utilizados pelo pub para controle dos diretórios em que se encontram as dependências baixadas na máquina local, com informações de versões. O primeiro

veio para substituir o segundo, então é provável que futuramente .packages deixará de existir.

- **pubspec.yaml**: as configurações e dependências imediatas do package.
- **pubspec.lock**: arquivo que lista todas as dependências imediatas e transitivas do package, com informações das versões suportadas. Chamamos de dependência imediata os packages que declaramos em nosso pubspec.yaml, já as transitivas são dependências indiretas. Ou seja, se dependemos do package A e esse por sua vez depende do B, A é direta e B é indireta.
- **.gitignore**: arquivo já criado com alguns padrões de arquivos e diretórios que não devem ser versionados.
- **README.md**: arquivo para escrita no padrão *markdown*, utilizado para introduzir o que é e como utilizar o package. É listado na página inicial do website pub.dev, para os packages publicados.
- **CHANGELOG.md**: assim como o README.md, este também é listado no pub.dev e contém o *changelog* das modificações de acordo com as diferentes versões.
- **analysis_options.yaml**: arquivo que contém regras de *linter* para boas práticas em Dart.
- **example/**: apesar de não obrigatório, é normal os packages do tipo library conterem este diretório com um exemplo prático de como utilizar o package.
- **teste/**: diretório onde ficam os testes unitários do projeto.
- **lib/**: diretório onde ficam os arquivos .dart de implementação do package.
- **lib/src**: é convenção colocar as libraries privadas neste diretório para controlar o acesso externo de outros packages.

E além destes, seguem alguns diretórios opcionais que você pode criar ou encontrar por aí:

- **bin/**: utilizado para definir scripts que são públicos e podem ser executados externamente, é comum em aplicações CLI.
- **tool/**: alguns packages de grande porte ou mais complexos, podem exigir alguns scripts ou ferramentas necessárias para sua execução e

desenvolvimento, como essas são privadas, devem estar neste diretório.

- **web/**: arquivos utilizados em projetos web.
- **doc/**: arquivos de documentação do projeto.

Com isso, conseguimos iniciar a customização do package. Então comece removendo os arquivos `logger_example.dart`, `logger_base.dart` e `logger_test.dart` gerados automaticamente, além de limpar o conteúdo de `logger.dart`, assim o preencheremos com nosso próprio código.

Modificando o `pubspec.yaml`

Por ser o arquivo mais importante, é o primeiro que vamos modificar:

```
name: logger
description: Um package para facilitar a criação de logs nas aplicações.
version: 1.0.0

environment:
  sdk: '>=2.16.0 <3.0.0'

dev_dependencies:
  lints: ^1.0.0
  test: ^1.16.0
```

Você pode personalizar conforme achar necessário. Vamos iniciar o package já na versão `1.0.0`, embora seja uma boa prática iniciar os packages com um versionamento menor, pois é normal ocorrerem mudanças e melhorias até que ele atinja sua versão `1.0.0` estável.

É interessante saber que Dart utiliza o que chamamos de versionamento semântico, onde os números são caracterizados como `MAJOR.MINOR.PATCH`. Para saber mais sobre seu funcionamento, visite o endereço <https://semver.org/lang/pt-BR/>.

Não possuímos nenhuma dependência direta que seja necessária após o build do package, por isso não precisamos definir a diretiva `dependencies`. Por outro lado, possuímos `dev_dependencies` para os packages `test` e

`pedantic`. O primeiro, `test`, é utilizado para criação de testes unitários e é comum todo *package* já o definir como dependência. O segundo, `lints`, é utilizado para definir regras de *linter*, que veremos em breve.

Como esse arquivo reflete as dependências do projeto, sempre que ele é modificado precisamos executar novamente o comando `dart pub get`, assim o SDK vai atualizar os metadados e as dependências.

Organizando os arquivos existentes

Quando criamos a library no capítulo anterior, havia dois arquivos, `log.dart` e `printer.dart`. Copie ambos para dentro do diretório `lib/src`. Atualmente esses arquivos fazem parte de uma mesma library denominada *logger*, mas se lembarmos de que um package é composto de N libraries e de que o ideal é sempre separarmos nosso código em mini *libraries*, podemos separar estes dois arquivos em duas libraries.

Começando pelo `printer.dart`:

```
import 'dart:io';
import 'log.dart';

class Printer {
    const Printer({this.inicio = '', this.fim = ''});
    final String inicio;
    final String fim;

    void log(Cores cor, Object object) {
        stdout.writeln(
            '${ansiCores[cor]}'
            '$inicio$object$fim'
            '$resetarCor',
        );
    }
}
```

- Removemos a diretiva `part of` para separar `printer.dart` do arquivo `log.dart`, agora ele passa a ter sua própria library.
- Como ele deixou de ser parte de outro arquivo, é preciso realizar os imports de `dart:io` e `log.dart`.

- O método `log()` precisa deixar de ser privado para ser acessível externamente.

Já em `log.dart` :

```
import 'printer.dart';

// Demais códigos
class Logger {
  //...
  void error(dynamic object) {
    if (_habilitado(Nivel.error))
      printer.log(Cores.vermelho, '[ERROR] $object');
  }
  //...
}
```

- Removemos a diretiva `library logger;` para manter esta library com o padrão sem nome, uma vez que não é obrigatório.
- Removemos a diretiva `part` que fazia referência ao arquivo `printer.dart`.
- Inserimos o import de `printer.dart` uma vez que precisamos utilizar o objeto `Printer`. Todas as chamadas para o método `_log()` foram alteradas para `log()`, que agora é público.

7.3 Exportando libraries

Uma vez que os packages são separados em várias mini *libraries* e todo o conteúdo de `lib/src` é privado, todo package deve possuir pelo menos uma library pública e que de preferência exporte o acesso para as outras libraries dentro de `lib/src`. Como ela é pública, deve ficar apenas no diretório `lib`, então podemos utilizar o arquivo `lib/logger.dart` :

```
export 'src/log.dart';
export 'src/printer.dart';
```

O comando `export` permite que essas libraries sejam exportadas junto à library atual. Ou seja, as APIs públicas de `log.dart` e `printer.dart` poderão ser acessadas através de `logger.dart`. Assim os usuários do nosso package precisam importar apenas um único arquivo nas suas aplicações.

Note que nem sempre toda library presente em `lib/src` deverá ser exportada. É muito comum ter arquivos utilitários que dizem respeito apenas ao funcionamento interno do package, e estes não são exportados. Afinal quem usa o package não precisa nem saber de sua existência.

Testando o package

Com os arquivos prontos, podemos testar se tudo está funcionando normalmente. Para isso, podemos utilizar o diretório `example`, que serve justamente para criarmos um exemplo executável que sirva de referência para que outras pessoas saibam como utilizar nosso package.

Crie então um arquivo `example/main.dart`.

```
import 'package:logger/logger.dart';

void main() {
    const logger = Logger(
        nivel: Nivel.info,
        printer: Printer(inicio: 'Customizado: {', fim: '}'),
    );
    logger.error('Este é um erro');
    logger.warning('Este é um warning');
    logger.info('Esta é uma info');
}
```

O funcionamento é exatamente como já conhecemos, só que desta vez estamos utilizando o package em vez de apenas usar a library. Note como o import está sendo feito através do '`package:logger/logger.dart`', onde `logger` é o nome do package que definimos no `pubspec.yaml`, e `logger.dart` é a nossa library pública que exporta as demais libraries.

Execute o exemplo com `dart run example/main.dart` e os três logs deverão ser impressos.

7.4 Adicionando linter para o código

Como pessoas desenvolvedoras, nossa principal atividade é escrever código. E também é o nosso papel garantir que este código esteja seguindo os princípios de boas práticas, não somente em performance, mas também em padrões de codificação definidos de acordo com cada linguagem.

E por mais que sejamos atentos, é natural acabar não percebendo algum trecho de código que contenha uma má prática durante as análises, por isso, existem automatizações dessas tarefas através de um *linter*. Um *linter* é um script que realiza análise estática do código-fonte de um programa, identificando possíveis problemas ou má práticas de acordo com regras personalizáveis e predefinidas.

Em Dart, essa análise estática do código é feita através do package `analyzer` e configurada com o arquivo `analysis_options.yaml`. E como este é um arquivo opcional, caso não seja definido no projeto, o *analyzer* utilizará verificações padrões. Por exemplo, levando em consideração que o arquivo `analysis_options.yaml` esteja vazio, e dado o seguinte código em um arquivo `lib/main.dart`:

```
class pessoa {
  pessoa({this.nome = null});
  final String? nome

  void DizerOi() {
    print('Olá! Me chamo $nome');
  }
}

main() {
  final julio = new pessoa(nome: 'Julio');
  julio.DizerOi();
}
```

Podemos mandar o *analyzer* identificar possíveis problemas com o nosso código executando `dart analyze lib`, onde `lib` é um diretório e o *analyzer* vai validar todos os arquivos dentro dele. Resultado:

```
Analyzing lib...
  error • main.dart:3:17 • Expected to find ';' . • expected_token
1 error found.
```

O *analyzer* identificou um *error*, indicando que está faltando um ; logo após o campo nome e, por conta disso, o código não vai compilar. Corrija o código para final String? nome; e rode novamente o *analyzer*. Desta vez não será encontrado nenhum problema e o código poderá ser executado normalmente com dart run main.dart .

Mas se eu dissesse para você que no código anterior ainda existem cinco más práticas que vão totalmente ao contrário dos padrões de codificação em Dart e devem ser consideradas como erro, você saberia informar quais são todas elas? Pause um pouco a leitura e tente identificá-las.

A seguir estão os problemas enumerados:

1. A classe está nomeada como pessoa , em vez de seguir o padrão CamelCase . O correto seria Pessoa .
2. No construtor, o campo nome está sendo inicializado com o valor null , o que é redundante e desnecessário, pois o campo já é nullable String? , que por padrão é null . O correto seria apenas {this.nome} .
3. O método Dizeroi() está nomeado com o padrão CamelCase , porém métodos e funções devem seguir o padrão lowerCamelCase , iniciando em minúsculo. O correto seria dizeroi() .
4. A função main() não possui retorno definido, sendo que toda função deveria definir explicitamente qual o seu retorno. O correto seria void main() .
5. Estamos criando um objeto com new pessoa() , além do nome do objeto que já sabemos estar errado, não deveríamos utilizar o modificador new sendo que este se tornou opcional e sua utilização não faz parte das boas práticas. O correto seria Pessoa() .

Conseguiu acertar os cinco problemas? Como na prática eles não afetam na execução e resultado do código, o *analyzer* acaba ignorando-os. Porém, eles

afetam diretamente na padronização e qualidade do código, pois uma vez que são definidas boas práticas, é importante segui-las.

Felizmente, o analyzer possui um linter como um plugin, que pode ser configurado através do `analysis_options.yaml` :

```
linter:  
  rules:  
    - camel_case_types  
    - non_constant_identifier_names  
    - avoid_init_to_null  
    - always_declare_return_types  
    - unnecessary_new
```

Dentro da diretiva `linter` é possível especificar `rules`, ou regras, para a análise do código. As cinco regras listadas no arquivo são justamente para validação dos cinco problemas que identificamos anteriormente. Uma delas é a regra `unnecessary_new`, cujo nome traduzido é `new` desnecessário e valida justamente a utilização do modificador `new`. Uma lista completa com todas as regras disponíveis para a linguagem pode ser vista em <https://dart-lang.github.io/linter/lints/>.

Uma vez que essas regras estão no arquivo, podemos rodar o *analyzer* novamente. Resultado:

```
Analyzing lib...  
info • main.dart:1:7 • Name types using UpperCamelCase. •  
camel_case_types  
info • main.dart:2:11 • Don't explicitly initialize variables to  
null. • avoid_init_to_null  
info • main.dart:5:8 • Name non-constant identifiers using  
lowerCamelCase. • non_constant_identifier_names  
info • main.dart:10:1 • The function main should have a return  
type but doesn't. Try adding a return type to the function. •  
always_declare_return_types  
info • main.dart:11:17 • Unnecessary new keyword. •  
unnecessary_new  
5 issues found.
```

E aí está, o *analyzer* informou exatamente os problemas que havíamos identificado. Ao corrigir os problemas e analisar novamente, nada será listado. Assim conseguimos manter a padronização e as boas práticas aplicadas em nosso código.

Utilizando regras de packages

Para definir essas regras do *linter*, você pode simplesmente pegar uma lista com todas elas e colocar em seu arquivo. Uma segunda opção é utilizar regras já definidas em outros packages criados exatamente para isso.

Até pouco tempo atrás não existia um *package* oficial recomendado pela equipe de Dart que deveria ser utilizado em todos os projetos. Existiam alguns diferentes e o mais famoso era o `pedantic`, que continha as regras utilizadas pela Google em seus projetos. Porém, recentemente foi criado o `lints`, um package com as regras oficiais de linter, que inclusive já é incluído como `dev_dependencies` de um novo projeto ao utilizar o `dart create`.

Para utilizá-lo, uma vez que já está declarado como dependência, podemos modificar o `analysis_options.yaml` para:

```
include: package:lints/recommended.yaml
```

A diretiva `include` permite apontar para um outro arquivo com regras, neste caso, para o `recommended.yaml`, que está definido no `package:lints`. Ele contém as regras que são recomendadas para todos os projetos, mas também existe o `package:lints/core.yaml` apenas com as regras críticas e principais.

Existem outras diretivas que podem ser utilizadas dentro do `analysis_options` para customizar mais o comportamento do linter, consulte a documentação para visualizar todas opções.

7.5 Import dinâmico de libraries

O objetivo do nosso *package logger* é que ele possa ser facilmente utilizado como dependência por outras aplicações, só que estamos utilizando a lib `dart:io` para imprimir os logs no console, o que o torna incompatível com aplicações web. O fato de depender de `dart:io` não deixa uma aplicação web nem ser compilada.

Felizmente, por ser uma linguagem multiplataforma conseguimos contornar essa situação e fazer com que:

- Caso a aplicação seja web, utilizaremos um simples `print()` para imprimir, mantendo a compatibilidade.
- Caso a aplicação não seja web, continuamos imprimindo com o `stdout` normalmente.

Para isso, precisamos separar a funcionalidade em duas libraries. A primeira será a especialista em imprimir com `stdout` e estará em um arquivo `printer_io.dart`:

```
import 'dart:io';

void log(Object object) {
    stdout.writeln('[IO] $object');
}
```

E a segunda será a especialista em imprimir através do `print()` no arquivo `printer_console.dart`:

```
void log(Object object) {
    print('[Console] $object');
}
```

Agora no `printer.dart` definimos um import dinâmico:

```
import 'log.dart';
import 'printer_console.dart' if (dart.library.io)
    'printer_io.dart' as printer;

class Printer {
    const Printer({this.inicio = '', this.fim = ''});
```

```

final String inicio;
final String fim;

void log(Cores cor, dynamic object) {
  printer.log(
    '${ansiCores[cor]}$inicio$object$fim$resetarCor',
  );
}
}

```

Utilizando um `if` em conjunto com o `import` o deixamos dinâmico. Nesse caso, será validado:

- Se a lib `dart.library.io` (a `dart:io`) está disponível, significa que não estamos compilando para a web, então importe `printer_io`. Caso contrário, importe `printer_console.dart`.

Como foi definido um alias `printer` para a library importada, posteriormente conseguimos chamar `printer.log()`, que vai executar a função `log()` da library importada em tempo de compilação. Uma coisa importante é que, para utilizar um import dinâmico, ambas as libraries devem possuir a mesma API pública de métodos e funções, assim como ambas as nossas libraries possuem uma função `log()` de mesma assinatura.

O import dinâmico também pode conter mais de um `if` para validação:

```

import 'src/default.dart'
  if (dart.library.io) 'src/cli.dart'
  if (dart.library.html) 'src/web.dart';

```

Cujo funcionamento é:

- Valida se `dart.library.io` está habilitado e importa `cli.dart`.
- Caso não, valida se `dart.library.html` está habilitado e importa `web.dart`.
- Caso não, importa `default.dart`.

Também é possível utilizar esse recurso em conjunto com o `export` e funciona exatamente igual. Basta substituir o comando `import` por `export` nos exemplos anteriores.

Testando o package na web

Com a importação dinâmica, o package se torna compatível com a web e podemos utilizá-lo sem problemas. Para testar, podemos utilizar a aplicação em que construímos o jogo do "Pedra, Papel, Tesoura, Lagarto e Spock", bastando incluir a dependência no seu `pubspec.yaml`.

Só que, como nosso package não foi publicado no <http://pub.dev>, que é o diretório oficial de packages públicos, não podemos simplesmente informar o nome da dependência e versão, aguardando que o `pub` saiba de onde realizar o download desta dependência. Precisamos importar de uma forma diferente:

```
dependencies:  
  logger:  
    path: '/Users/jhbitencourt/dev/projects/dart-  
book/code/pt_Br/na_pratica/07_logger_package'
```

É possível apontar dependências de packages locais com a diretiva `path:` , bastando informar qual o diretório completo em que está o package em nossa máquina. Com isso, ao rodar `dart pub get` , a dependência estará pronta para uso.

```
import 'package:logger/logger.dart';  
// Demais códigos...  
void main() {  
  const logger = Logger(nivel: Nivel.info);  
  logger.error('Teste erro');  
  logger.warning('Teste warning');  
  logger.info('Teste info');  
  //...  
}
```

Importe o package normalmente e crie logs pelo código. Rodando a aplicação web com `webdev serve` , você conseguirá ver os logs serem

impressos no console do navegador (aquele que conseguimos acessar pelas ferramentas de desenvolvedor, ou ao inspecionar elemento na página).

7.6 Documentando libraries

Ao desenvolver packages com o intuito de que outros desenvolvedores utilizem, ou principalmente, projetos open source onde qualquer pessoa possa se envolver no desenvolvimento, é crucial haver uma documentação do projeto.

Uma das formas de documentar é através de comentários no próprio código-fonte, e Dart permite comentários de três formas. A primeira são comentários de única linha, criados utilizando duas barras invertidas `//`. Eles aparecem bastante ao longo dos exemplos deste livro e servem para comentários rápidos e que são ignorados ao gerar a documentação, como os famosos `TODO`'s .

```
// TODO finalizar este método
void imprimirValor() {
    print(42); // Imprime sempre 42.
}
```

A segunda forma são comentários multilinhas demarcados com `/*` para iniciar e `*/` para encerrar. Todo o conteúdo que está no meio é ignorado pelo compilador e tratado como comentário.

```
/* // TODO finalizar este método
void imprimirValor() {
    print(42); // Imprime sempre 42.
    Todo esse trecho de código é tratado como comentário..
} */
```

Esse tipo de comentário embora permitido não é muito utilizado, apenas em raras exceções quando se precisa comentar um bloco de código grande temporariamente.

Já o terceiro tipo e o mais utilizado é conhecido por ser um comentário de documentação. Em vez de duas barras, são utilizadas três `///`.

```
// Código ANSI para resetar a cor do terminal
const resetarCor = '\x1b[m';
```

Serve para documentar tipos, métodos, funções, membros e qualquer coisa que a sua library possua. Pode se estender por várias linhas desde que cada nova linha inicie com `///`.

```
// Responsável por realizar a impressão de logs no terminal ou
// console, além de
// permitir customizar os logs impressos.
class Printer {
```

Também permite criar hiperlinks para outros membros caso preenchido entre colchetes `[Classe]` e utilizar elementos de formatação em markdown:

```
// Controla o [Nível] permitido do log para ser impresso, quanto
// mais crítico mais restrito é, e menos níveis são impressos.
// Abaixo as opções ordenadas pelo menos crítico:
//
// * [Nível.info] significa que serão impressos todos os níveis.
// * [Nível.warning] são impressos os logs de [Nível.warning]
//   e [Nível.error]
// * [Nível.error] é o nível mais restrito, e apenas
//   [Nível.error] são impressos.
final Nível nível;
```

Tudo isso é muito útil na hora de gerar a documentação através do `dartdoc`. Faça o teste, documente as classes do *logger* (no repositório do livro no GitHub você as encontra já documentadas), acesse o seu diretório e simplesmente execute `dart doc .`, onde o `.` (ponto) é uma referência ao diretório atual do projeto.

Isso será o suficiente para que seja gerado uma documentação completa em HTML no diretório `doc/api`, que pode facilmente ser visualizada localmente ou hospedada em algum servidor. O formato é exatamente no mesmo estilo da documentação da API oficial em <https://api.dart.dev>.



Documentação Logger.

Figura 7.1: Documentação Logger.

Assim concluímos a construção do logger e você encerra o capítulo com um maior conhecimento sobre a criação e disponibilização de packages. Uma vez que um package está pronto, ele pode facilmente ser publicado no <http://pub.dev> e compartilhado com milhares de desenvolvedores e desenvolvedoras do mundo todo.

Inclusive, quando publicamos packages no pub.dev, esta documentação é gerada e hospedada automaticamente, como a documentação do package `http` localizada em <https://pub.dev/documentation/http/latest/>.

O processo em si de publicar um package é muito simples e facilmente encontrado na documentação, por isso não abordaremos aqui no livro. Vale lembrar que, embora você seja livre para publicar qualquer coisa, desde que o nome de identificação seja único, é interessante você publicar algo que realmente faça sentido ser compartilhado com a comunidade de desenvolvedores. Então capriche no código, nas funcionalidades e na documentação do seu package.

7.7 Se liga aí

- Quando criar um package de aplicação é boa prática versionar o arquivo `pubspec.lock`, assim, ao trabalharem no mesmo projeto, outros desenvolvedores utilizarão as mesmas versões das dependências, garantindo o funcionamento. Por outro lado, se for um package do tipo library, este arquivo não deve ser versionado e sim, sempre gerado um novo com `pub get` para garantir a compatibilidade do seu package com a última versão das dependências.
- Além de poder utilizar dependências do repositório oficial `pub.dev`, ou da máquina local com a diretiva `path`, é possível baixar dependências hospedadas em um servidor privado utilizando a diretiva `hosted`:

```
dependencies:  
  logger:  
    hosted:  
      name: logger  
      url: http://meu-servidor-privado.com  
      version: ^0.1.0
```

- E também, o que pode ser útil em alguns casos, apontar para dependências de um repositório git :

```
dependencies:  
  logger:  
    git:  
      url: git@github.com:JHBitencourt/logger.git  
      ref: master
```

7.8 É com você

1 - Consulte a documentação de Dart para conhecer todas as diretivas possíveis de serem usadas no `pubspec.yaml`. E aproveite para entender o que é o versionamento semântico das dependências e a utilização da *caret syntax* (^) ao especificar as versões.

2 - Pesquise sobre como fazer a publicação de packages para o `pub.dev`. Mas lembre-se: uma vez que um package está publicado ele não pode mais ser removido, então não é uma boa prática hospedar packages de testes.

3 - Pesquise sobre as demais opções de customização do analyzer de Dart com o `analysis_options.yaml`, como excluir diretórios da análise, ignorar algumas regras, mudar a severidade, entre outras opções.

Até aqui

Pronto para criar os seus próprios packages e compartilhar com a comunidade? Pois agora você já tem as informações necessárias para iniciar suas criações, aproveite e acesse o `pub.dev` para buscar inspirações. Afinal, o lado bom do open source é justamente a possibilidade de você conseguir

também contribuir com outros projetos e auxiliar no desenvolvimento de packages de terceiros.

Uma outra opção muito comum também é criar packages com coisas comuns e que você acaba utilizando com frequência em seus projetos pessoais ou até mesmo na sua empresa, como um package de widgets com os padrões de UI da empresa para utilização nos aplicativos em Flutter, por exemplo. São várias as possibilidades.

Mas lembre-se de que, um código bem escrito sempre será mais atrativo e fácil de outros desenvolvedores entenderem e contribuírem, por isso sempre habilite o linter e faça uso em abundância de uma boa documentação do código. Mas enquanto você vai pensando nas ideias, vamos seguir para o próximo capítulo!

CAPÍTULO 8

Oriente seus objetos

Quantas vezes você já leu durante este livro que tudo em Dart são objetos? De fato, Orientação a Objetos é algo bastante presente em Dart, mas não apenas nela. Trata-se de um paradigma de programação muito aceito e usado no desenvolvimento das mais variadas aplicações, que auxilia na reutilização e manutenção do código com representações próximas ao que veríamos no mundo real, o que facilita o seu entendimento.

Aqui eu assumo que você esteja familiarizado de certa forma com os conceitos deste paradigma, e o objetivo deste capítulo é apresentar as diferenças e especificidades de como Dart os aborda. Portanto, vamos ver:

- A relação das classes e objetos;
- Como sobrescrever operadores;
- Os tipos de construtores existentes;
- Os métodos especiais *getters* e *setters*;
- Os diferentes relacionamentos entre objetos;
- As poderosas *extensions*.

8.1 Classes e Objetos

Um **objeto** em si representa a abstração principal dentre os pilares de POO (Programação Orientada a Objetos). Objetos são criados a partir de uma **classe**, que comumente é associada ao conceito de uma receita de bolo, pois ela dita como o objeto deve ser construído. Considere uma representação simples de um programador:

```
class Programador {}
```

Uma classe em Dart pode ser declarada apenas como alto nível, o que significa que não é possível ter classes aninhadas uma dentro da outra. Geralmente, a definição delas vem acompanhada dos atributos (variáveis de

instância) ou comportamentos (métodos) que o objeto possuirá, então podemos adicionar mais informações pertinentes a um programador:

```
class Programador {  
    String? nome;  
    double salario = 0.0;  
    List<String> tarefas = [];  
    List<String>? linguagens;  
  
    void trabalhar(){}
}
```

Essas variáveis de instância respeitam as mesmas regras que vimos até esse momento no livro. Variáveis declaradas como *nullable* não precisam ser inicializadas diretamente, como os atributos `nome` e `linguagens` acima. Para as demais, existe a possibilidade de já definir um valor padrão, como `salario`, que será inicializado como `0`, e `tarefas`, como uma lista vazia. A criação de um objeto é simples:

```
final programador = new Programador()  
..nome = 'Julio Bitencourt';  
programador.tarefas.add('Terminar livro de Dart');
```

O operador `new` é o responsável por criar uma nova instância para esse objeto em memória. Como já sabemos também, após a versão 2 de Dart, ele é opcional e implícito, o compilador vai se encarregar de adicioná-lo. É parte das boas práticas então não o utilizar.

Como visto no capítulo dos tipos, Dart possui uma classe especial chamada `Object` que representa o objeto pai de todas as demais classes. Portanto, para qualquer nova classe criada, seus objetos herdam diretamente dele, mesmo que essa herança não esteja explícita no código. Isso significa que os atributos definidos na classe `Object`, como o `runtimeType` ou o famoso método `toString()`, são herdados:

```
void main() {  
    final p1 = Programador()..nome = 'Julio';  
  
    print(p1.runtimeType); // > Programador
```

```
    print(p1.toString()); // > Instance of 'Programador'  
}
```

O `toString` é uma representação em string do estado atual do objeto e por padrão vai sempre imprimir `Instance of` objeto. Podemos facilmente personalizar esse comportamento sobrescrevendo o método na nossa classe, afinal faz mais sentido a representação em string de um programador ser o seu nome.

```
class Programador {  
    //código omitido  
    @override  
    String toString() => '$nome';  
}  
void main() {  
    final p1 = Programador()..nome = 'Julio';  
    print(p1); // > Julio  
}
```

Ao definir um método de mesmo nome e retorno na classe filha `Programador`, estamos sobrescrevendo o método da classe pai `Object`. A anotação `@override` é totalmente opcional, o método continuaria sendo sobreescrito sem ela, o único papel dela é funcionar como um indicativo de que este é um método sobreescrito, facilitando essa identificação na leitura do código. O uso dela ou não é uma escolha individual por projeto, mas é recomendado que seja padronizada em todo o código. Inclusive, existe uma regra de linter denominada `annotate_overrides`, onde o analyzer informará os locais não utilizados.

Com esta sobreescrita, o resultado do código é a impressão do `nome`. Note que dessa vez não chamamos explicitamente o método `p1.toString()`, isso porque, ao utilizar qualquer objeto em um contexto de `String`, o seu método `toString` será chamado automaticamente.

8.2 Sobrescrita de operadores

Um conceito muito importante para se entender ao trabalhar com diferentes instâncias de um mesmo objeto é a relação de igualdade entre eles. Em outras palavras, aquilo de determina se um objeto é ou não igual a outro. Considere:

```
void main() {  
    final p1 = Programador()..nome = 'Julio';  
    final p2 = Programador()..nome = 'Julio';  
  
    print(p1 == p2); // > false  
    print('hash p1: ${p1.hashCode} - hash p2: ${p2.hashCode}');  
    // > hash p1: 624437211 - hash p2: 125619715  
}
```

Por mais que os dois objetos `p1` e `p2` tenham o mesmo nome, ao validar através do operador `==` (ou `equals`), o resultado é `false`. Isso porque a implementação deste operador na classe `Object` vai validar como `true` apenas caso os dois objetos sejam os mesmos em memória. Mas como criamos dois objetos, teremos duas instâncias de `Programador` diferentes em memória.

Só que na maioria dos casos, quando estamos criando nossos próprios objetos, existe uma regra de negócio por traz que determina uma certa igualdade entre eles. No caso do exemplo, se duas instâncias de `Programador` possuírem o mesmo nome faz sentido que sejam considerados iguais. Em um exemplo real, isso se daria por algum campo de `id` ou uma combinação de campos que fossem únicos de cada objeto.

Felizmente, em Dart, além de poder sobrescrever métodos normais, podemos também sobrescrever operadores:

```
class Programador {  
    //código omitido  
    @override  
    bool operator ==(Object other)  
        => other is Programador && nome == other.nome;  
}
```

Como é um operador e não um simples método, a palavra-chave `operator` é necessária. O objeto recebido por parâmetro pode ter qualquer nome, mas é padrão nomear de `other` (outro). Já a validação é simples, se o objeto a ser validado é um `Programador` e possuir o mesmo nome, para o nosso caso significa que ambos são iguais. Assim `print(p1 == p2)` resulta em `true`.

Mas isso não é suficiente, ainda tem o `hashCode`, que é uma representação numérica da igualdade de um objeto, utilizada por algumas implementações como da classe `HashSet` ou `HashMap`. Então por consistência do nosso código, a igualdade do `==` sempre deve ser igual à do `hashCode`. Para o mesmo objeto passado, ambos devem retornar `true` ou `false` de forma igual.

```
class Programador {  
    //código omitido  
    @override  
    int get hashCode => nome.hashCode;  
}
```

O `hashCode`, por sua vez, não é um operador, mas é um outro método especial conhecido por *getter*, que veremos em breve. Por ora, como a classe `String` já implementa um `hashCode` customizado, vamos utilizá-lo para respeitar as mesmas regras do `==`. No dia a dia, você dificilmente precisará implementar manualmente o `==` ou `hashCode`, pois as IDEs utilizadas fazem esse trabalho de geração automática.

É válido reforçar que o fato de dois objetos serem iguais (`==` e `hashCode`) não significa que eles são os mesmos, e alterar a propriedade de um não vai alterar também a do outro. Um objeto só será idêntico ao outro se ambas as variáveis apontarem para o mesmo endereço de memória, o que pode ser verificado através da função `identical(p1, p2);`, também presente em `Object`.

8.3 Construtores

Construtores são métodos especiais para inicialização das instâncias dos objetos. Comumente são utilizados para definição de valores para as variáveis, mas também podem servir para a execução de qualquer código, assim como qualquer método. Existem alguns tipos diferentes de construtores em Dart.

Construtores padrão

Toda classe em Dart possui implicitamente um construtor padrão definido sem parâmetros. Para a classe `Programador`, temos o construtor `Programador(){}, o que permite criar novas instâncias dela. Facilmente, podemos alterar esse construtor padrão:`

```
class Programador {  
    Programador(String nome, List<String> linguagens) {  
        this.nome = nome;  
        this.linguagens = linguagens;  
    }  
    // código omitido  
}  
//inicialização:  
final p1 = Programador('Julio', ['Dart']);
```

Alteramos o construtor padrão obrigando a passar por parâmetro o `nome` e `linguagens`. Não é possível ter mais de um construtor do tipo padrão definido, então, ao tentar inicializar o objeto como anteriormente, `Programador()` sem parâmetros, vai ocorrer um erro.

Os construtores, assim como todos os métodos dentro da classe, possuem acesso ao `this`. O `this` é a referência para a instância atual do objeto, note que no escopo do construtor anterior possuímos duas variáveis de mesmo nome, `nome`, `linguagens`. O valor da variável seria atribuído a ela mesmo, mas com o `this` acessamos a variável de instância, enquanto sem ele referenciamos a variável local (passada por parâmetro).

Como essa utilização do construtor para inicializar as variáveis é muito comum, Dart possui um *syntax sugar* para isso. O mesmo construtor

poderia ser reescrito como a seguir, sem qualquer alteração da forma de instanciamento do objeto.

```
class Programador {  
    Programador(this.nome, this.linguagens);  
    // código omitido  
}
```

Automaticamente, o valor passado é atribuído à variável de mesmo nome. Mas como nem sempre esse comportamento de exigir todos os parâmetros é o desejado, outra opção é definir o construtor com o auxílio de parâmetros opcionais e valores default:

```
class Programador {  
    Programador({this.nome = 'Fulano', this.linguagens = const []});  
    // código omitido  
}  
//inicialização:  
final p1 = Programador();  
final p2 = Programador(nome: 'Julio');
```

As regras são as mesmas de funções normais, com a única diferença de que apenas valores constantes podem ser usados como valores default de parâmetros opcionais em construtores.

Construtores nomeados

Já sabemos que não é possível ter mais do que um construtor padrão, mas a necessidade de um objeto ter mais de um construtor é algo existente, e é por isso que existem os construtores nomeados. A diferença desse tipo para o padrão é que ele possui a atribuição de um identificador, de forma que você possa dar um melhor significado para o seu uso. Talvez faça sentido para nossa aplicação já definirmos as variáveis de nosso `Programador` baseado em uma linguagem, então podemos criar:

```
class Programador {  
    Programador.dart(this.nome) {  
        linguagens = ['Dart'];  
    }  
    // código omitido
```

```
}
```

//inicialização:

```
final p1 = Programador.dart('Eric Seidel');
```

As possibilidades são infinitas, mas os nomes nunca podem se repetir, independente de os parâmetros serem diferentes.

Construtores constantes

Uma terceira opção de construtores está associada diretamente ao conceito de objetos imutáveis, conceito este que é bastante utilizado por algumas linguagens de programação pois traz certos benefícios para manutenção e execução do código. São os construtores `const`.

```
class LinguagemProgramacao {
    const LinguagemProgramacao(this.nome);
    final String nome;
}
```

Um objeto pode conter um construtor `const` caso ele contenha apenas variáveis de instância definidas como `final`, para garantir a sua imutabilidade.

```
void main() {
    final primeira = const LinguagemProgramacao('Dart');
    final segunda = const LinguagemProgramacao('Dart');
    final terceira = LinguagemProgramacao('Dart');
    print(identical(primeira, segunda)); // > true
    print(identical(primeira, terceira)); // > false
}
```

Como visto no capítulo 3 ao abordar as variáveis, um objeto criado a partir de um construtor `const` é canonicalizado, o que vai garantir apenas uma instância dele em memória. No último código, as variáveis `primeira` e `segunda` apontam para o mesmo endereço de memória, afinal utilizaram o construtor `const`. É válido prestar atenção nisso, pois para que de fato o objeto seja criado como `const` é necessário informar na criação, caso contrário, um novo objeto pode ser criado com o construtor padrão, como a variável `terceira`, que aponta para um endereço de memória diferente.

Construtores factory

Existe um último conceito de construtores em Dart que possui um significado semântico diferente dos demais. Isso porque os construtores `factory` são utilizados quando podemos ou não retornar uma nova instância do objeto. Por exemplo, imagine que cada criação de um novo objeto é muito custosa, então é decidido que será criado um cache de instâncias.

```
class Programador {  
    Programador._internal(this.nome);  
  
    factory Programador(String nome) {  
        if(_cache.containsKey(nome)) {  
            return _cache[nome]!;  
        }  
        final novo = Programador._internal(nome);  
        _cache[nome] = novo;  
        return novo;  
    }  
  
    static final Map<String, Programador> _cache = {};  
    String nome;  
  
    static imprimirCache() {  
        print(_cache);  
    }  
}
```

Agora em vez de sempre retornar uma nova instância de `Programador`, o nosso construtor `factory` vai validar se o programador já existe em um `Map` que vai manter o cache, caso verdadeiro, essa instância já existente é retornada. Na prática, o construtor `factory` sempre vai utilizar algum outro tipo de construtor para de fato criar uma nova instância, como no caso anterior, em que utilizamos um construtor nomeado que também é privado `_internal`. Ele poderia ter qualquer outro nome, mas essa nomenclatura é um padrão utilizado para indicar que ele é um construtor apenas interno, privado ao objeto.

Uma classe pode conter atributos estáticos, como o Map `_cache`. Isso faz com que essa variável não seja de instância e sim da classe, pois ela passa a ser compartilhada por todas as instâncias daquela mesma classe, assim como o `imprimirCache`, que é um método estático. Justamente por pertencerem à classe, os métodos estáticos não possuem acesso ao modificador `this`, que se refere à instância. Esse último também é válido para os construtores `factory`, que não podem acessar o `this`.

```
void main() {
    final p1 = Programador('Julio Bitencourt');
    final p2 = Programador('Julio Bitencourt');
    print(identical(p1, p2)); // > true
    Programador.imprimirCache();
    // > {Julio Bitencourt: Instance of 'Programador'}
}
```

Executando o `main()`, validamos o funcionamento do cache, pois ao chamarmos o construtor `factory` duas vezes a mesma instância é retornada. E ao imprimir o cache, apenas ela estará presente. Note que, por ser estático, o método pode ser chamado diretamente pela própria classe `Programador.imprimirCache()`.

8.4 Encapsulamento

Um outro conceito presente em POO é a habilidade de encapsular dados ou tarefas apenas na classe responsável, de modo que esses dados fiquem protegidos de acessos exteriores (outras classes). Por exemplo, imagine que você tenha uma classe `DownloadArquivo` responsável por fazer o download de arquivos de um servidor através de um método `Arquivo download(){}`.

Como esse download é feito, de qual servidor, qual o protocolo de requisição e afins, isso não interessa para as demais classes; o que interessa para quem chama é que o método vai retornar este arquivo. Todas essas variáveis que influenciam como o download é realizado podem inclusive

mudar com o tempo. Neste caso, por estarem encapsuladas na classe `DownloadArquivo`, a mudança é feita somente nela.

Dois métodos muito famosos quando nos referimos a encapsulamento de dados são o `get` e `set`, utilizados para acesso e alteração de atributos das classes. E para garantir essa privacidade do estado dos objetos em linguagens orientadas a objetos, como o Java, é bastante comum definir todos seus atributos como privados e permitir o acesso apenas através do método `get()` e alteração pelo `set()`. Só que em Dart esse comportamento não é necessário (e nem recomendado), veja:

```
class Programador {  
  DateTime? nascimento;  
}  
void main() {  
  final p1 = Programador();  
  p1.nascimento = DateTime(1995, 12, 1, 2, 30);  
  print(p1.nascimento); // > 1995-12-01 02:30:00.000  
}
```

O `Programador` permite a definição de sua data de nascimento através do atributo `nascimento`. Agora imagine que surgiu uma nova regra em nosso sistema onde todas as datas de nascimento devem ser guardadas sem as horas, em certo momento seria muito trabalhoso mudar em todos os locais que criam um programador e definem uma data. Nesse caso bastaria fazer uso do `set` e `get`.

```
class Programador {  
  DateTime? _nascimento;  
  
  set nascimento(DateTime? value) {  
    if (value != null)  
      _nascimento = DateTime(value.year, value.month, value.day);  
  }  
  
  DateTime? get nascimento => _nascimento;  
  
  int get idade {  
    if (_nascimento == null) return 0;  
  }
```

```

    return DateTime.now()
        .difference(_nascimento!).inDays ~/ 365;
}
}

void main() {
    final p1 = Programador();
    p1.nascimento = DateTime(1995, 12, 1, 2, 30);
    print(p1.nascimento); // > 1995-12-01 00:00:00.000
    print(p1.idade); // > 26
}

```

Já sabemos que o caractere `_` no início de atributos e métodos os definem como privados, acessíveis apenas dentro da própria library. Com isso, é criado um `set` customizado para o nascimento, pegando o valor passado e salvando apenas o ano, mês e dia em uma nova data. Como o `_nascimento` está privado, é necessário também definir o `get` para acesso ao dado.

É possível inclusive criar um `get` sem ter um atributo, como o `get idade` do último código, que simplesmente calcula a idade do programador baseado em sua data de nascimento, com métodos utilitários da classe `DateTime`. O operador `~/` pega o valor inteiro da divisão.

Agora o que realmente interessa: você notou que tendo ou não o `get` e `set` o acesso ao atributo continuou sendo igual? A ideia de sempre encapsular os dados com *getters* e *setters* é muito difundida em linguagens como o Java, onde eles são métodos normais e o acesso ao dado acaba mudando, como `p1.nascimento`, `p1.setNascimento()` e `p1.getNascimento()`.

Só que em Dart `get` e `set` são métodos especiais e todo atributo de uma classe já os possui definidos implicitamente. Ter ou não um `set` e `get` customizado não interfere em nada a chamada no uso desse atributo, por isso são usados apenas quando é realmente necessário definir explicitamente e alterar o comportamento padrão. No geral, aqui não existe essa necessidade de deixar o atributo privado para encapsulá-lo.

8.5 Relacionamento entre objetos

Assim como no mundo real, as classes também são utilizadas para representar vários tipos de relacionamento entre os objetos, criando suas conexões de forma que tudo funcione em conjunto no programa final. Não necessariamente um objeto precisa ser da mesma classe para compartilhar atributos ou comportamentos, e Dart fornece algumas opções para definirmos essas relações, como herança, classes abstratas, interfaces e mixins.

Herança

No nosso dia a dia, lidamos com vários objetos com características, formatos e finalidades parecidas, como um lápis e uma caneta, que, embora diferentes, possuem uma funcionalidade muito parecida: permitir escrever. E seguindo a linha do exemplo do `Programador`, imagine que agora precisamos representar também um `Gerente` em nosso código, um outro tipo de funcionário da empresa.

Programador	Gerente
<code>nome : String</code> <code>salario: double</code> <code>tarefas: List<String></code> <code>linguagens: List<String></code> <code>trabalhar();</code>	<code>nome : String</code> <code>salario: double</code> <code>tarefas: List<String></code> <code>bonus: double</code> <code>trabalhar();</code>

Figura 8.1: Programador e Gerente.

Ambos são muito parecidos, com a diferença de que o `Programador` possui as linguagens de programação, enquanto o `Gerente` possui um valor de bônus pelo resultado dos projetos. Analisando o diagrama da imagem, faz sentido termos duas classes diferentes para representar objetos semelhantes? Afinal, estariámos duplicando código e possíveis

funcionalidades. A resposta é "não". Utilizando herança conseguimos compartilhar esses atributos e ações de forma mais concisa. Considere o diagrama:

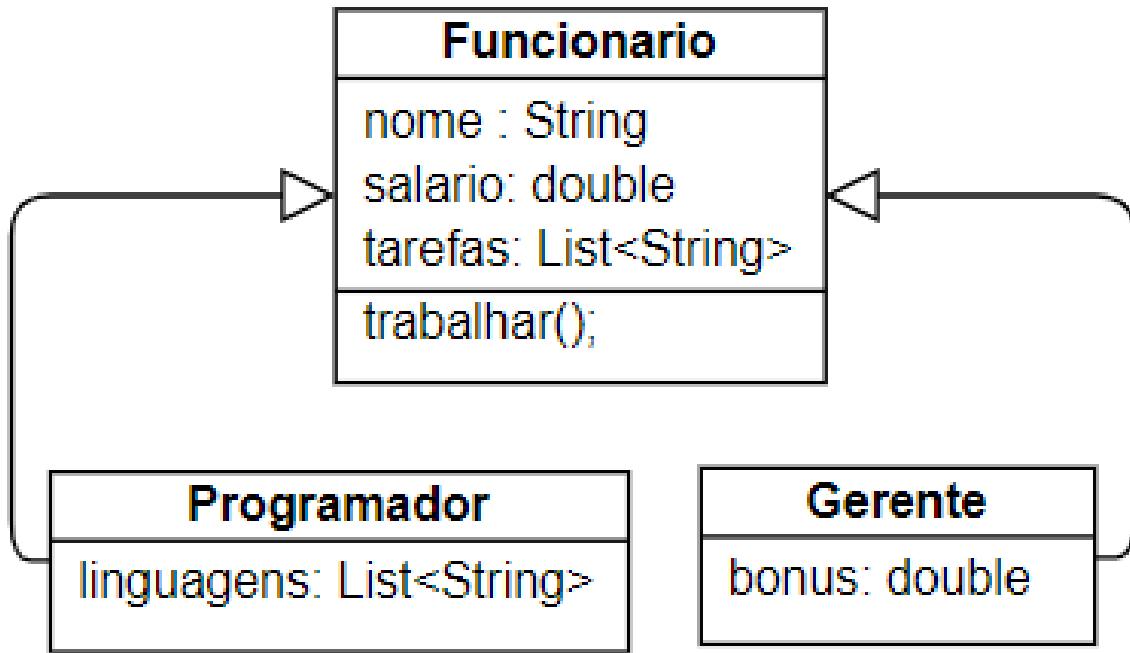


Figura 8.2: Programador e Gerente estendendo de funcionário.

Com uma superclasse `Funcionario` agrupamos as semelhanças, aquilo que é genérico e é pertinente para ambas as classes. A partir dela é possível criar subclasses como `Programador` e `Gerente`. É comum dizer que elas estendem ou herdam as características de `Funcionario`, evitando ter que escrever essas mesmas propriedades em dois locais distintos:

```
class Funcionario {
    String? nome;
    double salario = 0.0;
    List<String> tarefas = [];

    void trabalhar(){
        print('${this.runtimeType} trabalhando..');
    }
}
```

```

class Programador extends Funcionario {
  List<String>? linguagens;
}

class Gerente extends Funcionario {
  double? bonus;
}

```

Quando mencionamos que todos os objetos em Dart herdam de `Object`, este é um relacionamento de herança e ocorre de forma implícita. Para herdarmos de outra classe de forma explícita, é necessário utilizar o `extends`, definindo de qual superclasse serão herdados os atributos. Só é possível herdar diretamente de uma única classe, além de `Object`.

```

void main() {
  final programador = Programador()
    ..nome = 'Bill Gates'
    ..linguagens = ['.Net'];
  final gerente = Gerente()
    ..nome = 'Jeff Bezos'
    ..bonus = 500;

  programador.trabalhar(); // > Programador trabalhando..
  print('Programador é Funcionario? ${programador is
Funcionario}');
  // > Programador é Funcionario? true
  gerente.trabalhar(); // > Gerente trabalhando..
  print('Gerente é Funcionario? ${gerente is Funcionario}');
  // > Gerente é Funcionario? true
}

```

Além dos atributos, todos os métodos não privados também ficam disponíveis para as subclasses, como o `trabalhar()`. Ao instanciar o `Programador` ou `Gerente` é possível utilizar atributos que estão presentes em `Funcionario`. Note também que, ao executar o `trabalhar()` de `Funcionario`, é impresso o tipo conforme o tipo do objeto que instanciamos, que é o tipo real em tempo de execução.

Polimorfismo

Assim como podemos sobrescrever métodos do `Object`, como já fizemos com o `toString`, também podemos sobrescrever qualquer método de uma superclasse, como o próprio `trabalhar()` definido em `Funcionario`:

```
class Funcionario {  
    void trabalhar(){  
        print('Funcionario trabalhando..');  
    }  
}  
class Programador extends Funcionario {  
    @override  
    void trabalhar(){  
        print('Programador trabalhando..');  
    }  
}  
class Gerente extends Funcionario {  
    @override  
    void trabalhar(){  
        print('Gerente trabalhando..');  
        super.trabalhar();  
    }  
}
```

Basta criar um método na `subclasse` com mesmo nome do método na `superclasse`. Agora imagine que temos uma `Startup` e ela precise criar um novo projeto, botando todos os seus programadores e gerentes para trabalharem juntos.

```
class Startup {  
    Startup(this.funcionarios);  
    List<Funcionario> funcionarios;  
  
    void novoProjeto() {  
        funcionarios.forEach((f) => f.trabalhar());  
    }  
}
```

Em vez de manter uma lista de `Programador` e outra de `Gerente`, é possível ter apenas uma de `Funcionario`, afinal todos os programadores e gerentes são funcionários. Para deixar mais simples, o método

`novoProjeto()` apenas pega a lista de `funcionarios` e chama o método `trabalhar()` sobrescrito pelas subclasses.

```
void main() {
    Funcionario funcionario = Funcionario();
    Funcionario programador = Programador();
    Funcionario gerente = Gerente();
    final startup = Startup([funcionario, programador, gerente]);
    startup.novoProjeto();
}
```

O conceito de polimorfismo está na capacidade de um objeto de assumir N formas e executar em *runtime* uma ação diferente conforme sua forma atual. Então mesmo quando instanciamos um objeto do tipo `Programador`, podemos guardá-lo em uma variável de tipo `Funcionario` e tratá-lo como tal. Isso torna possível nossa `Startup` receber apenas funcionários que em *runtime* podem ser do tipo `Funcionario`, `Programador` ou `Gerente`. A saída desse programa é:

```
> Funcionario trabalhando..
> Programador trabalhando..
> Gerente trabalhando..
> Funcionario trabalhando..
```

Ao executar o método `novoProjeto()`, a primeira linha impressa no console mostra que foi chamado como esperado o método da classe `Funcionario`. Já a segunda linha mostra uma saída diferente. Mesmo que a variável seja do tipo `Funcionario`, Dart é inteligente o suficiente para identificar qual o tipo real instaciado e chamar o método correspondente dessa classe. O segundo objeto presente na lista é um `Programador` e, como o método foi sobrescrito nessa classe, será executado o método dela, ignorando o da superclasse.

A terceira linha também indica que o terceiro objeto da lista é um `Gerente` e que o seu método `trabalhar()` foi executado. Nossa lista contém apenas três funcionários, mas foi impressa uma quarta linha indicando que o método da classe `Funcionario` foi chamado novamente. Se você voltar

para o método da classe `Gerente`, vai perceber no final o trecho de código `super.trabalhar();`.

Assim como o `this` referencia o objeto atual, o `super` é utilizado para referenciar a superclasse do objeto, sendo possível acessar um atributo ou método dela. Então através do gerente foi chamada a execução do método no `Funcionario`, imprimindo a última linha.

Classes abstratas

Esse último trecho de código que criamos possui uma falha que pode induzir ao erro caso outra programadora ou programador o utilize. Estamos permitindo a criação de um um objeto `Funcionario`, enquanto o único propósito desta classe é funcionar como uma superclasse, da qual os demais tipos de funcionários estendem. Para impedir que ela seja instanciada, podemos fazer dela uma classe abstrata.

```
abstract class Funcionario {  
    String? nome;  
    double salario = 0.0;  
    List<String> tarefas = [];  
  
    void trabalhar();  
}
```

O único propósito de se criar uma classe abstrata é para que ela sirva exatamente de superclasse para ser estendida por outra classe. Elas podem conter atributos e métodos normais como qualquer outra classe, com a única diferença de que não pode ser instanciada, então, se tentarmos criar um `Funcionario()`, vai resultar em erro.

Outra mudança feita foi no método `trabalhar()`, note que agora ele não possui as chaves `{}` e é finalizado com um `;`. Isso indica que ele passou a ser um método abstrato, outra possibilidade para as classes abstratas.

Quando um método abstrato é criado em uma classe abstrata, todas as demais classes que a estenderem (`Programador` e `Gerente`, por exemplo)

são obrigadas a implementar este método, servindo como uma garantia de que toda subclasse de `Funcionario` vai poder `trabalhar()`.

