**Capítulo 11**

**Ferramentas, Linguagens e Frameworks para Desenvolvedores Ethereum**

Este capítulo é uma introdução às ferramentas, linguagens e frameworks utilizados para o desenvolvimento de contratos inteligentes Ethereum. Examinaremos diferentes métodos de desenvolvimento de contratos inteligentes para a blockchain Ethereum. Discutiremos em detalhes vários constructos da linguagem Solidity, que é atualmente a linguagem de desenvolvimento mais popular para contratos inteligentes na Ethereum.

Neste capítulo, cobriremos os seguintes tópicos:

* Linguagens
* Compiladores
* Ferramentas e bibliotecas
* Frameworks
* Desenvolvimento e implantação de contratos
* A linguagem Solidity

Há várias ferramentas disponíveis para desenvolvimento em Ethereum, incluindo clientes, IDEs e frameworks de desenvolvimento.

O conteúdo deste capítulo não inclui todos os frameworks e ferramentas existentes para desenvolvimento na Ethereum. Ele mostra as ferramentas e frameworks mais comumente utilizados, incluindo alguns que usaremos em nossos exemplos no capítulo seguinte.

Há vários recursos disponíveis relacionados a ferramentas de desenvolvimento para Ethereum no seguinte endereço:  
<http://ethdocs.org/en/latest/contracts-and-transactions/developer-tools.html#developer-tools>

**Linguagens**

Contratos inteligentes podem ser programados em uma variedade de linguagens para a blockchain Ethereum. Há três linguagens principais que podem ser usadas para escrever contratos, e três que têm interesse histórico:

* **Solidity**: Esta linguagem tornou-se um padrão para escrita de contratos na Ethereum. Esta linguagem é o foco deste capítulo.
* **Vyper**: Esta é uma linguagem experimental semelhante a Python que está sendo desenvolvida para trazer segurança, simplicidade e auditabilidade ao desenvolvimento de contratos inteligentes.
* **Yul**: Esta é uma linguagem intermediária que tem a capacidade de compilar para diferentes backends, como EVM e ewasm. Os objetivos de design do Yul incluem legibilidade, fluxo de controle fácil, otimização, verificação formal e simplicidade.
* **Mutan**: Esta é uma linguagem no estilo Go, que foi descontinuada no início de 2015 e não é mais usada.
* **LLL**: Esta é uma linguagem de baixo nível semelhante ao Lisp, daí o nome LLL. Também não é mais usada.
* **Serpent**: Esta é uma linguagem simples e limpa semelhante ao Python. Não é mais usada para desenvolvimento de contratos e não é suportada pela comunidade.

Como o código Solidity precisa ser compilado para bytecode, precisamos de um compilador para isso. Na próxima seção, introduziremos o compilador Solidity.

**O compilador Solidity**

Compiladores são usados para converter o código-fonte de contrato de alto nível para o formato que o ambiente de execução da Ethereum entende. O compilador Solidity, **solc**, é o mais comum em uso.

**solc** converte da linguagem Solidity de alto nível para bytecode da Máquina Virtual Ethereum (EVM), para que possa ser executado na blockchain pela EVM.

**Instalando o solc**

O **solc** pode ser instalado em um sistema operacional Linux Ubuntu usando o seguinte comando:

$ sudo apt-get install solc

Se os Personal Package Archives (PPAs) ainda não estiverem instalados, eles podem ser instalados executando os seguintes comandos:

$ sudo add-apt-repository ppa:ethereum/ethereum

$ sudo apt-get update

Para instalar o **solc** no macOS, execute os seguintes comandos:

$ brew tap ethereum/Ethereum

Este comando adicionará o repositório Ethereum à lista de fórmulas do brew:

$ brew install solidity

Esse comando produzirá uma saída longa e pode levar alguns minutos para ser concluído. Se nenhum erro for produzido, então eventualmente o Solidity será instalado.

Para verificar se o compilador Solidity está instalado e validar a versão do compilador, pode-se usar o seguinte comando:

$ solc –version

Este comando produzirá uma saída como a seguinte, exibindo a versão do compilador Solidity:

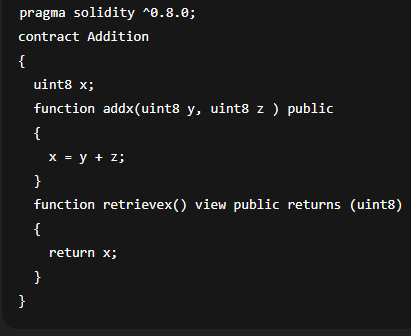
solc, the solidity compiler commandline interface

Version: 0.8.11+commit.d7f03943.Darwin.appleclang

Esta saída confirma que o compilador Solidity foi instalado com sucesso.