Interfaces

Nossa `Startup` está se modernizando e agora contratou um novo tipo de funcionário, só que um pouco diferente dos demais. Um funcionário robô! Isso mesmo, ele já está automatizado para fazer a limpeza e o café da galera, então vai trabalhar muito.

```
class Robo extends Funcionario {  
    double? bateria;  
    @override  
    void trabalhar() {  
        print('0100010110010110');  
    }  
}
```

Pronto, o robô já está preparado para ser instanciado e sair trabalhando, pois também já é um funcionário. Mas, além do seu atributo que mostra o nível de `bateria`, por ser um `Funcionario`, ele vai herdar todos os demais atributos da superclasse. Terá um nome (faz sentido), uma lista de tarefas (limpar o chão, fazer café etc.) e um... Salário? Não parece ter muito sentido pagar um salário ao robô. Isso significa que estamos espalhando código desnecessário pelas subclasses de `Funcionario`, pois acabamos de descobrir que nem todo funcionário receberá um salário.

Felizmente, Dart nos permite um outro tipo de relacionamento entre os objetos, que usaremos para resolver esse problema: as interfaces. Vamos criar uma que vai conter o salário.

```
class Assalariado {  
    Assalariado(this.salario);  
    double? salario;  
    void receber(){}
}
```

E nesse momento você está se perguntando onde está a interface, pois essa é só uma classe normal. Bem, toda classe em Dart já disponibiliza por padrão

uma interface implícita dela mesma, e essa interface pública possui todos os atributos e métodos da classe.



Figura 8.3: Classe Assalariado e a interface implícita.

Dessa forma, qualquer classe pode ser utilizada por outra como interface. Com isso, o atributo `salario` já pode ser removido de `Funcionario`, e apenas os funcionários que realmente possuem salário vão implementar `Assalariado`, através da palavra-chave `implements`.

```
abstract class Assalariado {
    Assalariado(this.salario);
    double salario;
    void receber();
}
class Programador extends Funcionario implements Assalariado {
    Programador({this.salario = 1000});
    List<String>? linguagens;
    @override
    double salario;
    @override
    void trabalhar(){
        print('Programador trabalhando..');
    }
    @override
    void receber() {
        print('Programador recebendo $salario');
    }
}
```

Classes abstratas também disponibilizam interfaces, então `Assalariado` pode ser alterada para `abstract`, uma vez que não precisamos instanciar um `Assalariado()` diretamente. Assim como `Funcionario`, um `Gerente` também recebe um salário, então o mesmo pode ser feito com esta classe.

Quando implementamos uma interface, é obrigatória a implementação de todos os atributos e métodos expostos por essa interface, como o método `receber()` implementado pelo `Programador`. A interface de `Assalariado` também exigia a definição de um `get` e `set` para o `salario`, então apenas definindo o atributo na classe cumprimos com esse requisito (lembra do `get` e `set` implícitos?). Já é possível adicionar o pagamento na `Startup`:

```
class Startup {  
    List<Funcionario> funcionarios;  
    List<Assalariado> assalariados;  
    Startup(this.funcionarios, this.assalariados);  
  
    void pagarFuncionarios() {  
        assalariados.forEach((a) => a.receber());  
    }  
}  
  
void main() {  
    Funcionario programador = Programador();  
    Funcionario gerente = Gerente();  
    Funcionario robo = Robo();  
  
    final startup = Startup(  
        [programador, gerente, robo],  
        [(programador as Assalariado), (gerente as Assalariado)],  
    );  
    startup.pagarFuncionarios();  
}
```

O método `pagarFuncionarios()` vai chamar o método `receber()` declarado na interface, para todos os assalariados da `Startup`. Note que o construtor recebe duas listas. Na primeira, que já vimos anteriormente, foi adicionado o `robo`, já a segunda possui algo diferente.

É esperado que se passe uma lista de assalariados, e a superclasse `Funcionario` não implementa a interface `Assalariado`, o que causaria um erro. Como declaramos explicitamente os objetos como `Funcionario`, o compilador não tem como saber em tempo de compilação qual o real tipo que aquela variável contém (poderia ser um robô que não é assalariado). Por isso, somos forçados a utilizar o operador `as` para converter explicitamente esses objetos para o tipo `Assalariado`, avisando o compilador de que sabemos o que estamos fazendo.

Ao rodar, o programa imprime as execuções do `receber()` conforme o esperado:

```
> Programador recebendo 1000.0
> Gerente recebendo 2000.0
```

Mixins

A solução de uso de interface é boa, pois isolamos o salário para apenas os funcionários que são de fato assalariados. Mas e se o método `receber()` deve possuir o mesmo comportamento para todos eles? Afinal, é apenas pegar o salário e guardar. Estaríamos obrigatoriamente colocando a mesma implementação do método em todas as subclasses de `Funcionario` que fossem `Assalariado`. Novamente, duplicação de código.

Existe uma terceira opção de relacionamento presente em Dart, e é algo que pode até ser uma novidade para quem vem de outras linguagens de programação: os *mixins*. São bastante utilizados para reaproveitar código entre as diferentes hierarquias de classes, pois a classe que usa o mixin possuirá acesso a todos os seus atributos e métodos, sem a necessidade de implementá-los (interface).

```
abstract class Assalariado {
  Assalariado(this.salario);
  double salario;
  void receber() {
    print('Pegando o salário $salario');
  }
}
```

Novamente, qualquer classe pode ser utilizada como um mixin por outra. E para prevenir que esse mixin fosse instanciado, antes era comum definir a classe como abstrata. Mas após a versão 2.1 do Dart, foi criada a palavra reservada `mixin` para definição de classes que possuem apenas esse propósito.

```
mixin Assalariado {
    double salario = 0;
    void receber() {
        print('Pegando o salário $salario');
    }
}
```

Essa alteração evita problemas com o código futuramente, pois uma classe utilizada como mixin não pode definir um construtor e deve estender apenas de `Object` - garantia que nós temos utilizando a palavra-chave `mixin`. Existe ainda a possibilidade de restringir o seu uso:

```
mixin Assalariado on Funcionario {
    double salario = 0;
    void receber() {
        print('Pegando o salário do ${super.nome}, valor de $salario');
    }
}
```

Utilizando o modificador `on`, estamos sinalizando que apenas classes que estendem (herança) ou implementam (interface) `Funcionario` podem utilizar `Assalariado` como mixin, pois o `Assalariado` depende de que essa classe tenha um recurso disponível pela superclasse, no nosso caso, o nome do funcionário, acessado por `super.nome`.

```
class Programador extends Funcionario with Assalariado {
    List<String>? linguagens;
    void trabalhar() {
        print('Programador trabalhando..');
    }
}
```

Para usar uma classe como mixin, é utilizada a palavra reservada `with`. Não é mais necessário que `Programador` defina o salário e o método

`receber()`, uma vez que eles estão no mixin. A classe `Startup` não precisa de alterações para funcionar. E para teste, vamos inserir os nomes no `main()`:

```
void main() {
    Funcionario programador = Programador()
        ..nome = 'Bill Gates'
        ..salario = 34000;
    Funcionario gerente = Gerente()
        ..nome = 'Douglas Adams'
        ..salario = 42000;
    Funcionario robo = Robo();

    final startup = Startup(
        [programador, gerente, robo],
        [(programador as Assalariado), (gerente as Assalariado)],
    );
    startup.pagarFuncionarios();
}
```

O seguinte será impresso:

```
> Pegando o salário do Bill Gates, valor de 34000.0
> Pegando o salário do Douglas Adams, valor de 42000.0
```

Mixins a fundo

Mixins são excelentes e dão uma liberdade para reaproveitar operações e estados entre toda a hierarquia de classes, e por isso é importante entender de fato como Dart os implementa. Muitas pessoas tendem a comparar o uso de mixin com o conceito de herança múltipla, que não é presente em Dart. De fato, é um recurso que permite reaproveitar código de múltiplas classes, mas Dart continua sendo uma linguagem de herança única. Vamos entender na prática:

```
abstract class A {
    String ola();
}

class B extends A {
    String ola() => 'Olá B';
```

```

}

class C extends A {
  String ola() => 'Olá C';
}

```

Dado esse trecho de código e **supondo** que Dart permitisse herança múltipla:

```

class D extends B, C {
  void dizerOla() {
    print(ola());
  }
}

```

Você saberia dizer o que seria impresso na tela ao executar `dizerOla()`? É difícil, nem o compilador saberia qual dos dois métodos executar, o método da superclasse `B` ou `C`. Isso é conhecido como o problema do diamante, gerado quando existe uma ambiguidade entre os métodos de superclasses diferentes. É um dos grandes problemas quando se permite herança múltipla.

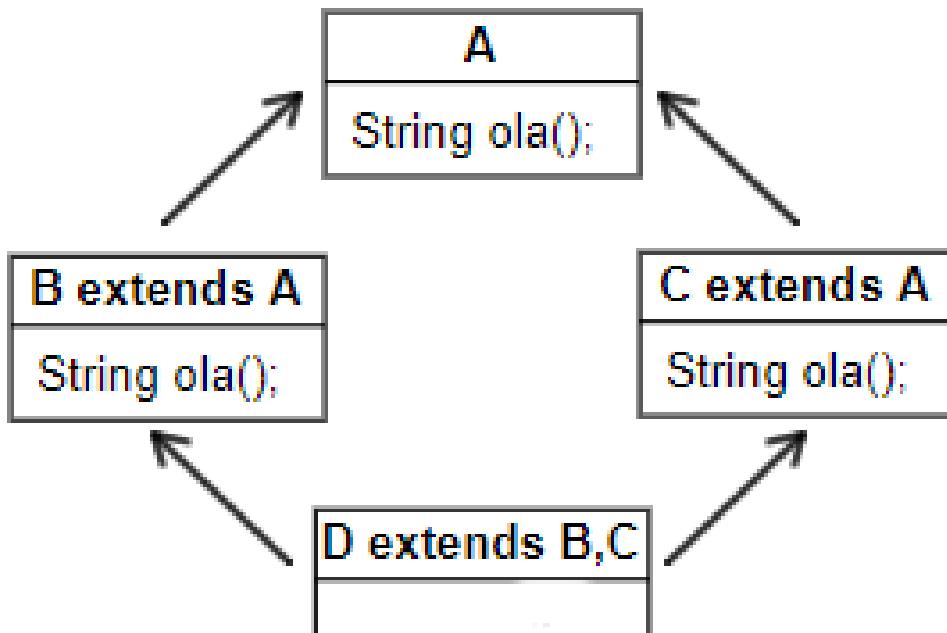


Figura 8.4: Problema do diamante.

Agora utilizando mixin em vez de herança:

```
abstract class A {
  String ola();
}

class B {
  String ola() => 'Olá B';
}

class C {
  String ola() => 'Olá C';
}

class D extends A with B, C {
  void dizerOla() {
    print(ola());
  }
}

void main() {
  D().dizerOla();
}
```

- D estende A , sendo uma subclasse de A .
- D define dois mixins, B e C .
- A define um método abstrato ola() , D não precisa implementá-lo pois ambos os mixins já implementam ola() .

Com base nesses fatos, você consegue novamente dizer o que será impresso ao executar o método dizerOla() ? Dessa vez não haverá problemas, e a resposta correta é Olá C , surpreso(a)?

O segredo é que, para adicionar os atributos e métodos de um mixin a uma classe, Dart cria uma nova classe implícita com essas funcionalidades, acima da classe atual, tornando esses métodos disponíveis para as camadas abaixo. Na prática, seria o mesmo que:

```
class AB = A with B;
class ABC = AB with C;
class D extends ABC {}
```

É criada uma espécie de hierarquia, o que permite uma classe possuir N mixins. E como C é o nível mais baixo dessa hierarquia, que possui a

implementação de `ola()`, ele é o executado. Portanto, a ordem de declaração dos mixins é muito importante.



Hierarquia de classes utilizando mixin.



Figura 8.5: Hierarquia de classes utilizando mixin.

Essa forma de implementação garante que nunca haverá ambiguidade entre os métodos, como existe no problema com diamantes.

Afinal, devo usar herança, interface ou mixin?

Se você ainda está se sentindo um pouco perdido com todos esses conceitos de relacionamento e não sabe exatamente quando aplicar cada um, a seguir está uma tabela com um resumo das principais diferenças discutidas durante este capítulo.

Herança	Interface	Mixin
A extends B	A implements B	A with B
É possível estender apenas uma única classe.	É possível implementar n classes.	É possível ter n mixins.
Vai possuir todos os atributos e métodos da superclasse. É obrigatório implementar os métodos abstratos.	Deve implementar todos os atributos e métodos da superclasse (abstratos ou não).	Uma forma de reaproveitar atributos e métodos em uma hierarquia de classes, sem necessidade de implementar nada extra.
Dado A extends B , semanticamente A é B, pois herda A de B .	Dado A implements B , semanticamente A é B, pois A implementa o contrato de abstração de B .	Dado A with B , semanticamente A contém B, pois A possui disponível toda a implementação de B .

8.6 As incríveis Extensions

Frequentemente realizamos operações comuns entre os tipos presentes no SDK, que não fazem parte da API disponível. Por exemplo, eventualmente precisamos capitalizar uma string, deixar sua primeira letra em maiúsculo para apresentar ao usuário com este padrão. Então é normal criar uma função parecida com essa:

```
String capitalizar(String texto) {
    if(texto.isEmpty) return texto;
    return '${texto[0].toUpperCase()}${texto.substring(1)}';
}
void main() {
    print(capitalizar('dart'));// > Dart
}
```

Essa é apenas uma das diversas funções utilitárias que acabam surgindo e são frequentemente agrupadas nos famosos arquivos `utils`, como `string_utils.dart`, presentes em praticamente todo projeto.

Só que em sua versão 2.7 Dart trouxe uma novidade que pode ser considerada um superpoder para os objetos, as chamadas **Extensions**.

```
extension on String {
    String capitalizar() =>
        this.isEmpty ? this
        : '${this[0].toUpperCase()}${this.substring(1)}';

    String operator &(String other) => '$this - $other';

    String get primeiraPalavra => split(' ').first;
    String get ultimaPalavra => split(' ').last;
}
```

Uma *extension* permite adicionar novos métodos (incluindo *getters* e *setters*) e operadores na API de objetos já existentes, como ilustrado acima na `String`. Ela tem acesso aos membros já existentes, como a chamada ao método `split()` do exemplo, da classe `String`. E o `this` faz referência à instância do objeto sendo utilizada no momento. Somente com isso já é possível utilizar os novos membros:

```
void main() {
  print('julio'.capitalizar()); // > Julio
  print('julio bitencourt'.primeiraPalavra()); // > julio
  print('julio bitencourt'.ultimaPalavra.capitalizar());
  // > Bitencourt
  print('julio' & 'bitencourt'); // > julio - bitencourt
}
```

Consegui perceber o poder disso? É como se a própria classe `String` tivesse originalmente essas novas funcionalidades! E isso pode ser feito com qualquer objeto. Ter a possibilidade de adicionar novas funções à API de um objeto de um *package* terceiro é algo fenomenal. Entretanto, elas não permitem definir novos atributos, construtores ou sobrescrever métodos do objeto original.

Uma `extension` sem nome é privada para a library em que ela se encontra, se você deseja utilizá-la em arquivos diferentes é necessário definir um nome de identificação. Supondo que ela esteja em um arquivo

```
string_extension.dart :
```

```
extension StringExtension on String {
  // código omitido
}
```

É necessário então importar esse arquivo onde for utilizá-lo.

```
import 'string_extension.dart';
void main() {
  print(StringExtension('julio').capitalizar());
  print('julio bitencourt'.primeiraPalavra());
}
```

Também existe a possibilidade de realizar as chamadas através do nome da `extension StringExtension('')`, para haver uma distinção caso ocorra algum conflito de membros iguais em extensions diferentes usadas ao mesmo tempo.

8.7 Se liga aí

- Ao sobrescrever `==`, você indica que instâncias do seu objeto possuem atributos que, quando iguais, devem representar um mesmo objeto, e essa implementação deve ser reflexiva, simétrica e transitiva. Da mesma forma, sempre sobrescreva o `hashCode` para refletir os mesmos resultados para objetos que são considerados iguais pelo `==`.
- Ao usar construtores, use a forma abreviada de inicialização em construtores sempre que possível e, fazendo isso, não declare o tipo da variável e não initialize um corpo com `{}`.

Ruim	Bom
<code>Pessoa(int idade) {this.idade = idade;}</code>	<code>Pessoa(this.idade);</code>
<code>Pessoa(int this.idade);</code>	<code>Pessoa(this.idade);</code>
<code>Pessoa(this.idade) {}</code>	<code>Pessoa(this.idade);</code>

- Embora ainda permitido, não utilize o modificador `new` na inicialização de objetos.

Ruim	Bom
<code>new Pessoa(23);</code>	<code>Pessoa(23);</code>

- Dê preferência à criação de mixins com o modificador `mixin`, ou nomeie a classe com o sufixo `Mixin`, indicando semanticamente a outros desenvolvedores que você garante seu uso como tal, sem que haja quebra de código futuramente.

Ruim	Bom
<code>abstract class Assalariado {}</code>	<code>mixin Assalariado {}</code>
<code>class Assalariado {}</code>	<code>class AssalariadoMixin {}</code>

- As extensions são incríveis, de fato, mas isso não significa que você deve sair criando tudo através delas. Deve existir uma cautela e análise se faz sentido definir tal funcionalidade para uma API através de uma

extension, ou é algo que necessita uma API separada para melhor controle e facilidade de uso.

8.8 É com você

1 - Construtores nomeados também podem ser privados. Um caso de uso é para a construção de um `Singleton`, que é a garantia de existência de apenas um objeto de um tipo criado. Pesquise como implementar esse padrão em Dart (dica: existe mais de uma forma).

2 - Pesquise sobre lista de inicializadores em construtores, uma outra forma de inicializar variáveis com construtores.

3 - É possível encadear chamadas para outros construtores da classe. Pesquise a respeito disso.

4 - Existe uma palavra reservada denominada `covariant` que pode ser utilizada em algumas situações na sobrescrita de métodos. Pesquise o seu funcionamento.

Até aqui

A implementação de todos esses conceitos de Orientação a Objetos é muito similar entre diferentes linguagens, então, se você já teve uma experiência prévia no assunto, este capítulo foi fácil de ser entendido. Ainda mais Dart, que leva muito em consideração nas suas escolhas de design a familiaridade que o programador ou programadora já tem ao vir com um *background* de diferentes linguagens de programação.

Algo que foge do escopo deste livro mas é extremamente recomendado inclusive, é o aprendizado dos famosos *Design Patterns*, uma série de padrão de soluções para problemas comuns relacionados ao paradigma de Orientação a Objetos amplamente divulgados e aceitos pela comunidade de desenvolvedores. Anote essa dica de estudo enquanto seguimos para o próximo assunto do livro logo ali no capítulo seguinte.

CAPÍTULO 9

Generics<T> e as estruturas de dados

Em computação, estrutura de dados é um dos ramos responsáveis por estudar os inúmeros mecanismos para armazenamento de dados. Se em algum momento da sua carreira você já precisou estudar este assunto, com certeza está familiarizado com o funcionamento de vetores, filas, pilhas, árvores, entre outros. Mas a verdade é que as linguagens de programação costumam abstrair todos esses conceitos, muitas vezes complexos, em estruturas prontas e de fácil uso. É exatamente o que Dart também faz.

Essas mesmas classes costumam ser implementadas utilizando um outro conceito conhecido por *generics*, uma forma de reforçar a segurança de tipos além do reaproveitamento de código. Então, durante este capítulo, vamos entender melhor:

- O funcionamento de *generics*;
- As estruturas de dados presentes na linguagem.

9.1 O que é esse tal de generics?

Ao contrário do que muitos podem achar, entender e usar *generics* é algo simples. Olhe estas duas listas vazias:

```
List vogais = [];
List<String> consoantes = [];
```

Na primeira, denominada `vogais`, o compilador vai instanciá-la como `List<dynamic>` por padrão, o que significa que é permitido adicionar qualquer objeto dentro dela, por mais que não faça qualquer sentido uma lista de vogais conter um número, por exemplo. Na segunda, estamos tipando-a explicitamente como `List<String>`, então apenas valores de strings podem ser adicionados, o que faz total sentido.

Generics é justamente isso, esse tipo que é informado entre < >, que funciona como um limitador ou um reforço dos tipos permitidos e utilizados pela instância sendo criada. É mais uma forma de auxiliar o *type system* da linguagem para obter os benefícios das validações estáticas de tipos.

Vamos ilustrar melhor com um exemplo prático. Imagine que possuímos um sistema de pedágio e precisamos implementar uma fila onde os carros vão entrar e aguardar até que chegue a sua vez de realizar o pagamento e seguir o caminho.

```
abstract class VeiculoAutomotor {}
class Carro extends VeiculoAutomotor {}
class FilaPedagio {
    List _veiculos = [];
    void entrarNaFila(veiculo) {
        _veiculos.add(veiculo);
    }
}
void main() {
    final fila = FilaPedagio();
    fila.entrarNaFila(Carro());
    fila.entrarNaFila(Carro());
}
```

Tudo está funcionando perfeitamente até que o pedágio precisa começar a cobrar de caminhões também, porém, um caminhão deve entrar em uma fila diferente, já que eles são mais lentos. Mas o(a) novo(a) programador(a) responsável pela alteração não prestou muita atenção e sem perceber definiu que caminhões poderiam entrar na mesma fila:

```
class Caminhao extends VeiculoAutomotor {}
void main() {
    final fila = FilaPedagio();
    fila.entrarNaFila(Carro());
    fila.entrarNaFila(Caminhao());
}
```

O código compilou perfeitamente, foi para produção, e o desastre estava feito. Culpa de quem implementou primeiro ou de quem fez a modificação? De ambos. É complicado porque a implementação da nossa fila permite que

seja inserido qualquer objeto nela e, como bem sabemos, o parâmetro do método `entrarNaFila()` não possui tipagem definida explicitamente, então ele é `dynamic`. E mesmo que estivesse tipado com a superclasse `VeiculoAutomotor`, ainda aceitaria ambos.

É necessário garantir que a instância de uma fila aceite sempre os mesmos tipos de veículos, então, a princípio a solução é separar em duas classes.

```
class FilaPedagioCarro {
    List _veiculos = [];
    void entrarNaFila(Carro veiculo) {
        _veiculos.add(veiculo);
    }
}
class FilaPedagioCaminhao {
    List _veiculos = [];
    void entrarNaFila(Caminhao veiculo) {
        _veiculos.add(veiculo);
    }
}
void main() {
    var filaCarro = FilaPedagioCarro();
    filaCarro.entrarNaFila(Carro());
    var filaCaminhao = FilaPedagioCaminhao();
    filaCaminhao.entrarNaFila(Caminhao());
}
```

Problema resolvido, temos duas filas diferentes e que aceitam apenas carros ou caminhões. Mas e se amanhã for necessária uma fila só para motos? Vamos continuar duplicando código? Essa solução definitivamente não é boa. Então é aí que entra o tal do *generics*:

```
class FilaPedagio<T> {
    List<T> _veiculos = [];
    void entrarNaFila(T veiculo) {
        _veiculos.add(veiculo);
    }
}
```

Ao utilizar o `<T>`, estamos tipando a classe de forma genérica. Agora é possível definir um tipo `T` em sua inicialização e reutilizá-lo em toda a classe. Supondo que criamos uma instância utilizando `FilaPedagio<Carro>()`, esse `T` se torna um `carro`. A lista de veículos que está definida como `List<T>` aceitará também apenas carros, assim como o método `entrarNaFila(T)`.

Esse caractere `T` é apenas um nome de referência genérico para o tipo, podendo ser qualquer outro (veremos uma padronização).

```
void main() {  
    final filaCarro = FilaPedagio<Carro>();  
    filaCarro.entrarNaFila(Carro());  
    filaCarro.entrarNaFila(Caminhao()); // Erro  
    final filaCaminhao = FilaPedagio<Caminhao>();  
    filaCaminhao.entrarNaFila(Caminhao());  
}
```

Com a mesma implementação de `FilaPedagio`, é possível garantir que uma instância dela vai conter apenas objetos de um tipo. Ao tentar inserir um caminhão na fila de carros, por exemplo, o compilador acusará um erro. Como já comentado, essa abordagem traz dois excelentes benefícios:

- O reutilização de código, tornando a implementação genérica.
- A garantia do uso de tipagem estática e os seus benefícios.

É possível definir mais de um tipo genérico a ser utilizado separando por vírgula, como `FilaPedagio<T, E, K, V>`. O limite de tipos é o bom senso, mas provavelmente você nunca precisará definir mais que dois tipos. Um bom exemplo de uso de mais de um tipo é a coleção `Map<K, V>`, que especifica um tipo para a chave `K` e outro para o valor `V`.

Restringindo um tipo

O governo instalou uma ciclovia que precisa passar pelo nosso pedágio. Segundo as leis de trânsito, uma bicicleta não deve ser cobrada ao passar por pedágios, e antes que alguma pessoa desavisada crie uma `FilaPedagio<Bicicleta>`, precisamos restringir o tipo aceito na fila.

```

class Bicicleta {}
class FilaPedagio<T extends VeiculoAutomotor> {
  List<T> _veiculos = [];
  void entrarNaFila(T veiculo) {
    _veiculos.add(veiculo);
  }
}
void main() {
  final filaCarro = FilaPedagio<Carro>();
  final filaCaminhao = FilaPedagio<Caminhao>();
  final filaBicicleta = FilaPedagio<Bicicleta>(); //Erro
}

```

Ao utilizar uma classe com o `extends` na definição do tipo genérico, é possível especificar que apenas ela ou as subclasses dela serão aceitas. No nosso caso, apenas `VeiculoAutomotor`, `Carro` ou `Caminhao`.

Métodos genéricos

Inicialmente, Dart suportava apenas a definição de *generics* em classes, mas posteriormente foi estendido para métodos e funções. Por exemplo:

```

T ultimo<T extends num>(List<T> itens) {
  T ultimo = itens.last;
  return ultimo;
}
void main() {
  print(ultimo<int>([10, 20, 30])); // > 30
  print(ultimo([1.4, 2, 42.0])); // > 42.0
}

```

É possível utilizar *generics* na definição de parâmetros, variáveis e até mesmo no retorno da função.

9.2 Se liga aí

- Utilize as convenções de nomenclatura para definir o nome do tipo genérico. Os mais comuns são `E`, `K`, `V`, `R` e `T`. Escolha o que mais

fizer sentido de acordo com o uso.

- E significa *element* (elemento), como `List<E>`, o tipo de elemento da lista.
- K *key* e V *value* são os famosos chave e valor usados no `Map<K, V>`.
- R de *return* é utilizado em casos para definição do tipo de retorno.

```
abstract class Classe<R> {  
  R metodo();  
}
```

- Para os demais casos, τ de type (tipo) é suficiente.

9.3 Estruturas de dados

Estrutura de dados na verdade é uma área da computação muito ampla e complexa de certa forma. Existem inúmeros algoritmos na literatura que abordam diferentes estratégias para se armazenar e manipular essas grandes quantidades de informações, sempre buscando o máximo de otimização.

As linguagens de programação costumam abstrair essa complexidade e disponibilizar em seu *core* as principais estruturas já implementadas, que é o que Dart também faz através da library `dart:core`, que contém as implementações padrão de `List`, `Set`, `Map` e `Queue`. Implementações que diferem dessas básicas também podem ser encontradas na library `dart:collection`.

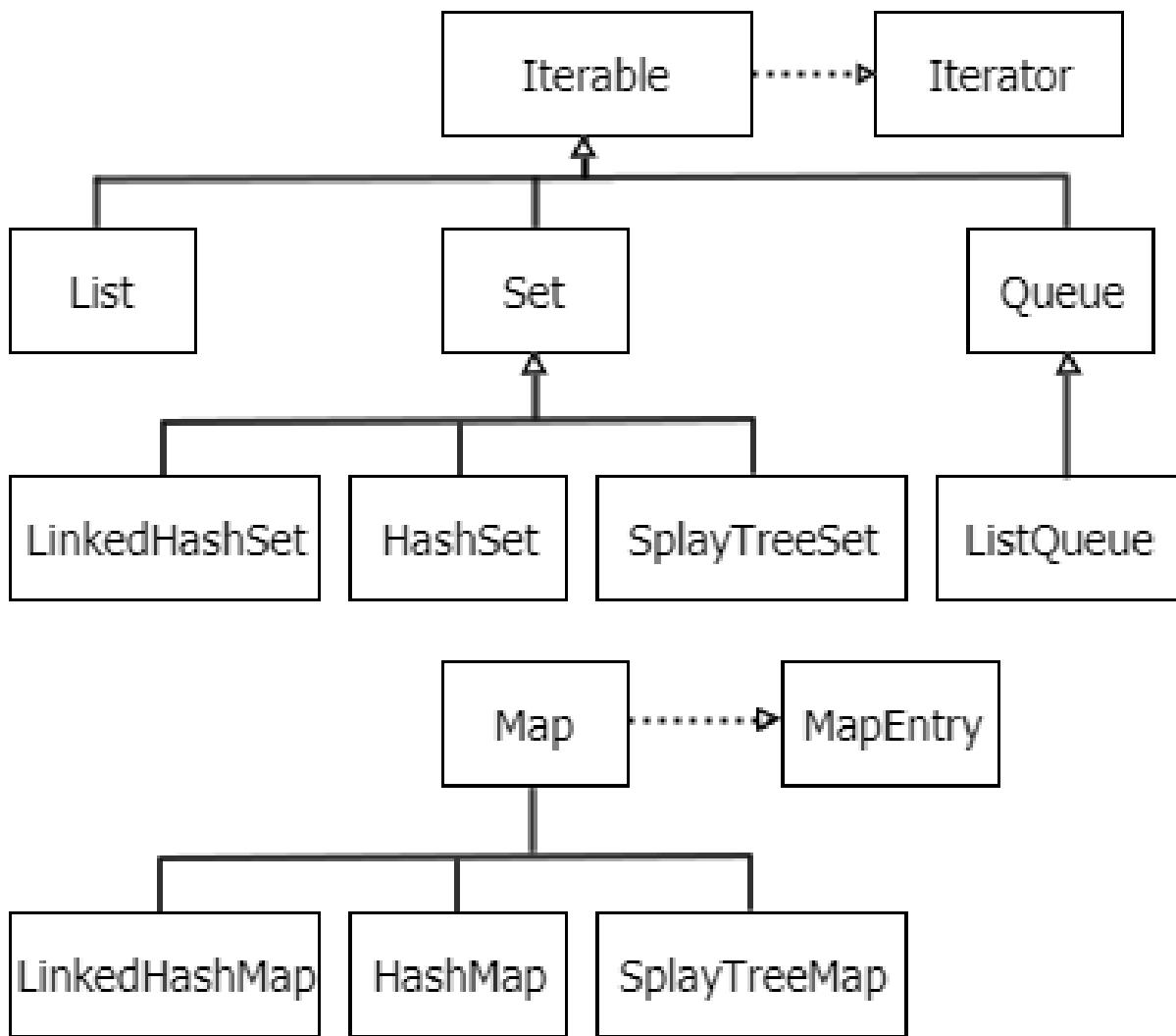


Figura 9.1: Organograma de classes de coleções.

O organograma da imagem representa as principais implementações disponíveis. A estrutura hierárquica não representa fielmente o que encontramos na implementação do código, uma vez que existem algumas classes, interfaces e mixins auxiliares entre elas, mas já dá a ideia necessária de como elas se organizam.

Você vai notar que essas estruturas são sempre ótimas candidatas para utilização de generics, justamente porque elas são criadas para manipular qualquer tipo de dado existente. Em mais de 90% das vezes você utilizará apenas `List` ou `Map`, mas vamos explorar outras opções.

List

Em várias linguagens, a estrutura mais básica para armazenamento de dados é o famoso `array`, que não existe em Dart. Esse papel é então desempenhado por uma simples `List`. As listas com certeza são a estrutura mais utilizada, afinal são ótimas para guardar informações sequencialmente. Eis uma lista em sua forma mais básica:

```
final vogais = [];
vogais.add('a');
vogais.add(42);
print(vogais); // ['a', 42]
```

Justamente por serem utilizadas com frequência, Dart disponibiliza uma forma especial de instanciar listas. Em vez de utilizar o construtor `List()`, são utilizados colchetes `[]`. Essa forma convencional de instanciar objetos utilizando algum caractere especial também é conhecida como forma literal.

No exemplo anterior, `vogais` é uma lista vazia, tipada como `List<dynamic>` pelo compilador, que permitirá a adição de objetos de qualquer tipo. Para evitar esse comportamento, o ideal é sempre utilizarmos generics para tipar a lista:

```
final risadas = <String>[];
risadas.addAll(['kkk', 'haha', 'rsrs']);
```

Agora `risadas` permite apenas a adição de strings. Inclusive, o método `addAll()` permite adicionar vários valores de uma outra lista. É possível também já definir valores na inicialização:

```
final alfabeto = ['a', 'b'];
alfabeto.add('c');
```

Dessa forma, o compilador infere o tipo da lista baseado nos objetos da inicialização, `alfabeto` é tipado como `List<String>`. Esta forma literal sempre cria uma lista expansível que permite a adição de novos valores. Para criar uma lista de tamanho fixo, deve ser utilizado o seu construtor `filled()`:

```
final vogais = List<String>.filled(5, 'e');
vogais[0] = 'a';
```

```
vogais[4] = 'u';
print(vogais); // > [a, e, e, e, u]
// vogais.add('a'); // Erro
```

Com ele, `vogais` se torna uma lista com cinco espaços para objetos. Antigamente os espaços da lista eram preenchidos com valores nulos, mas com *null-safety* agora é obrigatório definir um objeto para ocupar os valores, no exemplo, o 'e'. A tentativa de adicionar novos elementos resultará sempre em erro, sendo possível modificar apenas os existentes através da sua posição com o operador `[]`.

Isso porque cada elemento da lista é armazenado em um índice de acordo com a sua posição. Elas são estruturas de índice zero, o que indica que o primeiro elemento tem índice 0, o segundo tem índice 1, e assim por diante, sendo que o último elemento sempre terá índice igual ao tamanho da lista menos 1. Para criar uma lista imutável, de tamanho fixo onde os valores não podem ser alterados, há o construtor `List.unmodifiable()`.

É possível iterar seus valores através da ordem dos índices e existem quatro maneiras de fazer isso. A primeira e mais comum é através de um `for` convencional, que varre toda a lista com uma variável de controle `i` representando o índice dos elementos.

```
for(var i = 0; i < risadas.length; i++)
  print(risadas[i]);
```

Se o índice não é uma informação importante, é possível utilizar a segunda opção de `for`, também conhecida como `for-in`, que tem uma sintaxe mais amigável e itera a lista sequencialmente retornando o objeto `r` de cada posição.

```
for (final r in risadas)
  print(r);
```

A terceira forma é mais prática e funcional, com o `forEach`. É uma boa alternativa caso você apenas queira executar uma função para cada elemento da lista de forma sequencial.

```
risadas.forEach((r) => print(r));
```

Já a última forma conta com o uso explícito da interface `Iterable`, que veremos a seguir.

9.4 A interface Iterable

Como visto no organograma das classes, várias coleções implementam essa interface e o papel dela é definir um comportamento padrão da iteração sobre os elementos. Na verdade, as três formas vistas anteriormente, de iterar através dos tipos de `for`, só são possíveis com objetos que herdam de `Iterable`.

Essa interface disponibiliza um `Iterator` que funciona como um ponteiro para o próximo elemento da coleção, por exemplo:

```
Iterator i = risadas.iterator;
while (i.moveNext())
    print(i.current);
```

A cada chamada para o `moveNext()` é retornado um `bool` avisando se existe ou não um próximo elemento na coleção; caso exista, o `Iterator` alterará o seu ponteiro para o próximo elemento, guardando uma referência para ele na variável `current`. Por isso, é possível iterar também através de um `while`. O exemplo imprime todos os elementos de `risadas`.

Além do `Iterator`, a maioria dos métodos e atributos de `List`, padrões para manipulação dos dados, é definida na própria interface, por exemplo:

- `length` : propriedade que retorna a quantidade de elementos da coleção;
- `isEmpty` / `isNotEmpty` : propriedades que verificam se a coleção está vazia;
- `first` / `last` : propriedades para recuperar o primeiro e último elementos da coleção;
- `forEach()` : a própria implementação do `forEach` é definida na interface.

- `contains()` , `where()` , `take()` , `skip()` etc.

Set

O set é uma estrutura que tem em sua essência não permitir a existência de mais de um objeto igual armazenado, sendo essa validação feita através do `==` e `hashCode` . São três os tipos principais de sets: `LinkedHashSet` , `HashSet` e `SplayTreeSet` .

```
final megasena = Set<int>();
megasena.addAll([44, 35, 4, 11, 29, 4, 35, 57]);
print(megasena); // > {44, 35, 4, 11, 29, 57}

final vogais = <String>{'a', 'e', 'i', 'a', 'o', 'u'};
print(vogais); // > {a, e, i, o, u}
print(vogais.elementAt(1)); // e
print(vogais[0]); // Erro
```

Diferente de uma lista, um set pode ser criado através de um construtor padrão com `Set()` e também com sua forma literal `{}` . Ambas as formas produzem por padrão uma implementação de `LinkedHashSet` , sendo equivalente ao seguinte:

```
import 'dart:collection';
final megasena = LinkedHashSet();
```

Note que para utilizar o `LinkedHashSet` diretamente é necessário importar o pacote `dart:collection` . Esse pacote contém um conjunto de classes e utilidades extras para as coleções em Dart, com implementações de alta performance e otimizadas para execução tanto na VM quanto em JavaScript, e eventualmente é necessário importá-lo se desejado utilizar algo diferente do padrão.

Os elementos de um set não podem ser acessados diretamente pelo operador `[]` , pois esse é definido apenas em uma list. Porém, é possível acessar um elemento de um índice específico através do método `elementAt()` de `Iterable` . Um `LinkedHashSet` tem um funcionamento semelhante ao de

uma lista, pois ele mantém a ordem de inserção dos elementos conforme os valores impressos anteriormente no set `megasena`.

A segunda implementação é o `HashSet`. Considere:

```
final megasena = HashSet<int?>();
megasena.addAll([44, 35, 4, 11, null, 29, 4, null, 35, 57]);
print(megasena); // > {35, 4, 57, 11, null, 44, 29}
```

Diferente do `LinkedHashSet`, `HashSet` não mantém a ordem de inserção e sim ordena os elementos através da implementação do `hashCode` desses elementos. Essa ordenação via *hash* garante que operações como `add()`, `remove()` e `contains()` sejam feitas constantemente, com boa performance.

Já a terceira implementação é o `SplayTreeSet`, que mantém a ordem dos seus elementos com base na implementação do seu `compareTo()`, em vez do `hashCode`. E esse método é disponibilizado pela interface `Comparable`. Vamos ver mais sobre ela a seguir.

9.5 A interface Comparable

Essa interface determina como deve ser feita a comparação entre dois objetos através do seu método `int compareTo(objeto)`. Ela é essencial para casos em que é necessário ordenar objetos em uma coleção. Várias classes em todo o SDK a implementam e um exemplo clássico de uso é a ordenação de listas com o método `sort()`.

```
final vogais = ['e', 'i', 'a', 'o', 'u'];
vogais.sort();
print(vogais); // > [a, e, i, o, u]
```

A saída é a lista em ordem alfabética, pois a classe `String` implementa a classe `Comparable`. A implementação do `compareTo()` deve retornar um valor inteiro respeitando as regras:

- Retornar um número negativo `-1` caso o objeto deva estar antes do objeto passado por parâmetro.
- Retornar o `0` caso ambos objetos possuam a mesma ordem.
- Retornar um número positivo `1` caso o objeto deva ficar em uma ordem posterior ao objeto passado por parâmetro.

O `SplayTreeSet` então exige que todos os seus elementos implementem essa interface, fazendo com que ele não aceite um objeto `null`, por exemplo (por mais que seja tipado com um tipo *nullable*), diferente das outras implementações de set.

```
final megasena = SplayTreeSet<String>();
megasena.addAll(['44', '35', '4', '11', '4']);
print(megasena); // > {11, 35, 4, 44}
```

No exemplo, o set guardou os elementos de acordo com a implementação do `compareTo` da `String`, de forma alfabética. Pode não ser o comportamento ideal, pois numericamente é estranho o `11` ficar à frente do `4`. Mas o `SplayTreeSet` também permite especificar uma função de comparação no seu construtor:

```
final megasena = SplayTreeSet<String>((a, b) {
  return int.parse(a).compareTo(int.parse(b));
});
megasena.addAll(['44', '35', '4', '11', '4']);
print(megasena); // > {4, 11, 35, 44}
```

Dessa forma, definimos como os objetos devem ser comparados utilizando a própria implementação do `int` para o `compareTo()`, o resultado são os números impressos em ordem natural.

Queue

O conceito de *queue* representa a abstração de uma estrutura de dado para manipulação dos objetos através de uma fila FIFO (*first in, first out*), onde os objetos entram de um lado e saem de outro. Em Dart, ela é muito parecida com uma fila e também estende de `Iterable`, mas inclui métodos para manipulação de dados em ambas as pontas, início e fim.

```
final megasena = Queue<int>();
megasena.addAll([11, 35]);
megasena.addFirst(4); // > {4, 11, 35}
megasena.removeLast(); // > {4, 11}
megasena.addLast(44); // > {4, 11, 44}
megasena.removeFirst(); // > {11, 44}
```

De fato, ao instanciar uma `Queue()`, o retorno será uma implementação padrão de `ListQueue`, uma classe que utiliza internamente uma list para armazenar os dados, com a diferença de que mantém um ponteiro para o `head` (início) e `tail` (final). Não há uma forma literal para instanciar uma queue, isso pode ser feito apenas através do construtor.

Map

Um `Map`, ou também muito conhecido como dicionário, é uma estrutura para armazenamento de dados com chave-valor.

```
final clientes = Map<int, String>();
clientes[1] = 'Rafael';
clientes[1] = 'Juliana';
clientes.putIfAbsent(2, () => 'João');
clientes.putIfAbsent(2, () => 'Maria');
print(clientes); // {1: Juliana, 2: João}

final usuario = {'Nome':'Julio', 'Linguagens': ['dart', 'java']};
usuario.putIfAbsent('Github', () => 'JHBitencourt');
print(usuario.runtimeType); // > _InternalLinkedHashMap<String, Object>
```

Semelhante a um `set`, a forma literal de inicializar um `map` também é entre chaves, só que incluindo a chave e valor `{chave:valor}`, como em `usuario` acima. Repare também como o map utiliza dois valores generics, pois trabalha com dois tipos de dados ao mesmo tempo.

É possível inserir valores no map através do operador `[]` referenciando a chave e, caso já exista um valor com essa chave, ele será atualizado, pois não são permitidas chaves duplicadas. Uma outra forma de inserir um novo

valor é com o método `putIfAbsent`, que vai inserir um valor apenas se a chave não existir; caso já exista, nada é feito e o valor não é atualizado.

Com o mesmo operador ainda é possível trazer valores do map baseado na chave:

```
print(usuario['Nome']!.runtimeType); // > String  
print(usuario['Sobrenome']?.runtimeType); // > null
```

Mas existe um detalhe: o retorno é um objeto *nullable*, afinal a chave informada pode ou não existir no map. Por isso, é necessário fazer o *unboxing* do valor e caso você tenha certeza da existência do valor pode ser utilizado o *bang* `!`. Caso contrário, precisa validar se o valor é diferente de `null`.

Diferente das outras estruturas, o `Map` não é um `Iterable`, entretanto, tanto as chaves, quanto os valores e os elementos podem ser obtidos como `Iterable`.

```
Iterable chaves = clientes.keys;  
Iterable valores = clientes.values;  
Iterable<MapEntry> elementos = clientes.entries;  
print(elementos.first.key); // > 1  
print(elementos.first.value); // > Juliana
```

Um `MapEntry` representa um elemento do `Map` e possui a sua chave e valor. Existem três tipos de `Map`: `LinkedHashMap`, `HashMap` e `SplayTreeMap`. A diferença entre eles está na forma com que os elementos são ordenados internamente.

A ordenação de um map é sempre feita através das chaves (*keys*). Ao criar um map com o construtor padrão ou de forma literal, é utilizada a implementação `LinkedHashMap`, que mantém a ordem dos elementos de acordo com a inserção:

```
final map = LinkedHashMap.fromIterables(['3', '1', '2'], [1, 2, 3]);  
print(map); // > {3: 1, 1: 2, 2: 3}
```

O construtor `fromIterables` cria um map a partir de duas listas: a primeira, com as chaves e a segunda, com os valores. Já o `HashMap` possui uma melhor performance para busca e alteração de valores, mas não garante a ordem dos elementos:

```
final map = HashMap.fromIterables(['3', '1', '2'], [1, 2, 3]);
print(map); // > {1: 2, 3: 1, 2: 3}
```

E por fim, o `SplayTreeMap` ordena seus elementos com base na implementação do `compareTo` das chaves, em geral essa ordenação gera uma pior performance para busca e alterações de valores:

```
final map = SplayTreeMap<String, int>
  .fromIterables(['3', '1', '2'], [1, 2, 3],
  (a, b) {
    return int.parse(a).compareTo(int.parse(b));
  });
print(map); // > {1: 2, 2: 3, 3: 1}
```

Com todas essas coleções apresentadas, você já é capaz de diferenciar e escolher a melhor para cada problema, embora na maioria dos casos uma simples lista seja suficiente. É possível inclusive implementar a sua própria coleção, o pacote `dart:collection` possui diversas classes, interfaces e mixins que facilitam muito essa tarefa.

Também existem inúmeros outros métodos e atributos úteis para todas as estruturas de dados citadas neste capítulo, o que torna inviável citar todos. Por isso, reserve um tempo para explorar essas classes e os seus recursos disponíveis.

9.6 Se liga aí

- Inicialize as coleções de forma literal sempre que possível.

Ruim	Bom
<code>var set = Set();</code>	<code>var set = <String>{};</code>

Ruim	Bom
<code>var map = Map();</code>	<code>var map = {};</code>

- Não valide se uma coleção está vazia com o `length`. Ele pode ser lento e não é idiomático para o propósito.

ruim	bom
<code>if(list.length == 0)</code>	<code>if(list.isEmpty)</code>
<code>if(!list.isEmpty)</code>	<code>if(list.isNotEmpty)</code>

9.7 É com você

1 - Implemente a interface `comparable` no objeto `Programador` do capítulo anterior, de Orientação a Objetos. Após isso, salve os elementos em uma lista e ordene-a com `sort`.

2 - Pesquise e aplique conceitos da teoria dos conjuntos, união, interseção e diferença com a ajuda do `Set`. Dica: já existem métodos no `Set` para isso.

3 - Explore os métodos das coleções, crie exemplos para `contains`, `elementAt`, `firstWhere`, `lastWhere`, `expand`, `take`, `skip`, `reduce`, `fold`, `join` e `every`.

4 - O `Iterator` permite iterar os elementos a partir do primeiro elemento da lista. Seria possível iterar a coleção de trás para a frente? Pesquise sobre o `BidirectionalIterator` e tente implementá-lo.

Até aqui

Agora que exploramos os generics você vai começar a notar as diferentes APIs em Dart que utilizam este recurso, afinal, não é algo exclusivo das

estruturas de dados. Então aproveite essa funcionalidade para sempre tipar esses objetos e fornecer mais informações ao compilador a respeito dos tipos com que você deseja trabalhar. A partir do próximo capítulo, entraremos no mundo da programação assíncrona e veremos como Dart lida com esse assunto que muitas vezes é de certa forma temido.

CAPÍTULO 10

Concorrência

Um dos assuntos mais complexos quando se trata de desenvolvimento de software é a famosa concorrência, segundo a qual um ou mais processos diferentes podem ser executados no mesmo espaço de tempo. Se voltássemos algumas décadas atrás, além de sentir falta dos jogos de última geração e das redes sociais, encontrariíamos microprocessadores com um único core capaz de executar apenas uma tarefa por vez. Hoje, com a necessidade de cada vez mais os dispositivos serem portáteis e ágeis, a indústria do hardware avançou muito. Um desses avanços foi o aumento da quantidade de cores presente em cada microprocessador, surgindo então o paralelismo de tarefas, que permite a execução de dois ou mais processos simultaneamente, aumentando a agilidade e melhorando a experiência final para o usuário.

Mas de nada adianta um hardware com vários cores e uma alta capacidade de paralelismo se ele estiver em conjunto com um software que não aproveita corretamente. Executar um programa de décadas anteriores (feito para rodar em um único core) em um hardware atual não significa que a performance será boa. Ambos devem trabalhar em conjunto da melhor forma possível, como *yin* e *yang*.

Por conta disso, as linguagens de programação costumam oferecer abstrações para que os desenvolvedores programem de forma concorrente. Em Dart, isso é conhecido como `Isolate`. Então, neste capítulo, falaremos sobre:

- O que são as *isolates*;
- A importância do event loop para a execução dos programas em Dart;
- As diferenças de código assíncrono vs. síncrono;
- Como trabalhar de forma assíncrona com a API de *futures*;
- Os benefícios da *syntax sugar* `async` e `await` ;
- Como obter um *future* mais poderoso com um *completer*.

10.1 Isolates

Uma vez que as linguagens disponibilizam APIs e códigos prontos para trabalharmos com esses recursos, por que isso ainda é considerado complexo? Bem, caso o desenvolvedor ou desenvolvedora não saiba exatamente o comportamento do que está implementando, isso pode levar a aplicação para estados de erro que são extremamente difíceis de serem encontrados.

Um dos problemas clássicos é o uso de memória compartilhada. Imagine um sistema bancário onde várias *threads* acessam e modificam o mesmo endereço de memória que contém uma variável com o saldo disponível de valor 0. Supondo que cheguem dois depósitos diferentes ao mesmo tempo, um de 10 e outro de 20, uma *thread A* faz a leitura da variável de valor 0, incrementa 10 e sobrescreve esse valor na memória. Porém, antes que a *thread A* escrevesse esse novo valor, uma *thread B* fez a leitura do valor como 0, incrementa 20 e sobrescreve o valor da *thread* anterior.

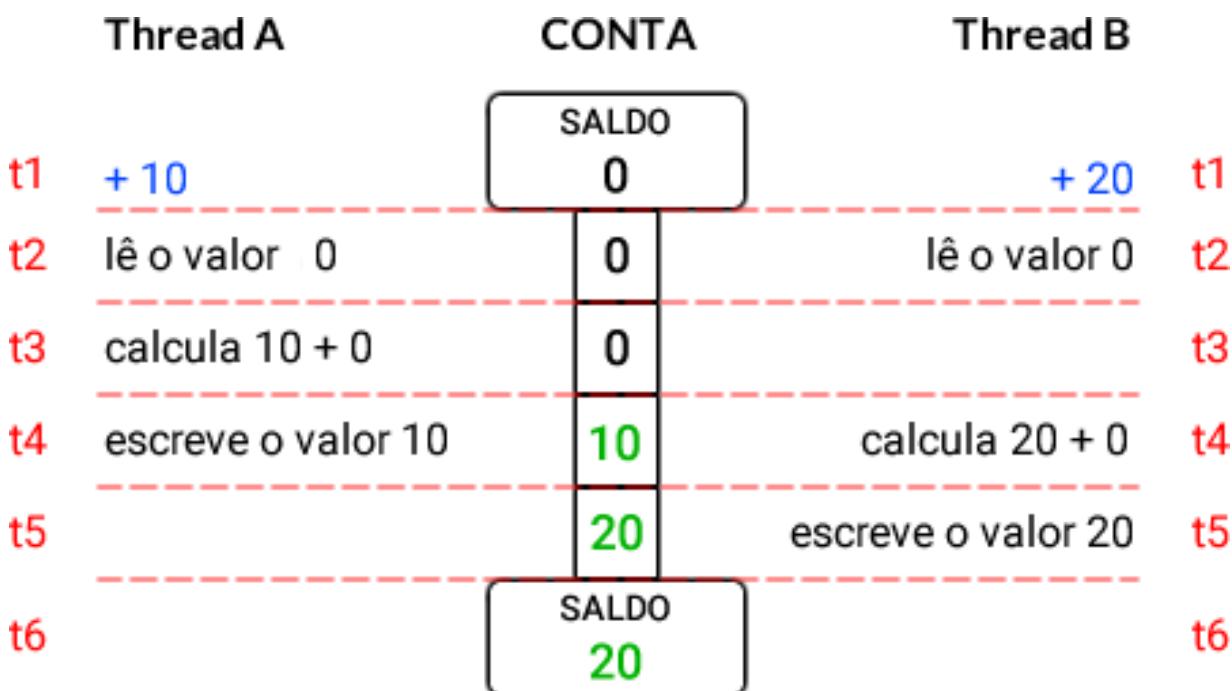


Figura 10.1: Problema em memória compartilhada sem lock.

No final, o valor acrescentado pela `thread A` é ignorado e aí o estrago está feito. O cliente terá um saldo de 20 em vez de 30 e muita dor de cabeça para resolver o problema. E é por conta disso que com certeza você já ouviu falar em *lock* de memórias para controle de acesso a informações críticas, que dependendo, também pode ter seu efeito negativo e impactar na performance do programa. Já pensou a tragédia que seria caso o processo nunca mais liberasse o *lock* da memória? (spoiler: isso acontece e é chamado de *deadlock*).

Em resumo, trabalhar com paralelismo de tarefas é um baita problema e acaba tornando o programa mais suscetível a erros. Porém, Dart traz uma abordagem um pouco diferente. A primeira coisa que você deve entender é que Dart é uma linguagem *single-thread*, ou seja, todo o código que executamos roda em uma única *thread*, ou melhor dizendo, em uma `Isolate`. Esse código é executado de forma sistemática, instrução após instrução, respeitando essa sequência.

O conceito de `Isolate` é na verdade semelhante ao de uma *thread*, com a diferença de que cada `Isolate` possui sua pilha de memória única, e esse estado não pode ser acessado por nenhuma outra `Isolate`, daí vem a origem desse nome, tudo é mantido isolado.

Esse simples fato é o suficiente para evitar os problemas de memória compartilhada e também dispensa a necessidade de ter que fazer *lock* para acesso aos dados. Mas, para entender melhor uma `Isolate`, vamos dar uma olhada em como nosso código em Dart é realmente executado por baixo dos panos.

10.2 Event Loop

Assim que utilizamos `dart run main.dart` para rodar um programa qualquer, Dart inicializa uma `Isolate` para executá-lo. Com ela, são criadas duas *queues* e um processo cíclico infinito que chamamos de *event loop*. Essas *queues* são do tipo FIFO (*first in, first out*), o primeiro elemento a entrar é também o primeiro a sair.

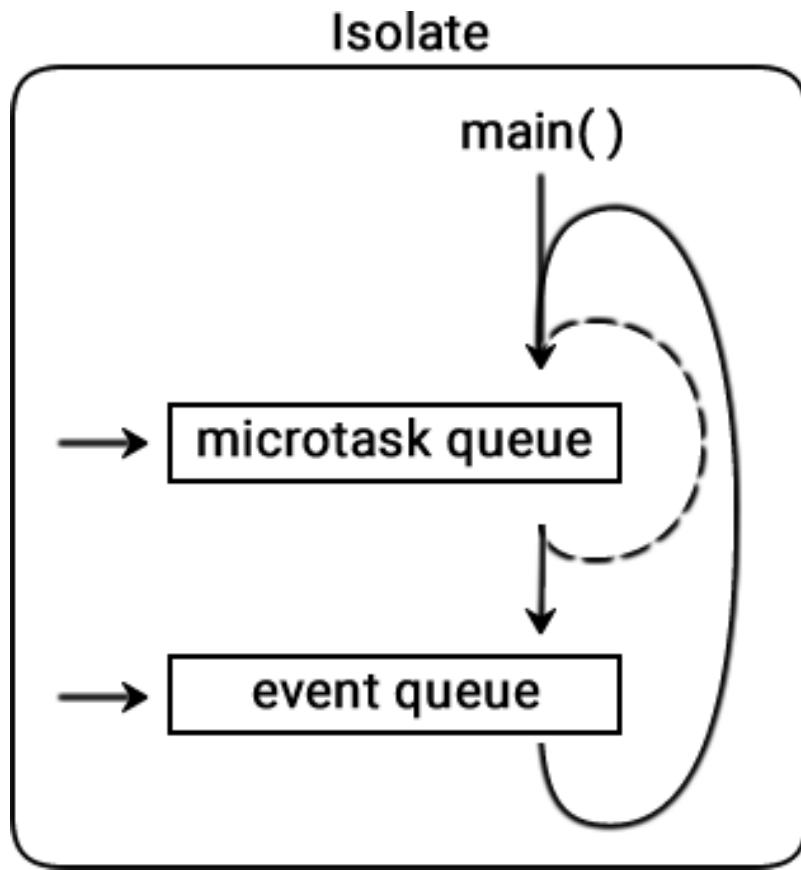


Figura 10.2: Event Loop.

O primeiro passo para a execução do programa é rodar toda a função `main()`, então é ela que a *isolate* chamará primeiro, junto de toda a árvore de métodos ou funções síncronas que são chamadas dentro dela. Ao finalizar, é inicializado o *event loop* que fica aguardando algum elemento chegar em uma das *queues* para ser executado.

A primeira queue, chamada de *microtask queue*, é usada para realizar tarefas internas assíncronas e que necessitam de uma prioridade maior de execução. Afinal, o *event loop* sempre vai executar primeiro todos os elementos presentes nela antes de executar algo presente na *event queue*. Ela pode ser utilizada internamente pelas libraries para liberar algum recurso ou realizar alguma atualização no estado de alguma variável. Mas é provável que você nunca precisará utilizar essa *queue* diretamente.

A segunda e mais interessante é chamada de *event queue*. A partir do momento em que a nossa aplicação está rodando, uma série de eventos

assíncronos pode ocorrer e não há como prevê-los ou determinar em qual ordem eles devem ser executados. Por exemplo, as ações que o usuário pode fazer, um clique do mouse em um botão, o upload de algum arquivo, um gesto de *swipe* em um aplicativo, todos são eventos que entram na fila dessa *queue* para execução.

Mas sem dúvida os principais usuários dessa *queue* são as APIs de *future* e *streams*, que exploraremos em seguida. Elas trabalham diretamente com o agendamento de eventos para execução. Mas, para assimilar esse conceito de execução com a prática, vamos fazer um debug detalhado e visual do trecho de código a seguir.

```
import 'dart:async';
void main () {
  print('Início main()');
  Timer.run(() => print('Event loop 1')); // #E1
  Timer.run(() { // #E2
    scheduleMicrotask(() => print('Microtask 3')); // #M3
    print('Event loop 2');
  });
  scheduleMicrotask(() => print('Microtask 1')); // #M1
  Timer.run(() => print('Event loop 3')); // #E3
  scheduleMicrotask(() => print('Microtask 2')); // #M2
  print('Fim main()');
}
```

Antes, para deixar claro, a library `dart:async` é necessária para manipular recursos de programação assíncrona. A função `scheduleMicrotask` é utilizada para adicionar elementos na *microtask queue*. Por outro lado, existem várias formas de adicionar eventos na *event queue*, uma delas é através de um `Timer`. Essa classe, como o nome bem indica, permite adicionar um evento para execução após uma determinada duração. Seu construtor `Timer.run()` é apenas um atalho para `Timer(Duration(seconds:0), () {})`, indicando que o evento deve ser imediatamente agendado para execução.

Mas o grande foco desse exemplo é na verdade a ordem em que os dados são impressos, ou melhor, em que os eventos das *queues* são executados:

```
> Início main()
> Fim main()
> Microtask 1
> Microtask 2
> Event loop 1
> Event loop 2
> Microtask 3
> Event loop 3
```

Os blocos de código estão nomeados com `#+número` para facilitar a explicação. Então vamos lá:

- O método `main()` dá início à execução do programa e imprime `Início main()`.
- O `Timer` adiciona os trechos de códigos `#E1` e `#E2` à fila da *event queue*. Nesse momento, ele está falando: "Event Loop, eu tenho dois eventos aqui, vou deixar nessa fila e, quando você tiver um tempo, execute-os".
- Com a função `scheduleMicrotask()` adicionamos o trecho de código `#M1` à fila da *microtask queue*. É o mesmo que falar: "Event Loop, eu tenho esse código aqui para ser executado, vou deixar nessa fila para você, mas ele precisa ser executado antes de qualquer outro evento".
- O `Timer` adiciona o trecho `#E3` à fila da *event queue*.
- O trecho `#M2` é adicionado à fila da *microtask queue*.
- O método `main()` imprime `Fim main()` e ele é finalizado.

Neste momento, este é o estado da aplicação:

```

Terminal
> Início main()
> Fim main()

```

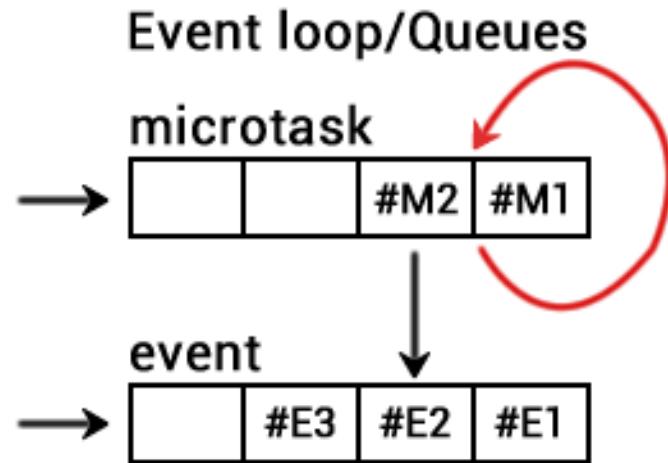


Figura 10.3: Análise event loop, etapa um.

Após o término do `main()`, é inicializado o processo do *event loop*. Repare que a ordem de inserção dos elementos nas queues é sempre respeitada.

- O *event loop* verifica se existem elementos na *microtask queue*. Como existem, todos eles são executados até que ela esteja vazia.
- O trecho `#M1` imprime Microtask 1 .
- O trecho `#M2` imprime Microtask 2 .

```

Terminal
> Início main()
> Fim main()
> Microtask 1
> Microtask 2

```

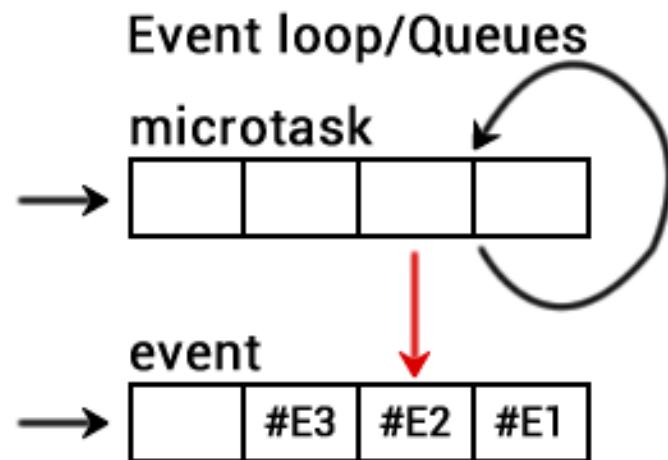


Figura 10.4: Análise event loop, etapa dois.

- Após limpar a *microtask queue*, o *event loop* verifica se existe algum elemento na *event queue*. Caso exista, ele pega apenas o primeiro da fila para execução.

- O trecho `#E1` imprime Event loop 1.

```
Terminal
> Início main()
> Fim main()
> Microtask 1
> Microtask 2
> Event loop 1
```

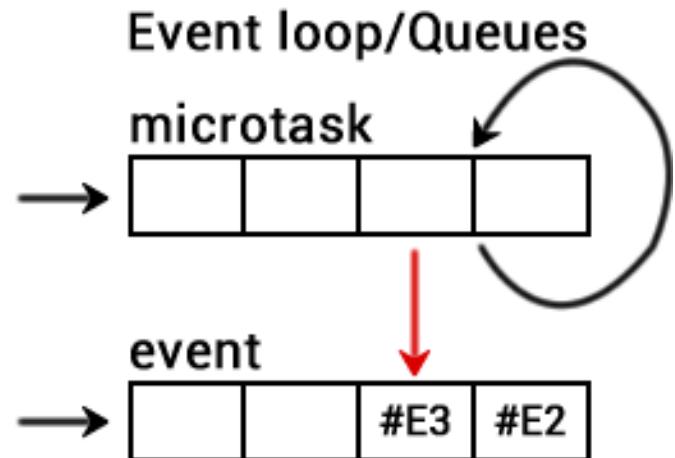


Figura 10.5: Análise event loop, etapa três.

- Neste momento é finalizada uma volta completa do *event loop*, e ele retorna ao início.
- Como dessa vez não existem elementos na *microtask queue*, ele pula direto para a *event queue* e pega o próximo elemento da fila.
- O trecho `#E2` inicia a execução e, se olharmos no seu código, ele utiliza a `scheduleMicrotask` para adicionar o trecho de código `#M3` na fila da *microtask queue*.
- O trecho `#E2` imprime Event loop 2 .

```
Terminal
> Início main()
> Fim main()
> Microtask 1
> Microtask 2
> Event loop 1
> Event loop 2
```

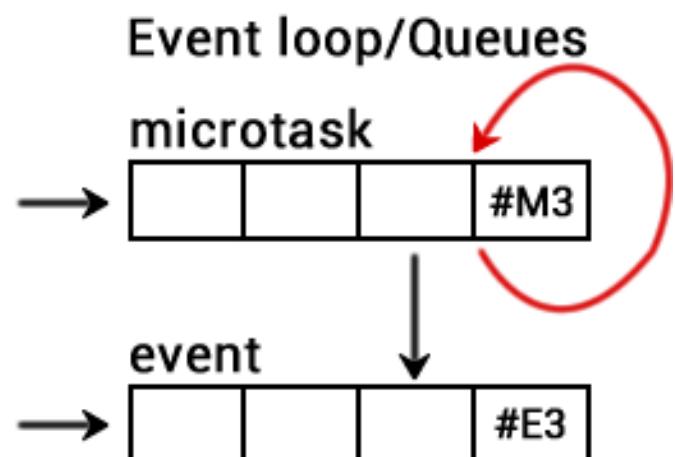


Figura 10.6: Análise event loop, etapa quatro.

- O *event loop* conclui mais uma volta.
- Dessa vez, existe um novo elemento na *microtask queue* e ele é executado.
- O trecho `#M3` imprime `Microtask 3`.
- Com a *microtask queue* vazia, ele parte para a *event queue* e pega o próximo elemento.
- O trecho `#E3` imprime `Event loop 3`.

Terminal

```
> Início main()
> Fim main()
> Microtask 1
> Microtask 2
> Event loop 1
> Event loop 2
> Microtask 3
> Event loop 3
```

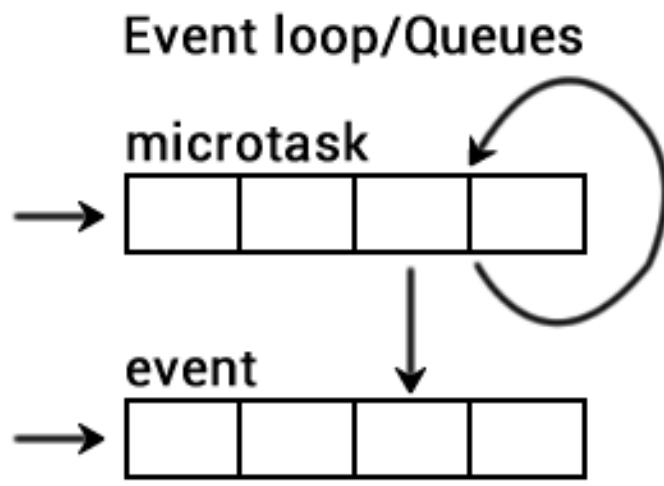


Figura 10.7: Análise event loop, etapa cinco.

Ao fim, como ambas as *queues* estão vazias, o *event loop* para sua execução. Mas caso chegasse qualquer outro elemento novo, ele retomaria e os executaria. Todo esse trabalho realizado pelo *event loop*, que simplificamos ao máximo para ficar fácil de entender, é essencial para o comportamento e execução dos programas em Dart de forma assíncrona. E por falar nisso, você já conhece as diferenças de síncrono e assíncrono?

10.3 Síncrono versus assíncrono

Mesmo após conhecer o funcionamento do *event loop*, a leitura do código assíncrono anterior ainda pode embaralhar um pouco a nossa mente. Afinal, nós seres humanos estamos acostumados a pensar de forma síncrona, pois é algo natural, o que com certeza facilita nosso entendimento e nos torna

também mais propensos a desenvolver um código que execute de forma síncrona. Por exemplo:

```
void main () {
    print('Início main()');
    int index = 0;
    while(index < 10) {
        print('Índice $index');
        index++;
    }
    print('Fim main()');
}
```

Ao ler o código acima, você consegue executá-lo mentalmente e acompanhar o seu fluxo de execução sem muito esforço. Faça esse exercício. Primeiro, o `Início main()` e depois o *loop* com `while` é executado, imprimindo o valor do `index` de 0 a 9 e finalizando o programa com `Fim main()`, tudo conforme o esperado. Um fluxo síncrono, linha a linha.

Agora, imagine que estamos desenvolvendo um aplicativo de clima e, na página inicial, entre as inúmeras informações apresentadas, tem a temperatura atual e a previsão para amanhã. Ambas são consultas que fazemos para uma API de um servidor externo e, por conta disso, o tempo de resposta é dependente desta comunicação. Vamos simular essas chamadas:

```
import 'dart:io';
void main () {
    print(temperaturaAtual());
    print(previsaoAmanha());
}
String temperaturaAtual() {
    sleep(const Duration(seconds: 3));
    return 'Está fazendo 28°';
}
String previsaoAmanha() {
    sleep(const Duration(seconds: 4));
    return 'Amanhã fará 35°';
}
```

Note a presença da função `sleep()` de `dart:io`. Com ela, conseguimos justamente fazer o programa dormir por alguns segundos, simulando o que seria uma requisição para um servidor real.

Ao rodar, o programa ficará três segundos sem responder e será impresso `Está fazendo 28°`, depois mais quatro segundos sem responder para só então imprimir `Amanhã fará 35°`. Ou seja, ainda estamos executando um código síncrono e, durante esses longos sete segundos, a chance de o usuário ter xingado e abandonado o seu sistema é grande. Tudo isso porque o programa roda em uma única `Isolate`:

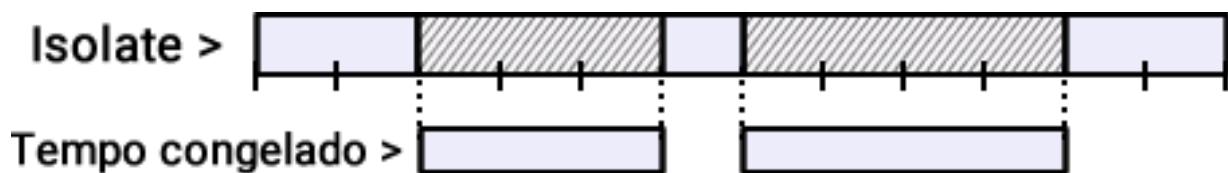


Figura 10.8: Isolate congelada com tarefas síncronas.

Enquanto buscávamos as informações em um servidor fictício, a nossa única *thread* principal ficou congelada, sem que o usuário pudesse realizar qualquer outra ação no sistema. No exemplo, foram três e quatro segundos, mas na vida real uma requisição para uma API externa possui um tempo variável sobre o qual não temos controle.

Além disso, uma única página web ou mobile pode fazer N requisições para APIs diferentes (um *dashboard*, por exemplo). Se impedirmos o usuário de interagir com o sistema, vai dar uma impressão de que ele está congelado, o que é uma péssima experiência de uso.

A solução para esse tipo de problema é utilizar recursos assíncronos. Quando vamos realizar alguma tarefa que poderá levar um tempo maior para obter uma resposta, devemos executá-la de forma assíncrona.

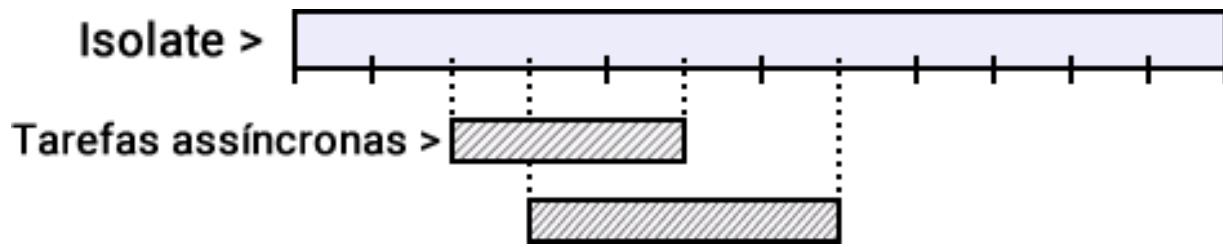


Figura 10.9: Isolate com tarefas assíncronas.

Assim a `Isolate` não fica ocupada em apenas uma tarefa, conseguindo responder a ações do usuário na aplicação, ou qualquer outro evento. Tudo isso obviamente por conta do *event loop* visto anteriormente.

Tarefas como requisições HTTP, operações com banco de dados, operações de IO, manipulação de imagens, entre outras, devem ser adicionadas à *event queue* para que sejam rodadas de forma assíncrona sem travar a execução da `Isolate` principal. Mas no dia a dia de um desenvolvedor Dart, não é comum termos que nos preocupar com adicionar eventos nas *queues*, pois esse trabalho é abstraído por outras libraries da própria linguagem. E dentre essas, podemos destacar como uma das principais, se não a mais, os *futures*.

10.4 Futures

Quando se fala em código assíncrono em Dart, é quase inevitável não assimilar com um *future*. Afinal, *future* funciona como uma API básica para programação assíncrona que trabalha acima do *event loop*, facilitando sua manipulação. De forma resumida, um `Future` representa um valor incompleto que em algum momento no futuro vai completar com algum resultado, muito similar ao conceito de uma *promise* em JavaScript.



Figura 10.10: Estados de um Future.

A ideia é simples, ao criar um `Future`, definimos um trecho de código que deve ser executado de forma assíncrona. Nesse momento, já obtemos de forma síncrona uma referência para um objeto do tipo `Future` que está no estado **incompleto**.

Nosso código é então enviado para a *event queue*, onde eventualmente vai ser executado pelo *event loop* em um dos seus ciclos. Internamente, a própria API utilizará um `Timer` para isso, exatamente como fizemos anteriormente no exemplo de análise do *event loop*.

Assim que esse evento for executado, o future será concluído com um valor ou com um erro. Sendo assim, um *future* pode ter três estados: **incompleto**, **completo com valor** ou **completo com erro**. Em linhas gerais, a anatomia de um *future* pode ser definida como:

```

Future<()
    // Função assíncrona para agendar na event queue
).then(
    // Callback, o que eu quero que execute quando
    // o Future concluir com valor
).catchError(
    // Callback, o que eu quero que execute quando
    // o Future concluir com erro
).whenComplete(
    // Callback, o que eu quero que execute quando
    // o Future concluir com valor ou erro
)

```

Figura 10.11: Anatomia de um Future.

Perceba como a leitura da estrutura de um future é objetiva: "Execute esta função `X`, quando acabar eu quero que o meu método `then` seja chamado com o seu *callback* e o valor do resultado; mas, caso der erro, em vez de chamar o `then` execute este outro método `catchError` e me informe qual foi o erro no callback dele. E lembre-se, independente do resultado, não esqueça de sempre executar o callback deste meu método `whenComplete`!". Traduzido para o código, fica assim:

```

void main() {
    final future = Future<int>(() {
        return 84 ~/ 2;
    }).then((int valor) {
        print('Future em estado Completo, valor: $valor');
    }).catchError((Object erro, StackTrace stacktrace) {
        print('Future em estado Completo, erro: $erro');
    })
}

```

```
    print(stacktrace);
}).whenComplete(() {
    print('Future finalizado');
});
print('$future em estado Incompleto');
}
```

Ao criar um `Future`, é passado no construtor a função que deve ser executada de forma assíncrona, no exemplo, um simples cálculo de divisão `84 ~/ 2` que retorna o valor truncado para `int`. Esse construtor retorna uma instância de `Future` em estado incompleto. Nessa mesma instância ainda é possível adicionar callbacks.

Com a ajuda do método `then()` é adicionado um callback para ser executado assim que o future for finalizado, obtendo por parâmetro o resultado (`valor`) dessa operação (`84 ~/ 2`). Como o future pode resultar em um erro, também é possível adicionar um callback para essa situação com o `catchError`, que vai receber o `erro` e o `stacktrace` por parâmetro. Com base nisso, você consegue prever o que esse programa imprime no console?

Bem, como bem sabemos todo código assíncrono é agendado para que o *event loop* execute nos próximos ciclos. Por conta disso, todo o código síncrono do `main()` é executado primeiro, imprimindo a última linha `Instance of 'Future<Null>' em estado Incompleto`. Em seguida, é impresso `Future em estado Completo, valor: 42`, indicando que o *event loop* executou a função assíncrona, o cálculo funcionou corretamente e o callback registrado no `then()` foi chamado.

Experimente trocar o cálculo para `84 ~/ 0` e você verá que a execução do future vai finalizar com um erro, dessa vez ignorando o `then()` e executando o callback do `catchError()`.

No final, o `whenComplete()` registra um callback que sempre será executado ao finalizar o future, independente de ele ter sido completado com um valor ou com um erro. Como se fosse um `finally` em um bloco `try catch`, por exemplo, que imprime `Future finalizado`.

Future encadeado

Você pode ter notado que encadeamos as chamadas dos métodos `Future().then().catchError().whenComplete()` no último exemplo para tratar os diferentes estados do nosso future. E isso só foi possível porque se olharmos para as assinaturas desses métodos o retorno de cada um é exatamente um novo `Future` sucessor, que completa com o valor do seu `Future` antecessor, compreendeu? Dê uma olhada na assinatura do método `then`, por exemplo:

```
Future<R> then<R>(FutureOr<R> onValue(T value),  
                      {Function? onError});
```

Nela está explícita a informação de que o retorno é um novo `Future<R>` (um objeto diferente) que será completado com o valor retornado pelo *callback* `onValue()` caso o future original complete com um valor, ou o valor retornado pelo `onError()` caso o future original complete com um erro. Então não se confunda, pois o *future* sucessor e o seu antecessor são objetos **diferentes**.

Com isso, se você precisa executar alguma ação que dependa de mais de uma chamada assíncrona, é completamente possível encadear os futures. Por exemplo, um cenário em que precisamos buscar o nome de um usuário em uma base de dados de forma assíncrona, só que para fazer isso é necessário antes buscar o id dele em um outro local.

```
void main() {  
    buscarId().then((int id) {  
        print('Id encontrado, buscando nome..');  
        return buscarNome(id);  
    }).then(print);  
    print('Buscando..');  
}  
Future<int> buscarId() {  
    return Future.delayed(const Duration(seconds: 3), () => 1);  
}  
Future<String> buscarNome(int id) {  
    return Future.delayed(const Duration(seconds: 3),
```

```
        () => 'JHBitencourt');
}
```

Repare como a API de `Future` utiliza generics para definir qual o tipo do objeto que será retornado quando o future for completado, como ilustrado nas funções de busca. O construtor `Future.delayed()` nos dá a possibilidade de adicionar um `Timer` com um tempo de `delay` antes da execução de um future, e é perfeito para ocasiões onde precisamos simular uma chamada assíncrona que possa demorar alguns segundos para ter um retorno, como nesse exemplo simulando uma consulta de três segundos a uma base de dados.

Então o código imprime `Buscando...`, e após três segundos o *callback* do primeiro future `buscarId()` é chamado, imprimindo `Id encontrado,` buscando `nome...`. Note que dentro desse *callback* retornamos um outro future de `buscarNome(int id)`. Então novamente após três segundos o novo *callback* também é chamado com a função `print`, que imprime o nome do usuário `JHBitencourt`.

Quando construímos aplicações reais é comum haver diversas chamadas assíncronas consecutivas onde o resultado de uma depende da outra. Algo como `Future().then().then().then()` é completamente possível (apenas um alerta, embora possível, sintaticamente existe uma forma melhor de se fazer isto, que veremos logo mais neste capítulo) e existe uma certa dependência, pois o retorno de um desses métodos influencia no retorno dos próximos métodos. Afinal o que acontece se o primeiro `then()` lança uma exceção e completa com erro?

Erros em uma cadeia assíncrona

O que vai acontecer é que esse mesmo erro vai ser disparado por toda a cadeia de métodos até que ele seja capturado e tratado de alguma forma. E se não for, bem, aí o usuário vai descobrir o que acontece em tempo de execução. Olhe só este exemplo:

```
Future<String> um() => Future.value('Primeiro');
Future<String> dois() => Future.error('Erro no dois()');
```

```

Future<String> tres() => Future.value('Terceiro');

void main() {
    um().then((_) => dois())
        .then((_) => tres())
        .then((String valor) {
            print('O valor é $valor');
        });
}
> Unhandled exception: Erro no dois()

```

Quando usamos `_` na tipagem de retorno de uma função, como `(_) =>` do exemplo, estamos indicando que o valor não será utilizado e ele pode ser ignorado. Como o primeiro `then()` na cadeia de métodos retorna um future com erro (já que ele chama a função `dois()`), o segundo `then()` nem executa o seu callback `onValue()` e também já resulta em um future com esse mesmo erro, passando-o à frente na cadeia. Consequentemente, esse erro é passado entre todos os demais métodos encadeados na esperança de que em algum momento seja feito a sua captura e tratamento, o que não acontece e, por isso, o programa encerra com uma exceção não capturada.

Mas já que é necessário capturar e tratar é só usar um bloco `try/catch` então, afinal todo mundo sabe que eles foram criados para isso:

```

void main() {
    try {
        um().then((_) => dois())
            .then((_) => tres())
            .then((String valor) {
                print('O valor é $valor');
            });
    } catch(e) {
        print('Capturado: $e');
    }
}

```

Só que não... Na verdade não é bem assim. O código acima ainda vai continuar com o mesmo problema de exceção não capturada. Isso porque os blocos `try/catch` não capturam originalmente erros assíncronos. Para esse

caso, é necessário utilizar a própria API de futures com o método `catchError`, que já conhecemos.

```
void main() {
    um().then((_) => dois())
        .then((_) => tres())
        .catchError((dynamic e) {
            print('Capturado: $e');
            return '42';
        }).then((String valor) {
            print('O valor é $valor');
        });
}

> Capturado: Erro no dois()
> O valor é 42
```

Dessa vez o erro também é transmitido pela cadeia, só que no meio dela está o `catchError()`, que magicamente o captura, faz seu tratamento e retorna um novo future com valor (perfeitamente sem erro) que dá sequência à execução das chamadas.

Uma outra alternativa é utilizar o *callback* `onError()` do próprio método `then()`.

```
void main() {
    um().then((_) => dois())
        .then((_) => tres(), onError: (dynamic e) {
            print('Capturado onError: $e');
            return '42 :)';
        }).catchError((dynamic e) {
            print('Capturado: $e');
            return '42';
        }).then((String valor) {
            print('O valor é $valor');
        });
}

> Capturado onError: Erro no dois()
> O valor é 42 :)
```

Nesse caso, o future do `then()` é completado com o valor retornado pelo `onError()`, que capturou e tratou a exceção. Por conta disso, o `catchError()` é ignorado já que não existe mais erro quando chega sua vez de executar, e a cadeia de métodos continua normalmente.

Podemos dizer que a diferença entre as duas formas de tratar o erro é mais semântica. Enquanto o `onError()` geralmente é criado para tratar erros no escopo de execução do seu próprio método `then()`, o `catchError()` tem um significado mais genérico de tratamento de erros de toda a cadeia. Então fique à vontade e modifique a ordem dos *callbacks*, seus retornos, experimente diferentes situações e sinta-se mais familiar com o comportamento da API de futures. Afinal, ela é a base da programação assíncrona na linguagem.

Enquanto isso, saiba que, embora seja possível encadear todos esses *callbacks*, hoje essa já não é a maneira mais adequada e aceita para realizar essas chamadas, pois a leitura do código é um pouco complexa por conta de todos esses *callbacks* e funções anônimas, não acha? Veremos uma alternativa melhor com `async` e `await` mais à frente no capítulo. Antes disso, é interessante conhecermos algumas variações de construtores e métodos de `Future` que produzem diferentes comportamentos.

Future.sync

Um future síncrono faz sentido? A princípio, não, mas esse construtor permite ao menos que a função passada seja executada imediatamente, de forma síncrona. Porém, o future só vai ser completado com esse resultado de forma assíncrona na próxima iteração do *event loop*.

```
void main() {
    Future<String>.sync(() {
        print('Função Future.sync() executada');
        return 'Future síncrono!';
    }).then(print);
    print('Future.sync():');
}
```

```
> Função Future.sync() executada  
> Future.sync():  
> Future síncrono?
```

Repare na ordem dos valores impressos para entender a diferença da execução para um future normal. Durante a execução do `main()`, a função do future já foi executada, porém o future só completou com seu valor na próxima iteração do *event loop*.

Future.microtask

Com o `Future.microtask()`, por outro lado, é possível agendar a função para execução na *microtask queue* em vez da *event queue*, dando uma maior prioridade na execução. Internamente, é utilizada a mesma função `scheduleMicrotask()`, que já conhecemos.

```
void main() {  
    Future(() => 'Future normal').then(print);  
    Future.microtask(() => 'Future microtask').then(print);  
}  
  
> Future microtask  
> Future normal
```

Mesmo que o código que está agendando a função execute depois, por utilizar a queue de *microtask*, ele tem uma prioridade maior de execução e o resultado é impresso antes.

Future.value

O construtor `Future.value()` permite definir um future já com um valor predefinido.

```
void main() {  
    Future<int>(() => Future.value(42)).then(print);  
    print('Future.value():');  
}  
  
> Future.value():  
> 42
```

Porém, esse valor só será completado no próximo loop do *event loop*. É exatamente o equivalente a `Future.sync(() => 42)`.

Future.error

Se é possível criar um que já completa com um valor predefinido, também é possível criar um que complete com um erro.

```
void main() {
    Future.error('ERRO')
        .then(
            (valor) => print('Não executa, o Future completa com erro...'),
            onError: (e) => print('Erro capturado no onError $e'),
        ).catchError((e) => print('Future.error() => $e'));
}

> Erro capturado no onError ERRO
```

O future é completado com erro na próxima iteração do *evento loop*. Note que o erro é capturado pelo parâmetro `onError` do `then`. Remova o parâmetro `onError` e você verá que ele será capturado pelo `catchError`.

Future.wait

Imagine que você deseja executar alguma ação quando todas as informações de diferentes fontes assíncronas forem carregadas, como mostrar uma mensagem para o seu usuário. Para isso, existe a possibilidade de aguardar a execução de uma lista de futures através do método estático `Future.wait()`.

Suponha que temos uma lista de três futures onde cada um é concluído em uma quantidade de segundos distinta, 1, 2 e 3 segundos respectivamente:

```
void main() {
    Future.wait<String>(listaFuture())
        .then(print, onError: print);
    print('Aguardando...');
}

List<Future<String>> listaFuture() => <int>[2, 1, 3]
```

```
.map((s) => Future<String>.delayed(Duration(seconds: s),
                                         () => 'Tempo: $s'))
.toList();

Future<String> comErro() =>
    Future<String>.error('Erro ao buscar dados.');
```

O método `wait()` vai retornar:

- Um novo `Future` com uma lista dos valores resultados, assim que todos os futures da lista finalizarem.
- Um novo `Future` com erro caso um dos futures da lista resulte em erro.

Esse novo `Future` completará apenas quando todos os futures forem completados, ou seja, o tempo para que ele complete será igual ao tempo de execução do future mais demorado da lista. Dessa forma, executando o exemplo é impresso:

```
> Aguardando..
> [Tempo: 2, Tempo: 1, Tempo: 3]
```

Troque `Future.wait(listaFuture())` por `Future.wait([comErro(),
...listaFuture()])` para ver o erro na prática.

Esse mesmo método ainda possui dois parâmetros opcionais. Experimente utilizá-los para praticar a diferença de comportamento:

- `eagerError` : um `bool` que caso seja `true`, já vai completar o `future` com erro assim que o primeiro `future` da lista resulte em erro, mesmo que ainda tenham `futures` pendentes.
- `cleanUp` : uma função que executará para cada valor de `future` retornado com sucesso. É útil para limpar algum recurso utilizado.

Future.any

O método também estático `Future.any()` é similar ao `wait()`, pois também recebe uma lista de `futures`. A diferença é que ele já vai completar

com o valor do primeiro future que completar, ignorando o resultado dos demais.

```
void main() {
    Future.any<String>(listaFuture())
        .then(print, onError: print);
    print('Aguardando..');
}

List<Future<String>> listaFuture() => <int>[2, 1, 5]
    .map((s) => Future<String>.delayed(Duration(seconds: s),
        () => 'Tempo: $s'))
    .toList();

Future<String> comErro() =>
    Future<String>.error('Erro ao buscar dados.');

> Aguardando..
> Tempo: 1
```

No exemplo, é impresso `Tempo: 1` após um segundo de pausa, pois esse foi o future que finalizou primeiro. Mas é importante saber que isso não para a execução dos demais futures que serão eventualmente finalizados.

Troque `Future.any(listaFuture())` por `Future.any([comErro(), ...listaFuture()])` para ver o retorno como erro.

10.5 Async e await

Até agora, já aprendemos que é possível trabalhar de forma assíncrona em Dart agendando eventos para execução nas *queues* do *event loop*, e existem algumas formas de agendar esses eventos, sendo que a principal API que abstrai e facilita esse processo é a de `future`.

Dentre as inúmeras formas que vimos de criar um `Future`, com todos seus métodos disponíveis e as infinitas chamadas encadeadas, com certeza a que você mais utilizará no dia a dia é através do modificador `async`, que é

frequentemente utilizado em conjunto com o modificador `await`. Afinal, ambos permitem a criação de funções e métodos assíncronos que possuem a seguinte estrutura:

O diagrama mostra o código `Future<T> metodoAssincrono() async { return await 'Retorno assíncrono'; }` com anotações manuscritas. Um círculo verde ao redor da palavra `Future` tem a legenda "Retorno sempre será um Future". Um círculo laranja ao redor da palavra `async` tem a legenda "Modificador async obrigatório". Um círculo azul ao redor da palavra `await` tem a legenda "Pode ou não conter um modificador await".

Figura 10.12: Anatomia de uma função `async`.

O modificador `async` define que a função em determinado momento vai executar algum código de forma assíncrona, e é justamente por isso que o retorno de uma função `async` obrigatoriamente sempre será um `Future`.

```
void main() {
    final r = resposta().then(print);
    print('Resposta: $r');
}

resposta() async {
    return 42;
}
```

Por mais que a função assíncrona `resposta()` esteja retornando um inteiro, nós a deixamos sem a declaração de tipo de retorno propositalmente (por mais que o analisador possa acusar um erro). Assim fica comprovado que implicitamente o compilador vai envelopar este `42` em um `Future` e vai definir o retorno como sendo de fato do tipo `Future`. Olhe o que é impresso:

```
> Resposta: Instance of 'Future<dynamic>'  
> 42
```

É o equivalente a adicionar um `return Future.value(42);`, por exemplo, pois o future também só será completado com o valor no próximo *loop* do *event loop*.

Quanto ao `await`, sua utilização dentro de uma função é opcional, mas ele indica que a partir dele a execução do método passa a ser assíncrona. Dessa forma, o fluxo síncrono é encerrado e o *future* incompleto é retornado imediatamente para a sua origem. Ficou confuso? Bem, então para ilustrar vamos ver o fluxo completo de execução de uma função assíncrona na prática. Considere este trecho de código:

```
void main() {
    final future = resposta();
    future.then(print);
    print('Resposta: $future');
}

Future<int> resposta() async {
    print('Isso é síncrono');
    final r = await 42;
    print('Isso é após o await');
    return r;
}

> Isso é síncrono
> Resposta: Instance of 'Future<int>'
> Isso é após o await
> 42
```

Agora repare no fluxo ilustrado com o passo a passo:

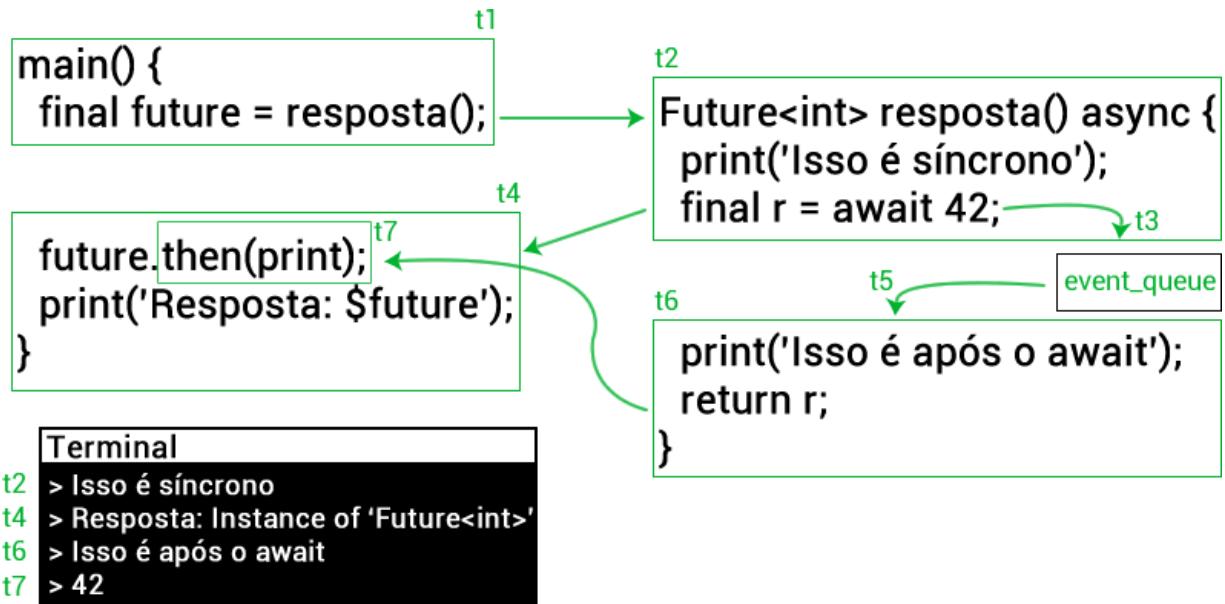


Figura 10.13: Fluxo de uma função async.

Vamos lá:

- t1: o `main()` inicia a execução e faz a chamada para a função `resposta()`.
- t2: a função `resposta()` inicia a execução normalmente e imprime `Isso é síncrono`. Porém, ao chegar na linha com o primeiro `await`, Dart entende que a partir dele o método deve continuar executando de forma assíncrona.
- t3: então `42` é entendido como `Future.value(42)` e enviado para a `event queue`. Nesse momento, o método já retorna um `Future` incompleto para quem o chamou.
- t4: a execução continua de forma síncrona no `main()`. Nele é registrado um *callback* para execução assim que o *future* for completado. Em seguida, é impresso `Resposta: Instance of 'Future<int>'`, demonstrando o *future* em estado incompleto.
- t5: com a finalização do `main()`, o *event loop* é iniciado, completando o evento que estava na `event queue` e retornando o resultado para a `resposta()`.
- t6: a função `resposta()` continua a sua execução imprimindo `Isso é após o await` e retornando o `42`, valor completado do `Future<int>`.

- t7: como esse future retornado pela `resposta()` possuía um *callback* registrado, ele é então executado e imprime o valor `42`.

Notou alguma semelhança com o que já estávamos fazendo só com a API de futures? A verdade é que `async` e `await` nasceram como uma sintaxe alternativa para a utilização de futures, praticamente um *syntax sugar*, uma vez que sua execução produz um resultado final igual ao de criar um future convencional. E é nesse momento que você me pergunta: "Por que criar uma nova sintaxe que faz a mesma coisa?".

Um jeito síncrono de trabalhar com assíncrono?

Bem, o fato é que esses modificadores surgiram na versão 1.9 da linguagem, onde a API de future padrão já era amplamente utilizada. E acabou se tornando um novo padrão recomendado na escrita de código assíncrono, pois a leitura tende a ficar mais clara e objetiva, já que se assemelha muito a um código síncrono e, como bem sabemos, nós seres humanos possuímos uma tendência a captar melhor informações síncronas.

Para efeitos de comparação, faça uma leitura rápida das duas funções `buscarNomeUsuario()` a seguir:

```
Future<int> buscarId() async => 42;
Future<String> buscarNome(int id) async => 'JHBitencourt';

// API Future pura
Future<String> buscarNomeUsuario() {
    return buscarId().then((int id) {
        return buscarNome(id);
    }).then((String nome) {
        return 'Usuário: $nome';
    });
}
// Async / await
Future<String> buscarNomeUsuario() async {
    final id = await buscarId();
    final nome = await buscarNome(id);
    return 'Usuário: $nome';
}
```

Ambas as funções possuem exatamente o mesmo comportamento e não precisa muita análise para notar o quanto mais rápido é a leitura e entendimento na versão com `async/await`. Nela, o fluxo de execução fica mais perceptível. Precisamos aguardar (`await`) o retorno da busca de `id` para utilizá-lo na busca do nome, e novamente aguardamos (`await`) o resultado com o `nome` para retorná-lo na função.

Na versão mais antiga, o código acaba ficando mais poluído, pois são utilizados muitos *callbacks* e funções anônimas na construção. Inclusive, qualquer código que utiliza a API de futures pura pode ser convertido para utilizar `async/await`, e essa é uma tarefa simples.

```
Future<String> buscarNomeUsuario() { 1
    return buscarId().then((int id) { 2
        return buscarNome(id);
    }).then((String nome) { 3
        return 'Usuário: $nome';
    });
}
```

Figura 10.14: Blocos de código com `then`.

Analizando a anatomia do código, conseguimos separar os trechos de código assíncrono, que produzem e dependem de um `Future`, delimitados pelo `then`. Ao converter essa função, cada *callback* vira uma execução com `await`:

```
Future<String> buscarNomeUsuario() async {  
    final id = await buscarId();  
    final nome = await buscarNome(id);  
    return 'Usuário: $nome';  
}
```

Figura 10.15: Blocos de código com `async` `await`.

Veja um outro exemplo, onde queremos chamar e imprimir o resultado de `buscarNomeUsuario()`:

```
void main() {  
    buscarNomeUsuario().then(print); // > Usuário: JHBitencourt  
}
```

Que pode ser transformado em:

```
Future<void> main() async {  
    print(await buscarNomeUsuario()); // > Usuário: JHBitencourt  
}
```

Muito fácil, concorda? Um ponto importante a ser lembrado é que, ao utilizar `await` em uma função, obrigatoriamente ela também deve ser definida como assíncrona com o modificador `async`.

Tratando erros com `async` `await`

Vamos lá, durante a leitura deste capítulo deixamos claro dois fatos a respeito de tratamento de erros assíncronos:

1. Ao trabalhar com a API de futures, conseguimos facilmente capturar erros assíncronos com o *callback* `catchError()` ou até mesmo com o parâmetro `onError` do também *callback* `then()`.
2. Erros assíncronos são diferentes dos erros síncronos tradicionais, e por isso não podem ser capturados com o famoso bloco `try/catch`.

Embora ambos sejam verdadeiros quando estamos trabalhando originalmente com a API de futures, o cenário muda um pouco com a utilização de `async/await`. Isso porque esses novos modificadores permitiram que os erros assíncronos também passassem a ser capturados pelo `try/catch`, reforçando o que discutimos sobre eles terem introduzido um jeito síncrono de trabalhar com o assíncrono, tornando o seguinte código possível:

```
void main() {
    buscarId();
    print('Capturando um erro assíncrono...');

}

Future<int> buscarId() async {
    try {
        return await Future<int>(() {
            return 42 ~/ 0;
        });
    } on UnsupportedError catch (e) {
        print('Erro capturado: $e');
    } catch (e) {
        print('Demais erros caem aqui');
    }
    return 0;
}

> Capturando um erro assíncrono...
> Erro capturado: IntegerDivisionByZeroException
```

Nesse caso, o future vai completar com um erro `IntegerDivisionByZeroException`, que é capturado com sucesso na primeira cláusula de `catch`. Para efeito de comparação, o mesmo código do método `buscarId()` a seguir utiliza apenas a API de *futures*:

```
Future<int> buscarId() async {
    return Future<int>(() {
        return 42 ~/ 0;
    }). catchError((dynamic e) {
        print('Erro capturado: $e');
        return 0;
    }, test: (dynamic e) => e is UnsupportedError)
```

```

    .catchError(
        (dynamic e) {
            print('Demais erros caem aqui');
            return 0;
        },
    );
}

```

10.6 Completer

Na grande maioria dos casos, trabalharemos com future utilizando algum dos seus próprios construtores predefinidos vistos ao longo do capítulo. Eles por si só já suprem quase todas as necessidades de criação e manipulação de futures na linguagem, mas algumas raras situações podem demandar a criação de um future de forma diferente. Para esses casos, existe o `Completer`. Um *completer* é uma forma mais poderosa de criar um *future* e eventualmente completá-lo de forma manual no futuro. Por conta disso, ele está fortemente acoplado a um `Future`, por exemplo:

```

import 'dart:async';
void main() {
    print('Buscando a resposta...');
    buscarResposta().then((resposta) {
        print('A resposta é $resposta');
    });
    print('para a vida, o universo e tudo mais...');
}

Future<String> buscarResposta() async {
    final completer = Completer<String>();
    Timer(Duration(seconds: 2), () {
        completer.complete('42');
    });
    return completer.future;
}

```

```

Buscando a resposta...
para a vida, o universo e tudo mais...

```

A resposta é 42

Ao criar um `Completer` implicitamente, também estamos criando um `Future` que pode ser acessado a qualquer momento através da própria propriedade `Completer.future`. E é exatamente este que é retornado, assim todo o comportamento assíncrono de quem chama esta função é mantido.

A mágica do `Completer` está na hora de completar esse `future`, que pode ser feito de forma manual a qualquer momento através de `Completer.complete()`. No exemplo, completamos o `future` após os dois segundos de um *timer*, mas esta decisão dependerá do seu problema, podendo inclusive completar com um erro utilizando `Completer.completeError(Object error, [StackTrace stackTrace])`.

Mas obviamente esse exemplo foi apenas para você entender o funcionamento de um `Completer`, uma vez que ele não era necessário nesse caso, e nem recomendado, já que um `Future.delayed()` cumpriria a mesma tarefa de aguardar dois segundos para retornar um valor. Um bom exemplo de utilização de um `Completer` seria para converter algum determinado código de uma API de *callbacks* para a API de *futures*. Por exemplo, imagine uma biblioteca de acesso a um banco de dados que funcione através de *callbacks*:

```
import 'dart:async';
class Database {
    void salvarUsuario(String usuario,
                        void Function(String) callback) {
        Timer(Duration(seconds: 2), () {
            callback('$usuario salvo');
        });
    }
}
void main() {
    Database().salvarUsuario('JHBitencourt', callbackUsuario);
}
void callbackUsuario(String resultado) {
    print('Callback executado: $resultado');
}
```

```
Callback executado: JHBitencourt salvo
```

A classe `Database` tem um método qualquer que recebe um usuário, faz qualquer coisa com ele lá dentro e, após concluir, chama a função de *callback* informada para retornar o resultado da operação. Só que talvez você não queira trabalhar com *callbacks* e deseje utilizar a própria API de futures presente em Dart. Para isso, bastaria criar alguma classe semelhante a um *proxy*:

```
class FutureDatabase {  
    Future<String> salvarUsuario(String usuario) {  
        final completer = Completer<String>();  
        Database().salvarUsuario(usuario, (String resultado) {  
            completer.complete(resultado);  
        });  
        return completer.future;  
    }  
}  
  
Future<void> main() async {  
    final resultado = await FutureDatabase()  
        .salvarUsuario('JHBitencourt');  
    print('Resultado: $resultado');  
}
```

```
Resultado: JHBitencourt salvo
```

A `FutureDatabase` serve como um intermediário que transforma a API de *callbacks* do banco em uma API de futures com a ajuda dos *completers*, assim é possível chamá-la diretamente sem se preocupar em utilizar *callbacks*. Como o retorno é um future, ele agora pode ser utilizado como tal, inclusive em conjunto com o `await`.