**Experimentando com o solc**

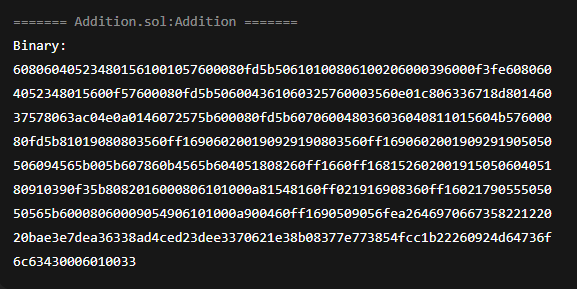
O **solc** suporta uma variedade de funções. Alguns exemplos são mostrados a seguir. Como exemplo, usaremos um contrato simples, Addition.sol:



Para ver o contrato inteligente no formato binário compilado, podemos usar o seguinte comando:

$ solc --bin Addition.sol

Este comando produzirá uma saída semelhante à seguinte:

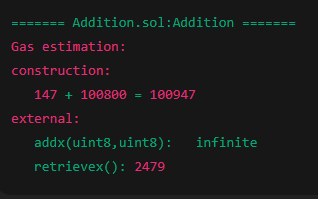


Essa saída mostra a tradução binária do código do contrato Addition.sol representado em hexadecimal.

Como uma taxa de gás é cobrada para cada operação que a EVM executa, é uma boa prática estimar o gás antes de implantar um contrato em uma rede ativa. Estimar o gás fornece uma boa aproximação de quanto gás será consumido pelas operações especificadas no código do contrato, o que dá uma indicação de quanto ether será necessário gastar para executar um contrato. Podemos usar a flag --gas para esse propósito, como mostrado no exemplo a seguir:

$ solc --gas Addition.sol

Isso dará a seguinte saída:

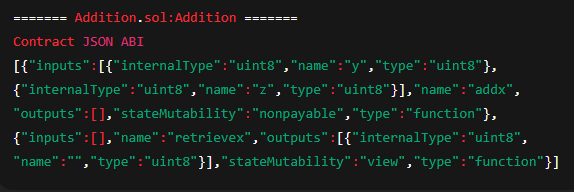


Essa saída mostra quanto uso de gás é esperado para essas operações no contrato inteligente Addition.sol. A estimativa de gás é mostrada ao lado de cada função. Por exemplo, a função retrievex() está estimada para usar 2479 de gás.

Podemos gerar a Interface Binária de Aplicativo (ABI) usando o **solc**, que é uma maneira padrão de interagir com os contratos:

$ solc --abi Addition.sol

Este comando produzirá um arquivo chamado Addition.abi como saída. O seguinte são os conteúdos do arquivo de saída Addition.abi:



A saída anterior exibe o conteúdo do arquivo Addition.abi, que está formatado em estilo JSON. Ele consiste em entradas e saídas juntamente com seus tipos. Vamos gerar e usar ABIs mais tarde neste capítulo para interagir com contratos inteligentes implantados.

Outro comando útil para compilar e produzir um arquivo compilado binário junto com um ABI é mostrado aqui:

$ solc --bin --abi -o bin Addition.sol

A mensagem exibida indicará se a execução do compilador foi bem-sucedida; caso contrário, erros serão relatados.

Compiler run successful. Artifact(s) can be found in directory bin.

Este comando produzirá uma mensagem e dois arquivos no diretório de saída bin:

* **Addition.abi**: Contém o ABI do contrato inteligente no formato JSON.
* **Addition.bin**: Contém a representação hexadecimal do binário do código do contrato inteligente.

**solc** é um comando muito poderoso e outras opções podem ser exploradas usando a flag --help, que exibirá opções detalhadas. No entanto, os comandos anteriores usados para compilação, geração de ABI e estimativa de gás devem ser suficientes para a maioria dos requisitos de desenvolvimento e implantação.

**Ferramentas, bibliotecas e frameworks**

Existem várias ferramentas e bibliotecas disponíveis para a Ethereum. As mais comuns são discutidas aqui. Nesta seção, primeiro instalaremos os pré-requisitos que são necessários para desenvolver aplicações para a Ethereum.

**Node.js**

Node.js é uma plataforma de desenvolvimento popular para executar código JavaScript, principalmente no lado do servidor backend; no entanto, também pode ser usado para frontends.

Como o Node.js é necessário para a maioria das ferramentas e bibliotecas, recomendamos instalá-lo primeiro. O Node.js pode ser instalado para seu sistema operacional seguindo as instruções no site oficial:  
<https://nodejs.org/en/>

A ABI codifica informações sobre funções e eventos de contratos inteligentes. Ela atua como uma interface entre o bytecode em nível EVM e o código do programa de contrato inteligente de alto nível. Para interagir com um contrato inteligente implantado na blockchain Ethereum, programas externos requerem uma ABI e o endereço do contrato inteligente.

**Ganache**

Às vezes, não é possível testar na testnet, e a mainnet obviamente não é um lugar para testar contratos. Uma rede privada pode ser demorada para configurar em alguns casos. O **Ganache** é uma blockchain pessoal simulada com uma interface de linha de comando ou uma interface gráfica amigável para visualizar transações, blocos e detalhes relevantes. Esta é uma blockchain pessoal totalmente funcional que suporta a simulação da Ethereum e seus diferentes hard forks, como Homestead, Byzantium, Istanbul, Petersburg ou London. Está disponível como CLI e também como GUI.