Mas para entender ainda mais a flexibilidade do *completer*, vamos ver um terceiro exemplo:

```
import 'dart:async';  
import 'dart:math';  
class CompletoComNumero {  
    late Completer<String> _completer;  
    int numero;
```

```

CompletoComNumero._internal(this.numero) {
    _completer = Completer();
    sorteio().listen(_validarNumero);
}

Stream<int> sorteio() async* {
    for (int i = 0; i < 5; i++) {
        if (_completer.isCompleted) break;

        await Future<void>.delayed(Duration(seconds: 1));
        final sorteado = Random().nextInt(5) + 1;
        print('Sorteio: $sorteado');

        yield sorteado;
    }
    if (!_completer.isCompleted) {
        _completer.completeError('O número não foi sorteado!');
    }
}

void _validarNumero(int numeroSorteado) {
    if (numeroSorteado == numero) {
        _completer.complete('O número $numero foi sorteado!');
    }
}

static Future<String> novo(int numero) async {
    final c = CompletoComNumero._internal(numero);
    return c._completer.future;
}
}

```

A `CompletoComNumero` é uma classe cujo único objetivo é verificar se um determinado número foi sorteado. O método `sorteio()` é um gerador assíncrono que gerará cinco números aleatórios (no máximo) que vão de 1 a 5. O resultado desse sorteio é informado através de um future que é controlado pelo completer interno, então, se dentre os cinco sorteios o número escolhido for sorteado, o completer completa o future com sucesso.

Caso o sorteio acabe e o mesmo número escolhido não saia, o completer completa o future com um erro.

Esse exemplo faz uso de recursos que ainda não vimos até então, principalmente no método `sorteio()`, que utiliza uma `stream` e função geradora (`async*`), que não são o foco deste momento. Então não se preocupe se essa parte do código parecer um pouco confusa, pois são assuntos que trataremos nos capítulos mais à frente. Por ora, repare apenas na liberdade que o completer oferece para manusear o future entre diversos métodos distintos, além da sua propriedade `_completer.isCompleted` para identificar se o sorteio pode ser encerrado, pois o número já foi sorteado, ou ele não foi sorteado e pode ser completado com erro. Para executar, basta instanciar a classe:

```
void main() {
    CompletoComNumero.novo(3).then(print).catchError(print);
}

Sorteio: 4
Sorteio: 2
Sorteio: 5
Sorteio: 3
0 número 3 foi sorteado!
```

O resultado impresso vai depender do número escolhido e dos números sorteados, mas se você está com azar, receberá o erro capturado por `catchError()` informando que o número não foi escolhido. No caso do exemplo, na quarta vez em que um número foi gerado, o 3 foi sorteado.

10.7 Se liga aí

- Utilize `async` e `await` em vez de `Future` sempre que possível. O código se torna mais conciso, fácil de entender e de dar manutenção.
- Por outro lado, não declare uma função como `async` sem haver a real necessidade.

```

// ruim
Future<Arquivo> download() async {
    return Future.any([downloadApi(), downloadBD()]);
}

// bom
Future<Arquivo> download() {
    return Future.any([downloadApi(), downloadBD()]);
}

```

- Utilize `Future<void>` como retorno quando a função assíncrona não deve produzir um resultado, ou esse resultado não é importante e não deve ser utilizado. É o equivalente ao utilizar um simples `void` em uma função síncrona.
- Não utilize um `Completer` sem necessidade. Após entender o seu funcionamento, você pode até pensar em criar algo do tipo:

```

Future<int> buscarResposta() {
    Future<int> future = calcular();
    Completer<int> completer = Completer();
    future.then((int resposta) {
        salvar(resposta);
        completer.complete(resposta);
    });
    return completer.future;
}

```

Mas isso é um *anti-pattern*, uma vez que utilizar um *completer* nessa situação (e na maioria das situações) é completamente desnecessário. Utilizar encadeamento de futures seria mais adequado:

```

Future<int> buscarResposta() {
    return calcular().then((int resposta) {
        salvar(resposta);
        return resposta;
    });
}

```

Ou, melhor ainda, com a sintaxe do `async/await`:

```
Future<int> buscarResposta() async {
  int resposta = await calcular();
  salvar(resposta);
  return resposta;
}
```

10.8 É com você

1 - Vimos neste capítulo algumas formas de se programar de forma assíncrona e adicionar eventos para execução na *event queue*: `Timer`, `Future` e `async await`. Porém, esses não são os únicos. Pesquise outras APIs que permitem fazer o mesmo.

2 - Pesquise sobre os construtores de `Future` não abordados neste capítulo: `Future.dowhile()` e `Future.forEach()`.

3 - O `FutureOr<T>` é um tipo em Dart que pode conter tanto um `Future<T>` quanto um `T`. Pesquise a respeito.

4 - Uma vez que a *microtask queue* possui uma prioridade maior de execução para a *event queue*, o que aconteceria se ela recebesse um número infinito de elementos para serem executados? Tente implementar um código que a chame infinitamente para ver o resultado.

5 - Atualize todos os exemplos que vimos neste capítulo com `Future().then()` para utilizar a sintaxe com `async` e `await`.

Até aqui

Entendemos o quão complexo pode ser ter que se preocupar com os problemas gerados por códigos executando em diferentes *threads*, justamente o que levou Dart a introduzir o conceito de *isolate* para amenizá-los. Também conhecemos a fundo como nossos códigos são de fato executados e a importância do *event loop* por trás disso tudo.

Com isso, passamos pela introdução ao mundo assíncrono que serve de base para muito código escrito em Dart, com um foco na API de futures, e pelos benefícios de se usar a *syntax sugar* `async await`. Nos capítulos a seguir, continuaremos explorando esses recursos, conhecendo as *streams*, aprofundando-nos nas *isolates* e até conhecendo as *zones*. Então continue, que com certeza você garantirá um bom "*future*" na linguagem!

CAPÍTULO 11

Na prática - Dart CLI

Como desenvolvedores, eventualmente precisamos utilizar o terminal do nosso sistema operacional para executar alguma tarefa, como instalar algum programa ou habilitar algum recurso.

Toda essa interação é feita através de aplicações conhecidas como CLI (*Command Line Interface*), que executam a partir de alguma interface de comandos. Até mesmo para rodar nossos exemplos com `dart run main.dart` estamos interagindo com uma interface de comandos do próprio SDK do Dart.

Neste projeto, vamos criar uma aplicação que seja capaz de nos informar o clima atual de uma determinada cidade através da linha de comandos, e melhor, utilizando apenas os recursos de Dart. Então durante o capítulo iremos:

- Criar requisições REST com o package `http` ;
- Entender o que é um `Converter` e utilizar a lib `dart:convert` ;
- Criar os parâmetros da aplicação com o package `args` ;
- Executar e compilar em diferentes formatos com `dart compile` .

11.1 API da Climatempo

A nossa aplicação CLI será capaz de consultar os dados reais de clima através de uma API RESTful disponibilizada pela Climatempo, uma empresa brasileira que fornece serviços de meteorologia em tempo real. O acesso a essas informações está disponível em <https://advisor.climatempo.com.br>.

Uma API (*Application Programming Interface*) funciona como uma interface para comunicação entre aplicações, enquanto a sigla RESTful representa uma API que implementa o REST (*Representational State Transfer*), um modelo que especifica uma série de princípios e regras para a padronização na transferência de dados na web, trabalhando muito bem integrado com o protocolo HTTP e os seus métodos (GET, POST etc.).

Para começar, acesse o site da API e efetue o cadastro da forma que desejar (e-mail, gmail ou GitHub). Após isso, você será redirecionado para um dashboard com algumas informações de status, velocidade, consumo diário, entre outros. Mas apenas duas coisas interessam para o desenvolvimento do projeto, a documentação e o *token*.

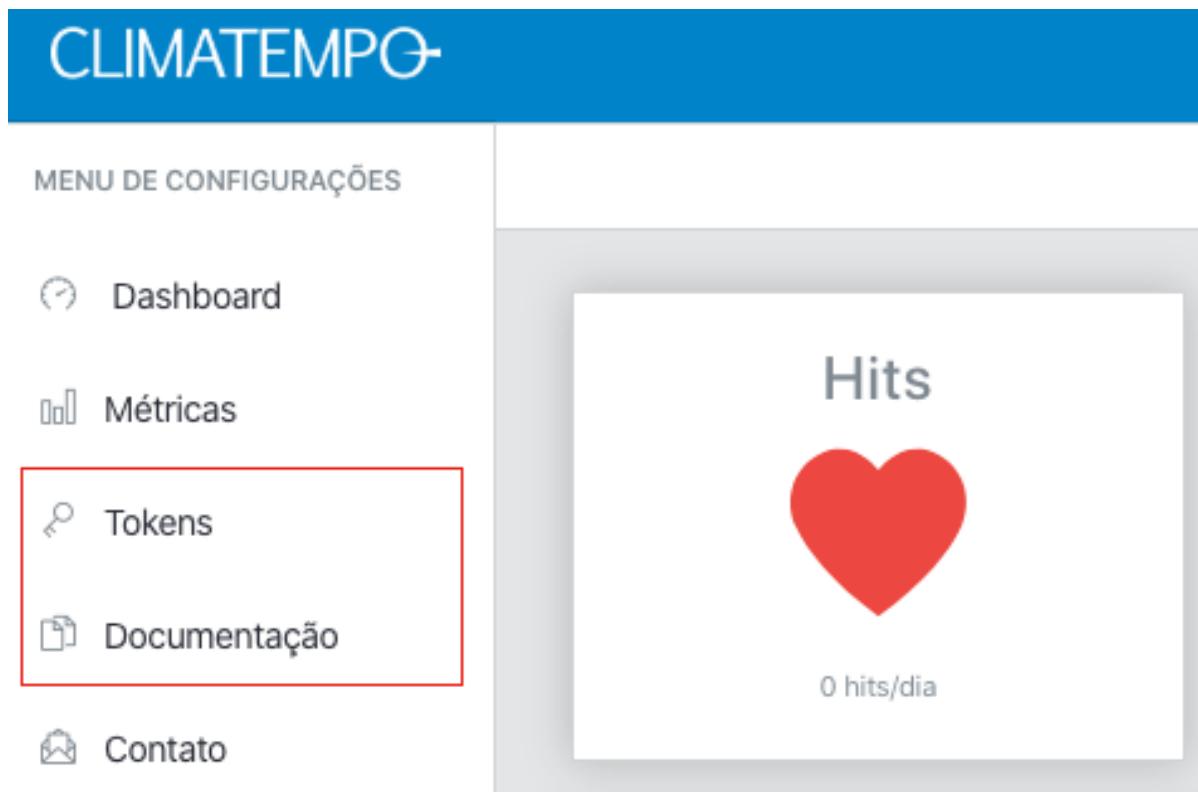


Figura 11.1: Dashboard API do Climatempo.

A documentação é sempre a nossa melhor amiga. Com ela, conseguimos ver quais são os métodos existentes para consulta, quais informações eles retornam e como devem ser utilizados. Existem vários, mas dentre as opções usaremos:

- **PUT /api-manager/user-token/{token}/locales**: registra um token para permitir acesso aos dados de uma cidade.
- **GET /api/v1/locale/city?name={nome}&state={estado}&token={token}**: permite consultar as cidades brasileiras disponíveis, os parâmetros name (nome) e state (estado) são opcionais.
- **GET /api/v1/weather/locale/{idCidade}/current?token={token}**: traz os dados atuais do clima de uma determinada cidade.

Como não estamos pagando para ter acesso a esses dados, existe um limite de 300 requisições diárias para o uso gratuito. E esse controle é feito através de um token necessário para acessar os dados. Para gerar um, acesse o menu de *token* e cadastre um novo projeto. Isso dará um código parecido com o da imagem a seguir:

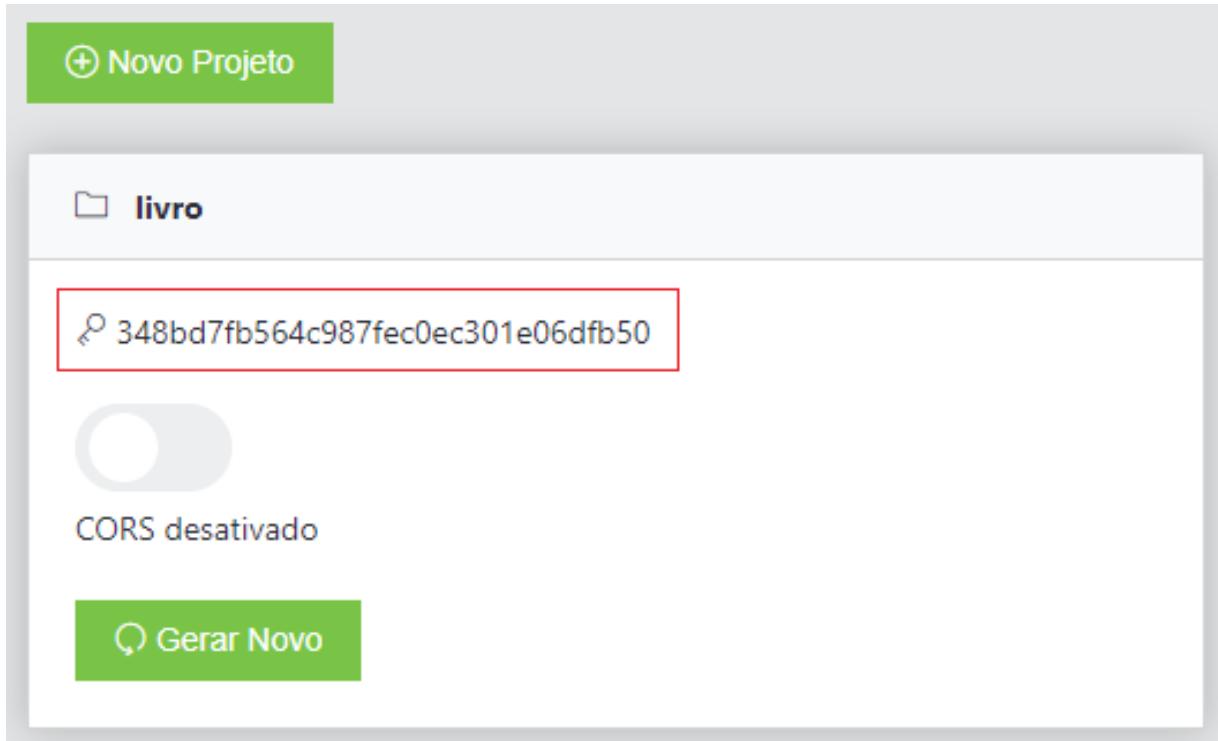


Figura 11.2: Token gerado para acesso aos dados.

Guarde esse valor pois ele será utilizado em seguida.

Criando o projeto

Já utilizamos o comando `dart create` para criar um projeto com os templates `web-simple` e `package-simple`. Desta vez, como o objetivo é uma aplicação via linha de comandos, usaremos o `console-full`.

```
dart create -t console-full climatempo && cd climatempo
```

Assim o esqueleto do projeto é criado dentro da pasta **climatempo**, e o próprio SDK já atualizou as dependências rodando `pub get`. Só com isso já é possível rodar o app usando `dart run`, que deverá imprimir `Hello world: 42!` no console, indicando que tudo está funcionando corretamente.

A estrutura de arquivos gerada pelo template é simples e semelhante ao que já vimos anteriormente, a diferença está em:

- **bin/climatempo.dart**: o diretório `bin` é particular das aplicações CLI, ele sempre contém o ponto de partida da aplicação, o arquivo que possui o método `main()`.
- **lib/climatempo.dart**: assim como na versão web, o diretório `lib` contém a implementação da aplicação. O `climatempo.dart` é um arquivo de exemplo gerado pelo template.
- **test/climatempo_test.dart**: já o diretório `test` contém todos os testes unitários da aplicação.

Portanto, como não nos interessam os testes unitários no momento, o diretório `test` pode ser removido, assim como o arquivo `lib/climatempo.dart` gerado automaticamente que não será utilizado. Já o `bin/climatempo.dart` pode ser alterado para apenas:

```
void main(List<String> args) {  
}
```

É típico de aplicações via linha de comando receberem argumentos para executar alguma operação. Quando executamos `dart run`, por exemplo, o `run` é um argumento que indica para a VM executar o projeto do diretório atual. Por padrão, esses argumentos podem ser recebidos na função `main` através de uma lista de strings.

```
dart run bin/climatempo.dart teste1 2 3
```

Esse comando resultaria em três parâmetros string, `['teste1', '2', '3']`, passados para a função `main` do arquivo `climatempo.dart`. Para nossa aplicação, vamos definir três comandos principais para serem utilizados como argumentos:

1. **[-h --help]**: vai mostrar uma descrição dos comandos existentes na aplicação.
2. **cidade [-n --nome] NomeCidade [-e --estado] SC**: vai buscar as cidades disponíveis para consultar o clima. Os parâmetros `nome` e `estado` são opcionais.
3. **agora [-i --id] 4765**: vai mostrar o clima atual de uma determinada cidade, que é identificada pelo parâmetro `id`.

11.2 Requisições com o package http

Dart possui uma implementação genérica construída acima da API de futures, que permite a realização de requisições HTTP e é acessível através do package `http`. Como ele não está disponível por padrão no SDK, é necessário adicionar sua dependência no já conhecido `pubspec.yaml`.

```
name: climatempo
description: Aplicação CLI para consulta de clima.
version: 1.0.0

environment:
  sdk: '>=2.16.0 <3.0.0'

dependencies:
  http: ^0.13.4

dev_dependencies:
  lints: ^1.0.0
  test: ^1.16.0
```

Coloque a última versão disponível para o package `http` presente em <http://pub.dev>, que atualmente pode ser diferente da listada aqui, afinal é

comum os packages serem atualizados com o tempo. Ao modificar informações desse arquivo, sempre rode `dart pub get` para resolver as dependências. Assim será possível utilizar o `http`:

```
import 'package:http/http.dart' as http;
void main(List<String> args) async {
  http.Response response = await http.get(
    Uri.parse('https://google.com'));
  print(response.body);
}
```

Esse package possui várias funções declaradas e, para não as chamar diretamente, já se tornou uma convenção definir um apelido, como o `as http`. Assim, todas as funções e classes presentes nele podem ser referenciadas a partir de `http..`.

Note como ele utiliza programação assíncrona. Ao fazer uma chamada `GET`, o retorno será um `Future<Response>`, mas como estamos aguardando a resposta com `await`, já pegamos diretamente a referência para um objeto do tipo `Response`.

Esse objeto representa a resposta da requisição. Dentre as várias propriedades que ele possui, como os *headers* da requisição ou o *status code* de retorno, existe o *body* com o corpo retornado. Nesse caso, como o *request* de teste foi feito para a URL do Google, o corpo é o próprio HTML da página, exatamente o que um navegador recebe para mostrar ao usuário quando acessamos uma URL.

Mas as requisições do projeto serão para uma API onde o retorno não é um HTML e sim um JSON.

11.3 Package convert e JSON

Uma tarefa comum ao desenvolver aplicações é a necessidade de converter um tipo de dado em outro. A lib `dart:convert` foi criada justamente para auxiliar nesse processo e contém a implementação de conversão de tipos

que são comuns, como JSON, UTF-8, Base64 e ASCII. Um `Converter<S, T>` é o tipo que representa a abstração básica:

```
abstract class Converter<S, T>
    extends StreamTransformerBase<S, T> {
  const Converter();
  T convert(S input);
  // código omitido
}
```

Ele é responsável por converter um tipo `S` em um tipo `T` através do método `convert()`. Quando uma conversão possui dois sentidos (pode ser revertida), são agrupados dois *converters* em um `Codec<S, T>`.

```
abstract class Codec<S, T> {
  const Codec();
  T encode(S input) => encoder.convert(input);
  S decode(T encoded) => decoder.convert(encoded);
  Converter<S, T> get encoder;
  Converter<T, S> get decoder;
  // código omitido
}
```

Nesse caso, um `Converter<S, T>` é responsável por fazer o `encode()`, processo de transformação de um objeto `S` em `T`, enquanto o outro `Converter<T, S>` é responsável pelo `decode()`, processo inverso que transforma um objeto `T` em `S`.

Se analisarmos um JSON, vemos que ele possui uma estrutura muito semelhante a um `Map`. Sempre haverá uma única chave e N valores:

```

{
  "livro": {
    "titulo": "Dart",
    "autor": "Julio", < "livro", < "titulo", "Dart" >>
    "capitulos": [
      "1 - Hello Dart", < "autor", "Julio" >
      "2 - O básico" < "capitulos", ["1...", "2..."] >
    ]
  }
}

```

Figura 11.3: Estrutura JSON e Map.

Como o JSON é apenas uma `String`, para converter esses dados bastaria um `Codec<Map<String, dynamic>, String>` com:

- `Converter<Map<String, dynamic>, String>` : responsável por fazer o encode transformando um `Map` em JSON.
- `Converter<String, Map<String, dynamic>>` : responsável por fazer o decode transformando um JSON em `Map`.

E felizmente já existe isso pronto.

```

import 'dart:convert';
void main() {
  final dados = '{"data": [{"id":1, "name":"Acre"}, {"id":2, "name":"Alagoas"}]}';
  Map<String, dynamic> map = json.decode(dados);
  print(map);
  // > {data: [{id: 1, name: Acre}, {id: 2, name: Alagoas}]}
}

```

A variável `json` vem da lib `dart:convert` e é a implementação de um `JsonCodec` que através do método `decode()` transforma a `String` com o JSON em um `Map`. Simples assim.

11.4 Consumindo a API do Climatempo

Já sabemos como fazer o `decode` de um JSON para um `Map`, mas embora seja possível trabalhar com um `Map` ele não representa exatamente o real significado dos dados que ele possui. Por isso, é comum criarmos objetos simples em Dart que vão cumprir esse papel. Alguns costumam nomear esses objetos simplesmente de `model`, então crie o diretório `lib/model` e, dentro dele, os arquivos `cidade.dart`, `tempo.dart` e `clima_tempo.dart`.

Eles vão conter exatamente a representação dos dados que precisamos e são retornados no JSON. Por exemplo, ao consultar o endpoint de busca de cidades, o retorno será uma lista com os resultados:

```
[  
  {  
    "id": 3477,  
    "name": "Florianópolis",  
    "state": "SC",  
    "country": "BR"  
  }  
]
```

E a representação em Dart pode ser um objeto `Cidade` dentro de `cidade.dart`.

```
class Cidade {  
  Cidade.fromJson(Map<String, dynamic> jsonMap)  
    : id = jsonMap['id'],  
      nome = jsonMap['name'],  
      estado = jsonMap['state'],  
      pais = jsonMap['country'];  
  
  final int id;  
  final String nome;  
  final String estado;  
  final String pais;  
  
  @override  
  String toString() => 'Id: $id - Nome: $nome - Estado: $estado,
```

```
País: $pais';  
}
```

Como todas as propriedades do nosso objeto são *non-nullable*, elas precisam obrigatoriamente ser inicializadas na declaração ou no construtor da classe. E é o construtor nomeado `Cidade.fromJson` que vai fazer a mágica. Ele recebe o `Map` gerado pelo `decode` por parâmetro e preenche os dados do objeto, mapeando cada chave do `Map` para uma propriedade de `Cidade` através dos inicializadores.

Como vamos apenas consultar dados, o objeto `Cidade` pode ser imutável e ter suas propriedades como `final`. Já o `toString` vai auxiliar ao mostrar os dados para o usuário. O outro endpoint que usaremos vai retornar os dados do clima atual da cidade:

```
{  
    "id": 3477,  
    "name": "Florianópolis",  
    "state": "SC",  
    "country": "BR"  
    "data": {  
        "temperature": 23.8,  
        "wind_direction": "NW",  
        "wind_velocity": 22,  
        "humidity": 43,  
        "condition": "Poucas nuvens",  
        "pressure": 1008,  
        "icon": "2",  
        "sensation": 27,  
        "date": "2017-10-01 12:37:00"  
    }  
}
```

Esse JSON de retorno possui informações de objetos distintos, então quebramos em partes. Dentro de `"data"` estão os dados do clima, que pode ser um objeto `Tempo` dentro de `tempo.dart`.

```
class Tempo {  
    Tempo.fromJson(Map<String, dynamic> jsonMap)  
        : temperatura = jsonMap['temperature'],
```

```

    velocidadeVento = jsonMap['wind_velocity'],
    humidade = jsonMap['humidity'],
    sensacao = jsonMap['sensation'],
    data = DateTime.parse(jsonMap['date']);

    final num temperatura;
    final num velocidadeVento;
    final num humidade;
    final num sensacao;
    final DateTime data;
}

```

Nem todas as informações retornadas pela API são necessárias, então pegamos apenas as principais. O campo `date` retornado é uma `String` que possui um formato de data e, por isso, é utilizado `DateTime.parse` para transformá-lo em um `DateTime`. Agora o `Tempo` pode ser utilizado em um outro objeto `ClimaTempo` dentro de `clima_tempo.dart`.

```

import 'cidade.dart';
import 'tempo.dart';
class ClimaTempo {
    ClimaTempo.fromJson(Map<String, dynamic> jsonMap)
        : cidade = Cidade.fromJson(jsonMap),
          tempo = Tempo.fromJson(jsonMap['data']);

    final Cidade cidade;
    final Tempo tempo;

    String toString() => '''
    Cidade: ${cidade.nome}, ${cidade.estado} - ${cidade.pais}
    ${tempo.data}
    Temperatura: ${tempo.temperatura} - Sensação: ${tempo.sensacao}
    Umidade: ${tempo.humidade} - Velocidade do Vento:
    ${tempo.velocidadeVento}
    ''';
}

```

As informações de `id`, `name`, `state` e `country` já representam uma `Cidade`, então reutilizamos esse mesmo tipo para manter a coerência e reaproveitamento do código. Repare como em `ClimaTempo.fromJson`

simplesmente chamamos os construtores `fromJson` de `Cidade` e `Tempo` para popular as propriedades.

Agora já é possível de fato consumir a API. Dentro de `lib` crie o `api.dart`. Ele vai ser o responsável por fazer os requests para a API do ClimaTempo e retornar os dados em forma dos objetos que mapeamos.

```
import 'dart:convert';
import 'package:http/http.dart' as http;
import 'model/cidade.dart';
import 'model/clima_tempo.dart';

const apiUrl = 'https://apiadvisor.climatempo.com.br';
const token = '348bd7fb564c987fec0ec301e06dfb50';
```

O valor da constante `token` deve ser trocado pelo token que você gerou anteriormente. Para consultar as cidades, usaremos:

- `https://apiadvisor.climatempo.com.br/api/v1/locale/city?name=Florianópolis&state=SC&token=TOKEN`

O nome e estado são opcionais, o que torna possível trazer todas as cidades brasileiras ou todas as cidades de algum estado em específico, algo facilmente replicado em Dart com os parâmetros opcionais.

```
// código omitido
Future<List<Cidade>> buscarCidades({String? nome, String? estado})
async {
    var url = '$apiBaseUrl/api/v1/locale/city?';
    if (nome != null) url += 'name=$nome';
    if (estado != null) url += '&state=$estado';

    final response =
        await http.get(Uri.parse('$url&token=$token'));
    if (response.statusCode != 200) throw response.body;

    final responseJson = json.decode(response.body);
    final cidades = <Cidade>[];
    responseJson.forEach((map)
        => cidades.add(Cidade.fromJson(map)));
}
```

```
    return cidades;
}
```

O método `buscarCidades` é assíncrono e, ao completar o seu `future`, ele retornará uma lista de cidades. A URL montada é passada para `http.get` e, neste momento, devido ao `await`, o processamento vai para a *event queue* e o método retorna um `future` incompleto. Ao completar a requisição, validamos qual foi o status de retorno. Um `statusCode 200` indica que a requisição ocorreu com sucesso, então se for diferente lançamos uma exceção com o conteúdo do `body` retornado.

Caso a exceção não seja lançada, o corpo da `response` vai possuir o JSON com uma lista de cidades que é decodificado com a ajuda do `json.decode()`. O resultado disso vai para a variável `responseJson`, que nesse momento contém uma lista de `Map`. Iterando sobre ela, cada `Map` é traduzido para uma `cidade` e adicionado à lista de `cidades`, que será o resultado retornado.

O próximo passo é consultar o clima atual de uma cidade, mas existe um problema. Na versão gratuita da API, cada token pode acessar informações de apenas uma cidade por vez. Por isso, é necessário sempre que consultar outra cidade registrá-la no token através do endpoint:

- `https://apiadvisor.climatempo.com.br/api-manager/user-token/TOKEN/locales`

```
Future<void> registrarCidade({required int idCidade}) async {
    final url =
        '$apiBaseUrl/api-manager/user-token/$token/locales';
    final map = {'localeId[]': '$idCidade'};
    await http.put(Uri.parse(url), body: map);
}
```

O `registrarCidade` é um pouco diferente. O retorno dele não é interessante para nós, por isso o `Future<void>`. Em vez de uma requisição `GET`, essa é uma requisição `PUT` para atualização de registro e, por convenção da API, é necessário passar na requisição um parâmetro `localeId[]` com o valor sendo o id da cidade a ser registrada. Então,

utilizando o `http.put()`, conseguimos passar a URL com os parâmetros no `body`.

Agora sim, o último endpoint é a consulta de clima.

- `https://apiadvisor.climatempo.com.br/api/v1/weather/locale/ID/current?token=TOKEN`

```
Future<ClimaTempo> climaAtual({required int idCidade}) async {
    final url = '$apiBaseUrl/api/v1/weather/locale/$idCidade/current?token=$token';
    final response = await http.get(Uri.parse(url));
    if (response.statusCode != 200) throw response.body;

    final responseJson = json.decode(response.body);
    return ClimaTempo.fromJson(responseJson);
}
```

O comportamento desse método é muito parecido com o que já foi visto, então não há mistérios. Retornará um objeto `ClimaTempo` preenchido com as informações do clima atual da cidade.

11.5 Tratando os argumentos de CLI

As ações do usuário no nosso aplicativo serão todas via comandos que são passados como argumentos para o `main()`. Existem padrões amplamente utilizados que ditam como deve ser a sintaxe e comportamento desses comandos, como os padrões do GNU e POSIX.

Dart possui um package `args` que auxilia na conversão e tradução dos parâmetros utilizados, seguindo esses padrões. Para usá-lo, adicione a dependência no `pubspec.yaml`.

```
# código omitido
dependencies:
  http: ^0.13.4
  args: ^2.3.0
```

A primeira coisa a se fazer quando utilizamos o `args` é criar um *parser*.

```
import 'package:args/args.dart';
void main(List<String> args) {
  final parser = ArgParser()
    ..addOption('nome', abbr: 'n', defaultsTo: 'Julio');
}
```

O `ArgParser` contém todas as regras e comandos que uma aplicação vai aceitar. No exemplo, o `addOption` determina uma opção denominada `nome`, que é usada como `--nome`. O `abbr` define a forma abreviada `n`, que é usada como `-n`. Já o `defaultsTo` permite setar um valor padrão caso o parâmetro não seja passado.

Com o *parser* criado, é possível então traduzir os argumentos passados.

```
void main(List<String> args) {
  final parser = ArgParser()
    ..addOption('nome', abbr: 'n', defaultsTo: 'Julio');
  ArgResults argsResult = parser.parse(args);
  print('Hello ${argsResult['nome']}');
}
```

Ao chamar o `parser.parse()` passando os `args`, o retorno será um objeto `ArgResults`. Esse objeto contém todos os argumentos passados e validados de acordo com as regras em um formato de Map com chave/valor. Para acessar o valor, basta buscar pelo nome da opção `argsResult['nome']`.

A seguir, os formatos de parâmetros permitidos por esse programa e o resultado impresso.

```
dart run bin/climatempo.dart -> Hello Julio
dart run bin/climatempo.dart --nome JHB -> Hello JHB
dart run bin/climatempo.dart --nome=JHB -> Hello JHB
dart run bin/climatempo.dart -n JHB -> Hello JHB
dart run bin/climatempo.dart -nJHB -> Hello JHB
dart run bin/climatempo.dart --nome "Julio Bitencourt" -> Hello
Julio Bitencourt
```

Voltando para o projeto, já conseguimos definir o *parser* da aplicação no `climatempo.dart`.

```
import 'package:args/args.dart';
ArgParser criarParser() {
    return ArgParser()
        ..addCommand(
            'cidade',
            ArgParser()
                ..addOption('nome',
                    abbr: 'n', valueHelp: 'Nome da cidade para consulta')
                ..addOption('estado',
                    abbr: 'e', valueHelp: 'Sigla do estado para consulta'))
        ..addCommand(
            'agora',
            ArgParser()
                ..addOption('id',
                    abbr: 'i', valueHelp: 'Id da cidade para consulta do tempo'))
        ..addFlag('help',
            abbr: 'h', help: 'Como utilizar o programa', negatable: false);
}
```

Em vez de adicionar somente opções, também usamos comandos com o `addCommand()`. Um comando pode ter um `ArgParser` único, o que garante N opções de parâmetros que podem ser utilizados em conjunto com ele. No nosso caso:

- `cidade [-n --nome] NomeCidade [-e --estado] sc` : o comando `cidade` indica a consulta pelas cidades brasileiras. Os parâmetros `nome` e `estado` são opcionais.
- `agora [-i --id] 4765` : o comando `agora` vai consultar o clima atual de uma cidade, cujo `id` é passado por parâmetro.

Já o `help` é um pouco diferente. É padrão as aplicações CLI possuírem esse parâmetro que descreve como ela deve ser utilizada. Execute `dart --help` e serão impressas as opções de parâmetros para a VM, por exemplo.

O `help` é cadastrado como uma *flag*, o `negatable` como `false` apenas indica que a flag não pode ser negada, com o comando `--no-flag`. Todos

os parâmetros `valueHelp` das opções cadastradas funcionam como a descrição que será apresentada ao utilizar o `--help`. Continuando com o `parser`:

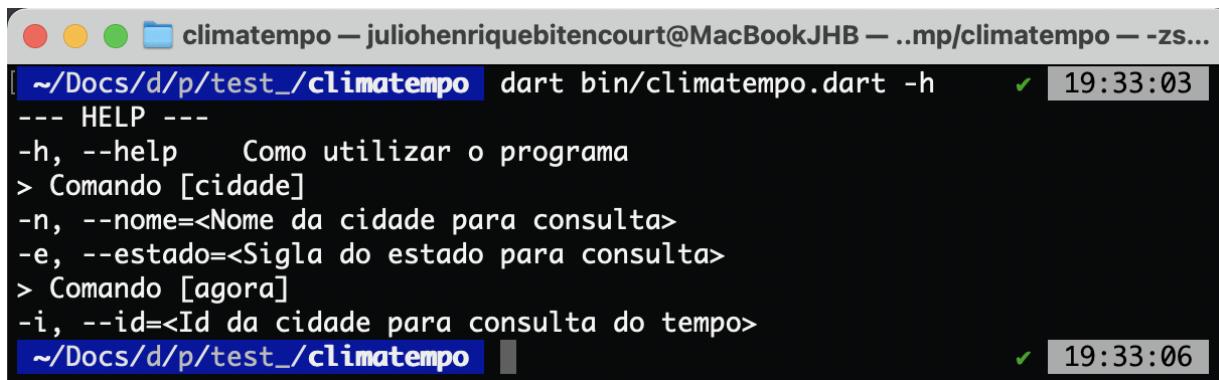
```
import 'dart:io';
import 'package:args/args.dart';
void main(List<String> args) async {
  final parser = criarParser();
  final argsResult = parser.parse(args);

  if (argsResult['help']) {
    mostrarAjuda(parser);
    exit(0);
  }
}
void mostrarAjuda(ArgParser parser) {
  print('--- HELP ---');
  print(parser.usage);
  for (var comando in parser.commands.entries) {
    print('> Comando [${comando.key}]');
    print(comando.value.usage);
  }
}
// código omitido
```

Os argumentos passados são mapeados para um `ArgsResult`, e a primeira coisa a ser feita é validar se o usuário solicitou o `help`. Como é uma flag, o `argsResult['help']` vai resultar em `true` caso esse parâmetro esteja presente no `argsResult`. E, com o auxílio da função `mostrarAjuda()`, varremos todo o `Map` de comandos presente no `ArgParser`, e o método `get usage` imprime todos esses valores de ajuda das opções.

A biblioteca `dart:io` possui a função `exit(0)`. Ela encerra o programa assim que é executada e recebe um *exit code* por parâmetro. Esse valor numérico é passado para o processo pai que executou o programa, informando qual o motivo da finalização. O código 0 indica uma finalização normal.

Neste ponto, já é possível testar a aplicação rodando `dart run bin/main.dart --help`.



The screenshot shows a terminal window with the following content:

```
climatempo — juliohenriquebitencourt@MacBookJHB — ..mp/climatempo — -zs...
[ ~/Docs/d/p/test_/climatempo ] dart bin/climatempo.dart -h ✓ 19:33:03
--- HELP ---
-h, --help      Como utilizar o programa
> Comando [cidade]
-n, --nome=<Nome da cidade para consulta>
-e, --estado=<Sigla do estado para consulta>
> Comando [agora]
-i, --id=<Id da cidade para consulta do tempo>
~/Docs/d/p/test_/climatempo ✓ 19:33:06
```

Figura 11.4: Comando help.

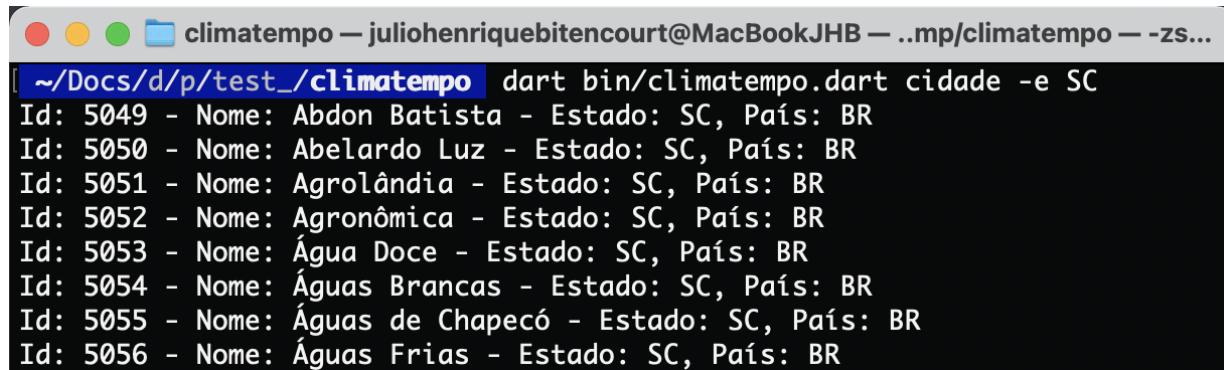
Agora é necessário tratar os comandos.

```
import 'package:climatempo/api.dart';
import 'package:climatempo/model/clima_tempo.dart';
void main(List<String> args) async {
    // código omitido
    final comando = argsResult.command;
    try {
        if (comando != null && comando.name == 'cidade') {
            final nomeCidade = comando['nome'];
            final estado = comando['estado'];

            final cidades = await buscarCidades(estado:
                nome: nomeCidade);
            cidades.forEach((c) => print(c));
        }
    } catch(e) {
        print(e);
    }
}
```

Com `argsResult.command` obtemos qual foi o comando utilizado. Se for `cidade`, é pego o valor das opções `nome` e `estado`, que podem ou não ser `null`. Então basta fazer a chamada para o `buscarCidades()` de `api.dart`

que implementamos anteriormente. O resultado dessa requisição é impresso no terminal.



```
[ ~ /Docs/d/p/test_/_climatempo ] dart bin/climatempo.dart cidade -e SC
Id: 5049 - Nome: Abdon Batista - Estado: SC, País: BR
Id: 5050 - Nome: Abelardo Luz - Estado: SC, País: BR
Id: 5051 - Nome: Agrolândia - Estado: SC, País: BR
Id: 5052 - Nome: Agronômica - Estado: SC, País: BR
Id: 5053 - Nome: Água Doce - Estado: SC, País: BR
Id: 5054 - Nome: Águas Brancas - Estado: SC, País: BR
Id: 5055 - Nome: Águas de Chapecó - Estado: SC, País: BR
Id: 5056 - Nome: Águas Frias - Estado: SC, País: BR
```

Figura 11.5: Comando cidade.

A mesma lógica é utilizada para o comando `agora`, que pode ser incluído também dentro do bloco `try / catch`:

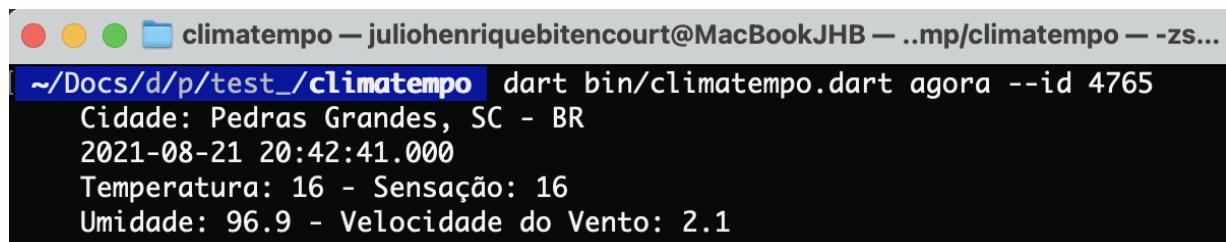
```
void main(List<String> args) async {
    // código omitido
    if (comando != null && comando.name == 'agora') {
        final id = comando['id'];
        if (id == null) {
            print('É obrigatório informar um [-id] de cidade');
            exit(2);
        }
        final tempo =
            await registrarCidadeEBuscarTempo(int.parse(id));
        print(tempo);
    }
}
```

O `id` da cidade é um parâmetro obrigatório, então se não for informado encerramos a aplicação com `exit(2)`. O código `2` indica para o processo pai que o app encerrou com erro. Com o `id` preenchido, ele é passado para a `registrarCidadeEBuscarTempo()`.

```
Future<ClimaTempo> registrarCidadeEBuscarTemp(int idCidade)
    async {
    await registrarCidade(idCidade: idCidade);
```

```
    return await climaAtual(idCidade: idCidade);  
}
```

Essa função possui um detalhe. Note que, com a ajuda do `await`, é possível primeiro chamar o `registrarCidade`, que vai permitir o acesso dessa cidade ao nosso token, e depois é consultado o seu `climaAtual()`. Usando essa estratégia, por mais que exista um limite de uma cidade para cada token, conseguimos consultar o clima de qualquer cidade desde que ela seja registrada previamente.



```
climatempo — juliohenriquebitencourt@MacBookJHB — ..mp/climatempo — -zs...  
~/Docs/d/p/test/_climatempo dart bin/climatempo.dart agora --id 4765  
Cidade: Pedras Grandes, SC - BR  
2021-08-21 20:42:41.000  
Temperatura: 16 - Sensação: 16  
Umidade: 96.9 - Velocidade do Vento: 2.1
```

Figura 11.6: Comando `agora`.

Com isso, o projeto está completo e com tudo funcionando! Como estamos utilizando um token gratuito da API, ela pode limitar esse uso de alguma forma, então é possível que eventualmente alguma requisição retorne um status diferente de 200 com algum erro, que será impresso no terminal. Por exemplo, existe um limite de tempo entre o registro de cidades diferentes para o mesmo token, o que pode ocasionar problemas ao utilizar o comando `agora` com uma outra cidade.

11.6 Rodando um app CLI de qualquer lugar

Até o momento, para rodar a aplicação, sempre foi necessário utilizar o comando `dart run` ou `dart bin/climatempo.dart` da Dart VM e estar dentro do diretório do projeto. O comando `run` inclusive também existia no `pub`, onde é possível executar:

```
pub run climatempo:climatempo -h
```

O primeiro `climatempo` é o nome do package, e o segundo, o do arquivo executável. Foi só posteriormente que este comando `run` foi incorporado à VM. Mas o interessante para nós é que seja possível rodar nosso app de qualquer lugar no nosso terminal e não apenas no seu diretório.

Lembra como no capítulo 4 utilizamos o `pub global` para ativar o package `webdev` globalmente? O conceito é o mesmo, precisamos ativar globalmente o script executável do nosso app. Para isso, é necessário modificar o `pubspec.yaml`.

```
# código omitido
executables:
  climatempo: climatempo
```

A seção `executables` permite definir diferentes scripts executáveis de entrada para a aplicação, assimilando um apelido `climatempo`: a um script `climatempo`, que se refere ao `climatempo.dart`. Isso porque uma aplicação pode ter mais de um script executável. Execute `dart pub get` e após isso conseguimos fazer a instalação global com `dart pub global activate --source path <path>`, substituindo o `<path>` pelo diretório do app:

```
dart pub global activate --source path
/Users/jhbitencourt/dev/climatempo
```

Isso é o suficiente. O comando `dart pub global list` já vai listar o script ativo e em qualquer diretório é possível executar a aplicação apenas com seu apelido.

```
climatempo --help
climatempo cidade -e SC
climatempo agora --id 4765
```

Compilando AOT

Ao rodar a aplicação, seja no seu diretório ou com um script ativo global, a Dart VM sempre está compilando para em seguida executar. Nossa app é pequeno e para estudos, mas quando falamos de aplicações que serão

utilizadas em produção, otimizações precisam ser realizadas, e esse é o papel do compilador AOT (*Ahead of Time*) presente na VM.

O ecossistema de Dart possui uma série de compiladores que produzem código otimizado para diferentes plataformas, desktop, web, mobile e *embedded devices*. Historicamente esses compiladores foram criados em aplicações separadas, então é comum você encontrar locais se referindo ao `dart2js`, `dart2aot` e `dart2native`, por exemplo. Esse último e mais recente, inclusive, foi introduzido na versão 2.6 da linguagem com o objetivo de compilar nativamente para as plataformas desktop: Windows, Mac e Linux.

Acontece que, com a unificação dos comandos em uma única interface, todos eles foram englobados ao comando `dart compile`. Então conseguimos executar:

```
dart compile aot-snapshot bin/climatempo.dart
```

Um arquivo `bin/climatempo.aot` é criado contendo o binário otimizado para execução e pode ser executado com o *runner* `dartaotruntime`:

```
dartaotruntime bin/climatempo.aot --help  
dartaotruntime bin/climatempo.aot cidade -e SC  
dartaotruntime bin/climatempo.aot agora --id 4765
```

Binários independentes

Já sabemos como rodar o projeto em seu diretório através do `pub global` e também do compilador AOT. Cada um com suas vantagens, mas todos com um defeito em comum: a dependência da Dart VM. Todos são executados através dela. Se quisermos enviar nossa aplicação CLI para um amigo testar em seu computador, por exemplo, ele vai precisar ter a Dart VM instalada em sua máquina.

Por conta disso, existe uma outra opção de compilar um binário executável independente para as plataformas Windows, Mac e Linux: compilando com o comando `exe`:

```
dart compile exe bin/climatempo.dart
```

Isso gera um arquivo binário `bin/climatempo.exe` com tudo o que é necessário para sua execução. Então para rodar é só executar diretamente:

```
climatempo -h  
climatempo.exe -h
```

Qualquer máquina, mesmo sem a Dart VM, vai conseguir rodá-lo. Porém, uma das limitações é que esse executável não é *cross-platform*, o que significa que ele executará apenas no SO que você utilizou para compilar. Se você compilar com Windows, ele executará apenas em máquinas com o Windows, e o mesmo vale para Mac e Linux.

Até aqui

Você teve uma rápida prévia de como funciona a construção de aplicações CLI e seus padrões, como consumir dados de uma API e transformar esse json em objetos, além de ver como executar e compilar em diferentes formas.

E seguindo esses mesmos conceitos vistos até aqui, você já pode implementar suas próprias aplicações via linha de comando e integrar com diferentes APIs. Inclusive, aproveite para pesquisar sobre o package `json_serializable`. Ele auxilia na transformação do `Map` com os dados do `json` no nosso objeto *model* e vice-versa.

A seguir, vamos nos aprofundar mais nos recursos assíncronos com as famosas *streams*.

CAPÍTULO 12

Stream é tão funcional...

Em Ciência da Computação, o termo *stream* é utilizado como o conceito de algo que transporta dados de forma sequencial, disponíveis no decorrer do tempo e respeitando sempre uma única direção. No mundo teórico, existem algumas abstrações que podem ajudar a entender o que isso representa, como um **cano** por onde passa a água, uma **rodovia** de mão única por onde os carros transitam ou até mesmo a sua **orelha** por onde são transportados os ruídos externos para o cérebro.

Em programação, existem diversas implementações que se originam dessa ideia. Um uso clássico está nas operações de I/O (input/output) feitas através de streams de escrita e leitura de bytes em arquivos. Durante o capítulo, além de focar em como Dart aborda e implementa sua API de streams, veremos:

- O que é uma *stream*;
- As diferenças entre inscrição única e *broadcast*;
- A importância de um `StreamController` ;
- Como criar, adicionar e ler dados de uma *stream*;
- Como manipular os seus dados através de suas operações.

12.1 O que é uma Stream?

As *streams* em si também representam uma outra forma de trabalhar de forma assíncrona, pois partem da ideia de que um dado pode ser disponibilizado no decorrer do tempo. Elas permitem uma melhor modularização do sistema, desacoplando e reaproveitando responsabilidades, por isso estão diretamente ligadas ao conceito de Programação Reativa.

Juntas, a API de *futures* e a API de *streams* representam a base da programação assíncrona em Dart. Porém, enquanto um *future* completa com um único resultado, uma *stream* pode produzir N resultados durante o seu ciclo de vida.

	Único	Múltiplo
sync	<code>String</code>	<code>Iterable<String></code>
async	<code>Future<String></code>	<code>Stream<String></code>

Figura 12.1: Comparativo de dados síncronos e assíncronos.

Assim como em um dado síncrono, o `Iterable<String>` representa um conjunto de dados comparado a uma `String`. Em um dado assíncrono, uma `Stream<String>` representa o mesmo para um `Future<String>`. Encare a *stream* como uma forma de fornecer uma sequência de dados assíncronos.

- - elemento
- ✗ - erro
- ✓ - completada

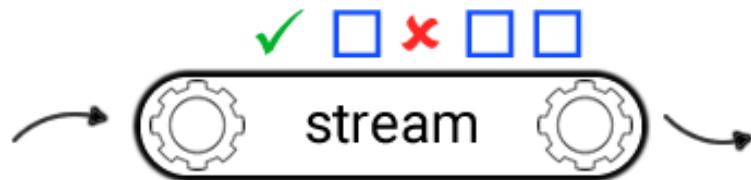


Figura 12.2: Uma stream é como uma esteira de dados.

Em uma analogia, uma *stream* pode ser comparada a uma *queue* ou a uma simples esteira, onde um ou mais eventos entram por um lado e saem do outro, sempre respeitando a ordem de entrada. Esses eventos podem ser:

- Um elemento de dado padrão;
- Uma informação de erro;
- Uma informação de finalizada.

Um evento com a informação de que a stream foi finalizada é disparado uma única vez e indica que ela não produzirá mais eventos a partir daquele momento, sendo assim, esse é sempre o último a ser gerado.

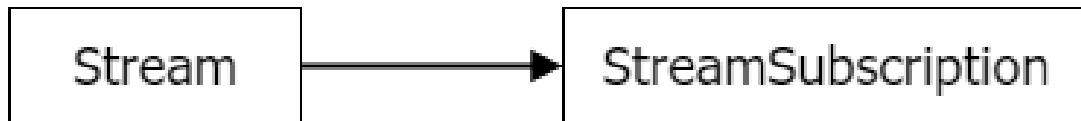


Figura 12.3: Uma stream possui uma StreamSubscription.

Os tipos `Stream` e `StreamSubscription` representam a estrutura básica de uma *stream* em Dart. Em um exemplo simples, dada uma stream que emitirá um elemento 'A' , ela pode ser construída com:

```
Stream stream = Stream.value('A');
```

O construtor `value()` cria um objeto do tipo `Stream` já com um valor a ser processado. Uma vez que a função de uma stream é transmitir informações, essa mesma informação só vai ter algum propósito se houver alguém que a consuma. Então é possível definir um *listener* (ou uma inscrição) em uma stream, de forma que toda informação ao chegar no final da "esteira" seja capturada.

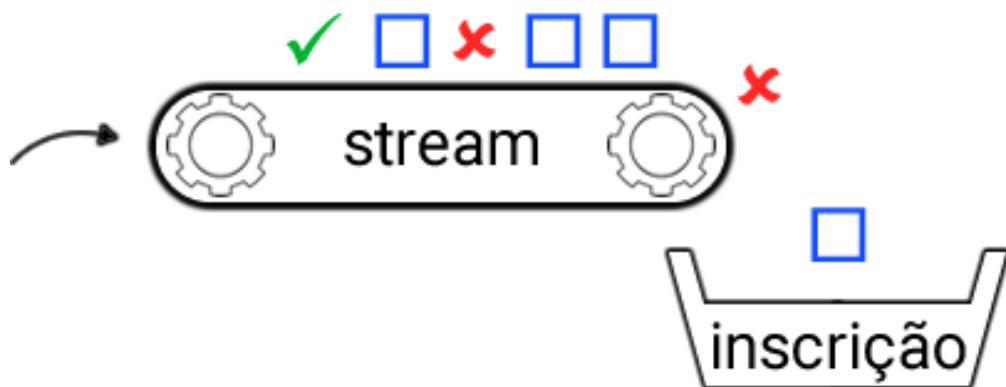


Figura 12.4: StreamSubscription captura os dados da Stream.

```
import 'dart:async';
void main() {
  print('Início main()');
  final stream = Stream<String>.value('A');
```

```
StreamSubscription subscription = stream.listen((dados) {  
    print('Novo evento: $dados');  
});  
print('Fim main()');  
}  
  
> Início main()  
> Fim main()  
> Novo evento: A
```

Com o método `listen()`, é criado um callback que será chamado sempre que um novo dado sair da stream. O retorno síncrono deste método é uma instância de `StreamSubscription`, que representa uma inscrição para receber os eventos transmitidos. Essa mesma instância permite manipular posteriormente todos os callbacks associados a ela, por isso não é difícil encontrar um `stream.listen(null)` apenas para obter uma instância de inscrição.

Por trabalhar de forma assíncrona, esses eventos também são processados pela *event queue*, o que faz o callback ser executado apenas após a finalização do método `main()`. E independente da quantidade de eventos transmitidos, uma inscrição, enquanto estiver ativa, sempre escutará todos os eventos.

```
void main() {  
    final stream = Stream.fromIterable(['A', 'E', 'I']);  
    final subscription = stream.listen(null);  
    subscription.onData((dados) {  
        print('Novo evento: $dados');  
    });  
}  
  
> Novo evento: A  
> Novo evento: E  
> Novo evento: I
```

O `fromIterable()` cria uma `Stream` a partir dos elementos de um `Iterable`, que serão transmitidos um a um, por isso o callback é chamado três vezes. E, como já sabemos, uma stream também pode transmitir uma informação de erro ou de que ela foi finalizada.

```

void main() {
    final stream = Stream.error('StreamErro');
    final subscription = stream.listen((dados) {
        print('Novo evento: $dados');
    });
    subscription.onError((e) {
        print('Erro capturado: $e');
    });
    subscription.onDone(() => print('Stream finalizada'));
}

```

> Erro capturado: StreamErro
> Stream finalizada

A mesma instância de `StreamSubscription` ainda permite a adição dos callbacks `onError` e `onDone`. O construtor `error()` cria uma `Stream` que produz um erro capturado por `onError()`. Note que, ao executar esse código, o `onDone()`, que captura a informação de stream finalizada, também é disparado. Isso porque no momento estamos criando as streams através de seus construtores, com valores fixos, o que a faz ser encerrada assim que os eventos são processados.

Uma inscrição pode ser cancelada a qualquer momento através do método `cancel()`. A partir desse momento, ela deixará de receber qualquer evento transmitido pela stream. Faz parte das boas práticas sempre encerrar inscrições que não serão mais necessárias, mantendo o código coeso.

12.2 Inscrição única versus broadcast

Por padrão, uma stream permite que exista apenas um único *listener* em todo o seu ciclo de vida. Ao tentar ouvir uma stream mais de uma vez, o código resultará em erro.

```

void main() async {
    final stream = Stream.value('42');
    final inscricaoUm = stream.listen(print);

```

```
    final inscricaoDois = stream.listen(print); // Erro!
}
```

A segunda tentativa de inscrever-se lança a exceção 'Bad state: Stream has already been listened to.' . Até mesmo se a primeira inscrição for cancelada, com `inscricaoUm.cancel()` , nenhuma outra inscrição para essa mesma stream é permitida.

Para aceitar esse comportamento, existe um outro tipo de stream conhecido como *broadcast*. Como o próprio nome indica, com ela é possível transmitir informações para vários receptores, ou seja, obter várias inscrições para uma única stream, sendo que todas receberão sempre os mesmos eventos.

```
void main() async {
  final stream = Stream.value('42').asBroadcastStream();
  stream.listen(( dado ) => print('Inscrição 1 - $dado'));
  stream.listen(( dado ) => print('Inscrição 2 - $dado'));
}

> Inscrição 1 - 42
> Inscrição 2 - 42
```

Toda `Stream` padrão pode se tornar uma nova `Stream` do tipo *broadcast* a partir do método `asBroadcastStream()` . Assim, cada chamada ao método `listen()` desta produzirá uma `StreamSubscription` diferente com seus próprios métodos de callback.

No momento é o suficiente saber que essa capacidade de inscrições permitida é a principal característica que separa uma stream de inscrição única para uma do tipo broadcast. Mas não é a única, existem outras implicações que serão discutidas mais à frente. Antes, precisamos conhecer mais sobre o controle de uma `Stream` .

12.3 StreamController

Grande parte das streams que acabamos utilizando durante o desenvolvimento no dia a dia já estão implementadas e fazem parte do SDK

ou de packages terceiros. Mas eventualmente é necessário criar nossa própria `Stream` e ter um controle total sobre ela. É nesse ponto que um `StreamController` é útil.

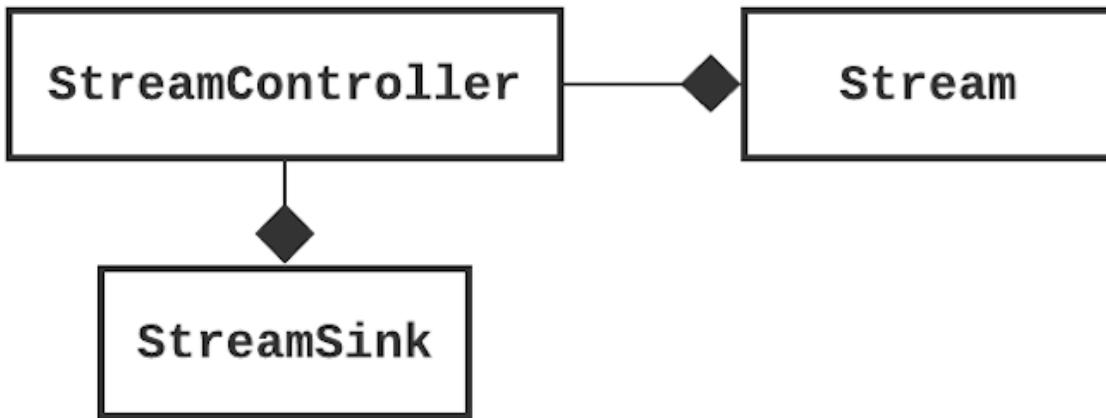


Figura 12.5: Organograma `StreamController`.

As streams geradas a partir dos construtores nomeados da classe `Stream` possuem o problema de serem criadas com valores fixos e finitos. Mas não necessariamente saberemos, já em sua criação, todos os valores que uma stream deve transmitir até o seu encerramento. Inclusive, é bem improvável e foge um pouco do conceito de uma stream saber isso.

Por exemplo, no capítulo 4, criamos o jogo e cadastramos um *listener* na `stream` de eventos do *mouse* para ser possível identificar os cliques nos botões. No momento de sua criação é impossível saber quais eventos serão transmitidos, pois depende da manipulação do usuário e esses eventos são processados à medida que ele interage com o sistema. A `Stream` transmite os eventos de *click* (ação) aos interessados (inscrições), e estes respondem de alguma forma gerando uma resposta (reação), no nosso caso, o início do jogo.

Por conta disso, a criação de uma `Stream` costuma sempre estar associada a um `StreamController`.

```
import 'dart:async';
void main() {
```

```

final controller = StreamController();
Stream stream = controller.stream;
}

```

Todo `StreamController` já possui por padrão uma única `Stream` atrelada a ele, acessível através do próprio atributo `stream`, que permite criar novas inscrições. Até aí tudo normal, mas as diferenças começam com a forma em que os dados entram nessa `stream`.

Interface Sink

Um `controller` fornece a possibilidade de adição de elementos sob demanda na `stream` através de um *sink*.

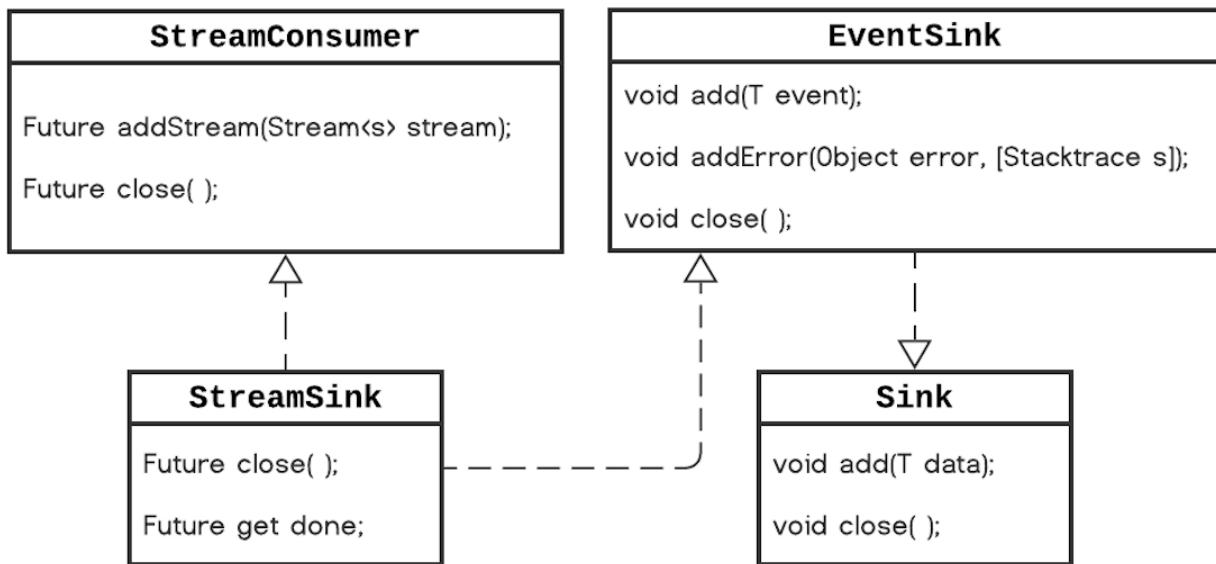


Figura 12.6: Organograma de classes Sink.

A interface `Sink` é uma representação genérica de destino de dados. Múltiplos dados podem ser inseridos em chamadas consecutivas ao `add()` até que, em determinado momento, quando não há mais dados, a *sink* deve ser encerrada com o `close()`.

Já a `EventSink`, que implementa a interface `Sink`, é uma *sink* com a possibilidade de adicionar um erro através de seu método `addError()`, resultando em uma *sink* apta para trabalhar com computações assíncronas.

que podem completar com um valor ou erro. O `StreamController`, entretanto, possui uma implementação de `sink` do tipo `StreamSink`, que por sua vez implementa `EventSink`.

```
import 'dart:async';
void main() {
    final controller = StreamController();
    final stream = controller.stream;
    final subscription = stream.listen(( dado ) {
        print('Novo evento: $dado');
    });
    subscription.onError(( e ) {
        print('Erro capturado: $e');
    });

    StreamSink sink = controller.sink;
    sink.add(42);
    sink.addError('Erro');
}

> Novo evento: 42
> Erro capturado: Erro
```

Através de seus métodos `add()` e `addError()` é possível adicionar novos elementos a qualquer momento para serem processados na `stream`, permitindo agora um controle muito maior. Mas essa não é a única forma de adicionar elementos ao `sink`.

Se você voltar ao organograma das classes de `sink`, verá que `StreamSink` também implementa a interface `StreamConsumer`. Ela representa uma `sink` que pode receber múltiplas `streams` como entrada de dados através do método `addStream()`, onde todos os eventos da `stream` passada são consumidos e inseridos na nova `stream`. E, quando necessário, ela também define o `close()`, que avisa ao `sink` que nenhuma outra `stream` será processada.

```
import 'dart:async';
void main() {
    final controller = StreamController();
    controller.stream.listen(( dado ) {
```

```

    print('Novo evento: $dado');
});

StreamSink sink = controller.sink;
final future = sink.addStream(
    Stream.fromIterable(['A', 'B', 'C']));
future.whenComplete(() => print('Todos eventos processados'));
}

> Novo evento: A
> Novo evento: B
> Novo evento: C
> Todos eventos processados

```

O retorno de `addStream()` é um `Future` que indica quando todos os eventos daquela stream foram processados. Como usamos novamente o construtor `Stream.fromIterable` com valores finitos, a stream é encerrada após emitir o evento '`c`' , completando o seu processamento. Considere o próximo exemplo:

```

import 'dart:async';
void main() {
    final controllerUm = StreamController();
    controllerUm.stream.listen((dado) {
        print('Novo evento StreamUm: $dado');
    }, onDone: () => print('StreamUm finalizada'));

    final controllerDois = StreamController.broadcast();

    final future = controllerUm.sink
        .addStream(controllerDois.stream);
    future.whenComplete(() => print('Todos eventos processados'));

    controllerDois.stream.listen((dado) {
        print('Novo evento StreamDois: $dado');
    }, onDone: () => print('StreamDois finalizada'));
    controllerDois.sink.add(42);
}

> Novo evento StreamDois: 42
> Novo evento StreamUm: 42

```

Agora o sink do `controllerUm` consome os dados da stream do `controllerDois`. Dessa forma, ao adicionar um evento `42` ao sink do `controllerDois`, ambas as streams vão processar esse dado. Mas existem ainda duas observações importantes a respeito desse exemplo.

A primeira é que o construtor `StreamController.broadcast` produz um controller que contém uma stream do tipo broadcast. Isso é necessário pois, além da inscrição que criamos através do `listen`, o método `addStream()` do sink também acaba registrando outra inscrição para consumir os eventos internamente.

A segunda é que dessa vez o `Future` retornado pelo `addStream()` não foi completado, assim como os callbacks de `onDone` registrados nas streams também não foram executados. Diferente do exemplo utilizando `Stream.fromIterable`, uma stream criada através de um controller não possui valores finitos, ela ficará sempre aberta aguardando a entrada de novos eventos.

Então uma forma de avisar que os eventos encerraram é utilizando o próprio `close()` do sink:

```
import 'dart:async';
void main() {
    final controllerUm = StreamController();
    controllerUm.stream.listen((dados) {
        print('Novo evento StreamUm: $dados');
    }, onDone: () => print('StreamUm finalizada'));

    final controllerDois = StreamController.broadcast();
    final future = controllerUm.sink
        .addStream(controllerDois.stream);
    future.whenComplete(() {
        print('Todos eventos processados');
        controllerUm.sink.close();
    });

    controllerDois.stream.listen((dados) {
        print('Novo evento StreamDois: $dados');
    }, onDone: () => print('StreamDois finalizada'));
}
```

```
    controllerDois.sink.add(42);
    controllerDois.sink.close();
}

> Novo evento StreamDois: 42
> Novo evento StreamUm: 42
> StreamDois finalizada
> Todos eventos processados
> StreamUm finalizada
```

Isso garante o encerramento da stream dois, e nenhum outro método do sink do `controllerDois` pode ser chamado para adição de novos elementos a partir do momento em que o `close()` é processado. E o sink do `controllerUm` também é avisado de que todos os eventos da stream passada em `addStream()` foram processados, encerrando também a stream um.

O objetivo desse último exemplo é alertar que o ideal é sempre encerrar os recursos quando finalizar a utilização de alguma stream, evitando qualquer possibilidade de ocorrerem *memory leaks* nas aplicações. Caso necessário, uma forma de confirmar o encerramento de uma sink é através do seu atributo `controller.sink.done`, um `Future` que completará assim que a `streamSink` for finalizada.

StreamController versus StreamSink

O `StreamController` possui um `sink` que utilizamos para adicionar elementos a serem processados pela stream. Mas o que pode gerar confusão para algumas pessoas é que o próprio `StreamController` é um `StreamSink`, pois ele implementa a sua interface.

```
abstract class StreamController<T> implements StreamSink<T> {
    StreamSink<T> get sink;
}
```

Isso faz com que todos os métodos do `streamSink` sejam acessíveis diretamente pelo `StreamController`:

StreamController	sink
controller.add()	controller.sink.add()
controller.addError()	controller.sink.addError()
controller.addStream()	controller.sink.addStream()
controller.close()	controller.sink.close()
controller.done	controller.sink.done

Na prática, as duas formas se comportarão internamente de maneira igual. A diferença está mais associada ao encapsulamento de informações. Considere um método que recebe um `StreamSink` :

```
import 'dart:async';
void acessaSink(StreamSink sink) {
  sink.add(42);
  if (sink is StreamController) {
    sink.stream.listen(print);
  }
}
```

E é passado para ele um `StreamController` :

```
acessaSink(StreamController());
```

A validação feita em `sink is StreamController` resulta em verdadeiro, e com isso se tem acesso a todos os atributos e métodos da classe `StreamController`. Por isso, ao rodar o código, é impresso 42 através da inscrição efetuada na stream. Por outro lado, ao passar o atributo `sink`, nada será impresso, pois o `sink` não é um `StreamController` :

```
acessaSink(StreamController().sink);
```

Isso resulta em um melhor acoplamento de informações, afinal o método em questão precisa lidar apenas com informações de um sink e não deveria poder manipular o controller desta stream.

Estados do StreamController

Já conhecemos as duas formas de se criar um `StreamController` com os seus construtores *factory*. A primeira é quando queremos uma stream de inscrição única:

```
factory StreamController({void onListen(), void onPause(),
    void onResume(), FutureOr<void> onCancel(), bool sync = false})
```

Ela permite registrar vários callbacks que refletem ações na medida em que o controller for utilizado. Os parâmetros são:

- **onListen**: chamado assim que uma inscrição for registrada na stream. É comum e faz parte das boas práticas disparar eventos para uma stream apenas quando ela possuir inscrições, e o `onListen` é utilizado para essas situações.
- **onPause**: chamado quando a stream for pausada através da sua inscrição.
- **onResume**: chamado quando a stream pausada for resumida através da sua inscrição.
- **onCancel**: chamado quando a inscrição é cancelada, encerrando o seu recebimento de eventos. Se for necessário executar alguma ação assíncrona, é possível retornar um `Future`.
- **sync**: por padrão é `false`, mas pode ser utilizado para obter um `StreamController` síncrono, onde os eventos são disparados imediatamente de forma síncrona para os *listeners*. Deve ser utilizado com cuidado, pois, caso contrário, pode resultar em comportamentos errados. Provavelmente você nunca precisará utilizar um controller síncrono.

Esse controller de inscrição única possui um ciclo de vida que pode ser definido como:

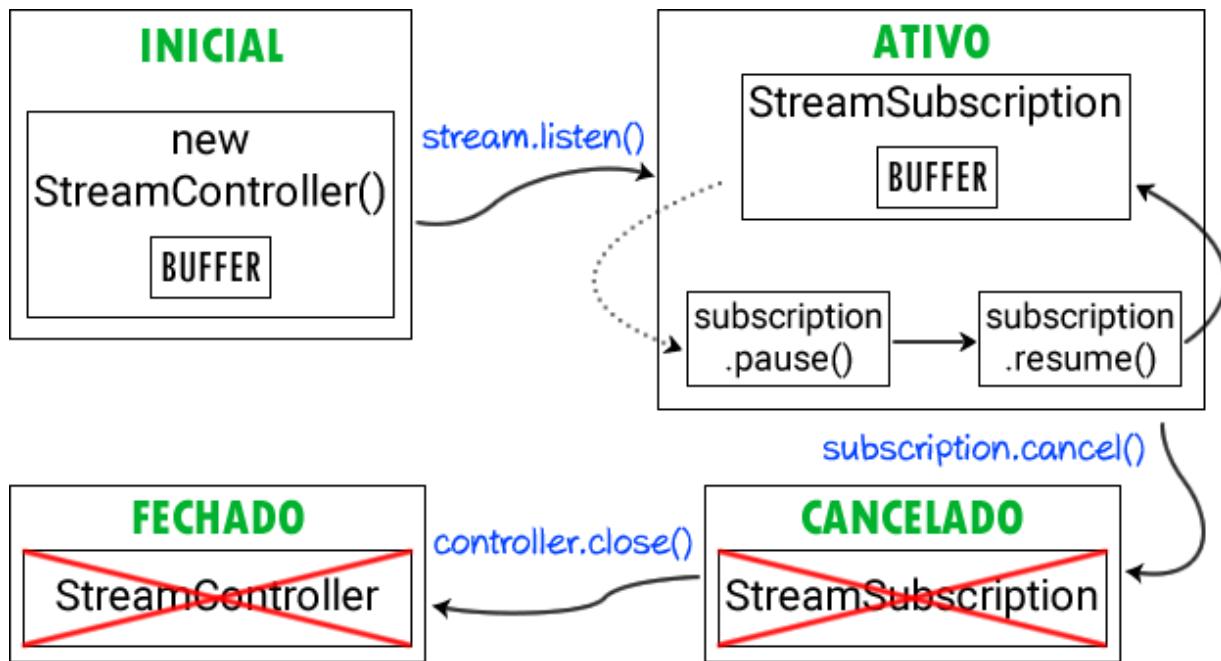


Figura 12.7: Ciclo de vida StreamController inscrição única.

1. Estado inicial: quando criado, o controller se encontra no estado **inicial**, onde não possui nenhuma inscrição. Todos os eventos adicionados nesse momento irão para um buffer de eventos.
2. Estado ativo: assim que uma inscrição é registrada na stream, o controller passa para o estado **ativo**. Nele, a stream funciona normalmente. Se houver eventos no buffer do controller, eles serão emitidos primeiro mantendo a mesma ordem; após isso novos eventos são emitidos. A qualquer momento, a `StreamSubscription` pode pausar, disparando o `onPause` do controller. Com isso, é possível e indicado pausar a fonte de dados para não emitir mais eventos. Caso a fonte de dados continue alimentando a stream, os eventos vão para um buffer interno da inscrição até que ela seja resumida e receba esses eventos.
3. Estado cancelado: a inscrição pode ser cancelada a qualquer momento e, a partir disso, o controller também entra no estado de cancelado disparando o seu callback `onCancel`. Após isso, por ser de inscrição única, nenhuma outra inscrição pode ser registrada e o controller passa novamente a ficar sem inscrição. Nesse estado ainda é possível adicionar eventos na stream, mas note que já não faz mais sentido, pois

ninguém vai recebê-los, então é recomendado encerrar a fonte de dados.

4. Estado fechado: esse é o estado final de um controller. Nele não é mais permitido adicionar eventos e a tentativa resulta em uma exceção. Não é possível reabrir um controller fechado.

No exemplo a seguir, vemos os callbacks de um controller de inscrição única em ação:

```
import 'dart:async';
void dispararEventos(StreamController controller) async {
  for (var i = 1; i <= 6; i++) {
    if (i == 5) {
      controller.sink.addError('Erro no número $i');
      continue;
    }
    await Future.delayed(Duration(seconds: 1), () {
      controller.sink.add(i);
    });
  }
  controller.sink.close();
  print('StreamController em estado FECHADO');
}

void main() {
  StreamController<int>? controller;
  controller = StreamController<int>(
    onListen: () {
      print('StreamController em estado ATIVO');
      dispararEventos(controller!);
    },
    onResume: () => print('StreamController resumido'),
    onPause: () => print('StreamController pausado'),
    onCancel: () => print('StreamController em estado CANCELADO'),
    sync: false);
  controller.sink.add(0);

  print('StreamController em estado INICIAL');
  StreamSubscription? inscricao;
  Future.delayed(Duration(seconds: 2), () {
```

```

    inscricao = controller!.stream.listen((int dado) {
      print('Número: $dado');
      if (dado == 1) {
        print('Inscrição pausada');
        inscricao!.pause(Future.delayed(Duration(seconds: 1),
            () => print('Inscrição resumida')));
      }
    }, onError: (print), onDone: () => print('onDone'),
    cancelOnError: true);
  });

> StreamController em estado INICIAL
> StreamController em estado ATIVO
> Número: 0
> Número: 1
> Inscrição pausada
> StreamController pausado
> Inscrição resumida
> Número: 2
> StreamController resumido
> Número: 3
> Número: 4
> StreamController em estado CANCELADO
> Erro no número 5
> StreamController em estado FECHADO

```

Alguns pontos-chaves para entendimento total do ciclo de vida do controller:

- Ainda em estado inicial, o evento `0` foi adicionado através do sink. Nesse momento, ele é mantido no buffer interno do controller.
- Assim que o future de dois segundos é completado, é registrada uma inscrição na stream. Isso faz com que o callback `onListen` dispare chamando a função `dispararEventos()`, que a cada segundo adiciona números como eventos na stream. A inscrição recebe o evento que estava no buffer e agora o controller está no estado ativo.
- A inscrição recebe o evento `1` e é pausada. O método `pause()` da inscrição pode receber um future por parâmetro que indica quando a inscrição deve ser resumida; no exemplo, dura apenas um segundo. Os

callbacks `onPause` e `onResume` são disparados respectivamente. Uma outra forma de resumir a inscrição é chamando diretamente o seu método `resume()`.

- Enquanto pausada, os eventos [2, 3] são adicionados e vão para o buffer interno da inscrição e, quando reiniciada, ela recebe os eventos deste buffer.
- O evento de número 5 é um evento de erro e como a inscrição foi criada com o parâmetro `cancelOnError = true`, o primeiro erro recebido faz ela ser cancelada automaticamente disparando o callback `onCancel`. Assim, o controller entra no estado cancelado.
- O evento 6 ainda é disparado, mas já não há mais nenhuma inscrição no controller. Então é chamado o seu método `close()` e ele entra no estado final de fechado.

A segunda forma de criar um controller é com o seu construtor nomeado `StreamController.broadcast`:

```
factory StreamController.broadcast(  
    {void onListen(), void onCancel(), bool sync = false})
```

O resultado é um controller que permite múltiplas inscrições e, por conta disso, os parâmetros possuem diferenças. A primeira delas é a ausência dos callbacks `onPause` e `onResume`, afinal, quando o controller deve pausar ou resumir agora que existem várias inscrições em diferentes situações? Esse papel agora é controlado exclusivamente pela inscrição. Quanto aos parâmetros:

- **onListen**: chamado sempre que o controller não possui inscrições e recebe uma inscrição. Isso pode ser na primeira vez em que recebe uma inscrição ao ser criado, ou após ter sido cancelado e receber uma nova inscrição.
- **onCancel**: chamado sempre que a última inscrição ativa for cancelada.
- **sync**: segue os mesmos princípios do parâmetro do controller de inscrição única.

Algumas mudanças também são visíveis ao olharmos o seu ciclo de vida:

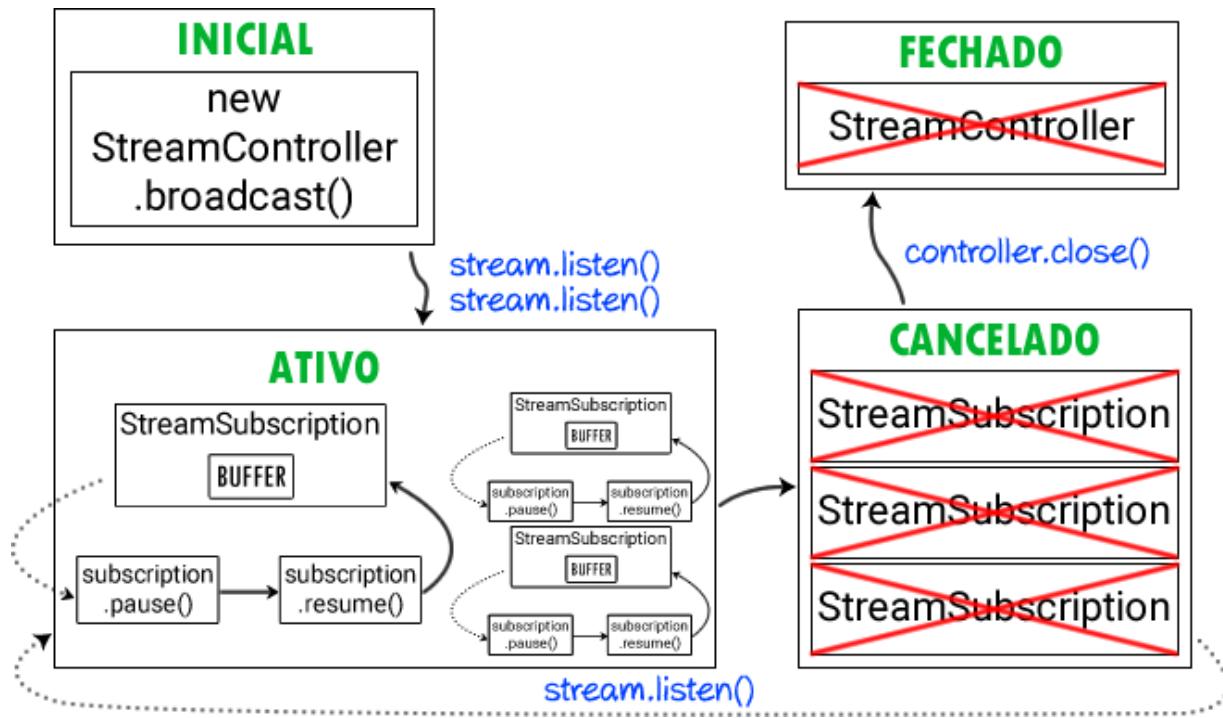


Figura 12.8: Ciclo de vida StreamController broadcast.

1. Estado inicial: nesse estado, o controller não possui inscrições registradas. Os eventos adicionados nesse momento são descartados uma vez que o *controller broadcast* não possui buffer.
2. Estado ativo: estado principal de funcionamento da stream. O controller entra nesse estado assim que a sua primeira inscrição é registrada ou é feita uma nova inscrição após ter sido cancelada. As inscrições podem pausar ou resumir a qualquer momento, não alterando a emissão de eventos pelo controller. Esse controle deve ser feito unicamente pelas inscrições, que possuem um buffer interno de eventos.
3. Estado cancelado: assim que a última inscrição ativa for cancelada, o controller entrará no estado de cancelado. Nesse momento é possível enviar eventos, porém eles serão descartados. Note que existe um ciclo entre o estado de cancelado e ativo, pois no controller de tipo *broadcast* é possível registrar novas inscrições mesmo após o controller ser cancelado, voltando a ficar ativo.
4. Estado fechado: é o estado final do controller após chamar seu método `close()`. Nesse estado não são aceitos mais eventos e a tentativa de

adição resulta em exceção. Novas inscrições são aceitas, mas são encerradas automaticamente. Não é possível reabrir um controller fechado.

Controller em funcionamento:

```
import 'dart:async';
void dispararEventos(StreamController controller) async {
    for (var i = 1; i <= 6; i++) {
        if (i == 3) {
            controller.sink.addError('Erro no número $i');
            continue;
        }
        await Future.delayed(Duration(seconds: 1), () {
            controller.sink.add(i);
        });
    }
    controller.sink.close();
    print('StreamController em estado FECHADO');
}

void criarInscricao(StreamController<int> controller, int valor) {
    StreamSubscription? inscricao;
    Future.delayed(Duration(seconds: 2), () {
        inscricao = controller.stream.listen((int dado) {
            print('[Inscrição $valor] número: $dado');
            if (dado == valor) {
                print('[Inscrição $valor] pausada');
                inscricao!.pause(Future.delayed(Duration(seconds: 2),
                    () => print('[Inscrição $valor] resumida')));
            }
            if (dado == valor + 3) {
                print('[Inscrição $valor] cancelada');
                inscricao!.cancel();
            }
        }, onError: (e) => print('[Inscrição $valor] $e'),
            onDone: () => print('[Inscrição $valor] onDone'),
            cancelOnError: false);
    });
}

void main() {
    StreamController<int>? controller;
```

```

controller = StreamController<int>.broadcast(
    onListen: () {
        print('StreamController em estado ATIVO');
        dispararEventos(controller!);
    },
    onCancel:()=>print('StreamController em estado CANCELADO'),
    sync: false);
controller.sink.add(0);
print('StreamController em estado INICIAL');
criarInscricao(controller, 1);
criarInscricao(controller, 2);
Future.delayed(Duration(seconds: 8), () {
    criarInscricao(controller!, 3);
});
}

StreamController em estado INICIAL
StreamController em estado ATIVO
[Inscrição 1] número: 1
[Inscrição 1] pausada
[Inscrição 2] número: 1
[Inscrição 2] número: 2
[Inscrição 2] pausada
[Inscrição 1] resumida
[Inscrição 1] número: 2
[Inscrição 1] Erro no número 3
[Inscrição 1] número: 4
[Inscrição 1] cancelada
[Inscrição 2] resumida
[Inscrição 2] Erro no número 3
[Inscrição 2] número: 4
[Inscrição 2] número: 5
[Inscrição 2] cancelada
StreamController em estado CANCELADO
StreamController em estado FECHADO
[Inscrição 3] onDone

```

- Ainda em estado inicial, o evento `0` foi adicionado através do `sink`, mas dessa vez ele é descartado, pois o `controller broadcast` não possui buffer.

- Assim que os futures de dois segundos completam, são registradas duas inscrições na stream com o método `criarInscricao()`, passando o controller para o estado ativo.
- Ambas as inscrições são pausadas em tempos diferentes e, quando resumidas, recebem os eventos que ficaram no buffer da inscrição, incluindo o evento de erro que dessa vez não cancela a inscrição, pois o parâmetro `cancelOn Error` está `false`.
- A primeira inscrição é cancelada quando recebe o evento `4`, deixando de receber os próximos eventos emitidos. Já a segunda é cancelada no evento `5` e, como é a última inscrição ativa, acaba disparando o `onCancel` e passando o controller para o estado cancelado.
- O evento `6` ainda é inserido mas é descartado pois não existem inscrições ativas. O controller é então encerrado passando para o estado de fechado.
- Note que mesmo depois de fechado uma terceira inscrição é registrada, diferente do controller de inscrição única que dispararia um erro. Mas por estar fechado, a inscrição recebe imediatamente um evento de finalizado e é encerrada automaticamente.

Agora separe um momento, pare e analise com atenção os trechos de código que ilustram o ciclo de vida dos diferentes controllers. Não deixe de experimentar, criar seus próprios callbacks e exemplos para ter o domínio e entender de fato o comportamento em diferentes situações. Com certeza esse conhecimento vai facilitar futuramente a codificação e correção de eventuais erros em seus próprios programas.

12.4 Stream e o await

Ao introduzir programação assíncrona e falar sobre future no capítulo 10, conhecemos o `await` e o seu comportamento, assim como os benefícios de sua utilização. Acontece que também é possível utilizar o `await` em conjunto com uma stream e obter um comportamento semelhante.

Por exemplo, o comportamento natural ao criar uma inscrição para consumo de eventos de uma stream é que isso seja processado pelo *event loop*:

```
void main() {
    print('início main()');
    final stream = Stream.fromIterable(['4', '2']);
    stream.listen(( dado ) {
        print('Novo evento: $dado');
    });
    print('fim main()');
}

> início main()
> fim main()
> Novo evento: 4
> Novo evento: 2
```

O resultado impresso deixa bem explícita a ordem de execução das ações. O callback da inscrição é processado apenas após a finalização do método `main()`. Agora, utilizando o `await`:

```
Future<void> main() async {
    print('início main()');
    final stream = Stream.fromIterable(['4', '2']);
    await for(var dado in stream) {
        print('Novo evento: $dado');
    }
    print('fim main()');
}

> início main()
> Novo evento: 4
> Novo evento: 2
> fim main()
```

A sintaxe `await for` é utilizada em conjunto com uma stream se comportando exatamente como um `for` normal, onde cada elemento é um novo evento que sai da stream. A diferença é que, por usar `await`, obviamente a execução do método é pausada até que todos os eventos da stream sejam processados, o que altera a ordem de execução do código.

Eventualmente essa stream pode resultar em um erro, o qual é capturado com um simples try-catch :

```
void main() async {
  print('início main()');
  final stream = Stream.error(['Error']);
  try {
    await for(var dado in stream) {
      print('Novo evento: $dado');
    }
  } catch(e) {
    print('Erro capturado: $e');
  }
  print('fim main()');
}

> início main()
> Erro capturado: [Error]
> fim main()
```

Mas é importante notar que um evento de erro nesse caso suspende a execução do `for`, descartando todos os demais eventos que podem vir a ser processados posteriormente. Isso demonstra que o `await for` não é equivalente a uma inscrição feita através de uma `StreamSubscription`, e sim uma *syntax sugar* que pode ser útil em alguns casos. E existe um outro detalhe importante:

```
import 'dart:async';
void main() async {
  print('início main()');
  final controller = StreamController();
  controller.sink.add('42');
  await for(var dado in controller.stream) {
    print('Novo evento: $dado');
  }
  print('fim main()');
}

> início main()
> Novo evento: 42
```

Com base em seu conhecimento até agora, você consegue entender porque o `fim main()` não foi e nem será impresso no exemplo anterior? A resposta é simples e está relacionada com o `StreamController`. O `await for` sempre processará todos os eventos da stream e ele só sabe que todos os eventos foram processados quando essa mesma stream é encerrada, mas acontece que uma stream de um controller nunca é encerrada de forma automática.

Por conta disso, deve-se ter o cuidado de utilizar o `await for` apenas com streams que sabemos que possuem dados finitos, como a leitura de um arquivo de texto, por exemplo.

12.5 Operações em Streams

A classe `Stream` possui uma série de propriedades e métodos prontos para utilizarmos, que facilitam quando precisamos lidar com diferentes dados de uma stream. A maioria desses métodos são similares a funcionalidades existentes na classe `Iterable`, que permitem a manipulação, modificação ou busca de dados.

Como as streams retornam dados no decorrer do tempo, a maioria dessas operações retornam um `future`, que pode resultar em um erro caso o evento da stream também seja um erro.

Propriedades

A seguir, um exemplo de uso das propriedades existentes, seguido da explicação de cada uma:

```
void main() {
    final stream = Stream.fromIterable(['AA', 'AE',
                                         'AI', 'AO', 'AU']).asBroadcastStream();
    stream.first.then(print); // > AA
    stream.last.then(print); // > AU
    stream.isEmpty.then(print); // > false
    stream.length.then(print); // > 5
```

```

    print(stream.isBroadcast()); // > true
    stream.single.catchError((e) => e.toString()).then(print);
    // > Bad state: Too many elements
}

```

- **first**: completa com o primeiro evento emitido pela stream.
- **last**: completa com o último evento emitido pela stream.
- **isEmpty**: se a stream emitir algum evento, o future completa com `true`, mas, se ela encerrar sem emitir eventos, ele completa com `false`.
- **length**: completa com a quantidade de elementos emitidos assim que a stream for encerrada.
- **single**: completa com o único evento emitido pela stream. Mas completa com um erro caso esse evento seja um erro, a stream encerre vazia ou emita mais de um evento.
- **isBroadcast**: indica se a stream em questão é ou não do tipo *broadcast*.

Métodos de busca

Os métodos de busca são semelhantes às propriedades `first`, `last` e `single`, porém atrelados a uma condição booleana para validação.

```

void main() {
    final stream = Stream.fromIterable(['AA', 'AE',
                                         'AI', 'AO', 'AU']).asBroadcastStream();
    stream.elementAt(2).then(print); // > AI
    stream.firstWhere((valor) => valor.startsWith('A'),
                     orElse: () => 'Nenhum valor começa com A').then(print); // > AA
    stream.lastWhere((valor) => valor.startsWith('A'),
                     orElse: () => 'Nenhum valor termina com A').then(print); // > AU
    stream.singleWhere((valor) => valor.startsWith('A'))
        .then(print).catchError(print);
    // > Bad state: Too many elements
}

```

- **elementAt**: completa com um elemento emitido em um índice de acordo com a ordem dos elementos emitidos, iniciando de 0. Pode

completar com erro caso a stream emita um erro ou não emita um elemento com o índice passado.

- **firstWhere**: completa com o primeiro elemento que satisfaça a condição passada como `test`. Caso a stream encerre sem nenhum elemento para essa condição, o future é completado com um erro ou com o valor do parâmetro opcional `orElse()`.
- **lastWhere**: possui o mesmo comportamento de `firstWhere`, com a diferença de que o future completa apenas após a stream encerrar e com o último elemento que satisfaça a condição.
- **singleWhere**: possui o mesmo comportamento de `firstWhere` e `lastWhere`, com a diferença de que o future completa com erro caso nenhum ou mais de um elemento satisfaçam a condição.

Métodos de validação

Métodos que registram validações na stream, que serão executadas para cada valor. O resultado dessa validação é retornado em um future booleano.

```
void main() {
    final stream = Stream.fromIterable(['AA', 'AE',
                                         'AI', 'AO', 'AU']).asBroadcastStream();
    stream.any((valor) => valor.endsWith('E'))
        .then(print); // > true
    stream.every((valor) => valor.startsWith('A'))
        .then(print); // > true
    stream.contains('AO').then(print); // > true
}
```

- **any()**: completa com `true` caso algum elemento da stream satisfaça a condição passada em `test`. Caso contrário, completa com `false`.
- **every()**: completa com `true` caso todos os elementos da stream satisfaçam a condição passada em `test`. Caso contrário, completa com `false`.
- **contains()**: compara cada elemento da stream com o objeto passado utilizando `==`. Caso algum objeto seja igual ao future retornado, completa com `true`, caso contrário, com `false`.

Métodos de modificação

Todos os métodos que modificam o comportamento dos elementos emitidos por uma stream, na verdade, produzem uma nova stream que serve como um "filtro" para os dados da stream original. São inúmeras as possibilidades:

asBroadcastStream

Este método cria uma nova stream do tipo *broadcast* que produz os mesmos eventos da stream utilizada para chamá-lo. Ou seja, é criado um `StreamSubscription` para a primeira, o que significa que, se ela for do tipo inscrição única, nenhuma outra inscrição pode ser feita nela a partir da chamada deste método.

Os callbacks `onListen` e `onCancel` são opcionais. O primeiro é chamado assim que alguma inscrição é feita nessa nova stream retornada. Já o segundo é chamado quando todas as inscrições existentes foram canceladas.

```
import 'dart:async';
void main() {
    final controller = StreamController();
    final single = controller.stream;
    final broadcast = single.asBroadcastStream(onListen:
        (StreamSubscription s) {
            print('Inscrição realizada');
        }, onCancel: (StreamSubscription s) {
            print('Todas inscrições canceladas');
        });

    final subscription = broadcast.listen(null);
    subscription.onData((e) {
        print('Novo elemento em broadcast $e');
        subscription.cancel();
    });
    controller.sink.add('42');
}

> Inscrição realizada
> Novo elemento em broadcast 42
```

> Todas inscrições canceladas

Note que o evento foi adicionado à stream `single` e também foi transmitido para a stream `broadcast`, pois esta criou uma inscrição na primeira.

distinct

Cria uma nova stream que nunca produzirá dois eventos consecutivos iguais. Essa validação da igualdade é feita através do método `equals` passado por parâmetro, ou pelo `==` nos objetos.

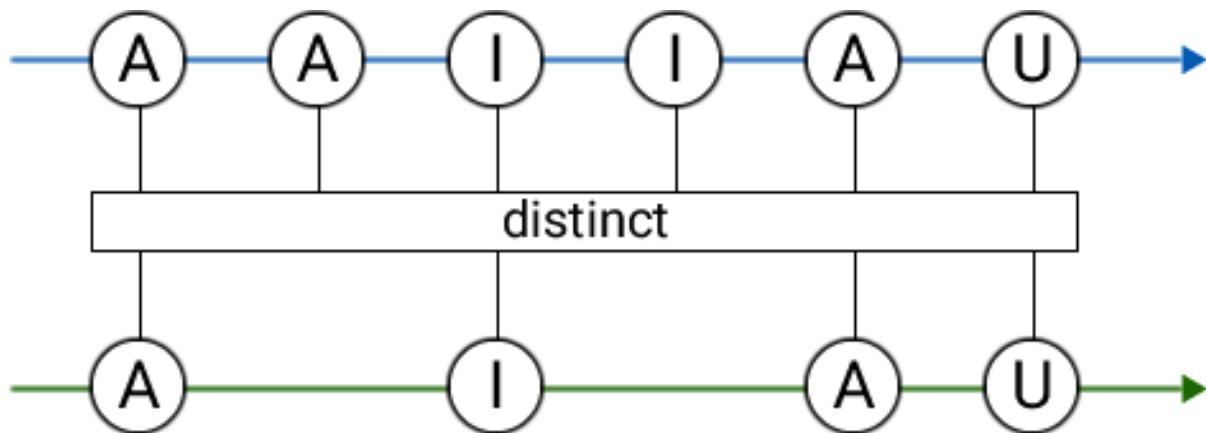


Figura 12.9: Modificador distinct.

```
void main() {
    final stream = Stream
        .fromIterable(['A', 'A', 'I', 'I', 'A', 'U']);
    final distinctStream = stream.distinct();
    distinctStream.listen(print);
}
```

```
> A
> I
> A
> U
```

map

Cria uma nova stream que transforma cada evento transmitido em um novo através da função informada.

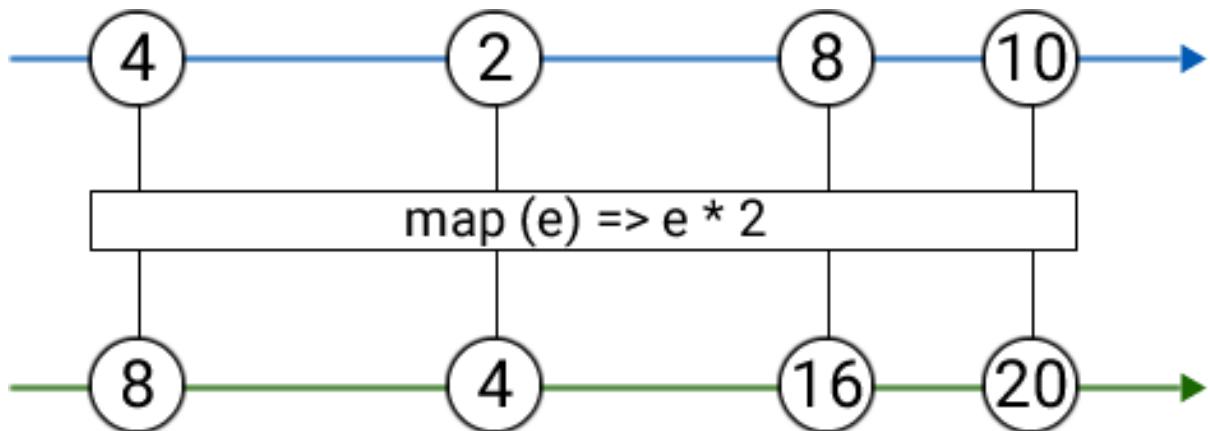


Figura 12.10: Modificador map.

```
void main() {
    final stream = Stream.fromIterable([4, 2]);
    final mapStream = stream.map((e) => e * 2);
    mapStream.listen(print);
}

> 8
> 4
```

expand

Similar ao `map`, com a diferença de que neste modificador cada evento pode ser transformado em N eventos na nova stream através da função passada por parâmetro.

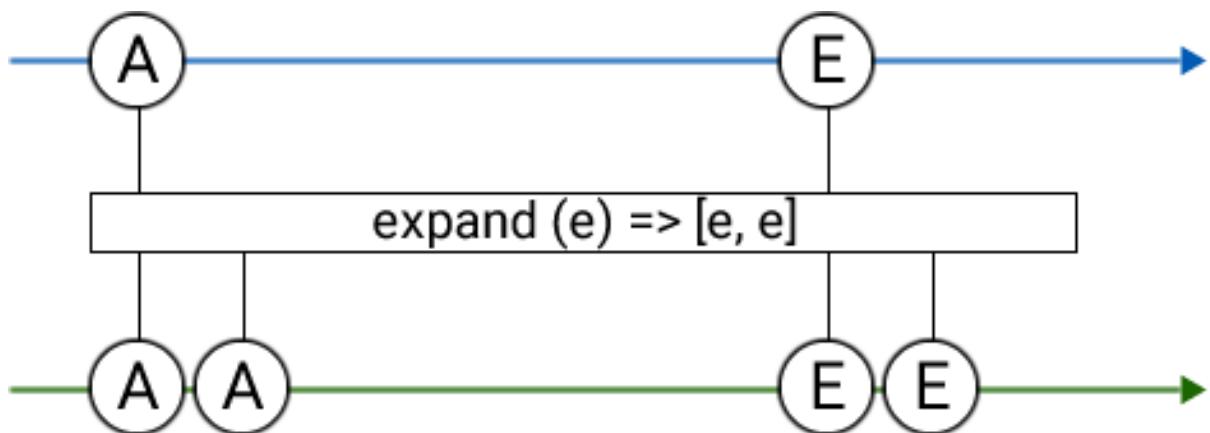


Figura 12.11: Modificador expand.

```

void main() {
    final stream = Stream.fromIterable(['A', 'E']);
    final expandStream = stream.expand((e) => [e, e]);
    expandStream.listen(print);
}

> A
> A
> E
> E

```

asyncMap

Possui o mesmo comportamento do `map`, com a diferença de que a função passada pode retornar um future. Sendo assim, cada evento será transmitido para a nova stream apenas quando esse future completar. No exemplo a seguir, cada evento leva dois segundos para ser transmitido em função do `Future.delayed`.

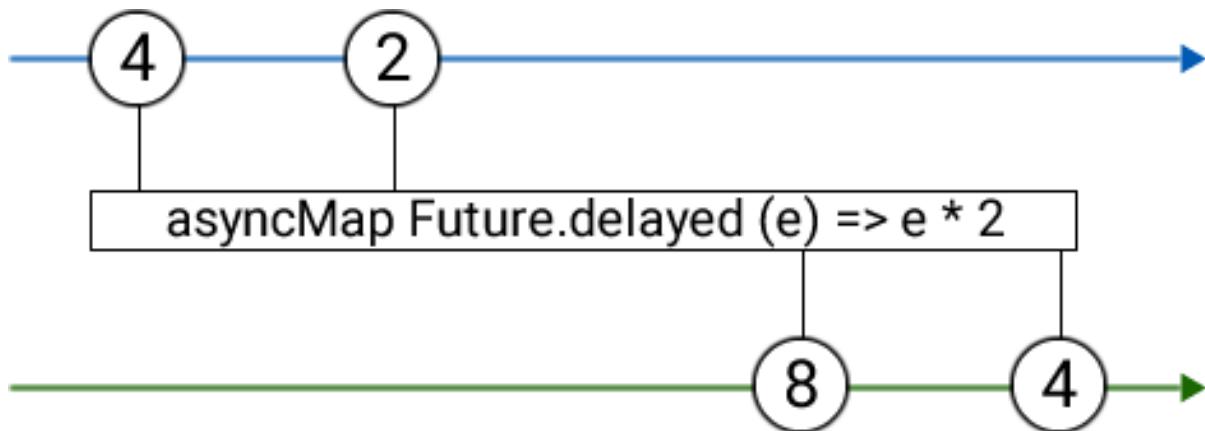


Figura 12.12: Modificador `asyncMap`.

```

void main() {
    final stream = Stream.fromIterable([4, 2]);
    final mapStream = stream.asyncMap((e) =>
        Future.delayed(Duration(seconds: 2), () => e * 2));
    mapStream.listen(print);
}

> 8
> 4

```

asyncExpand

Possui o mesmo comportamento do `expand`, a diferença é que na função passada por parâmetro, em vez de um `Iterable`, é retornada uma outra `Stream`. Cada elemento emitido pela stream de origem é repassado para a função, que pode transformá-lo e reemitir novos eventos. No exemplo a seguir, cada evento é duplicado:

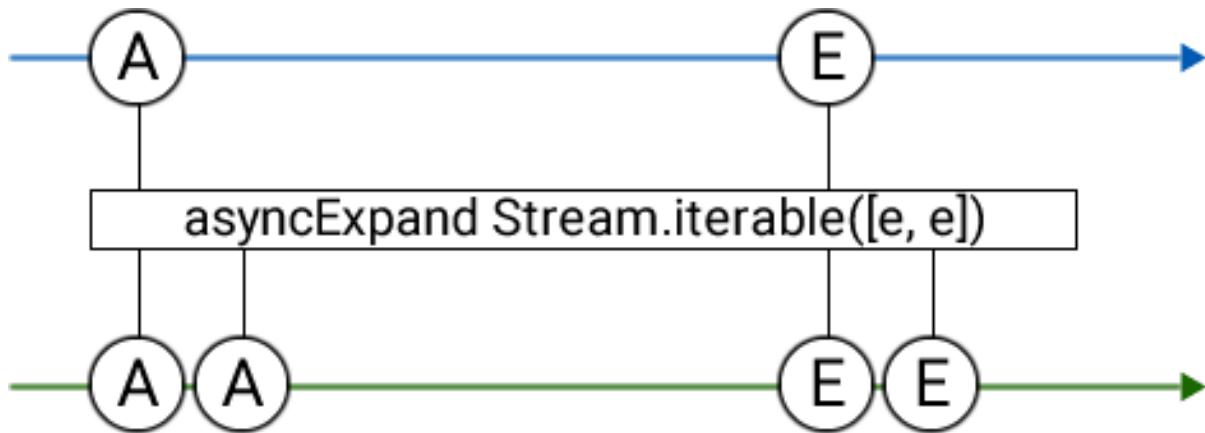


Figura 12.13: Modificador `asyncExpand`.

```
void main() {
    final stream = Stream.fromIterable(['A', 'E']);
    final expandStream = stream.asyncExpand(
        (e) => Stream.fromIterable([e, e]));
    expandStream.listen(print);
}

> A
> A
> E
> E
```

skip

Método que ignora os primeiros X eventos da stream de acordo com o valor informado por parâmetro. A partir desse valor, todos os eventos são emitidos normalmente.

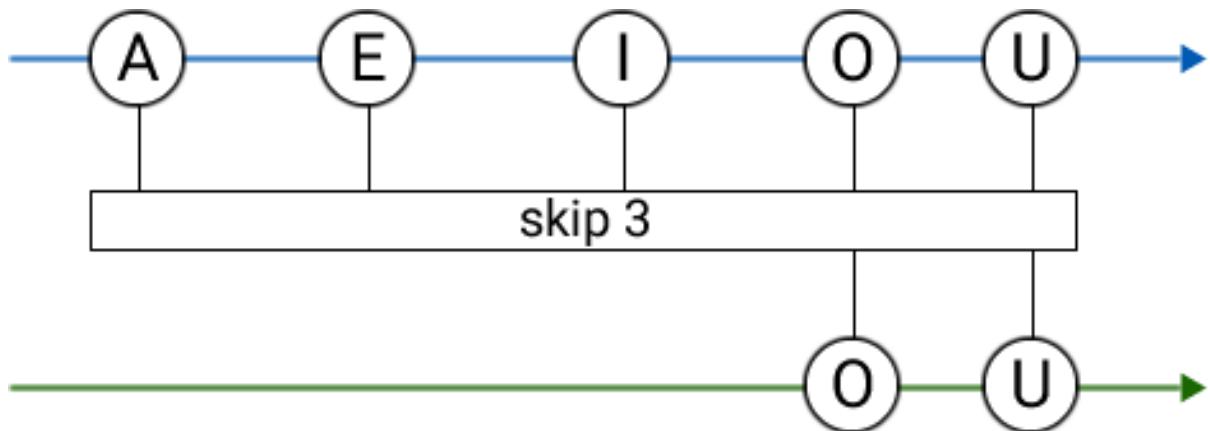


Figura 12.14: Modificador skip.

```
void main() {
    final stream = Stream.fromIterable(['A', 'E', 'I', 'O', 'U']);
    final skipStream = stream.skip(3);
    skipStream.listen(print);
}

> 0
> U
```

skipWhile

Similar ao `skip`, com a diferença de que os elementos são ignorados até que a função informada retorne `false` em algum elemento.

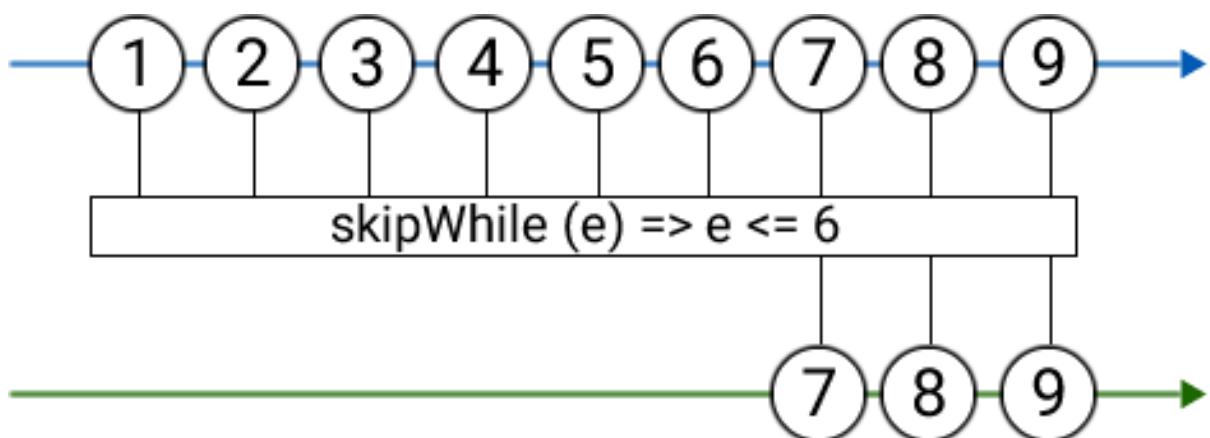


Figura 12.15: Modificador skipWhile.

```

void main() {
    final stream = Stream.fromIterable([1,2,3,4,5,6,7,8,9]);
    final skipStream = stream.skipWhile((e) => e <= 6);
    skipStream.listen(print);
}

> 7
> 8
> 9

```

take

Cria uma stream que emite os primeiros eventos até que a quantidade informada seja atingida.

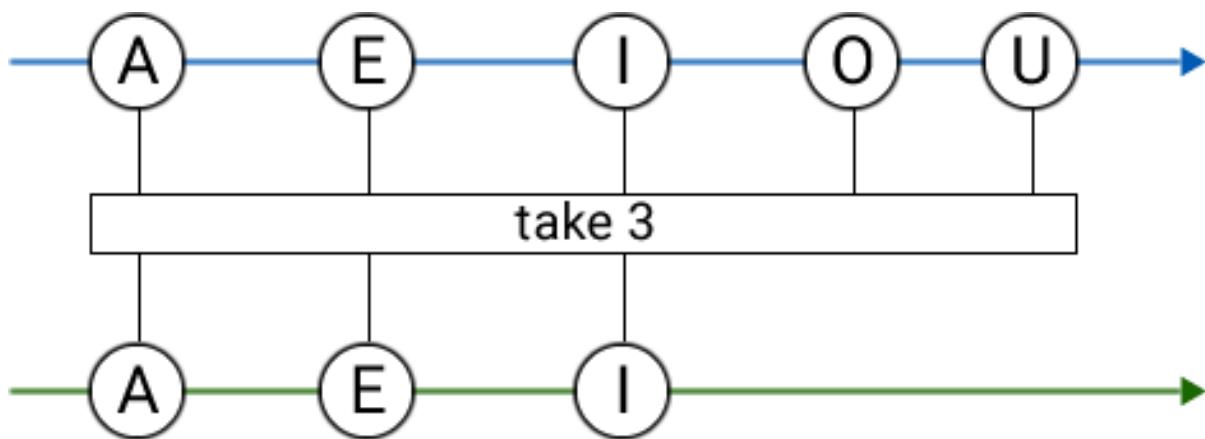


Figura 12.16: Modificador take.

```

void main() {
    final stream = Stream.fromIterable(['A', 'E', 'I', 'O', 'U']);
    final takeStream = stream.take(3);
    takeStream.listen(print);
}

> A
> E
> I

```

takeWhile

Similar ao `take`, com a diferença de que os elementos são emitidos até que a função informada retorne `false` em algum elemento.

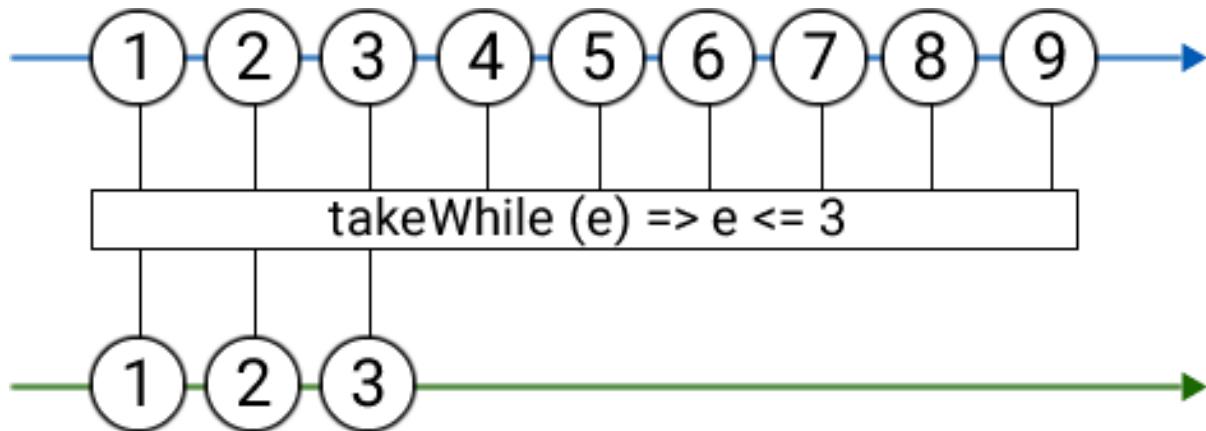


Figura 12.17: Modificador `takeWhile`.

```
void main() {  
    final stream = Stream.fromIterable([1,2,3,4,5,6,7,8,9]);  
    final takeStream = stream.takeWhile((e) => e <= 3);  
    takeStream.listen(print);  
}  
  
> 1  
> 2  
> 3
```

where

Cria uma stream que só emite os elementos que satisfazem a função informada retornando `true`.

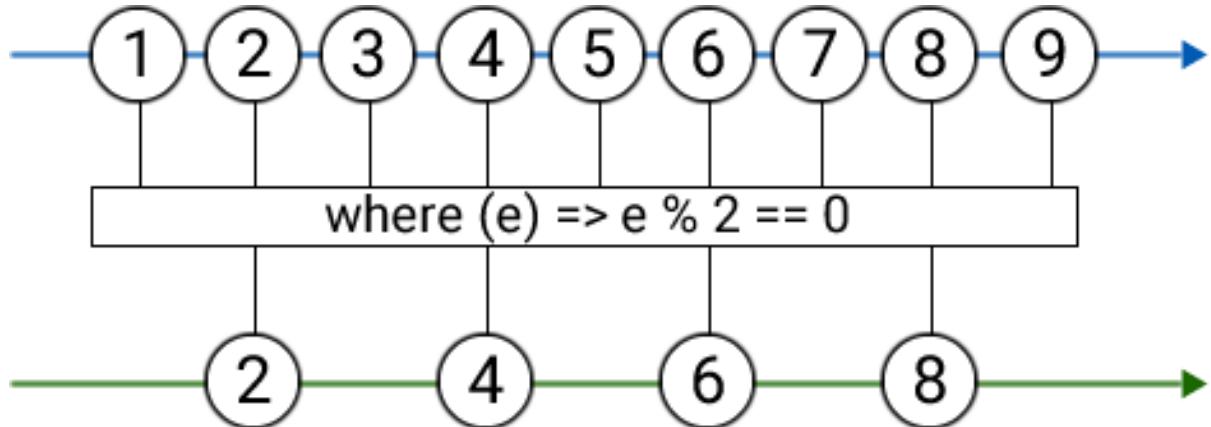


Figura 12.18: Modificador where.

```
void main() {
    final stream = Stream.fromIterable([1,2,3,4,5,6,7,8,9]);
    final whereStream = stream.where((e) => e % 2 == 0);
    whereStream.listen(print);
}

> 2
> 4
> 6
> 8
```

timeout

Retorna uma stream que emite os mesmos elementos da stream original, porém ela inicia um contador regressivo a cada evento emitido. Caso não seja emitido um novo elemento dentro do tempo especificado, o callback passado por parâmetro é executado.

Esse mesmo callback recebe por parâmetro um `EventSink` que permite adicionar elementos na stream. Caso o callback não seja especificado, a stream emitirá um `TimeoutException`. O contador reinicia a cada elemento emitido, até que a stream seja encerrada.

```
import 'dart:async';
void main() {
    final controller = StreamController();
    controller.sink.add(4);
```

```

    Timer(Duration(seconds: 3), () {
      controller.sink.add(2);
      controller.close();
    });

    final timeoutStream = controller.stream.timeout(
      Duration(seconds: 2),
      onTimeout: (sink) {
        sink.add('Timeout excedido');
      },
    );
    timeoutStream.listen(print);
  }

> 4
> Timeout excedido
> 2

```

Métodos utilitários

Alguns métodos não produzem uma nova stream, mas ainda assim são úteis em algumas situações.

toList e toSet

Retornam um future que completa assim que a stream é encerrada, agrupando todos os elementos em uma `List` ou em um `Set`. Se a stream emitir um erro, o future completa com esse mesmo erro.

```

void main() {
  final stream = Stream.fromIterable(['A', 'E', 'E', 'O', 'O'])
    .asBroadcastStream();
  stream.toList().then(print);
  stream.toSet().then(print);
}

> [A, E, E, O, O]
> {A, E, O}

```

drain

Cria um future que ignora todos os eventos emitidos e só completa quando a stream emite algum erro ou é encerrada. Caso seja encerrada, o future completa com o valor passado por parâmetro.

```
import 'dart:async';
void main() {
    final controller = StreamController();
    controller.sink.add(4);
    controller.sink.add(2);
    controller.close();
    controller.stream.drain('Stream encerrada').then(print);
}
> Stream encerrada
```

forEach

Permite executar alguma ação para cada evento emitido pela stream. Quando a stream é encerrada, o future retornado completa.

```
void main() {
    final stream = Stream.fromIterable([4, 2]);
    stream.forEach((e) => print('Elemento: $e'))
        .then((v) => print('Future completo'));
}
> Elemento: 4
> Elemento: 2
> Future completo
```

fold

Retorna um future que completa assim que a stream é encerrada, com um valor agrupado de todos os elementos emitidos, gerado através da função informada. Por parâmetro também é possível definir um valor inicial para combinar com o primeiro elemento da stream.

```
void main() {
    final streamUm = Stream.fromIterable(['A', 'E', 'I', 'O', 'U']);
    final streamDois = Stream.fromIterable([1, 2, 3, 4, 5, 6, 7, 8, 9]);
    streamUm.fold('Vogais->', (a, b) => '[${a}, ${b}]').then(print);
```

```
    streamDois.fold(10, (int a, int b) => a+b).then(print);  
}
```

```
> [[[Vogais->,A],E],I],0],U]  
> 55
```

reduce

Assim como o `fold`, este método retorna um future que vai completar com o resultado assim que todos os elementos da stream forem reduzidos a um único objeto. A diferença é que ele não possui um valor inicial passado.

```
void main() {  
    final streamUm = Stream.fromIterable(['A', 'E', 'I', 'O', 'U']);  
    final streamDois = Stream.fromIterable([1, 2, 3, 4, 5, 6, 7, 8, 9]);  
    streamUm.reduce((a,b) => '[$a,$b]').then(print);  
    streamDois.reduce((a,b) => a+b).then(print);  
}  
  
> [[[A,E],I],0],U]  
> 45
```

join

Retorna um future que completa com uma `String` assim que a stream é encerrada, unindo todos os elementos emitidos através da `String` passada como `separador`.

```
void main() {  
    final stream = Stream.fromIterable(['A', 'E', 'I', 'O', 'U']);  
    stream.join('-').then(print);  
}  
  
> A-E-I-O-U
```

cast

Gera uma stream que força todos os elementos emitidos a serem do tipo `R` passado. A tentativa de adicionar um objeto de outro tipo resulta em um

erro em tempo de execução.

```
import 'dart:async';
void main() {
    final controller = StreamController();
    controller.sink.add(42);
    final novaStream = controller.stream.cast<String>();
    novaStream.listen(print);
}

> Unhandled exception:
> type 'int' is not a subtype of type 'String' in type cast
```

handleError

Cria uma stream que envelopa e intercepta erros que ocorrem na stream original. A cada erro, o callback `onError()` é executado e eles podem ser filtrados através da função `test()`.

Pode ser útil ao utilizar em conjunto com o `await for`, pois captura o erro e evita que o `for` seja encerrado, diferente do tratamento de erro com `try-catch`.

```
import 'dart:async';
void main() async {
    final controller = StreamController();
    controller.sink.add('4');
    controller.sink.addError('Erro');
    controller.sink.add('2');
    final streamError = controller.stream.handleError((e) {
        print('Erro capturado $e');
    });
    await for (var dado in streamError) {
        print('Novo evento: $dado');
    }
}

> Novo evento: 4
> Erro capturado Erro
> Novo evento: 2
```

pipe

Adiciona todos os eventos da stream no `StreamConsumer` informado através de seu método `addStream`. Retorna um future que completa assim que todos os elementos são adicionados e a stream é encerrada.

```
import 'dart:async';
void main() async {
    final controllerUm = StreamController();
    controllerUm.sink.add('4');
    controllerUm.sink.add('2');
    controllerUm.close();

    final controllerDois = StreamController();
    controllerUm.stream.pipe(controllerDois).then((v) =>
        print('Eventos adicionados')
    );
    controllerDois.stream.listen(print);
}

> 4
> 2
> Eventos adicionados
```

Até aqui

Já deu para ver como uma stream é muito mais poderosa do que uma simples forma desacoplada de emitir dados. Olha essa quantidade de operações disponíveis! Obviamente ninguém precisa decorar o que cada um desses métodos de manipulação dos dados faz ou como funciona, para isso existem os livros e documentações, mas ter o conhecimento de que eles existem e conseguir combiná-los abre um leque gigante de possibilidades.

Isso é só a ponta do iceberg, afinal as streams são a base do paradigma de Programação Reativa. Inclusive, existe um projeto open source denominado ReactiveX (<https://reactivex.io>) que implementa esse paradigma entre diferentes linguagens de programação. Em Dart não é diferente, o package `rxdart` é a implementação oficial construída acima da API de streams, que

estende suas funcionalidades adicionando "superpoderes". Vale a pena conferir.

No próximo capítulo, ainda continuaremos nesse mesmo assunto, pois ainda há muito a ser explorado com as streams.

CAPÍTULO 13

Um pouco mais sobre streams

Agora que você já conhece o básico sobre as streams, chegou a hora de dar um passo a mais e conhecer outros aspectos da API. Por isso, durante este capítulo, vamos:

- Aprender as diferentes formas de transformar uma stream;
- Nos aprofundar nas diferenças entre inscrição única e broadcast;
- Desvendar os métodos geradores;
- Praticar criando um server HTTP.

13.1 Transformando uma Stream

Sabemos que uma stream consiste em uma "esteira" de eventos que entram de um lado e saem do outro, onde os interessados podem registrar *listeners* para capturá-los. E são muito comuns situações onde o mesmo evento que entra não é o que sai, pois no meio da esteira ele sofreu alguma transformação. Essas transformações podem ocorrer de várias formas e por diferentes motivos, como uma stream de leitura de arquivos, por exemplo, onde entram eventos em forma de bytes e saem strings com os textos.

No capítulo anterior conhecemos diversas formas de transformar os dados de uma stream através de implementações padrões da própria classe, como `map` , `expand` , `where` entre outros. Mas existe uma forma muito mais poderosa (e geralmente mais complicada) de transformar uma stream, através de um `StreamTransformer` .

A verdade é que todos esses métodos padrões podem inclusive ser implementados através de um *transformer*, embora não seja recomendado. Mas repare que essas mesmas transformações sempre ocorrem no mesmo padrão:

1. Uma função capture os dados que entram na stream;

2. Algum processamento é feito com esse dado (filtro, transformação, agrupamento etc.);
3. A saída é uma nova stream com os dados processados.

Por isso, um transformer nada mais é do que uma manipulação feita nos dados de uma stream de entrada que gera novos dados em uma stream de saída. A interface `StreamTransformer` define dois métodos para serem implementados, `bind()` e `cast()` :

```
abstract class StreamTransformer<S, T> {
    Stream<T> bind(Stream<S> stream);
    StreamTransformer<RS, RT> cast<RS, RT>();
}
```

O `cast()` funciona de forma similar à do próprio `cast()` de `stream`, garantindo em tempo de execução que todos os elementos que entram no transformer são do tipo `S` e saem como `T`. Já o `bind()` é o principal, pois é onde ocorre a transformação retornando uma nova stream de eventos computados através da stream de origem.

Existem algumas formas de criar um transformer e nenhuma pode ser considerada melhor que a outra. Vamos explorar as diferenças a seguir e, para exemplificar, precisaremos de uma transformação como exemplo. Imagine que existe uma stream da qual esperamos que os elementos emitidos sejam apenas letras do alfabeto, e precisamos criar um transformer customizado que:

- Permita definir um prefixo e um sufixo para cada elemento dessa stream;
- Emite um erro 'Elemento inválido' caso o elemento não seja uma letra do alfabeto.

Então, levando em consideração que seja definido o prefixo `[` e o sufixo `]` para os elementos:

```
['A', 'b', 'CC', 'D', '2', '@']
```

Esta seria a saída da nova stream:

```
['[A]', '[b]', 'Elemento inválido', '[D]', 'Elemento inválido',
'Elemento inválido']
```

Implementando a interface StreamTransformer

A primeira opção para criar um `StreamTransformer` é implementar a sua própria interface. Pode-se dizer que dentre as opções essa é a mais complexa, pois exige um pouco mais de código. Primeiro, veja o código da interface completo:

```
import 'dart:async';
class AlfabetoTransformer
  implements StreamTransformer<String, String> {
  AlfabetoTransformer({this.sufixo = '', this.prefixo = ''}) {
    _controller = StreamController(
      onListen: _onListen,
      onCancel: _onCancel,
      onPause: () => _subscription.pause,
      onResume: () => _subscription.resume);
  }

  AlfabetoTransformer.broadcast({this.sufixo = '',
                                this.prefixo = ''}) {
    _controller = StreamController
      .broadcast(onListen: _onListen, onCancel: _onCancel);
  }

  late StreamController<String> _controller;
  late StreamSubscription<String> _subscription;
  late Stream<String> _streamEntrada;
  String sufixo;
  String prefixo;

  void _onListen() {
    _subscription = _streamEntrada.listen(_onData,
      onError: _controller.addError, onDone: _controller.close);
  }

  void _onCancel() {
    _subscription.cancel();
```

```

    }

    void _onData(String dado) {
        if (dado.length == 1 && RegExp('[a-zA-Z]').hasMatch(dado)) {
            _controller.sink.add('$prefixo$dado$sufixo');
        } else {
            _controller.sink.addError('Elemento inválido');
        }
    }

    @override
    Stream<String> bind(Stream<String> stream) {
        _streamEntrada = stream;
        return _controller.stream;
    }

    @override
    StreamTransformer<RS, RT> cast<RS, RT>()
        => StreamTransformer.castFrom(this);
}

```

Pontos dessa implementação a serem observados:

- Esse transformer vai tratar apenas de dados do tipo `String` tanto na entrada quanto na saída. Por conta disso, ele não precisa ser genérico e todos os objetos já foram tipados como `String`.
- O objeto `_streamEntrada` mantém uma referência para a stream de entrada, e o `_subscription` mantém a referência para a inscrição que realizamos nessa mesma stream para consumir os seus dados.
- Já para a stream de saída, é utilizado um `StreamController` internamente para gerenciar sua criação.
- `sufixo` e `prefixo` são os parâmetros opcionais e específicos do nosso transformer.
- Existem dois construtores e em ambos é feito a inicialização do `_controller`. A diferença é que o construtor nomeado `AlfabetoTransformer.broadcast` produz uma stream do tipo `broadcast`.

- A implementação do método `cast` é obrigatória pois ele faz parte do contrato da interface. Geralmente, é utilizada a própria implementação existente em `StreamTransformer.castFrom`.
- Quando utilizamos o transformer em alguma stream através de seu método `stream.transform()`, o método `bind()` do transformer é chamado internamente e esse mesmo `bind` é a segunda implementação obrigatória da interface. Nele, recebemos a stream de entrada e retornamos a nova stream gerada pelo nosso controller.
- O callback `onListen` do controller é ideal para criarmos a inscrição a fim de obter os elementos da stream de entrada.
- No callback `_onData()` da inscrição é onde finalmente definimos quais são as regras específicas para o transformer. Para todo dado que entra na stream de entrada:
 - Se possuir apenas um caractere e for uma letra do alfabeto, adicionamos o dado na stream de saída com o prefixo e sufixo.
 - Caso contrário, adicionamos na stream de saída um erro com a frase `Elemento inválido`.

```
void main() {
    final stream = Stream
        .fromIterable(['A', 'b', 'CC', 'D', '2', '@']);
    final streamTransformada = stream
        .transform(AlfabetoTransformer(sufixo: ']', prefixo: '['));
    streamTransformada.listen(print, onError: print);
}

> [A]
> [b]
> Elemento inválido
> [D]
> Elemento inválido
> Elemento inválido
```

A partir de uma stream de entrada qualquer, pode-se então utilizar o seu método `transform()` para aplicar o `AlfabetoTransformer`. A stream retornada já irá conter as novas regras definidas, gerando o resultado de saída esperado.

Construtor padrão do StreamTransformer

A classe `StreamTransformer` possui um construtor padrão que permite definir de forma mais rápida e curta um transformer.

```
const factory StreamTransformer(StreamSubscription<T> onListen  
(Stream<S> stream, bool cancelOnError))
```

Basta fornecer por parâmetro um callback `onListen`, que é disparado sempre que a stream de saída possuir uma nova inscrição, podendo ser chamado várias vezes caso a stream seja do tipo *broadcast*. Essa função recebe por parâmetro a stream de entrada e um booleano indicando se a inscrição deve ou não cancelar em caso de erro.

```
import 'dart:async';  
StreamTransformer<String, String> criarTransformer({  
    String sufixo = '',  
    String prefixo = '',  
}) {  
    return StreamTransformer<String, String>(  
        (Stream<String> streamEntrada, bool cancelOnError) {  
            late StreamController<String> controller;  
            late StreamSubscription<String> subscription;  
            controller = StreamController<String>(  
                onListen: () {  
                    subscription = streamEntrada.listen(( dado ) {  
                        if ( dado.length == 1 && RegExp('[a-zA-Z]').hasMatch(dado) ) {  
                            controller.sink.add('$prefixo$dado$sufixo');  
                        } else {  
                            controller.sink.addError('Elemento inválido');  
                        }  
                    },  
                    onDone: controller.close,  
                    onError: controller.addError,  
                    cancelOnError: cancelOnError);  
                },  
                onPause: () => subscription.pause(),  
                onResume: () => subscription.resume(),  
                onCancel: () => subscription.cancel(),
```

```

    );
    return controller.stream.listen(null);
});
}

```

Dentro do `onListen` é então onde ocorre toda a mágica do transformer. A implementação é de certa forma parecida com a implementação da classe, com a ajuda de uma inscrição na stream de entrada e um controller para a stream de saída. Porém, note que o retorno da função não é a stream em si, mas sim uma inscrição gerada a partir dela.

```

void main() {
    final stream = Stream.fromIterable(['z', '0', '%', 'U']);
    final streamTransformada = stream
        .transform(criarTransformer(sufixo: '-', prefixo: '-'));
    streamTransformada.listen(print, onError: print);
}

> -z-
> Elemento inválido
> Elemento inválido
> -U-

```

Assim basta transforma a stream passando a função `criarTransformer()`.

Estendendo StreamTransformerBase

Uma terceira opção para implementar um transformer é estender a classe `StreamTransformerBase`, classe que contém a implementação padrão para todos os métodos, exceto o `bind`.

```

import 'dart:async';
class AlfabetoTransformer
    extends StreamTransformerBase<String, String> {
    AlfabetoTransformer({
        String sufixo = '',
        String prefixo = '',
    }) : _transformer = criarTransformer(sufixo, prefixo);

    final StreamTransformer<String, String> _transformer;

```

```

@Override
Stream<String> bind(Stream<String> stream) =>
    _transformer.bind(stream);

static StreamTransformer<String, String> criarTransformer(
    String sufixo, String prefixo) {
    return StreamTransformer<String, String>(
        (Stream<String> inputStream, bool cancelOnError) {
            late StreamController<String> controller;
            late StreamSubscription<String> subscription;

            controller = StreamController<String>(
                onListen: () {
                    subscription = inputStream.listen(
                        ( dado) {
                            if ( dado.length == 1 &&
                                RegExp('^[a-zA-Z]').hasMatch(dado)) {
                                controller.sink.add( '$prefixo$dado$sufixo' );
                            } else {
                                controller.sink.addError( 'Elemento inválido' );
                            }
                        },
                        onDone: controller.close,
                        onError: controller.addError,
                    );
                },
                onPause: () => subscription.pause(),
                onResume: () => subscription.resume(),
                onCancel: () => subscription.cancel(),
            );
            return controller.stream.listen(null);
        });
}
}

```

Nesse tipo de implementação, é comum utilizar internamente um `StreamTransformer` próprio, que é criado inclusive através do seu construtor padrão, exatamente a mesma implementação que utilizamos no

método anterior. Na função `bind`, basta chamar a própria função `bind` do transformer criado.

Essa abordagem é muito utilizada pelo package `rxdart` na criação dos seus transformers. Apenas relembrando, `rxdart` é uma biblioteca com a implementação de ReactiveX para Dart, uma API de programação assíncrona com streams seguindo o padrão *observable* e com os princípios de Programação Reativa.

```
void main() {
    final stream = Stream.fromIterable(['FF', 'f', '42', '-']);
    final streamTransformada = stream
        .transform(AlfabetoTransformer(sufixo: '/', prefixo: '/'));
    streamTransformada.listen(print, onError: print);
}

> Elemento inválido
> /f/
> Elemento inválido
> Elemento inválido
```

Construtor nomeado `fromHandlers`

A quarta opção e a mais utilizada por ser prática, é a criação do transformer através do construtor nomeado `StreamTransformer.fromHandlers`.

```
factory StreamTransformer.fromHandlers(
    void handleData(S data, EventSink<T> sink),
    void handleError(Object error, StackTrace stackTrace,
        EventSink<T> sink), void handleDone(EventSink<T> sink))
```

São fornecidos três callbacks para gerenciar os eventos emitidos pela stream de entrada, eventos normais, de erro e de finalização. Cada um deles recebe por parâmetro um `EventSink` que pode ser utilizado para adicionar os eventos transformados na stream de saída.

```
import 'dart:async';
StreamTransformer<String, String> criarTransformer({
    String sufixo = '',
    String prefixo = '',
```

```

}) {
    return StreamTransformer.fromHandlers(
        handleData: (String dado, EventSink<String> sink) {
            if (dado.length == 1 && RegExp('[a-zA-Z]').hasMatch(dado)) {
                sink.add('$prefixo$dado$sufixo');
            } else {
                sink.addError('Elemento inválido');
            }
        },
        handleError: (Object error, StackTrace stackTrace,
                      EventSink<String> sink) {
            sink.addError('Erro stream');
        },
        handleDone: (EventSink<String> sink) {
            sink.add('Stream finalizada');
        });
}

```

No `handleData`, executado cada vez que a stream de entrada emite um evento, definimos as regras do transformer utilizando o `sink` para produzir as saídas. Em caso de erro na stream de entrada ou da sua própria finalização, também será produzida uma resposta para a stream de saída.

```

void main() {
    StreamController<String>? controller;
    controller = StreamController<String>(onListen: () {
        controller!.sink.add('j');
        controller.sink.add('42');
        controller.sink.addError('Erro!');
        controller.close();
    });
    final streamTransformada = controller.stream
        .transform(criarTransformer(sufixo: '*', prefixo: '*'));
    streamTransformada.listen(print, onError: print);
}

> *j*
> Elemento inválido
> Erro stream
> Stream finalizada

```

A inscrição na stream de entrada dispara o `onListen()` do controller, emitindo os eventos que passam pela transformação e acabam saindo pela

stream transformada.

Construtor nomeado fromBind

Como todos os métodos que modificam uma stream produzem uma nova stream como saída, podemos facilmente encadear os dados que saem de uma stream para entrarem em outra, criando uma sequência de etapas por onde esses dados transitam:

```
stream.take(...).where(...).distinct(...);
```

Dentre essas etapas, podem existir também transformers customizados e, se analisarmos bem, podemos dizer que o próprio conjunto de todas essas etapas também é um transformer. E seria muito bom se fosse possível reunir essa sequência de manipulações em algo que possuísse um significado e pudesse ser reutilizado em vários locais. Então isso nos leva ao

`StreamTransformer.fromBind .`

```
factory StreamTransformer.fromBind(Stream<T> Function(Stream<S>) bind)
```

Esse construtor recebe apenas a função por parâmetro e, como a função `bind` recebe e retorna uma stream, ele acaba sendo muito útil para a criação de transformers que são uma combinação de outros transformers e operações. Considere que já possuímos um transformer que identifica e modifica as letras do alfabeto. Agora precisamos de outro que, além disso tudo, imprima as letras em forma minúscula. Com o `bind` é simples:

```
import 'dart:async';
final alfabetoMinusculoTransformer = StreamTransformer.fromBind(
  (Stream<String> stream) => stream
    .transform(AlfabetoTransformer(sufixo: '}', prefixo: '{'))
    .map(( dado ) => dado.toLowerCase()));

void main() {
  final stream = Stream.fromIterable(['F', 'f', '42', 'J']);
  final streamTransformada = stream
    .transform(alfabetoMinusculoTransformer);
```

```

    streamTransformada.listen(print, onError: print);
}

> {f}
> {f}
> Elemento inválido
> {j}

```

Assim concluímos todas as formas de se criar um transformer. Entenda que esse é um recurso muito poderoso e também muito comum no trabalho com streams, e eventualmente você vai se deparar com algum problema ou obstáculo que pode ser facilmente solucionado criando um transformer para os seus dados.

13.2 Inscrição única versus broadcast, um pouco além

Já sabemos de cor e salteado a principal diferença entre os tipos de streams, que é relacionada à quantidade de inscrições que ela permite. Mas existem algumas outras implicações que são pouco exploradas mas afetam diretamente o comportamento da stream, e influenciam na escolha correta de qual tipo utilizar.

	Inscrição única	Broadcast
Inscrições	1	Infinito
Ciclo de vida bem definido	Sim	Não
Facilidade de uso	Mais difícil	Mais fácil
Pode perder eventos	Não	Sim

Ciclo de vida

Conhecer o ciclo de vida de uma stream é extremamente importante, pois ajuda a descartar possíveis problemas com utilização de recursos desnecessários ou *memory leak*. As streams de inscrição única possuem um

ciclo de vida muito bem definido, começam a emitir eventos assim que uma inscrição é registrada e terminam assim que ela é cancelada ou a stream encerrada.

Já as *streams broadcast* não possuem um padrão de ciclo de vida bem definido. Elas devem em algum momento ser encerradas? Se sim, quando? Supondo que existam N inscrições ativas, elas devem ser encerradas quando todas forem canceladas, não permitindo mais inscrições? Ou devem ser encerradas quando um evento X ocorrer? Ou apenas quando forem encerradas explicitamente através do seu controller? Todas são perguntas específicas para a necessidade e dependem da implementação, por isso, o ciclo de vida pode divergir e devemos sempre prestar mais atenção ao utilizá-las.

Facilidade de uso

O próprio ciclo de vida possui um certo peso complementar ao julgarmos a facilidade de uso de uma stream. Mas, além dele, as streams de inscrição única possuem uma característica mais crítica que tende a dificultar a sua utilização no dia a dia.

Basicamente, todas as propriedades e operações que exploramos de uma stream registram internamente uma inscrição que é cancelada ao finalizar o método. O que significa que, se ela for de inscrição única, após isso ela se tornará completamente inútil e descartável.

```
void main() {
    final stream = Stream.fromIterable(['Stream', 'inscrição',
                                         'única']);
    stream.first.then(print); // > Stream
    //stream.listen(null); > Bad state: Stream has already been
    listened to.
}
```

Uma simples chamada para o `first` ou `isEmpty`, por exemplo, já inutiliza a stream impedindo novas inscrições, algo que não acontece se for `broadcast`, o que facilita sua utilização, pois esse tipo de preocupação não é necessário.

Perda de eventos

A partir do momento em que criamos uma stream já é possível adicionar eventos para serem processados, mas isso não significa que nesse momento já exista alguma inscrição registrada para consumi-los. Por conta disso, é característica das streams de inscrição única criarem um buffer de eventos internamente para armazenar os eventos recebidos até que haja uma inscrição.

```
import 'dart:async';
Stream<int> get streamNumeros {
    final controller = StreamController<int>();
    var i = 1;
    Timer.periodic(Duration(seconds: 1), (timer) {
        if(i == 10) timer.cancel();
        controller.sink.add(i++);
    });
    return controller.stream;
}

void main() {
    final stream = streamNumeros;
    Future.delayed(Duration(seconds: 5), () {
        stream.listen(print);
    });
}
```

No exemplo, um `Timer` adiciona a cada segundo um valor inteiro na stream até ser cancelado quando atingir o valor 10. Através do `Future.delayed`, é registrada uma inscrição na stream apenas após cinco segundos. Ao executar, o `controller` vai armazenar os eventos [1, 2, 3, 4, 5] em seu buffer interno. Com o future disparado, a inscrição receberá todos esses eventos que estavam no buffer respeitando a mesma ordem. A partir daí, a mesma inscrição receberá os próximos eventos a cada segundo até atingir o 10º.

Agora, o mesmo exemplo com uma stream do tipo broadcast
`StreamController<int>.broadcast()` :

```

void main() {
    final stream = streamNumeros;
    Future.delayed(Duration(seconds: 7), () {
        stream.listen((e) {print('Inscrição A $e');});
    });
    Future.delayed(Duration(seconds: 9), () {
        stream.listen((e) {print('Inscrição B $e');});
    });
}
> Inscrição A 8
> Inscrição A 9
> Inscrição A 10
> Inscrição B 10

```

Repare que a primeira inscrição é efetuada apenas após sete segundos de espera, recebendo em seguida os eventos [8, 9, 10], assim como a segunda inscrição efetuada após nove segundos recebe apenas o último evento [10]. Isso porque as streams do tipo broadcast não possuem buffer de eventos, então todas as inscrições novas recebem apenas os novos eventos a partir dela.

Todos os eventos adicionados na stream enquanto ela não possui nenhuma inscrição são simplesmente descartados, por isso, dizemos que em *streams broadcast* ocorre perda de eventos. Por outro lado, toda StreamSubscription cadastrada funciona de forma individual sem qualquer comunicação entre elas e são inteligentes o suficiente para manter um buffer próprio quando pausadas. Olhe esta outra situação:

```

void main() {
    final stream = streamNumeros;
    Future.delayed(Duration(seconds: 2), () {
        final inscricao = stream.listen(print);
        Future.delayed(Duration(seconds: 2), () {
            inscricao.pause();
            Future.delayed(Duration(seconds: 3), () {
                inscricao.resume();
            });
        });
    });
}

```

```
});  
}
```

Os primeiros dois eventos, [1, 2] , são ignorados conforme esperado, então a inscrição é efetuada e recebe [3, 4] . Nesse momento, o segundo future interno é disparado e pausa a inscrição. Mais três segundos se passam até que a inscrição é resumida, nesse meio tempo os eventos [5, 6, 7] foram emitidos normalmente e mantidos no buffer interno da inscrição. Ao resumir, a inscrição recebe todos os eventos do buffer e se comporta normalmente recebendo os próximos eventos [8, 9, 10] a cada segundo.

Obviamente, os exemplos usados são simplistas e a própria implementação dos buffers é otimizada. Mas nesse momento, é valido deixar claro que nenhuma forma de buffer deve ser explorada de forma extrema.

Faz parte das boas práticas não adicionar eventos na stream de inscrição única enquanto não houver uma inscrição. Para isso, pode ser utilizado o callback `onListen` do `StreamController` , que é disparado exatamente quando uma nova inscrição é registrada.

Da mesma forma, para as do tipo broadcast, é sempre interessante avaliar se é realmente necessário pausar alguma inscrição e, ao resumir, receber todos os eventos disparados enquanto a inscrição estava pausada. Na maioria das vezes, faz mais sentido simplesmente cancelar a inscrição e efetuar uma nova quando preciso. Todos os eventos de DOM (ações de mouse, botão etc.), por exemplo, são streams do tipo broadcast e devem seguir esse comportamento, afinal só faz sentido receber um clique do mouse no momento real em que ele é efetuado.

13.3 Geradores

Agora que dominamos a programação assíncrona em Dart, entendemos como funciona a execução da isolate, exploramos o comportamento dos futures e desbravamos a utilização de streams, estamos aptos a conhecer um recurso interessante da linguagem: os geradores. Eles permitem gerar uma

sequência de valores de forma *lazy*, ou seja, um por um, e possuem dois tipos, geradores síncronos e assíncronos.

sync*	Iterable<T>
async*	Stream<T>

Figura 13.1: Tipos de geradores.

Geradores síncronos

As funções demarcadas com o modificador `sync*` são conhecidas como geradores síncronos e retornam obrigatoriamente um `Iterable` de valores criados sob demanda.

Retorno sempre
será um
Iterable

Modificador sync* obrigatório

```
Iterable<T> geradorSincrono() sync* {  
    yield 42;  
    yield* [4, 2];  
}
```

Modificadores opcionais `yield` e `yield*`,
adicionam valores ao Iterable de retorno

Figura 13.2: Assinatura de uma função geradora síncrona.

```
import 'dart:io';  
Iterable<int> numeros() sync* {  
    print('Gerador iniciado');  
    for (int i = 0; i < 3; i++) {  
        sleep(Duration(seconds: 2));  
        yield i;  
    }  
    print('Gerador finalizado');  
}  
void main() {  
    print('Início main');
```

```

final iterable = numeros();
print('Começo iteração');

Iterator i = iterable.iterator;
while (i.moveNext())
    print('Número: ${i.current}');

print('Término main');
}

> Início main
> Começo iteração:
> Gerador iniciado
> Número: 0
> Número: 1
> Número: 2
> Gerador finalizado
> Término main

```

Dado o exemplo, seguem as observações importantes:

- A função `numeros()` gera um `Iterable` com os valores `[0, 1, 2]`;
- O modificador `yield` é utilizado para adicionar um valor ao `Iterable` de retorno. No momento em que ele é executado, o valor já é inserido e a origem já pode consumi-lo, porém, a função geradora também continua sua execução, por isso falamos que os valores são gerados de forma *lazy*. A cada valor gerado é executado um `sleep()` de dois segundos para facilitar esse entendimento;
- Repare na ordem em que os valores foram impressos. Mesmo após ter sido criada, a função só é executada de fato quando alguém consome o seu `Iterable` utilizando o `Iterator` para iterar sobre os valores. O mesmo aconteceria com um `for` normal que utiliza internamente o `Iterator`.
- Como o gerador é síncrono, o `main()` só continua a execução após todos os seus valores serem emitidos e iterados.

Existe ainda o modificador `yield*`, que permite adicionar os dados de um `Iterable` de entrada ao `Iterable` de saída da função.

```

Iterable<dynamic> letrasNumeros() sync* {
    for (int i = 1; i < 3; i++)
        yield i;
    yield* letras();
}

Iterable<String> letras() sync* {
    yield* ['A', 'B'];
}

void main() async {
    for(var i in letrasNumeros())
        print('Valor: ${i}');
}

```

> Valor: 1
> Valor: 2
> Valor: A
> Valor: B

É possível inclusive adicionar a saída de um gerador a outro, criando uma sequência de geradores.

Geradores assíncronos

Similar aos síncronos, as funções demarcadas com o modificador `async*` são conhecidas como geradores assíncronos e retornam obrigatoriamente uma `Stream` de valores criados sob demanda.

```

Stream<T> geradorAssincrono() async* {
    yield 42;
    yield* Stream.fromIterable([4, 2]);
}

```

Figura 13.3: Assinatura de uma função geradora assíncrona.

```

Stream<int> numeros() async* {
    print('Gerador iniciado');
}

```

```

    for (int i = 0; i < 3; i++) {
        await Future.delayed(Duration(seconds: 2));
        yield i;
    }
    print('Gerador finalizado');
}
void main() {
    print('Início main');
    final stream = numeros();
    stream.listen((i) => print('Número: $i'));
    print('Término main');
}
> Início main
> Término main
> Gerador iniciado
> Número: 0
> Número: 1
> Número: 2
> Gerador finalizado

```

Dado o exemplo, seguem as observações importantes:

- A função `numeros()` gera uma `Stream` com os eventos `[0, 1, 2]`;
- O modificador `yield` é utilizado para adicionar um evento na stream de retorno. O future aguardando dois segundos ajuda a entender que esse valor é adicionado de forma *lazy*, e pode ser consumido normalmente à medida que é adicionado.
- A ordem de execução é um pouco diferente, como é assíncrono o método `main()` finaliza primeiro, para só então a stream executar. Mas note que a função geradora também é sob demanda e só é executada quando alguma inscrição for registrada na stream. Altere para consumir a stream com um `await for`, e o `main()` aguardará a execução.

Também funciona em conjunto com o `yield*`:

```

Stream<dynamic> letrasNumeros() async* {
    yield* letras();
    for (int i = 1; i < 3; i++)

```

```

    yield i;
}
Stream<String> letras() async* {
    yield* Stream.fromIterable(['A', 'B']);
}
void main() async {
    await for(var i in letrasNumeros())
        print('Valor: ${i}');
}

> Valor: A
> Valor: B
> Valor: 1
> Valor: 2

```

13.4 Na prática - Server HTTP

Para finalizar o estudo de streams, vamos implementar uma nova funcionalidade no app ClimaTempo, desenvolvido no capítulo 11. A ideia é criar um novo comando que deixará a aplicação em alerta, aguardando avisos de possíveis desastres naturais que podem ocorrer na região, como chuvas de granizo ou vendavais.

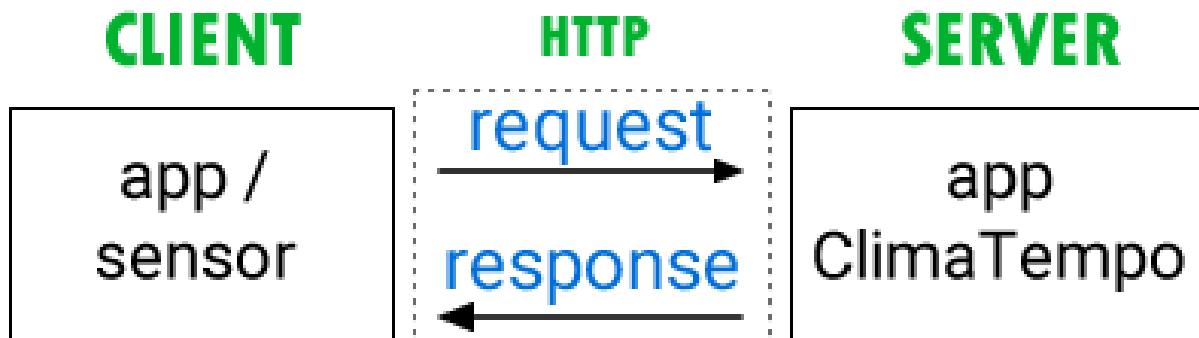


Figura 13.4: Comunicação Client/Server via HTTP.

Esses avisos são enviados de uma segunda aplicação que, nesse caso, poderia ser um sensor que identifica os riscos. Obviamente, esse é um exemplo lúdico criado justamente para exemplificar a criação e

disponibilização de um server em Dart. Na vida real, a comunicação entre aplicações desse gênero seria mais bem planejada e segura.

Server em Dart

O protocolo HTTP (*Hypertext Transfer Protocol*) é um padrão de comunicação para envio de dados entre aplicações na internet. Essa transferência de dados é feita entre um *client* e um *server*. Quando navegamos na internet, por exemplo, o *client* costuma ser um navegador, e o *server* é um servidor hospedado em alguma máquina acessível pela web.

No projeto ClimaTempo, já utilizamos esse protocolo para se comunicar com a API. Nesse caso, nossa aplicação era o *client* que gerava requisições para a API, que era o *server*. Essas requisições foram realizadas com a ajuda do package `http`, mas também é possível através de `dart:html` e do package que vamos utilizar agora, `dart:io`. E para não gerar confusões é importante deixar claro as diferenças entre as implementações de requisição *client*:

- **dart:io:** além de possuir todas as implementações de operações de input e output, criação de arquivos e *sockets*, contém as classes utilizadas para criar requisições *client* e disponibilização de *server* seguindo o protocolo HTTP. Pode-se dizer que esse contém as implementações de mais "baixo nível".
- **dart:html:** aplicações web que rodam no browser não possuem acesso ao package `dart:io`. Por isso, as requisições HTTP nessas aplicações são feitas através de classes existentes nesse package separado, criadas para funcionar exclusivamente na web.
- **http:** package que possui uma abstração de mais alto nível e funciona acima do `dart:io` e `dart:html`, unindo ambos. É considerado mais fácil de usar e por funcionar em todos os locais (CLI, web, server side e mobile) acaba sendo a maneira recomendada para trabalhar com requisições.

Para haver a comunicação, o server precisa estar disponível em um endereço fixo aguardando uma conexão. Esse endereço é composto de um

host (máquina conectada e acessível através de um IP) e porta, sendo que esse IP pode ser IPv4 ou IPv6.

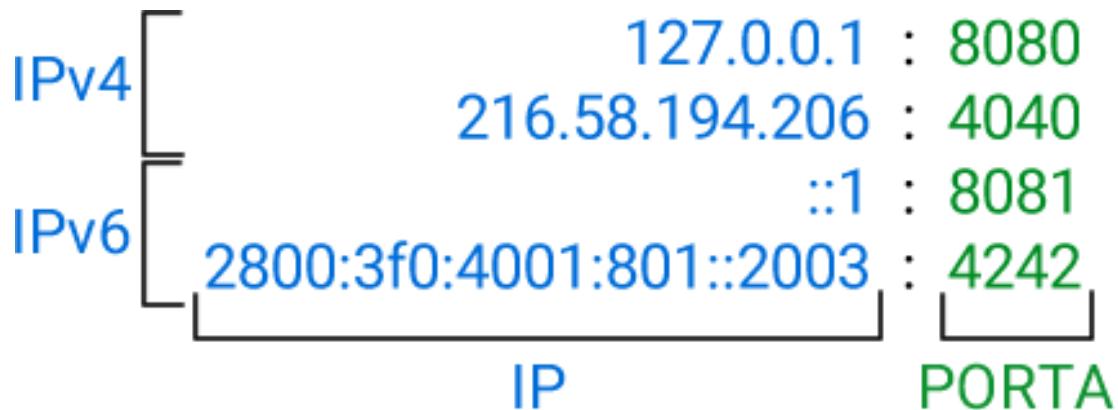


Figura 13.5: Endereço ip:porta.

Em Dart, um server pode ser criado e exposto facilmente com `dart:io`:

```
import 'dart:io';
void main() async {
  var server = await HttpServer.bind('127.0.0.1', 8080);
  await for (HttpRequest request in server) {
    request.response.write('Primeiro server em Dart!');
    await request.response.close();
  }
}
```

Experimente acessar um browser e digitar a URL `127.0.0.1:8080`. Assim que você apertar *enter*, o request será feito e a resposta com o texto "Primeiro server em Dart!" será impressa na tela. Simples assim!

O método `HttpServer.bind()` cria um server atrelado a um host e uma porta que ficará escutando nesse endereço. Por conta do suporte natural de Dart à programação assíncrona, ele já estará apto a receber N requisições simultaneamente. Para cada request recebido, uma mensagem é escrita na `response` e ela é encerrada, e, nesse momento, a resposta retorna para a origem, no caso o nosso browser.

Repare que o funcionamento de um server está diretamente atrelado ao funcionamento de streams, e na prática um `HttpServer` é uma

`Stream<HttpRequest>` , que por sua vez é uma `Stream<Uint8List>` :

```
abstract class HttpServer implements Stream<HttpRequest> {}
abstract class HttpRequest implements Stream<Uint8List> {}
```

Por isso, conseguimos utilizar o `await for` e ficar escutando os requests que chegam para a stream, da mesma forma que cada request é uma stream por meio da qual podemos escutar os dados que chegam com o request. Vamos desvendar melhor isso tudo implementando a nova funcionalidade no app ClimaTempo.

Funcionalidade de alertas

Neste momento é importante você ter implementado o projeto ClimaTempo, pois daremos continuidade ao mesmo código criado. Se houver qualquer problema, acesse o repositório do livro no GitHub para ter acesso às implementações organizadas e separadas por capítulo.

No mesmo projeto, crie um novo arquivo no caminho `lib/alerta_server.dart` com a classe `AlertaServer` :

```
import 'dart:convert';
import 'dart:io';
class AlertaServer {
  Stream<String> start() async* {}
```

O método `start()` vai ser o responsável por inicializar o server e, como ele é gerador, para cada request que chega vai ser reemitido o aviso em uma stream de saída.

```
Stream<String> start() async* {
  var server = await HttpServer.bind(
    InternetAddress.loopbackIPv4,
    8080,
  );
  await for (HttpRequest request in server) {}
```

A classe `InternetAddress` representa um endereço de um host na internet. Com a constante `loopbackIPv4`, obtemos o endereço de *loopback* no padrão IPv4, que é o próprio `127.0.0.1`. Assim um server é inicializado no endereço local `127.0.0.1:8080` e já está apto a receber requests.

```
Stream<String> start() async* {
    // código omitido
    await for (HttpRequest request in server) {
        final contentType = request.headers.contentType != null
            ? request.headers.contentType!.mimeType : '';
        final response = request.response;
    }
}
```

Um objeto `HttpRequest` contém todas as informações necessárias sobre o request feito. Dentre elas, as mais importantes e úteis são:

- **request.method**: o método HTTP utilizado, GET, PUT, POST etc.
- **request.headers**: objeto `HttpHeaders` que contém todas as informações passadas no header da requisição, como o `content-type`, que indica o tipo de dado enviado, ou o `authorization` com um token de autorização.
- **request.uri**: objeto `Uri` que contém o caminho solicitado, útil para identificar diferentes endpoints de uma API, por exemplo.
- **request.response**: objeto `HttpResponse` que contém as informações de resposta do request, que será retornado para o *client*.

```
Stream<String> start() async* {
    // código omitido
    await for (HttpRequest request in server) {
        // código omitido
        if (request.method == 'POST'
            && contentType == 'application/json') {
            final requestMap = await utf8.decoder.bind(request).join();
            final String? aviso = jsonDecode(requestMap)['aviso'];
            if (aviso != null) {
                response..statusCode = HttpStatus.ok
                    ..write('Informação recebida!');
            }
            yield aviso;
        }
    }
}
```

```

    } else {
        response..statusCode = HttpStatus.badRequest
            ..write('Formato de request errado..');
    }
} else {
    response..statusCode = HttpStatus.methodNotAllowed
        ..write('É aceito apenas request POST');
}
await request.response.close();
}
}

```

O nosso server de avisos vai aguardar uma requisição do tipo `POST` com um JSON no padrão:

```
{"aviso": "Risco de granizo nas próximas horas"}
```

Caso o método e o tipo de dado não sejam esses, a resposta é preenchida com o `statusCode 405`, que significa método não permitido, e com o método `write()` é escrita uma mensagem de retorno. A classe `HttpStatus` contém as constantes com os códigos padrões de retorno.

Caso a requisição possua os requisitos necessários, precisamos obter o aviso, que é a informação principal. Essa informação vem em JSON no corpo da requisição e, para recuperá-la, é preciso estar atento a alguns detalhes.

O `HttpRequest` é uma stream de `uint8List`. Então os dados passados no corpo da requisição são serializados e inseridos nessa stream como `Uint8List`, que é basicamente uma lista de inteiros 8-bit. Faz-se necessário voltar essa informação para o seu formato original, que é uma `String`. Conversão de tipos é algo que se encaixa perfeitamente no conceito de um `Converter`.

O package `dart:convert` é então a solução, pois ele possui o `codec utf8`. Como já vimos, um `codec` é composto de dois `converters`, um `encoder` e um outro `decoder`. O `decoder` do `utf8` é responsável por transformar `List<int>` em `String`, exatamente o que precisamos.

Um outro detalhe importante é que um `Converter` também é um `StreamTransformer`, pois ele estende de `streamTransformerBase`, sendo possível utilizar o seu método `bind()` que recebe a stream e retorna os dados transformados. Usando o `join()` nessa nova stream, obtemos a `String` com o JSON original.

Finalmente, com o decoder de JSON, conseguimos acessar os dados passados para a requisição. Com isso, já conseguimos validar a existência do parâmetro `aviso` e, se ele não estiver presente, já retornamos um *status code* 400 (*bad request*).

Caso esteja presente, ele é emitido para a stream de saída do gerador com o `yield`. É necessário então avisar o *client* de que tudo deu certo, preenchendo o `response` com o *status code* 200 e uma mensagem de retorno. Nesse simples método, reunimos na prática todos os conhecimentos adquiridos até aqui sobre programação assíncrona, streams e converters.

Para finalizar os ajustes, falta definir o comando que vai startar o server. Então volte no arquivo `climatempo.dart` e adicione ao final da função `main()`:

```
import 'package:climatempo/alerta_server.dart';
void main(List<String> args) async {
    final parser = criarParser();
    final argsResult = parser.parse(args);
    // código omitido
    if (comando != null && comando.name == 'alerta') {
        AlertaServer().start().listen(print);
    }
    // código omitido
}
```

Não se esqueça também de adicionar o novo comando ao método `criarParser()` com `..addCommand('alerta')`. Agora basta executar com `dart run bin/climatempo.dart alerta` ou `climatempo alerta` caso você gere o binário.

Criando o Client

Utilize o comando `dart create`, já aprendido, para criar uma aplicação CLI que simulará um sensor que detecta os avisos e os dispara para o app do ClimaTempo. No arquivo principal, adicione:

```
import 'dart:convert';
import 'dart:io';
void main(List<String> args) async {
  var aviso = args.isNotEmpty ? args.first : '-';
  final jsonMap = {'aviso': aviso};
}
```

Essa aplicação vai ser muito simples. Pegamos o primeiro argumento passado para a função, que será o aviso a ser enviado e, com ele, construímos um map que posteriormente será transformado em JSON.

```
void main(List<String> args) async {
  // código emitido
  HttpClientRequest request = await HttpClient()
    .post(InternetAddress.loopbackIPv4.host, 8080, '-')
    ..headers.contentType = ContentType.json
    ..write(jsonEncode(jsonMap));
}
```

A classe `HttpClient` dessa vez é que permite criar os requests com seus métodos `get()`, `put()`, `post()` e assim por diante, de acordo com o tipo de requisição. O próprio método `post()` recebe o host, a porta e o caminho, todos ajustados de acordo com o endereço do server.

O retorno desse método é um `Future<HttpClientRequest>`, que representa de fato um request e permite o preenchimento de todas as informações necessárias, como o próprio `contentType` e o JSON transformado através de `jsonEncode()`. Para finalizar a função, insira:

```
void main(List<String> args) async {
  // código emitido
  HttpClientResponse response = await request.close();
  response.transform(utf8.decoder).forEach(print);
}
```

Ao encerrar o request com `request.close()`, ele é disparado e enviado para o server. O retorno desse método é um future que completará com um `HttpClientResponse`, justamente a resposta de retorno gerada pelo server.

Só que `HttpClientResponse` é também uma `Stream<List<int>>` que precisa ser decodificada. E como o próprio `utf8.decoder` é um transformer, ele pode ser utilizado em conjunto com o método `transform()` e o resultado dessa nova stream é impresso. Com isso, o `client` está finalizado e pronto para ser testado. Certifique-se de que o server do ClimaTempo está rodando e execute o `client`:

```
dart run bin/main.dart "Risco de granizo nas próximas horas"
```

O server deve imprimir no console o aviso `Risco de granizo nas próximas horas` da mesma forma que o `client` deve imprimir a resposta `Informação recebida!`. Pronto, as duas aplicações em Dart estão se comunicando perfeitamente!

13.5 Se liga aí

- Entenda e utilize os callbacks do `StreamController` com sabedoria. Comece a inserir dados na stream apenas após a chamada do `onListen` indicando que há inscrição, pause e resuma a fonte de dados (*stream*) de acordo com o `onPause` e `onResume`. O `onCancel` também é um bom candidato para parar a fonte de dados e liberar algum possível recurso. Evitando o uso excessivo do buffer do controller, você estará reduzindo o consumo de memória e prevenindo possíveis *memory leaks*.
- Evite pausar inscrições do tipo broadcast, pois os eventos mantidos em buffer enquanto pausada não são interessantes. Nesse caso, é melhor liberar recursos, cancelar a inscrição e criar uma nova quando necessário.
- Não crie transformers desnecessariamente se os próprios métodos da stream são o suficiente para obter o resultado necessário, mesmo que utilizados em conjunto.

13.6 É com você

1 - Pesquise sobre RxDart, uma lib com a implementação de RX (Reactive Extensions) para Dart. Ela adiciona uma série de capacidades extras para as streams e é construída com base no padrão *observer* em conjunto com a programação reativa.

2 - Crie uma stream que emita todos os valores ímpares de 0 a 1.000. Utilize um gerador para isso.

3 - Também com um gerador, crie um programa que receba um valor numérico e imprima todos os divisores dele.

4 - Uma utilização também bastante comum para um transformer é a validação de dados, impedindo ou permitindo que um determinado dado passe pela stream. Crie um transformer para validação de e-mails e outro para telefones.

5 - Pesquise sobre outras classes utilitárias de streams, como `StreamIterator` da lib `dart:async`, ou classes, como `StreamQueue`, `StreamGroup`, `StreamSplitter` e `StreamZip` do package utilitário `package:async`.

6 - Além das opções apresentadas neste capítulo para a criação de server e requisições *client*, ainda existe o package `shelf` com utilitários para criação de servers e o package `dio`, muito utilizado para requisições com Flutter. Pesquise sobre eles.

Até aqui

Agora você está apto a trabalhar com maestria utilizando as streams, identificando suas peculiaridades e customizando os seus dados facilmente com a ajuda de diferentes transformers. Nossa exemplo com o server mostrou um pouco de como elas podem ser utilizadas na prática em algo comum no dia a dia através das requisições HTTP. Mas existem inúmeros outros cenários, e, mais do que nunca, você também começará a perceber como várias outras APIs são criadas com base nas streams.

Para continuar nossa jornada, no capítulo seguinte voltaremos a nos aprofundar nas isolates e conhecer um novo conceito também do mundo assíncrono em Dart: as *zones*.

CAPÍTULO 14

Um pouco mais sobre Isolates e Zones

Neste ponto do livro já estamos cientes de que Dart é uma linguagem *single-thread*, possui uma *isolate* principal que é responsável pela execução do programa, e através do seu *event loop* é possível executar e gerenciar tarefas que devem rodar de forma assíncrona. Por isso, APIs como a de *future* e *stream* possuem um nível mais alto de abstração e facilitam o trabalho assíncrono, rodando acima do *event loop* e adicionando eventos em suas *queues* de *microtask* e *event*.

Mas ainda está faltando entender melhor duas peças desse grande quebra-cabeça assíncrono de Dart: a primeira, já mencionada anteriormente, são as isolates, enquanto a segunda são as extensões dinâmicas de chamadas assíncronas, ou simplesmente *zones*. Então neste capítulo veremos:

- Como criar e gerenciar uma nova isolate;
- As diferenças de comunicação unidirecional e bidirecional entre isolates;
- A importância das zones e seu funcionamento;
- Como sobrescrever algumas funcionalidades utilizando zones.

14.1 Criando uma nova Isolate

Todas essas facilidades que vimos para se programar de forma assíncrona são excelentes, pois deixam os programas mais responsivos a interações e garantem uma melhor experiência de usuário. Mas, executar tarefas assíncronas não é o mesmo que executar tarefas em paralelo, em processos diferentes e aproveitando processadores *multi core*. Tarefas que envolvem um processamento mais pesado como tratamento de imagem ou leitura e escrita de arquivos, por exemplo, mesmo que sendo assíncronas, podem resultar em um gargalo de execução para a *thread(isolate)* principal.

Como todo código em Dart roda por padrão em uma única isolate, é nesse momento que entra o conceito de múltiplas isolates para separação dessas tarefas. As isolates podem ser acessadas e gerenciadas através do package `dart:isolate`, como ilustrado a seguir:

```
import 'dart:isolate';
void main() async {
  Isolate mainIsolate = Isolate.current;
  print('Executando na isolate: ${mainIsolate.debugName}');
  mainIsolate.kill(priority: Isolate.immediate);
  print('fim do main()');
}

> Executando na isolate: {main}
> isolate terminated by Isolate.kill
```

A propriedade `Isolate.current` retorna uma referência para a isolate em que o código atual está sendo executado. Um objeto do tipo `Isolate` é quem permite referenciar e manipular uma isolate, conforme o exemplo. Nele, o atributo `debugName` imprime `main`. Este é o apelido utilizado para a identificação, comprovando que estamos executando na isolate principal.

Com o método `kill` é possível encerrar por completo o processo de execução de uma isolate, onde `Isolate.immediate` (ou o valor 0) significa o mais rápido possível e `Isolate.beforeNextEvent` (ou 1) agenda o encerramento para o próximo ciclo do *event loop*. E novamente, por estar rodando na isolate principal, o programa é encerrado sem imprimir `fim` do `main()`.

Uma nova isolate pode ser criada a partir do seu método `spawn`:

```
external static Future<Isolate> spawn<T>(
  void entryPoint(T message), T message, {bool paused = false,
  bool errorsAreFatal = true, SendPort? onExit, SendPort? onError,
  String? debugName});
```

Esse método estático possui dois parâmetros que são obrigatórios. O primeiro, `entryPoint(T message)`, é uma função qualquer que precisa ser de alto nível e receber um parâmetro também obrigatório, essa é a função

que será executada assim que a nova isolate iniciar. O segundo, `T message`, é um objeto qualquer que funciona como uma mensagem, ele é quem será passado por parâmetro para a isolate na função `entryPoint`, por isso, ambos devem ser do mesmo tipo `T`. Os demais parâmetros opcionais são:

- **bool paused**: por padrão é `false` e define se a `isolate` vai iniciar pausada; caso `true`, para ela executar é necessário ser resumida através de `isolate.resume()`.
- **bool errorsAreFatal**: determina se a `isolate` deve ser encerrada caso ocorra algum erro não capturado dentro dela.
- **SendPort onExit**: listener que será executado assim que a isolate encerrar.
- **SendPort onError**: listener que será executado assim que ocorrer um erro na `isolate`.
- **String debugName**: nome de identificação da isolate, utilizado para facilitar operações de `debug`.

Vamos ver essa criação na prática:

```
import 'dart:isolate';
void main() async {
  Isolate mainIsolate = Isolate.current;
  print('Executando na isolate: ${mainIsolate.debugName}');
  await Isolate.spawn(funcaoEntrada, 'Olá nova Isolate.',
                      debugName: 'novaIsolate');
}
void funcaoEntrada(String parametro) {
  final isolate = Isolate.current;
  print('Executando na isolate: ${isolate.debugName}');
  print('Parâmetro: $parametro');
}

> Executando na isolate: {main}
> Executando na isolate: {novaIsolate}
> Parâmetro: Olá nova Isolate.
```

Com a ajuda de `Isolate.current` é possível ver claramente que `funcaoEntrada()` roda em uma isolate separada identificada por `novaIsolate`. E entender essa separação é crucial.

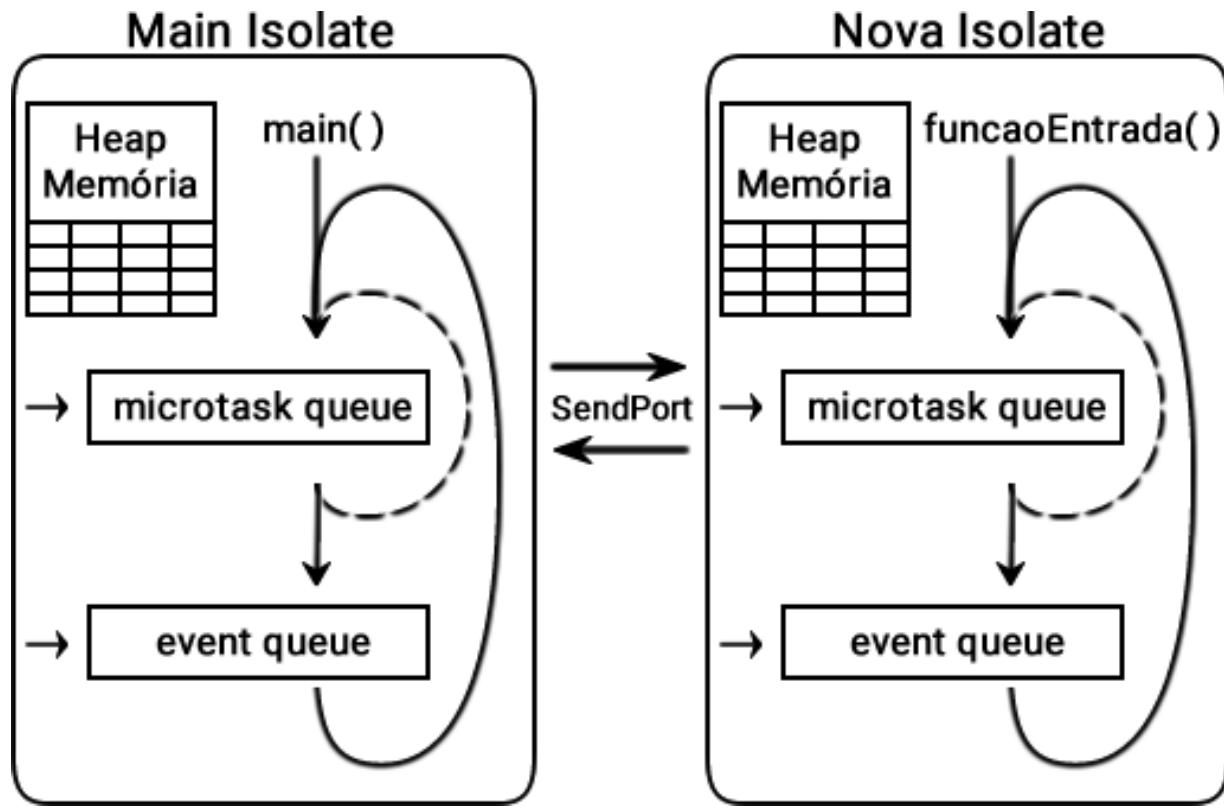


Figura 14.1: Isolates separadas.

Cada isolate funciona de forma independente com o seu próprio *event loop*, *queues*, e *heap* de memória, nada é compartilhado. A única forma de comunicação entre elas é através do parâmetro de mensagem definido ao criá-la, por isso, esse objeto é importante e costuma ser do tipo `SendPort`. Assim, pode-se dizer que essa comunicação é feita através de portas.

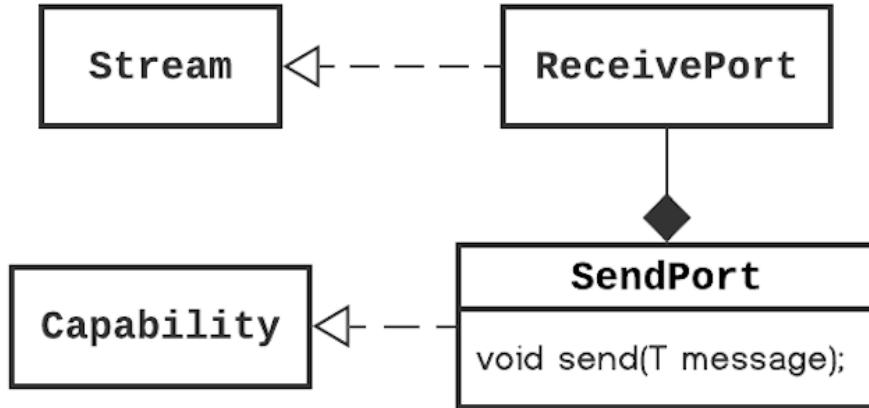


Figura 14.2: Organograma ReceivePort.

O objeto `ReceivePort` é um receptor de mensagens, por isso implementa `Stream`, afinal através de uma inscrição essas mensagens podem ser facilmente capturadas. Todo `ReceivePort` possui um objeto `SendPort`, que é por onde essas mensagens serão enviadas.

Um `SendPort` também implementa `Capability`. Um objeto `capability` é infalsificável, significando que ele pode ser transmitido entre diferentes isolates e sempre manterá a sua equalidade. É impossível criar um outro objeto que seja igual a um objeto `capability`.

14.2 Comunicação unidirecional

A forma mais simples de comunicação entre isolates é do tipo unidirecional, onde apenas um dos lados envia a mensagem e o outro recebe. Nesse caso, se uma **isolate A** inicia o processo de uma **isolate B**, a **isolate A** sempre será a receptora de informações, assim como a **isolate B** será a emissora. Essa isolate criada a partir da isolate principal também é comumente chamada de **worker**.

Um dos principais motivos de se criar uma nova isolate **worker** é apenas para processar alguma tarefa demorada e devolver uma resposta depois de um tempo, como a leitura de um arquivo grande por exemplo. E como é uma comunicação unidirecional e única (apenas uma mensagem) pode ser

utilizado o próprio método `exit()` da API para retornar uma resposta assim que a isolate encerrar o processamento.

Comunicação unidirecional única

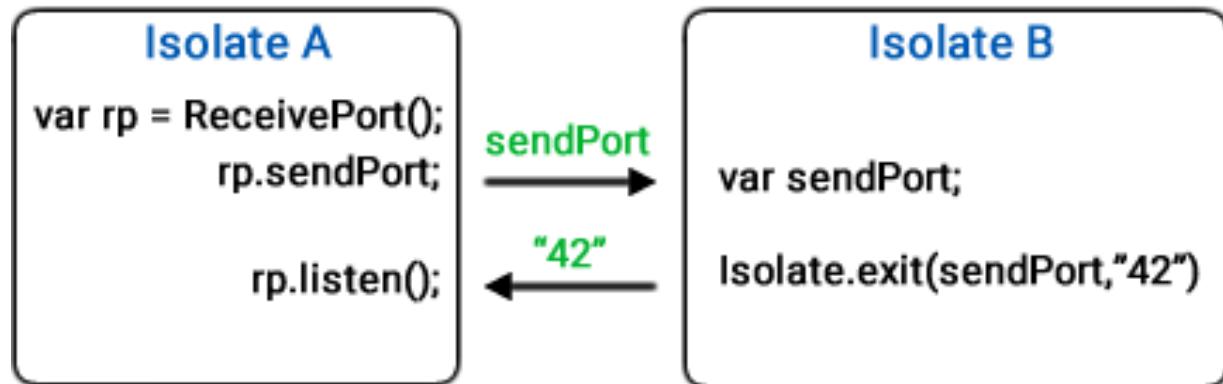


Figura 14.3: Comunicação unidirecional única

```
import 'dart:isolate';
void main() async {
    final receivePort = ReceivePort();
    await Isolate.spawn(workerIsolate, receivePort.sendPort);
    receivePort.first.then(( dado ) =>
        print('Mensagem recebida: $dado'));
}
void workerIsolate(SendPort sendPort) async {
    final resposta = await Future.delayed(Duration(seconds: 2),
        () => 'Conteúdo do arquivo');
    Isolate.exit(sendPort, resposta);
}
```

O objeto `sendPort` é passado por parâmetro para a nova isolate. Uma vez estando lá, ele possui a referência e conhecimento para realizar a transmissão de mensagens para o seu receptor `ReceivePort`, tornando-se a ponte de comunicação entre elas. Por isso, ao encerrar a nova isolate com seu método `exit()` é possível definir a mensagem de retorno utilizando esta ponte de comunicação.

Um ponto muito importante a ser ressaltado é que as isolates sofreram uma refatoração com algumas melhorias de performance na versão 2.15 da linguagem, onde foi introduzido um novo conceito de *isolate group*. Agora,

isolates criadas através do método `spawn()` fazem parte de um mesmo grupo que podem compartilhar algumas estruturas de dados internas que impactam diretamente na performance das mesmas, levando a um consumo menor de memória.

Ainda assim as isolates continuam completamente isoladas, cada uma com seu heap de memória, event loop e processamento separados. Porém, apenas isolates em um mesmo grupo podem se comunicar com a mensagem enviada através do método `exit()`.

Ainda sobre comunicação, também existem casos onde a isolate *worker* precisa enviar constantemente informações sobre o andamento de alguma tarefa que ela esteja executando, como o download e processamento de um arquivo grande, por exemplo. Para essa situação onde a comunicação unidirecional deve enviar várias mensagens, é feito o uso do método `send()` de `SendPort`.

Comunicação unidirecional frequente

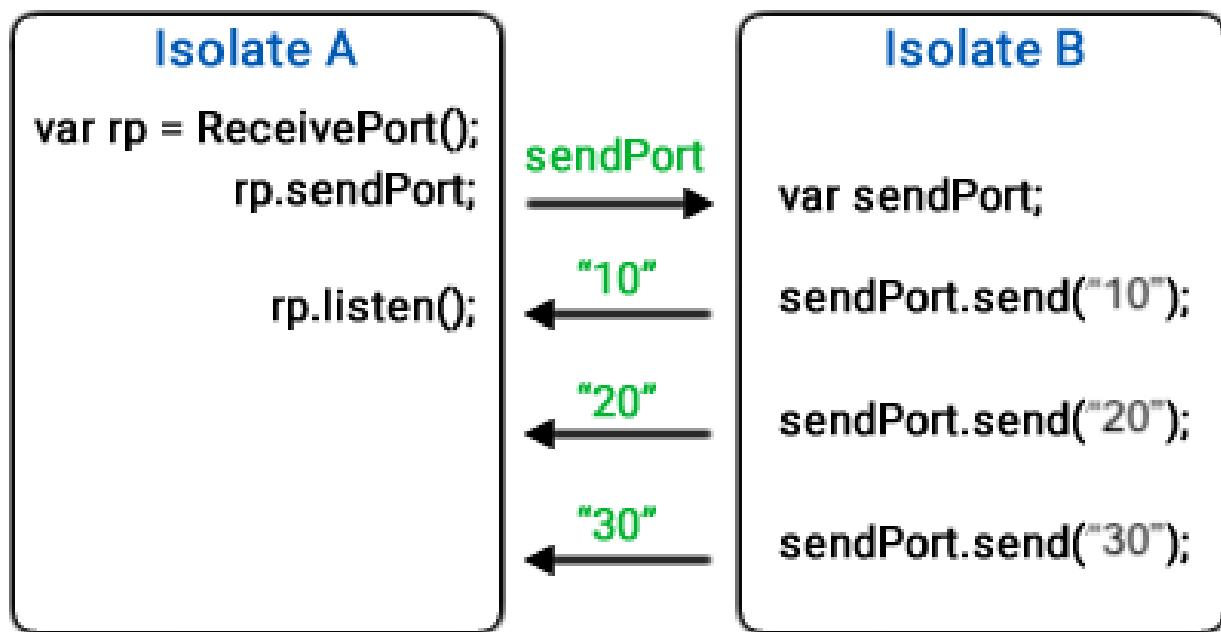


Figura 14.4: Comunicação unidirecional com várias mensagens

```
import 'dart:async';
import 'dart:isolate';
```

```

void main() async {
  final receivePort = ReceivePort();
  await Isolate.spawn(workerIsolate, receivePort.sendPort);
  receivePort.listen((dados) {
    print('Mensagem recebida: $dados');
  });
}

void workerIsolate(SendPort sendPort) async {
  sendPort.send('Upload iniciado');
  Timer.periodic(Duration(milliseconds: 100), (timer) {
    if (timer.tick % 10 == 0) sendPort
      .send('Upload ${timer.tick}%');

    if (timer.tick == 100) {
      timer.cancel();
      sendPort.send('Upload encerrado');
    }
  });
}

> Mensagem recebida: Upload iniciado
> Mensagem recebida: Upload 10%
> Mensagem recebida: Upload 20%
...
> Mensagem recebida: Upload 100%
> Mensagem recebida: Upload encerrado

```

O exemplo anterior simula o upload de um arquivo grande com a ajuda de um `Timer`. A cada iteração de 100 milissegundos o seu atributo `tick` é incrementado em 1, e a cada 10 iterações a `isolate` principal é atualizada com o status atual do upload através da porcentagem. Até que chegue a 100% e o timer e a `isolate` sejam encerrados.

Com o `send()` as mensagens são disparadas e adicionadas na stream do receptor, que justamente por ser uma stream, permite recuperá-las normalmente através de uma simples inscrição. Existem algumas restrições referente ao tipo do objeto que pode ser enviado, mas são casos muito específicos (como um `Socket`, por exemplo), então no geral você não encontrará problemas.

14.3 Comunicação bidirecional

Em alguns casos pode ser necessário que ambas isolates sejam capazes de enviar e receber informações, criando assim uma comunicação bidirecional.

Para isso ser possível, ainda utilizamos as portas `ReceivePort` e `SendPort`, porém, agora é necessário que ambas possuam uma referência para o `sendPort` do receptor da outra isolate. A nova isolate precisa encontrar uma forma de enviar para a isolate de origem uma cópia de sua porta de entrada, o que pode ser feito facilmente através da própria porta de entrada da isolate de origem.

Calma que não é tão confuso quanto parece. Tudo isso é muito similar ao processo de *handshake* utilizado em diversos protocolos de comunicação como FTP ou TCP, onde duas máquinas precisam confirmar que estão aptas para troca de informações. Nessa analogia as máquinas seriam as isolates.

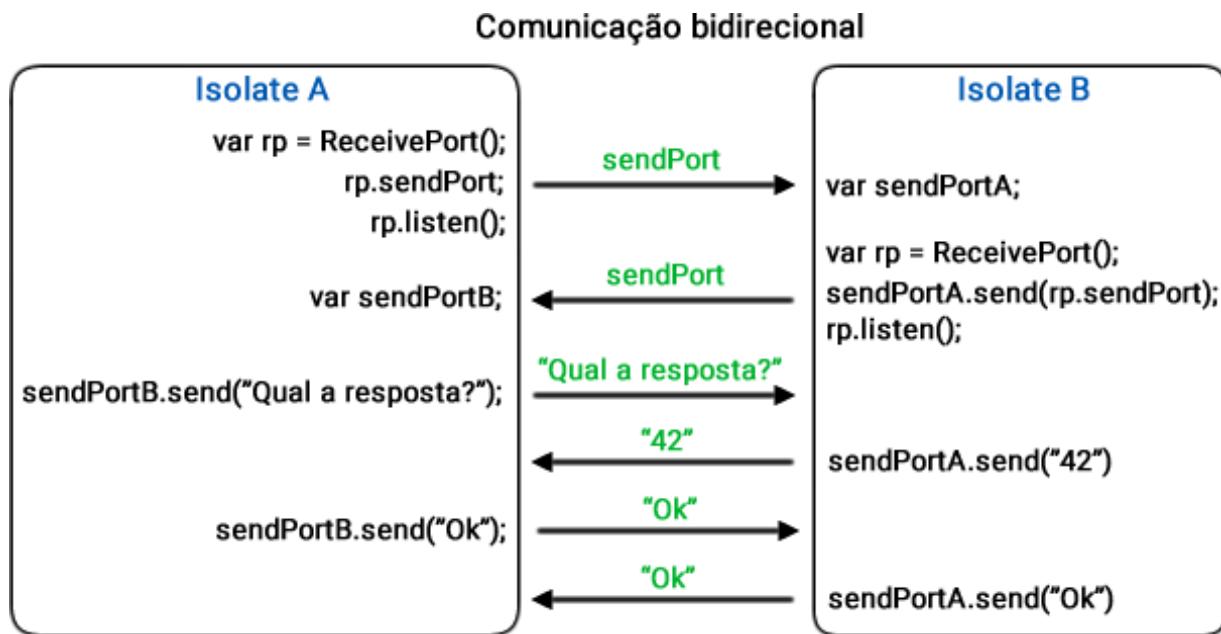


Figura 14.5: Comunicação bidirecional

É necessário um pouco mais de código para estabelecer essa comunicação, inclusive contendo vários conceitos de programação assíncrona que vimos até o momento:

```
import 'dart:async';
import 'dart:isolate';
Future<SendPort> isolateBidirecional(
    ReceivePort receivePort, void onMessage(var message)) async {
    final completer = Completer<SendPort>();
    print('[MAIN Isolate]: Iniciado comunicação, aguardando
handshake');
    receivePort.listen((mensagem) {
        if(mensagem is SendPort) {
            completer.complete(mensagem);
        } else {
            onMessage(mensagem);
        }
    });
    await Isolate.spawn(novaIsolate, receivePort.sendPort);
    return completer.future;
}

void novaIsolate(SendPort sendPort) {
    print('[NOVA Isolate]: Solicitação de comunicação recebida,
enviando resposta');
    final receivePort = ReceivePort();
    receivePort.listen((mensagem) {
        print('[NOVA Isolate]: Mensagem recebida $mensagem');
    });
    sendPort.send(receivePort.sendPort);

    Timer.periodic(Duration(seconds: 2), (timer){
        sendPort.send(timer.tick);
        if(timer.tick == 10) timer.cancel();
    });
}

void main() async {
    final receivePort = ReceivePort();
    SendPort sendPort = await isolateBidirecional(
        receivePort, (mensagem) {
            print('[MAIN Isolate]: Mensagem recebida $mensagem');
        });
    print('[MAIN Isolate]: Handshake concluído, comunicação
```

```

bidirecional estabelecida.');
Timer.periodic(Duration(seconds: 1), (timer){
  sendPort.send(timer.tick);
  if(timer.tick == 10) timer.cancel();
});
}

> [MAIN Isolate]: Iniciado comunicação, aguardando handshake
> [NOVA Isolate]: Solicitação de comunicação recebida, enviando
resposta
> [MAIN Isolate]: Handshake concluído, comunicação bidirecional
estabelecida.
> [NOVA Isolate]: Mensagem recebida 1
> [MAIN Isolate]: Mensagem recebida 1
> [NOVA Isolate]: Mensagem recebida 2
> [NOVA Isolate]: Mensagem recebida 3
> [MAIN Isolate]: Mensagem recebida 2
...

```

Os pontos principais são:

- O `isolateBidirecional()` é o método que vai inicializar uma nova isolate. Ele recebe por parâmetro o `receivePort` e a função que deve ser executada cada vez que chegar uma nova mensagem. Seu retorno é um `Future<SendPort>` que será completado com a porta para envio de mensagens para a nova isolate.
- Ao criar a nova isolate, é passado o `sendPort` da isolate principal. Esse seria o primeiro contato do *handshake* para iniciar a comunicação. É através dessa porta que a nova isolate vai responder com o seu próprio `sendPort`. Quando esse `sendPort` chega à isolate original, o `future` pode ser completado, sinalizando que o processo de *handshake* foi concluído com sucesso.
- Com a referência do `sendPort` obtida por ambas isolates, elas estão aptas para trocar qualquer tipo de informação de forma bidirecional. Então cada uma inicializa um `timer` e começa a trocar mensagens até que ele atinja o contador 10, encerrando a comunicação.

14.4 Controlando uma Isolate

Uma vez que uma nova isolate foi criada existem alguns métodos que ajudam a gerenciá-la e podem ser úteis, como a habilidade de pausar, resumir ou até mesmo encerrar por completo a sua execução. Todo esse controle é feito pela própria instância de `Isolate` retornada por

`Isolate.spawn()` :

```
import 'dart:async';
import 'dart:isolate';
void main() async {
    final receivePort = ReceivePort();
    final isolate = await Isolate.spawn(novaIsolate,
                                         receivePort.sendPort);

    final capability = Capability();
    Future.delayed(Duration(seconds: 1), () {
        isolate.pause(capability);
    });
    Future.delayed(Duration(seconds: 3), () {
        isolate.resume(capability);
    });

    receivePort.listen(( dado ) {
        print('Mensagem recebida: $dado');
        if(dado == 7) {
            isolate.kill(priority: Isolate.immediate);
            receivePort.close();
        }
    });
}
void novaIsolate(SendPort sendPort) async {
    Timer.periodic(Duration(milliseconds: 500), (timer) {
        sendPort.send(timer.tick);
    });
}

> Mensagem recebida: 1
> Mensagem recebida: 2
```

```
> Mensagem recebida: 6  
> Mensagem recebida: 7
```

Ao pausar uma isolate, é possível passar por parâmetro um objeto `Capability`, que, caso não seja informado, será criado e retornado pelo próprio método `pause()`.

Acontece que para poder resumir a isolate deve ser utilizada exatamente essa mesma instância de `capability` que foi usada para pausar. Por isso, falamos que um objeto `Capability` é infalsificável. Nenhum outro objeto diferente vai conseguir resumi-la, o que leva as pessoas a utilizarem o atributo `isolate.pauseCapability` da isolate para esse controle.

No exemplo, a nova isolate executa um timer que envia inteiros a cada 500 milissegundos, como as mensagens **1** e **2**, mas após um segundo na isolate principal, ela é pausada. Nesse momento, é importante entender que o *event loop* dessa isolate foi pausado, assim, novos eventos não serão executados, como o envio das mensagens. Mas, por outro lado, isso não pausa processos que já iniciaram a sua execução, como o próprio timer ou um *loop while(true)*.

Após 3 segundos a isolate é resumida, assim seu *event loop* volta a executar e as mensagens são disparadas, enviando **6** e **7**. Repare que por trás dos panos o timer continuou sendo incrementado a cada 500 milissegundos. A isolate principal possui uma validação e, ao receber a mensagem **7**, a nova isolate é encerrada através do método `kill()`.

Como `receivePort` é uma stream, ela continuará aberta e impedindo o encerramento do processo da *isolate main*. Para impedir que isso aconteça e seguindo as boas práticas de streams, ele também é encerrado com `receivePort.close();`.

Cadastrando listeners

Além de ter o controle, ainda é permitido cadastrar alguns *listeners* para algumas ações que ocorrem na isolate.

```

import 'dart:async';
import 'dart:isolate';
void main() async {
  final receivePort = ReceivePort();
  receivePort.listen((dados) {
    print('Mensagem recebida: $dados');
  });
  final isolate = await Isolate.spawn(novaIsolate,
      receivePort.sendPort, errorsAreFatal: true);

  isolate.addOnExitListener(receivePort.sendPort,
      response: 'Isolate finalizada');
  isolate.addErrorListener(receivePort.sendPort);
  Future.delayed(Duration(seconds: 1), () {
    isolate.ping(receivePort.sendPort, response:
        'Ping com sucesso');
  });
}

void novaIsolate(SendPort sendPort) async {
  Future.delayed(Duration(seconds: 2), () {
    throw TimeoutException('Exceção na isolate');
  });
}

> Mensagem recebida: Ping com sucesso
> Mensagem recebida: [TimeoutException: Exceção na isolate, #0
novaIsolate.<anonymous closure> #1      new Future.delayed.
<anonymous closure> (dart:async/future.dart:316:39)
...
> Mensagem recebida: Isolate finalizada

```

São três as opções disponíveis:

- `addOnExitListener` : define um objeto como resposta que será disparado para a porta informada sempre que a isolate for finalizada.
- `addErrorListener` : permite cadastrar uma porta para receber as mensagens sempre que a isolate disparar um erro não capturável. A mensagem será uma lista com o nome do erro e o seu *stacktrace*.

- `ping` : não é necessariamente um *listener*, mas é útil para validar se uma determinada isolate está ativa e funcionando. Caso sim, o objeto passado será enviado como resposta para a porta informada.

Todos recebem uma `SendPort` por parâmetro, que é por onde as mensagens serão enviadas. Pode ser utilizada a mesma porta para todos, da mesma forma que pode ser criada uma para cada, por motivos de organização.

No exemplo, todos os *listeners* foram registrados para entregarem as mensagens na mesma porta. Após 1 segundo, é disparado um ping para a nova isolate, que responde com a mensagem `Ping com sucesso`. Com 2 segundos, o future na isolate completa e dispara uma `TimeoutException`, nesse momento a mensagem contendo o erro é disparada para a isolate principal.

Como a nova isolate foi criada com o parâmetro `errorsAreFatal: true`, o primeiro erro que ela dispara também a faz ser encerrada, o que também acaba disparando a mensagem `Isolate finalizada` para a isolate principal.

14.5 Isolate através de uma URI

A API de isolates também fornece uma opção para conexão e disparo de novas isolates que não necessariamente se encontram no mesmo arquivo, projeto ou até mesmo máquina.

O método estático `Isolate.spawnUri()` é quem permite esta conexão, ele recebe três parâmetros obrigatórios. O primeiro é um objeto `URI` que contém o endereço da isolate, podendo ser um arquivo separado ou até mesmo uma URL HTTP. O segundo é uma lista de strings que serão passadas como argumentos para o método `main()` desse arquivo, que é quem dará início a nova isolate. O terceiro parâmetro é a mensagem, normalmente a própria porta de comunicação `SendPort`, que também é repassada para o `main()`.

Na prática, considerando o seguinte código em um arquivo `isolate.dart` :

```
import 'dart:isolate';
void main(List<String> args, SendPort sendPort) {
  print('[${Isolate.current.debugName}] - Args ${args}');
  sendPort.send('Esse livro é muito massa mesmo!');
}
```

Esse arquivo que vai ser inicializado com uma nova isolate precisa obrigatoriamente conter uma função `main()`, que pode ou não conter os parâmetros, mas deve seguir um dos padrões a seguir:

```
main() {}
main(args) {}
main(args, mensagem) {}
```

Agora em um arquivo `main.dart`, na mesma pasta para facilitar o teste, o disparo da nova isolate:

```
import 'dart:isolate';
void main() async {
  final receivePort = ReceivePort();
  receivePort.listen((dados) {
    print('[IsolatePrincipal] - $dados');
  });

  await Isolate.spawnUri(Uri.parse('isolate.dart'),
    ['Args 1', 'Args 2'],
    receivePort.sendPort, debugName: 'IsolateSeparada');
}

> [IsolateSeparada] - Args {[Args 1, Args 2]}
> [IsolatePrincipal] - Esse livro é muito massa mesmo!
```

A comunicação e comportamento dessa nova isolate disparada pelo `spawnUri()` continua a mesma de uma isolate disparada pelo `spawn()`, com a diferença de que ela não possuirá todas as melhorias de performance proporcionadas por fazer parte de um grupo (*isolate group*), afinal ela está localizada em algum outro local/ambiente. Com isso, a escolha de qual

método utilizar estará atrelada a este detalhe, dependendo da necessidade de cada projeto.

Assim, encerramos o conhecimento sobre o funcionamento das isolates e você já está consciente da importância das mesmas. Afinal, conceitos de *threads* e programação paralela sempre costumam ser um dos assuntos mais complexos ao explorar uma nova linguagem de programação. Agora nos resta conhecer o último recurso relacionado ao mundo assíncrono: as zones.

14.6 Zones

Dentre todos os assuntos abordados deste universo assíncrono de Dart, as zones com certeza estão entre os menos conhecidos e explorados como um todo. É muito comum pessoas programarem na linguagem e nunca terem sequer ouvido falar da existência de uma zone, pois este é um recurso de mais baixo nível geralmente utilizado pelo compilador ou APIs já existentes.

Mas, ainda assim, é possível utilizarmos uma zone a nosso favor em algumas situações. Para começo de conversa, uma zone representa um ambiente estável entre chamadas assíncronas, como se fosse uma espécie de área virtual dentro de uma isolate, que engloba a execução do nosso código.

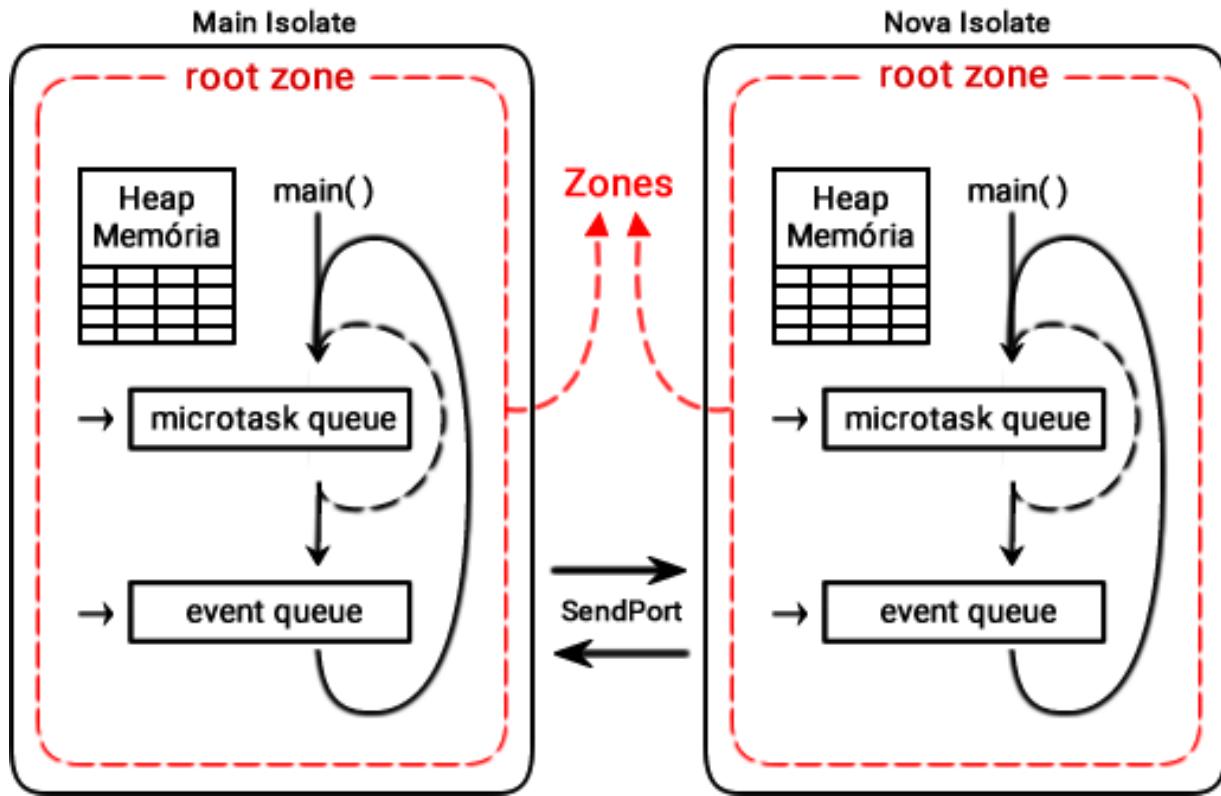


Figura 14.6: Zones e sua área virtual.

Todo código existente é obrigatoriamente executado no contexto (dentro da área virtual) de uma zone padrão conhecida como *root zone* (zone raiz), seja através do método `main()` principal ou em uma nova isolate criada através do `spawn()`. Essa zone *root* é criada e utilizada automaticamente, além de possuir todas as implementações padrões de comportamento e operações de uma zone.

```
import 'dart:async';
void main() {
  final root = Zone.current;
  print(root); // > Instance of '_RootZone'
}
```

A instância que representa a zone atual de execução é sempre acessível através da constante `Zone.current`, permitindo a sua manipulação, que no exemplo referencia a implementação de `_RootZone`. Novas zones podem

ser criadas a partir de uma zone já existente, através do seu método `fork()`, e consequentemente herda suas características:

```
import 'dart:async';
void main() {
    final root = Zone.current;
    print('Zone principal: $root');

    final novaZone = root.fork();
    novaZone.run(() {
        print('Zone nova: ${Zone.current}');
        print('Zone nova pai: ${Zone.current.parent}');
    });
}

> Zone principal: Instance of '_RootZone'
> Zone nova: Instance of '_CustomZone'
> Zone nova pai: Instance of '_RootZone'
```

No exemplo, uma nova zone é criada a partir da zone principal, que passa a ser sua zone pai. Para executar um código dentro de uma zone basta chamar seu método `run()` com a função desejada. A forma mais comum de se criar e executar trechos de código em uma zone separada, entretanto, é através da função `runZoned()` de `dart:async`.

```
R runZoned<R>(R body(), {Map<Object?, Object?>? zoneValues,
  ZoneSpecification? zoneSpecification,
  @Deprecated Function? onError})
```

Essa função funciona como um atalho para o que vimos anteriormente, pois ela criará a zone fazendo um `fork()` de `Zone.current` e executará o código passado em `R body()` com o método `run()`.

```
import 'dart:async';
void main() {
    runZoned(() {
        print('Zone nova: ${Zone.current}');
        print('Zone nova pai: ${Zone.current.parent}');
    });
}
```

```
> Zone nova: Instance of '_CustomZone'  
> Zone nova pai: Instance of '_RootZone'
```

Repare como o parâmetro `onError` está *deprecated*, o que indica que será removido em versões futuras. Os demais parâmetros desta função permitem a customização da zone e veremos suas funcionalidades a seguir.

Hierarquia de zones

Não há um limite para a quantidade de zones criadas, ao criar uma nova zone ela é aninhada em uma espécie de hierarquia onde a zone atual passa a conter esta nova zone. Considere esse exemplo:

```
import 'dart:async';  
void main() {  
    a();  
    Future? future;  
    runZoned(() {  
        future = Future(a).then(b);  
    });  
    future?.then(c);  
}  
  
a() => print('a');  
b(valor) => runZoned(() => a());  
c(valor) => print('c');
```

Como bem sabemos, `runZoned()` cria novas zones, então no código anterior existem 3 zones distintas de execução. Na prática, esse trecho de código estará separado em contextos de execução, como na ilustração a seguir, que representa em qual zone cada trecho é executado:

```

void main() {
    a();
    Future? future;
    runZoned(() {
        future = Future(a).then(b);
    });
    future?.then(c);
}
a() => print('a');
b(valor) => runZoned(() => a());
c(valor) => print('c');

```



Figura 14.7: Hierarquia de zones

Como a função `a()` é chamada nas três zones, ela em algum momento estará presente em todas. De acordo com a hierarquia existente, a `zone#1` ou `root` é pai da `zone#2`, que por sua vez é pai da `zone#3`. A próxima imagem ilustra a ordem de execução do código, levando em conta sua zone atual:

```
main()
```

```
  a()
```

```
    runZoned()
```

```
      Função anônima  
      Future(a)  
      then
```

Event Loop executa de forma assíncrona a função agendada

```
  a()
```

Future completa, chamando a função b

```
  b()
```

```
    runZoned()
```

```
      Função anônima  
      a()
```

O callback then executa no main()

```
c()
```

Figura 14.8: Hierarquia de zones

O ponto-chave é que uma chamada assíncrona sempre executará no contexto da zone em que ela foi criada. A própria função `a()` executou nas três zones diferentes. Mesmo que na `zone#2` a sua execução tenha sido agendada pelo *event loop*, ela sempre executará na mesma zone em que ocorreu esse agendamento. Da mesma forma, note que, por mais que o future tenha sido criado na `zone#2`, a função `c()` executou na `zone#1`, pois o agendamento do *callback* `future.then(c)` ocorreu nessa zone.

Outro ponto é que a função `b()` que executa na `zone#2` acaba criando uma terceira zone `zone#3`. Nesse momento, a `zone#1` contém a `zone#2`, que contém a `zone#3`, definindo uma hierarquia onde os filhos sempre herdam características de execução do pai.

Por que criar uma zone?

Certo, já sabemos o que é e como funciona uma zone, agora, para que raios elas servem? Bom, justamente por elas terem essa característica de envolver o código dentro dessa área virtual de execução, você consegue customizar alguns comportamentos de tudo que executar dentro deste contexto, ou até mesmo armazenar variáveis de acesso global.

Curiosamente várias das funcionalidades de Dart que vimos durante o livro executam através da zone atual, como o agendamento de uma *microtask*, um *timer* ou até mesmo a simples função `print()`, por exemplo, o que permite que sejam customizadas. Uma outra característica muito utilizada é conseguir capturar e centralizar todos os erros que ocorrerem dentro de uma determinada zone. Vamos explorar essas situações no restante do capítulo.

14.7 Uma zone livre de erros

Os erros inesperados são algo que realmente precisamos evitar a qualquer custo, e não é à toa que Dart fornece inúmeras formas diferentes de se capturar tanto erros síncronos quanto assíncronos. Uma zone também acaba sendo uma excelente aliada neste quesito, pois facilmente é possível capturar qualquer erro que ocorra dentro dela.

Uma das formas de se criar uma zone livre de erros era justamente utilizando o próprio método `runZoned()` e definindo seu parâmetro `onError`. Mas devido às modificações da *null-safety* na linguagem esse parâmetro foi depreciado, levando à inserção de um novo método `runZonedGuarded()` para cumprir esse papel, separando as responsabilidades e melhorando a API de zones.

```
R? runZonedGuarded<R>(R body(), void onError(Object error,  
StackTrace stack), {Map<Object?, Object?>? zoneValues,  
ZoneSpecification? zoneSpecification});
```

O uso de ambos é muito similar, basta definir no parâmetro `body()` todo o código envolto nesta nova zone:

```
import 'dart:async';
void main() {
  runZonedGuarded(() {
    mainZonedGuarded();
  }, (e, s) {
    print('Erro capturado: $e');
  });
}
void mainZonedGuarded() {
  Future(() => throw TimeoutException('Um erro qualquer'));
  Future(() => throw 'Outro erro qualquer');
  Future(() => throw 'Mais um erro qualquer');
}

> Erro capturado: TimeoutException: Um erro qualquer
> Erro capturado: Outro erro qualquer
> Erro capturado: Mais um erro qualquer
```

E assim como `runZoned()`, uma nova zone será criada realizando um *fork* da zone atual. Chamamos esta nova zone de "livre de erros" porque é obrigatória a definição do parâmetro `onError()`, que capturará todos os erros síncronos e assíncronos originados dentro desta zone.

A vantagem em se definir zones livres de erros é que elas acabam funcionando como um bloco `try/catch` global, centralizando os problemas. São excelentes para capturar erros de toda a aplicação, ou apenas uma parte, se desejado, e salvá-los em arquivos de log ou também enviá-los para ferramentas de análises de erros como o Crashlytics ou o Sentry.

Repare como no exemplo nenhum dos três erros assíncronos disparados pararam a execução do código, e todos foram capturados pela zone, o que nos leva a definir que zones livres de erro são zones seguras que se mantém estáveis entre diferentes chamadas assíncronas.

Propagação de erros entre zones

É natural que erros não capturados se propaguem no código até que acabem encerrando a execução do programa ou processo. Entre as zones livres de

erro, um erro, além de não encerrar sua execução, pode ter um comportamento um pouco diferente.

```
import 'dart:async';
void main() {
    runZonedGuarded(() {
        runZoned(() {
            Future(() => throw TimeoutException(
                'Um erro qualquer na zone#3'));
        });
    }, (e, s) {
        print('Erro capturado na zone#2 $e');
    });
}
> Erro capturado na zone#2 TimeoutException: Um erro qualquer na
zone#3
```

Um erro que ocorre em uma determinada zone e não é tratado nela é propagado para as zones pais da hierarquia e qualquer uma dessas poderá capturá-lo, encerrando a sua propagação. Como acontece no exemplo onde um erro lançado na zone#3 é capturado apenas em sua zone pai zone#2 . Então, isso significa que um erro assíncrono nunca sairá de uma zone livre de erros. Neste outro exemplo fica ainda mais claro esse comportamento:

```
import 'dart:async';
void main() {
    Future? future;
    runZonedGuarded(() {
        future = Future(() => throw TimeoutException(
            'Um erro qualquer'));
    }, (e, s) {
        print('Erro capturado na zone $e');
    });
    future!.catchError((e) => print('Nunca executa'));
}
> Erro capturado na zone TimeoutException: Um erro qualquer
```

Mesmo que o *callback* `catchError()` do `future` exista, ele nunca será executado pois o erro disparado pelo `future` é capturado pela zone interna,

impedindo que ele saia dela. Mas da mesma forma que não saem, erros externos também não entram nessas zones, olhe só:

```
import 'dart:async';
void main() {
    final future = Future(() => throw TimeoutException(
        'Um erro qualquer'));
    future.whenComplete(() => print('whenComplete zone#1'));
    runZonedGuarded(() {
        future.whenComplete(() => print('whenComplete zone#2'));
        print('Dentro zone#2');
    }, (e, s) {
        print('Erro capturado na zone#2 $e');
    });
    runZoned(() {
        future.whenComplete(() => print('whenComplete zone#3'));
    });
}
> Dentro #zone2
> whenComplete zone#1
> whenComplete zone#3
> Unhandled exception:
> TimeoutException: Um erro qualquer
> #0      main.<anonymous closure>... stacktrace
```

No exemplo, são criadas duas zones internas que registram um *callback* `whenComplete` para um `future` criado na zone *root*. A primeira delas é uma zone livre de erros. Ao executar o programa, o `future` completará com um erro e, de todos os *callbacks*, apenas o da zone livre de erros não é executado, indicando que um erro externo foi impedido de entrar nesta zone, por mais que seu código tenha sido executado normalmente.

14.8 Valores locais

Uma zone possui a capacidade de armazenar valores que se tornam disponíveis e acessíveis em qualquer lugar dentro de seu escopo. Essas

variáveis locais são definidas através de um `Map` pelo parâmetro `zoneValues`.

```
import 'dart:async';
void main() {
  runZoned(() {
    print(Zone.current['resposta']); // > 42
  }, zoneValues: {'resposta': 42});
}
```

Esses valores são acessados através de sua chave e o operador `[]` da zone atual, como o `Zone.current['resposta']` do exemplo. Por mais que esse map de valores aceite qualquer tipo, por padrão, é comum definir essas chaves como *symbols*:

```
import 'dart:async';
void main() {
  runZoned(() {
    print(Zone.current[#resposta]); // > 42
  }, zoneValues: {#resposta: 42});
}
```

Além de armazenar uma variável que deva estar disponível em qualquer local dentro da zone, esses valores também podem ser úteis para algumas situações de *debug*, pois eles podem ser sobreescritos. Olhe só:

```
import 'dart:async';
void main() {
  runZoned(() {
    logZone();
    runZoned(() {
      logZone();
      print('Resposta ${Zone.current[#resposta]}');
    }, zoneValues: {#nomeZone: 'Zone B'});
  }, zoneValues: {#nomeZone: 'Zone A', #resposta: 42});
}
void logZone() => print('Rodando na ${Zone.current[#nomeZone]}');

> Rodando na Zone A
> Rodando na Zone B
> Resposta 42
```

As zones filhas sempre herdam valores de suas zones pais, de acordo com a hierarquia existente, o que permite a sobrescrição. Por exemplo, o `#nomeZone` poderia ser utilizado para criar logs de identificação de em qual zone um determinado código está sendo executado, para fins de *debug* da aplicação.

14.9 Sobrescrevendo funcionalidades com ZoneSpecification

Já que algumas funcionalidades de Dart executam através da zone atual, podemos facilmente sobrescrevê-las. Repare neste trecho de código do SDK que possui a implementação de um `Timer`, onde o trabalho é delegado justamente para execução na zone atual:

```
factory Timer(Duration duration, void Function() callback) {
  if (Zone.current == Zone.root) {
    return Zone.current.createTimer(duration, callback);
  }
  return Zone.current
    .createTimer(duration,
                Zone.current.bindCallbackGuarded(callback));
}
```

Isso possibilita que o comportamento dessas tarefas seja customizado ao rodarem em uma determinada zone de execução. Digamos que você precisa que, ao utilizar o `print()`, seja sempre impressa junto das mensagens a data atual, ou qualquer outra informação. Para isso, bastaria modificar o comportamento desta função dentro da sua zone:

```
import 'dart:async';
void main() {
  runZoned(() {
    print('Teste log');
    print('42');
  }, zoneSpecification: ZoneSpecification(
    print: (Zone self, ZoneDelegate parent, Zone zone,
           String line) {
      parent.print(zone, '${DateTime.now()} - $line');
    }
  ));
}
```

```

        }
    ));
}

> 2022-01-22 15:03:22.226552 - Teste log
> 2022-01-22 15:03:22.230763 - 42

```

Pronto, agora toda vez que a função `print` for utilizada dentro desta zone, ela imprimirá a data e hora junto à mensagem. Isso é feito através do último parâmetro de `runZoned()`, dentre os já aprendidos, onde é permitido especificar um objeto do tipo `ZoneSpecification` para modificar um comportamento da zone.

Essas customizações são feitas através de vários *handlers* disponíveis, dentre os quais o `PrintHandler` é utilizado para customizar a função `print()`. Todos os *handlers* possuem 3 parâmetros padrão seguidos dos parâmetros da função original, então, se a assinatura de `print` é `print(String line)`, o `PrintHandler` é:

```
typedef void PrintHandler(Zone self, ZoneDelegate parent,
                           Zone zone, String line);
```

Já para o `scheduleMicrotask(void callback())`:

```
typedef void ScheduleMicrotaskHandler(Zone self,
                                       ZoneDelegate parent, Zone zone, void f());
```

E assim por diante, sendo que esses 3 parâmetros em comum são:

1. `Zone self` : a própria zone em que o *handler* está definido.
2. `ZoneDelegate parent` : um objeto que representa a zone pai, e é utilizado para delegar a execução da função para a zone acima da hierarquia. As funções do *delegate* são as mesmas definidas nos *handlers*, porém com apenas 1 parâmetro em comum, que é uma `Zone`. A função `print`, por exemplo, é definida no *delegate* como `void print(Zone zone, String line)`.
3. `Zone zone` : a `zone` em que originou a chamada da função. Algumas operações precisam saber qual a função original para execução, como uma *microtask* através de `scheduleMicrotask`, que deve sempre

executar na zone original. Esse é o mesmo parâmetro que deve ser repassado para o *delegate*.

Para facilitar essa associação, considere uma hierarquia de zone com a seguinte estrutura:

```
runZoned(() { // #zone1
    runZoned(() { // #zone2
        runZoned(() { // #zone3
            scheduleMicrotask(() => print(42));
        });
    }, zoneSpecification: ZoneSpecification(
        scheduleMicrotask: (Zone self, ZoneDelegate parent,
                            Zone zone, void f()) {
            parent.scheduleMicrotask(zone, f);
        }
    ));
});
```

São três *zones* criadas na ordem `#zone1 > #zone2 > #zone3`. E mesmo que a *specification* definida na `#zone2` não faça nada além de delegar para a zone superior, ao ser executada os três parâmetros vão referenciar:

1. `Zone self` : a própria `#zone2` .
2. `ZoneDelegate parent` : não contém a referência em si, mas como *delegate* representa a `#zone1` .
3. `Zone zone` : a `#zone3` que originou a requisição.

Todo esse poder de customização permitido pelas *zones* pode ser útil para várias situações. O próprio `runZonedGuarded()` que utilizamos para criar zones livres de erro é um atalho para o `HandleUncaughtErrorHandler` presente na `ZoneSpecification` . Então além de criar logs globais de erros que podem ser exportados para posterior análise, existem packages como o `stack_trace` , que através de zones fornecem um formato mais agradável de *stack trace* do erro para melhor análise.

Uma outra possibilidade seria analisar o tempo de execução de uma zone. Para isso, basta você customizar o seu método `run()` e utilizar um `Stopwatch` nele. Enfim, tudo dependerá da sua criatividade e necessidade,

mas um fato é que assim que você navegar pelo código-fonte de `dart:core` e principalmente `dart:async`, notará que as zones são profundamente utilizadas. Conhecer esse recurso será um acréscimo grande para o caminho de proficiência na linguagem.

14.10 Se liga aí

- Realize tarefas que demandem um maior processamento em uma isolate separada, priorize a experiência do usuário com sua aplicação e evite congelar ou prejudicar a execução da isolate principal.

14.11 É com você

1 - O Flutter possui o `compute()`, a maneira mais indicada para se criar uma nova isolate, funcionando como um atalho para `Isolate.spawn()`. Pesquise sobre isso.

2 - Aprendemos como estabelecer uma comunicação bidirecional entre isolates utilizando apenas `ReceivePort` e `SendPort`. Existe um package chamado `stream_channel`, no qual o objeto `IsolateChannel` facilita a comunicação bidirecional em uma isolate. Utilize-o para implementar a comunicação entre duas isolates.

Até aqui

As isolates são de fato uma mão na roda para processar tarefas pesadas, não é mesmo? Afinal, manter a isolate principal da aplicação livre para processar novas requisições do usuário é sempre uma excelente prática. Durante o capítulo pudemos nos aprofundar em como distribuir essas tarefas entre diferentes isolates e principalmente como manter a comunicação entre elas. Também foi exemplificado o conceito prático das zones, que, embora sejam muito utilizadas internamente pelo SDK, são pouco exploradas e até conhecidas por desenvolvedores no geral. Mesmo

sendo uma ferramenta muito poderosa, pois podem englobar e modificar o comportamento de todo o código dentro da aplicação, inclusive deixando-o mais seguro e livre de erros.

Agora, estamos quase chegando ao fim do livro, mas ainda não acabou! No capítulo a seguir veremos sobre manipulação de arquivos e como as isolates e zones podem ajudar nesta tarefa.

CAPÍTULO 15

Na prática - Gerando arquivos

Trabalhar com arquivos é uma tarefa muito comum no mundo da programação. Eventualmente você vai se deparar com algum relatório que precisa ser feito para a área de negócios, algum arquivo de importação de dados no sistema ou até mesmo um `log.txt` que precisa ser criado com erros da sua própria aplicação.

Então, nada mais justo do que abordarmos a manipulação de arquivos neste que é o último capítulo do livro, junto a um velho conhecido nosso: o projeto ClimaTempo. Nele já construímos uma CLI que consulta cidades e dados do clima em uma API de terceiros, recebe avisos através de um *server* disponível, e agora nosso papel será salvar esses dados consultados em arquivos locais que funcionarão como *logs*.

Mas antes de colocar a mão na massa precisamos entender o funcionamento básico dos arquivos em Dart. Portanto neste capítulo vamos:

- Aprender a manipular arquivos;
- Utilizar arquivos na prática com o app ClimaTempo;
- Criar isolates separadas para manipular arquivos;
- Capturar erros em arquivos de logs com a ajuda das zones.

15.1 Manipulando arquivos em Dart

Manipular um arquivo pode parecer uma tarefa difícil, e de fato é em algumas linguagens, mas você vai se surpreender com quão fácil é essa tarefa em Dart. Ao longo do livro, utilizamos `dart:io` para solucionar alguns problemas e é essa mesma library, de *input* e *output* que disponibiliza uma API para manipulação de diretórios e arquivos.

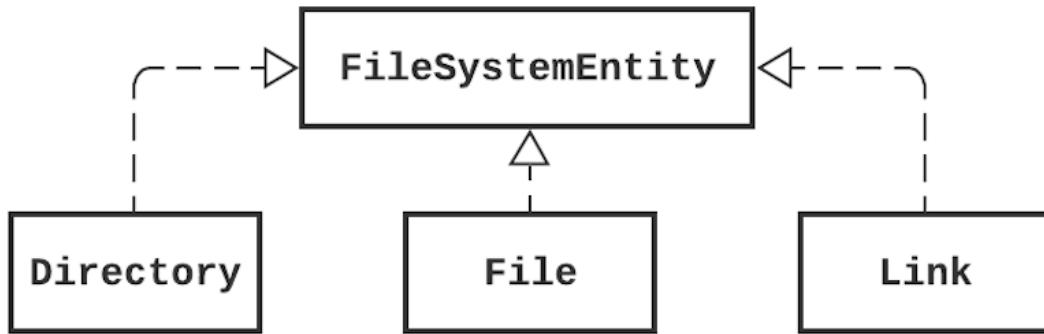


Figura 15.1: Organograma classes de I/O

Existem 4 classes principais que compõem essa estrutura. A primeira delas é a `FileSystemEntity`, que representa a classe mãe com várias implementações e métodos em comum definidos. Dificilmente precisaremos nos preocupar com ela, pois de fato usamos diretamente uma de suas 3 implementações.

- `Link` : dentre as três talvez seja a menos utilizada, serve para criar e manipular *symlinks*, um tipo especial de arquivo que contém uma referência para outro arquivo ou diretório existente.
- `Directory` : classe que fornece o necessário para criação e gerenciamento de diretórios.
- `File` : provavelmente a mais utilizada, permite a criação e gerenciamento de arquivos.

Criando um diretório

Criar um diretório com `Directory` é fácil:

```

import 'dart:io';
Future<void> main() async {
    final dir = Directory('dir/teste');
    await dir.create(recursive: true);
    print('Diretórios criados: $dir');
    // > Diretórios criados: Directory: 'dir/teste'
}

```

É possível informar um caminho absoluto como `C:/Users/julio/` ou um caminho relativo conforme o exemplo. Nesse último caso, será usado como referência o `Directory.current`, que aponta para o diretório atual. O método assíncrono `create()` é quem de fato vai criar os diretórios, e o seu parâmetro `recursive` indica se devem ser criadas todas as pastas informadas ou só a última.

Ao rodar o exemplo, o diretório `dir/teste` será criado na raiz do projeto. Remover um diretório também é uma tarefa simples:

```
import 'dart:io';
Future<void> main() async {
    final dir = Directory('dir/teste');
    if(await dir.exists()) {
        await dir.delete(recursive: true);
        print('Diretório removido: $dir');
        // > Diretório removido: Directory: 'dir/teste'
    }
}
```

A tentativa de remover um diretório não existente resulta em erro, por isso validamos se ele existe através de `dir.exists()`. Só que nesse caso o `recursive` de `delete()` funciona um pouco diferente. Por mais que esteja `true`, ele deletará apenas o diretório `teste`, pois nesse contexto ele é usado para indicar se os arquivos de um diretório não vazio devem também ser removidos.

Criando e escrevendo em um arquivo

A criação e remoção de um arquivo são semelhantes às de um diretório.

```
import 'dart:io';
Future<void> main() async {
    final file = File('teste.txt');
    await file.create();
    print('Arquivo criado: $file');
    // > Arquivo criado: File: 'teste.txt'
}
```

Com o método `create()`, um arquivo `teste.txt` será adicionado na raiz do projeto, assim como o `delete()` faria sua remoção. Mas um arquivo vazio de nada adianta, por isso temos o `writeAsString()`:

```
import 'dart:io';
import 'dart:convert';
Future<void> main() async {
  final file = File('teste.txt');
  await file.writeAsString('linha um', encoding: Utf8Codec());
  await file.writeAsString('\nlinha dois', mode: FileMode.append);
}
```

Esse método abre o arquivo (caso não exista, ele é criado), grava a `String` informada utilizando a codificação passada em `encoding`, que por padrão já é UTF8, e ao terminar fecha o arquivo. O parâmetro `mode` com `FileMode.append` garante que o texto informado é adicionado ao final, não descartando o que já existe no arquivo, diferente do padrão `FileMode.write`, que substitui todo o conteúdo. Muito fácil, não é?

E como nem sempre tratamos de arquivos de texto (poderia ser um vídeo ou uma música), é interessante conhecer a existência do método equivalente `writeAsBytes(List<int> bytes)`, que funciona da mesma forma, porém para escrita de `bytes`.

Escrevendo em um arquivo sob demanda

Agora que conhecemos muito bem streams, sabemos que elas são uma forma de produzir dados sob demanda, uma vez que podem emitir diversas vezes dados durante o seu ciclo de vida, e essas mesmas streams podem funcionar como fontes de dados para escrita de um arquivo.

Se a cada dado emitido pela stream utilizássemos o método anterior de escrita, estaríamos sempre abrindo o arquivo, escrevendo o desejado e encerrando a conexão, o que não é exatamente uma boa prática. Uma forma mais conveniente é através do método `openWrite()`, que cria e disponibiliza uma *sink* do tipo `Iosink` para adição de dados na stream de escrita do arquivo.

```

import 'dart:io';
Future<void> main() async {
  final file = File('teste.txt');
  IOSink sink = file.openWrite(mode: FileMode.write);

  await for(String texto in gerarStream()) {
    sink.write(texto);
    await sink.flush();
  }
  await sink.close();
}

Stream<String> gerarStream() async* {
  for(int i = 0; i <= 10; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield '$i\n';
  }
}

```

Com o auxílio de um gerador, cada valor emitido pela stream é adicionado ao sink com seu método `write()`, que funciona para textos; `sink.add()` seria o equivalente para *bytes*. Já o método `flush()` sincroniza os dados que podem ser mantidos em *buffer*, garantindo sua escrita. Por fim, como já estamos acostumados, é sempre necessário encerrar um sink com o `close()`.

Após os 10 segundos durante a execução do exemplo, o arquivo `teste.txt` conterá como resultado todos os valores de 0 a 10 impressos linha a linha. Obviamente, esse exemplo é um pouco lúdico para ser necessário utilizar uma stream, mas no dia a dia algumas situações refletem essa necessidade, como o download de um arquivo grande que pode ser feito através de *chunks* (ou pedaços) de dados.

Lendo um arquivo

Assim como escrever em um arquivo, ler o seu conteúdo é bem simples.

```

import 'dart:io';
import 'dart:convert';
Future<void> main() async {

```

```
final file = File('teste.txt');
String texto = await file.readAsString(encoding: Utf8Codec());
print(texto);
}
```

O método `readAsString()` de `file` abre o arquivo, lê o texto de acordo com o `encoding` informado (que já é UTF8, por padrão) e retorna um `future` que completa com a `String`. O resultado do programa é todo o conteúdo impresso no console. Para arquivos que não são de texto, também é possível ler em `bytes` com o `readAsBytes()`.

Lendo um arquivo sob demanda

Para finalizar, a leitura de um arquivo também pode ser realizada através de `streams` com o `openRead()`.

```
import 'dart:io';
import 'dart:convert';
Future<void> main() async {
  final file = File('teste.txt');

  Stream<List<int>> streamBytes = file.openRead();
  Stream<String> linhas = utf8.decoder.bind(streamBytes)
    .transform(LineSplitter());
  await for (var linha in linhas) {
    print('Valor da linha: ${linha}');
  }
}
```

Esse método vai fornecer uma stream de `Stream<List<int>>` que representam os bits do arquivo. Para um ser humano não significa muita coisa, mas o processo de transformação deste para um conteúdo inteligível já é conhecido. Basta usarmos o `decoder` do `utf8` para converter esses dados para `String`, que, se combinado com o `transformer` `LineSplitter`, resultará em uma `Stream<String>` que emite todo o conteúdo do arquivo separado por linhas.

Sendo assim, se o arquivo `teste.txt` contém:

```
Linha 1  
Linha 2
```

O programa acima vai imprimir:

```
valor da linha: Linha 1  
valor da linha: Linha 2
```

No geral, a manipulação de arquivos e diretórios em Dart é relativamente simples, as subclasses de `FileSystemEntity` possuem diversos outros métodos utilitários como `exists()` ou `rename()` que eventualmente são necessários. E, conforme esperado, todos esses métodos que utilizamos são assíncronos, pois manipulação de arquivos depende de acessos ao Sistema Operacional que podem demandar um certo tempo.

Acontece que cada um desses métodos possui um método síncrono equivalente `readAsString()` / `readAsStringSync()`, `exists()` / `existsSync()` e assim por diante. A diferença entre eles é apenas na execução de forma síncrona e assíncrona. Fazer a leitura de um arquivo com `readAsStringSync()` vai congelar a *isolate* principal até que seja finalizada, por isso é sempre aconselhável a utilização da forma assíncrona para evitar estes bloqueios.

15.2 Climatempo - Salvando consultas

Agora que sabemos manipular arquivos, podemos partir para a parte interessante! Para concluir o projeto, você precisará ter a aplicação ClimaTempo criada nos capítulos anteriores funcionando, através da sua própria implementação ou pegando o código no GitHub do livro. Isso é essencial, pois daremos continuidade nas funcionalidades criadas anteriormente.

Atualmente, a aplicação consulta as cidades disponíveis e a situação do clima atual em cada uma delas. A ideia é agora salvar em arquivos de texto o retorno de todas essas consultas em uma espécie de log. Para isso,

introduziremos uma nova *flag* `-s` / `--salvar` para os comandos já existentes.

```
> clima_tempo cidade [-n --nome] "Pedras Grandes" [-e --estado] sc  
[-s --salvar]  
> clima_tempo agora [-i --id] 4765 [-s --salvar]
```

Sempre que o usuário desejar salvar o resultado do comando, bastará inserir este novo parâmetro no final. Então o primeiro passo é definir no *parser* existente esses novos parâmetros.

```
ArgParser criarParser() {  
    return ArgParser()  
        ..addCommand('cidade', ArgParser()  
            ..addOption('nome', abbr: 'n',  
                valueHelp: 'Nome da cidade para consulta')  
            ..addOption('estado', abbr: 'e',  
                valueHelp: 'Sigla do estado para consulta')  
            ..addFlag('salvar', abbr: 's',  
                help: 'Salvar o resultado em arquivo de log',  
                negatable: false))  
        ..addCommand('agora', ArgParser()  
            ..addOption('id', abbr: 'i',  
                valueHelp: 'Id da cidade para consulta do tempo')  
            ..addFlag('salvar', abbr: 's',  
                help: 'Salvar o resultado em arquivo de log',  
                negatable: false))  
        ..addCommand('alerta')  
        ..addFlag('help', abbr: 'h',  
            help: 'Como utilizar o programa', negatable: false);  
}
```

Como é um parâmetro opcional que pode ou não estar presente, o `salvar` também pode normalmente ser uma *flag* no *parser* dos comandos `agora` e `cidade`.

Criando o arquivo de cidades

Relembrando o comando `cidade`, ele pode ser utilizado para listar cidades por um nome `[-n "Nome"]`, por um estado `[-e SC]`, pelo conjunto dos

dois, ou até mesmo sem qualquer parâmetro, o que retornaria todas as cidades disponíveis para consulta de clima no Brasil. E todas essas combinações listam as cidades de acordo com o que já estabelecemos no `toString()` do objeto `Cidade`:

```
Id: 5049 - Nome: Abdon Batista - Estado: SC, País: BR  
Id: 5050 - Nome: Abelardo Luz - Estado: SC, País: BR  
Id: 5051 - Nome: Agrolândia - Estado: SC, País: BR  
...
```

É exatamente essa listagem que vamos pegar e salvar em um arquivo denominado `cidades.txt`, que ficará na pasta raiz do projeto, em `log/cidades.txt`. Então, crie um arquivo `log.dart` dentro da pasta `lib`, que será o responsável pelo gerenciamento dos arquivos de log. Vamos trabalhar nele agora.

Mas antes de tentar salvar um arquivo em qualquer diretório, é preciso se certificar de que este existe; caso contrário, `dart:io` lançará uma exceção.

```
import 'dart:io';  
Future<void> _validarDiretorio() async {  
    final dir = Directory('log');  
    if(!(await dir.exists())) {  
        await dir.create();  
    }  
}
```

A função `_validarDiretorio()` fará exatamente isso, cria o diretório `/log` caso ele não exista na raiz do projeto. Com isso, já é seguro manipular arquivos dentro dele, e o primeiro será o `cidades.txt`.

```
import 'model/cidade.dart';  
Future<void> salvarCidades(List<Cidade> cidades) async {  
    await _validarDiretorio();  
    final file = File('log/cidades.txt');  
  
    final sink = file.openWrite(mode: FileMode.write);  
    cidades.forEach((c) => sink.writeln(c));  
  
    await sink.flush();
```

```
    await sink.close();
}
```

A implementação de `salvarCidades` é bastante simples. Como recebemos uma lista de `cidades` e cada uma será uma linha no arquivo, abrimos a escrita do arquivo através de uma stream, em modo `FileMode.write`. Ou seja, a cada execução todo o conteúdo do arquivo é substituído.

Depois, para cada cidade é inserido o conteúdo do seu `toString()` ao `sink` da stream, onde o método `writeln()` sempre adicionará uma nova linha após sua escrita. Ao final basta realizar o `flush()` e garantir o fechamento do `sink`. Com isso, é só voltar ao `climatempo.dart` e associar a *flag* que criamos com o salvamento do arquivo.

```
import 'package:climatempo/log.dart';
void main(List<String> args) async {
    // código omitido
    final comando = argsResult.command;
    try {
        if (comando != null && comando.name == 'cidade') {
            final nomeCidade = comando['nome'];
            final estado = comando['estado'];

            final cidades = await buscarCidades(estado,
                                                nome: nomeCidade);
            if (comando['salvar']) await salvarCidades(cidades);

            cidades.forEach((c) => print(c));
        }
    } catch(e) {
        print(e);
    }
    // código omitido
}
```

Todo o código acima já é existente no arquivo, com exceção do `import` e a linha onde se valida se a *flag* `salvar` está presente no `ArgResults` e, caso ela esteja, chamamos o método `salvarCidades()`, que realizará a escrita no arquivo antes de apresentar os resultados na linha de comando.

Com isso, executando `dart run bin/climatempo.dart cidade -e SC -s` todas as cidades de Santa Catarina serão salvas no arquivo. Experimente executar sem o parâmetro do estado e você terá no arquivo todas as cidades brasileiras.

Criando o arquivo de clima

O próximo objetivo é também salvar as consultas de clima de uma cidade em arquivos de log, só que dessa vez cada cidade deve ter um arquivo separado. Primeiro, vamos relembrar o funcionamento do comando `agora`. Ao executar `dart run bin/climatempo.dart agora -i 4915`, onde `4915` é o `id` de uma das cidades existentes, a saída é:

```
Cidade: Florianópolis, SC - BR  
2021-11-26 19:35:24.000  
Temperatura: 24 - Sensação: 24  
Umidade: 74 - Velocidade do Vento: 13
```

Um resumo de informações térmicas da cidade no momento da consulta. Esse resumo deve ser salvo em um arquivo `log/4915.txt`, que guardará o histórico de todas as consultas para a cidade Florianópolis. Cada cidade vai possuir um arquivo de log diferente com o nome sendo o seu `id`. Para isso, volte em `log.dart` e insira a função `salvarTempo()`, que criará os arquivos das cidades:

```
import 'package:climatempo/model/clima_tempo.dart';  
Future<void> salvarTempo(ClimaTempo climaTempo) async {  
    await _validarDiretorio();  
  
    final file = File('log/${climaTempo.cidade.id}.txt');  
    await file.writeString('===\n$climaTempo',  
                           mode: FileMode.append);  
}
```

O objeto `climaTempo` também contém todas as informações necessárias em seu `toString()`. Note que usamos o `id` da cidade para referenciar o nome do arquivo, então as consultas de cidades iguais sempre ficarão nos mesmos

arquivos, da mesma forma que o modo de escrita é o `FileMode.append` para sempre manter o conteúdo já salvo no arquivo.

Agora ficou faltando associar a *flag* com a chamada desta nova função no `climatempo.dart`.

```
void main(List<String> args) async {
    // código omitido
    if (comando != null && comando.name == 'agora') {
        final id = comando['id'];
        if (id == null) {
            print('É obrigatório informar um [-id] de cidade');
            exit(2);
        }

        final tempo =
            await registrarCidadeEBuscarTempo(int.parse(id));
        if (comando['salvar']) await salvarTempo(tempo);
        print(tempo);
    }
}
```

Novamente, todo esse código já existe no arquivo, com exceção da linha que faz a chamada para a função `salvarTempo()`. Agora é só testar adicionando a *flag* ao final do comando, como `dart run bin/climatempo.dart agora -i 4915 -s`. Rode algumas vezes e veja que todas elas tiveram o conteúdo salvo em `4915.txt` no formato a seguir:

```
===
Cidade: Florianópolis, SC - BR
2022-01-27 00:32:39.000
Temperatura: 27 - Sensação: 30
Umidade: 84 - Velocidade do Vento: 9
===
Cidade: Florianópolis, SC - BR
2022-01-27 00:37:22.000
Temperatura: 27 - Sensação: 30
Umidade: 84 - Velocidade do Vento: 9
```

15.3 Arquivos em isolates separadas

Em todos os nossos exemplos a manipulação de arquivos foi de certa forma simples em termos de tempo de execução. Tanto salvar os dados das cidades quanto do clima são rápidos o suficiente para não interferir na execução da *isolate* principal. E utilizar o assincronismo presente em Dart é suficiente.

Mas estamos cientes de que manipular arquivos do sistema operacional pode eventualmente ser custoso, dependendo do que precisa ser feito. Por isso, trabalhar com arquivos é um dos casos de uso que podem ser executados em uma isolate separada, e é isso que faremos com os arquivos da aplicação ClimaTempo.

Primeiro, faremos para o arquivo criado pelo comando `cidade`, então volte em `log.dart` para fazer as modificações necessárias.

```
import 'dart:isolate';
void _registrarCallbackFinalizacao(Isolate isolate) {
    final receivePort = ReceivePort();
    receivePort.listen((mensagem) {
        print(mensagem);
        receivePort.close();
    });
    isolate.addOnExitListener(receivePort.sendPort,
        response: 'Arquivo salvo com sucesso!');
}

Future<void> salvarCidadesIsolate(List<Cidade> cidades) async {
    final isolate = await Isolate.spawn(salvarCidades, cidades);
    _registrarCallbackFinalizacao(isolate);
}

Future<void> salvarCidades(List<Cidade> cidades) async {
    // código omitido
}
```

O método `salvarCidades()`, que é utilizado atualmente para criação do arquivo, continua exatamente igual. A diferença é que ele agora é utilizado

como função de entrada para a nova *isolate*, *spawnada* através do novo método `salvarCidadesIsolate()`.

No momento em que jogamos essa tarefa de salvar o arquivo para uma *isolate* separada, a *isolate* principal pode dar continuidade na execução já imprimindo todas as cidades no console. Essa nova *isolate* pode demorar o tempo que for, pois já não tem tanta interferência na execução principal, mesmo que demore 1 milissegundo ou 1 minuto. Entretanto, para sabermos quando a mesma é encerrada, usamos um `ReceivePort` para cadastrar um *listener* que vai imprimir 'Arquivo salvo com sucesso!' assim que ela encerrar.

Agora, para o outro arquivo de clima, basta utilizar a mesma lógica, criar um novo método que dispara uma nova *isolate* para executar o método já existente:

```
Future<void> salvarTempoIsolate(ClimaTempo climaTempo) async {
  final isolate = await Isolate.spawn(salvarTempo, climaTempo);
  _registrarCallbackFinalizacao(isolate);
}

Future<void> salvarTempo(ClimaTempo climaTempo) async {
  // código omitido
}
```

Para utilizar esses novos métodos, entretanto, é necessário alterar as chamadas em `climatempo.dart`. No método `main()`, troque a chamada de `salvarCidades()` para `salvarCidadesIsolate()` e, de `salvarTempo()`, para `salvarTempoIsolate()`. E pronto, agora os arquivos serão criados em *isolates* separadas.

15.4 Log de erros com zones

Nunca desenvolvemos códigos querendo que ocorram erros durante seu uso, mas toda programadora precavida sabe que eventualmente erros em tempo de execução ocorrem, principalmente quando há uma dependência

na comunicação com aplicações externas. É prática comum as aplicações e servidores de aplicações capturarem esses erros em arquivos de *logs* para posterior análise e solução pela equipe.

Por isso, vamos capturar todos os possíveis erros do ClimaTempo e guardar seu *stacktrace* em um arquivo diário de *log*, que terá o nome no formato `DDMMAAAA.txt` (dia, mês e ano) e também ficará no diretório de *logs* da aplicação. A implementação dessa funcionalidade a essa altura do campeonato é bem óbvia, e é um bom desafio caso você queira implementar antes de ver a solução a seguir.

Em `log.dart` podemos adicionar então esta nova função:

```
Future<void> salvarErro(Object erro, StackTrace stackTrace)
  async {
    await _validarDiretorio();

    final data = DateTime.now();
    final ano = data.year;
    final mes = _doisDigitos(data.month);
    final dia = _doisDigitos(data.day);
    final file = File('log/$dia$mes$ano.txt');

    await file.writeAsString('${data.toIso8601String()} : $erro\n',
        mode: FileMode.append);
    await file.writeAsString('${data.toIso8601String()} :
$stackTrace',
        mode: FileMode.append);
  }

  String _doisDigitos(int numero) {
    if (numero >= 10) return '$numero';
    return '0$numero';
}
```

A função `salvarErro()` recebe por parâmetro os objetos que serão salvos, sendo esses os mesmos disparados pelas exceções. O `erro` é uma breve descrição do próprio erro, enquanto `stacktrace` contém toda sua pilha de rastreamento. A data atual é pega para montar o nome do arquivo, afinal, todos os erros ocorridos no mesmo dia ficarão no mesmo arquivo.

Já a `_doisDigitos()` é uma função utilitária que ajuda a formatar meses e dias que contenham apenas 1 dígito, para manter o padrão de número de caracteres na nomeação dos arquivos. A escrita dos dados é feita em modo `append` para sempre manter o conteúdo existente nos arquivos.

Agora precisamos apenas capturar todos os erros da aplicação e encaminhar para essa função. Essa é uma tarefa que pode ser facilmente realizada com a ajuda de uma *zone* livre de erros. Para isso, é necessário encapsular todo o código do método `main()` em uma nova *zone*.

```
void main(List<String> args) {
    runZonedGuarded(() {
        executar(args);
    }, (e, s) {
        print(e);
        print(s);
        salvarErro(e, s);
    });
}

Future<void> executar(List<String> args) async {
    final parser = criarParser();
    // código omitido
}
```

Exatamente todo o código que estava dentro do `main()` foi extraído para a função `executar()`. Essa agora é chamada dentro de uma *zone* customizada, que, ao capturar qualquer erro, além de imprimir no console vai salvar em nosso arquivo de *logs*. Como capturar os erros agora será responsabilidade da *zone*, remova também qualquer bloco `try/catch` que existir no arquivo.

Agora, para testar, basta forçar a ocorrência de qualquer erro, e um jeito fácil é alterando o token utilizado para acesso à API, que fica em `api.dart`. Modifique qualquer coisa desse token e a requisição se tornará inválida, então ao executar qualquer comando uma exceção será lançada e salva corretamente no arquivo, conforme planejado.

Bom, com o fim deste capítulo você conclui oficialmente o aprendizado de todos os assuntos relacionados à concorrência presentes em Dart. Passamos pelo *event loop*, *event queue*, *microtask queue*, *futures*, *async/await*, *completers*, *streams*, geradores, *isolates* e finalmente *zones*. Neste momento, com certeza trabalhar de forma assíncrona e também em paralelo já não soa tão complexo, e obviamente, quanto mais se pratica, mais natural todos esses conceitos ficarão.

15.5 Se liga aí

- Ao trabalhar com manipulação de arquivos prefira utilizar os métodos assíncronos presentes em `dart:io`, ao invés de suas versões síncronas.

15.6 É com você

1 - Envolvemos nosso app ClimaTempo em uma *zone* livre de erros para poder capturá-los e salvá-los em um log. Porém, as novas *isolates* que criamos executam separadas e, consequentemente, em uma outra *zone*. Envolva essas *isolates* em suas próprias *zones* livres de erro e salve esses erros em um arquivo `io.txt`.

2 - Existe uma forma melhor de capturar todos os erros de uma *isolate* sem utilizar a *zone*? Altere o exercício anterior para utilizar esta forma. Dica: analisar os *listeners* da *isolate*.

3 - Novamente no projeto ClimaTempo, altere para salvar todos os arquivos em diretórios organizados, como abaixo:

- **log:** contém o arquivo `cidades.txt`
- **log/clima:** contém os arquivos de climas das cidades
- **log/erro:** contém os logs de erros do CLI

- **log/erro/io:** contém os logs de erros que podem ocorrer nas *isolates* de criação de arquivos

CAPÍTULO 16

Até mais, e obrigado pelos peixes!

E é assim, com uma frase do *Guia do Mochileiro das Galáxias*, que me despeço de vocês neste fim de livro. Foi uma longa e intensa jornada! Toda a estrutura de conteúdo do livro foi pensada de forma estratégica para passarmos pelos conceitos básicos e ir evoluindo por todo o *core* da linguagem, de forma que qualquer pessoa, independente do conhecimento prévio, pudesse tirar o melhor proveito. E eu espero ter contribuído para sua evolução profissional de alguma forma.

Com o conhecimento adquirido aqui você já tem capacidade de entender qualquer código escrito em Dart e criar seus próprios programas. Mas pode ter certeza de que não para por aqui. Isso pode ter sido apenas o início da sua caminhada infinita no mundo da programação, afinal o aprendizado é constante à medida que as tecnologias vão se atualizando e, pode confiar em mim, isso ocorre com frequência.

Aqui no livro mesmo foi preciso priorizar as principais funcionalidades e deixar de fora outras que também são de extrema importância em algumas situações específicas. Por isso deixo a seguir algumas dicas de assuntos para você se aprofundar e evoluir ainda mais:

Metadata

Nada mais são do que as *annotations* que podemos utilizar para inserir informações ao próprio código. Elas são opcionais e não afetam o comportamento do código em si, mas podem auxiliar na sua manutenibilidade.

Ao longo do livro, utilizamos alguns exemplos, como o `@deprecated`, que informa outros desenvolvedores que um determinado trecho de código está depreciado e será removido futuramente. O fato de ela estar ali não vai influenciar na execução do código, mas vai adicionar um metadado importante para identificação não só do desenvolvedor, mas também para

ferramentas como a própria IDE, que consegue identificar e já informar quem estiver programando para atualizar o seu código.

Outra *annotation* muito usada é o `@override` para identificar métodos herdados de classes pais. Enfim, você pode se aprofundar e ver como criar seus próprios metadados, inclusive, esses mesmos metadados são muito úteis quando alinhados ao assunto a seguir.

Geração de código

O que fazer quando existe uma tarefa manual que precisa ser feita repetidas vezes incansavelmente? Um bom programador preguiçoso dirá: encontre um padrão e automatize o processo! E é exatamente para isso que utilizamos código para gerar ainda mais código.

São várias as situações onde isso é útil. No capítulo 11, por exemplo, ao consumir os dados json de uma API, definimos métodos manualmente como `fromJson()` e `toJson()` para converter de json para objetos e vice-versa. Acontece que existe o package `json_serialization`, que gera esses métodos automaticamente, e inclusive utiliza a annotation `@JsonSerializable` para identificar as classes que devem ser geradas.

Esses geradores são criados através de builders definidos no package `build`, mas os packages `build_runner` e `source_gen` também ajudam no processo. Pesquise sobre eles para se aprofundar.

Testes unitários

Quem nunca subiu o código para produção, deixou para fazer os testes depois e esse "depois" nunca chegou? Pode confessar, faz parte do aprendizado. Não ter abordado o assunto testes no livro não faz deles menos importante. É todo um universo e também filosofia de desenvolvimento (como o próprio TDD), que existem livros inteiros focados apenas nisso.

Todo desenvolvedor profissional sabe a importância que eles têm para manter a qualidade das entregas. Aliás, os unitários são os mais famosos, mas existem diversos outros tipos de testes como de componentes ou de

integração, que também possuem suporte em Dart. Os principais packages que auxiliam nesse processo todo são o `test` e `mockito`, vale a pena conferir.

DevTools

DevTools consiste em um conjunto de ferramentas para *debug* e análise de performance que são simplesmente indispensáveis quando trabalhamos profissionalmente com aplicações Dart e/ou Flutter.

O processo de *debugging*, por exemplo, é algo que faz parte do dia a dia de um desenvolvedor. Definir *breakpoints*, inspecionar variáveis, avaliar exceções e verificar o *stack* de chamadas são coisas naturais e de fácil acesso através de uma IDE, mas que só são possíveis por conta do DevTools. Algumas das demais ferramentas existentes são:

- Acompanhamento de logs.
- Análise de tamanho das aplicações.
- Inspeção de UI/widgets e estado de um app Flutter.
- Diagnóstico de performance, uso de memória e CPU.
- Análise de tráfego de *network*.

Interface para funções externas - FFI

O mundo da programação é muito amplo e vem sendo construído a décadas, e durante todo esse tempo muitas soluções para problemas de software foram construídas, comprovadas e testadas incansavelmente nas mais diversas linguagens de programação.

Só que nem toda linguagem consegue se comunicar nativamente e chamar funções existentes em bibliotecas construídas por outras linguagens, e é aí que entra o conceito de *foreign function interface* (FFI) para funcionar como uma interface de comunicação entre os dois mundos.

Dart possui sua própria implementação disponível em `dart:ffi`, que busca criar esta interoperabilidade com a linguagem C, ou seja, é possível chamar funções nativas utilizando nosso próprio código escrito em Dart. Quem se

beneficia muito com isso é o Flutter, podendo interagir com as bibliotecas nativas das mais diferentes plataformas em que executa.

Shelf

Uma das maravilhas de Dart está em sua versatilidade de ser utilizada em diferentes plataformas, e uma dessas possibilidades está no backend com a criação de web servers e disponibilização de APIs. Durante os exemplos do livro chegamos a criar um server HTTP simples utilizando apenas o package `dart:http`, mas existem alguns outros frameworks construídos acima dele que acabam trazendo mais funcionalidades.

Os projetos mais famosos nessa área eram o Aqueduct e o Angel, que infelizmente foram descontinuados pelos seus criadores recentemente, então o Shelf tem se popularizado. Criado pela própria equipe de desenvolvimento da linguagem, é uma alternativa leve e rápida para utilizar Dart no backend, seja na construção de APIs REST ou *web sockets*.

Não espere um framework de web server robusto como um Django ou Laravel com diversas funcionalidades, até porque esse não é o seu objetivo atualmente. Mas ele acaba servindo como uma alternativa simples se você deseja manter a mesma stack de Dart no seu projeto como um todo.

Explore o Flutter!

Não tem como negar a ascensão do Flutter nos últimos anos, que deve ser inclusive o motivo de você querer conhecer Dart e estar lendo este livro. Criado em 2015 e lançado em 2017, vem ganhando popularidade como um dos melhores, se não o melhor, framework *cross-platform* para criação de aplicações que rodam nativamente em diferentes plataformas: mobile, web, desktop e até mesmo sistemas embarcados.

Com uma comunidade grande, um ecossistema maduro, uma boa documentação, fácil de começar e aprender, com vários apps já em produção, grandes empresas adotando e performance muito similar aos apps nativos, com certeza é uma excelente opção se você deseja desenvolver mobile.

Contribua para projetos Open Source

Em toda a sua carreira na área de tecnologia ou até mesmo fora dela, você será impactado direta ou indiretamente por projetos *open source*. Eles representam um papel muito importante na evolução da indústria de software como um todo, por isso, fazer parte disso lhe traz muitos benefícios.

Independente de você ser iniciante ou veterano existem inúmeras formas de contribuir, seja com código, organização de tarefas, análise de PRs (*Pull Requests*) ou uma simples documentação. Fazer-se presente nesse cenário pode lhe abrir inúmeras portas e alavancar a sua carreira, além de impulsionar o seu ganho de experiência e senso de comunidade, que é extremamente importante para o trabalho em equipe.

Tanto Dart quanto o Flutter, embora comandados pelo Google, são projetos totalmente open source, onde qualquer um pode se engajar em auxiliar de alguma forma no crescimento do ecossistema. E não são os únicos, então encontre um projeto com que você tenha afinidade e comece a acompanhar seu desenvolvimento e participar ativamente. Acesse sua página do GitHub e veja como contribuir - geralmente essas instruções estão localizadas em um arquivo de nome `CONTRIBUTING.md` .

Aprenda uma outra linguagem de programação

Se Dart não é sua primeira linguagem, você já deve ter vivenciado na prática o quanto essa frase é verdadeira: "A próxima linguagem de programação que você estudar será muito mais fácil de aprender, e assim sucessivamente".

A Engenharia de Software é uma ciência que consiste na criação de metodologias e definição de conceitos (muitas vezes abstratos) para criação, manutenção e evolução de software. É algo independente de uma linguagem de programação específica, o que faz com que as linguagens tenham muitas semelhanças tanto em seus conceitos quanto até mesmo a padrões de sintaxe.

É por isso que quando estudamos uma nova linguagem de programação estamos muitas vezes estudando um conceito já conhecido, porém, de um ponto de vista de implementação diferente, o que enriquece ainda mais o seu conhecimento como pessoa engenheira de software. Portanto, a cada linguagem nova conseguimos identificar esses padrões tornando-a mais fácil de aprender como um todo, o que é gratificante, pois é nesse exato estágio que você perceberá que está se tornando cada vez mais experiente.

Sem contar que diferentes linguagens seguem diferentes paradigmas, implementações que diferem em performance e são usadas para diferentes propósitos, e não existe uma bala de prata. Estar exposto ao funcionamento de várias delas vai lhe abrir um leque maior de opções para tomar a melhor decisão ao criar um determinado projeto, sem a possibilidade de agir como um *fan boy* por ter o conhecimento restrito a uma única opção.

Até mais, e obrigado pelos peixes!