**ganache-cli**

Ganache é útil quando testes rápidos são necessários e nenhuma testnet está disponível. Ele simula o comportamento do cliente geth da Ethereum e permite desenvolvimento e testes mais rápidos. A linha de comando do Ganache está disponível via npm como um pacote Node.js. Sendo assim, o Node.js já deve ter sido instalado e o gerenciador de pacotes npm deve estar disponível. O ganache pode ser instalado usando este comando:

$ npm install -g ganache

Para iniciar a interface de linha de comando do ganache, simplesmente emita este comando, mantenha-o executando em segundo plano e abra outro terminal para trabalhar no desenvolvimento de contratos:

$ ganache

Quando o Ganache for executado, ele gerará automaticamente 10 contas e chaves privadas, junto com uma carteira HD. Ele começará a escutar conexões de entrada na porta TCP 8545.

O **ganache-cli** possui várias flags para personalizar a cadeia simulada de acordo com seus requisitos. Por exemplo, a flag -a permite especificar o número de contas a serem geradas na inicialização. Da mesma forma, -b permite ao usuário configurar o tempo de bloco para mineração.

Ajuda detalhada está disponível usando o seguinte comando:

$ ganache –help

O Ganache é uma ferramenta de linha de comando. No entanto, às vezes, é desejável ter uma ferramenta totalmente funcional com uma interface gráfica rica.

**Ganache UI**

O Ganache UI é baseado em uma implementação JavaScript da blockchain Ethereum, com um explorador de blocos embutido e mineração, facilitando o teste local no sistema. Você pode visualizar transações, blocos e endereços em detalhes na interface gráfica. Ele pode ser baixado em:  
<https://www.trufflesuite.com/ganache>

Quando você inicia o Ganache pela primeira vez, ele perguntará se você deseja criar uma blockchain rápida ou criar um novo espaço de trabalho que pode ser salvo, e também possui opções de configuração avançada:

<**IMAGEM**>

**Figura 11.1: Criando um espaço de trabalho**

Selecione a opção **QUICKSTART** ou **NEW WORKSPACE**, conforme necessário. Para uma configuração temporária mais rápida com opções padrão, o que pode ser útil para testes simples, você pode escolher **QUICKSTART**. Escolheremos **NEW WORKSPACE**, pois queremos explorar recursos mais avançados.

Se **NEW WORKSPACE** for selecionado, há várias opções disponíveis para configurar a blockchain. Uma das opções de configuração é **WORKSPACE NAME**, onde você pode especificar um nome para seu projeto. Além disso, projetos Truffle também podem ser adicionados aqui — cobriremos o Truffle em mais detalhes mais adiante no capítulo.

Outras opções incluem **SERVER**, **ACCOUNTS & KEYS**, **CHAIN** e **ADVANCED**. A aba **SERVER** é usada para configurar a conectividade RPC especificando o nome do host, número da porta e ID da rede:

<**IMAGEM**>

**Figura 11.2: Configuração do servidor**

**ACCOUNTS & KEYS** fornece opções para configurar o saldo e o número de contas a serem geradas. A opção **CHAIN** fornece uma interface de configuração para especificar o limite de gás, preço do gás e o hard fork que precisa ser simulado, como Byzantine ou Petersburg.

A opção **ADVANCED** está disponível para configurar configurações relacionadas a logs e análises. Uma vez que você tenha todas as opções de configuração definidas, salve o espaço de trabalho selecionando **SAVE WORKSPACE**, e a visualização principal de transações da blockchain pessoal do Ganache será exibida:

<**IMAGEM**>

**Figura 11.3: Visualização principal do Ganache**

Com isso, concluímos nossa introdução ao Ganache, uma ferramenta essencial usada no desenvolvimento blockchain. Agora passaremos para diferentes frameworks de desenvolvimento que estão disponíveis para a Ethereum.

**Truffle**

**Truffle** (disponível em <https://www.trufflesuite.com>) é um ambiente de desenvolvimento que facilita e simplifica o teste e a implantação de contratos Ethereum. O Truffle fornece compilação e vinculação de contratos juntamente com um framework de testes automatizados usando **Mocha** e **Chai**. Ele também torna mais fácil implantar os contratos em qualquer blockchain Ethereum privada, pública ou de teste (testnet). Além disso, é fornecido um pipeline de ativos, que facilita o processamento de todos os arquivos JavaScript, tornando-os prontos para uso por um navegador.

Antes da instalação, presume-se que o Node.js esteja disponível, o que pode ser consultado conforme mostrado aqui:

$ node -v

v16.15.0

Se o Node.js ainda não estiver disponível, a instalação do Node.js é necessária primeiro para instalar o Truffle. A instalação do Truffle é muito simples e pode ser feita usando o seguinte comando via Node Package Manager (npm):

$ npm install truffle –g

Isso levará alguns minutos; uma vez instalado, o comando truffle pode ser usado para exibir informações de ajuda e verificar se foi instalado corretamente.

Digite truffle no terminal para exibir a ajuda de uso:

$ truffle

Isso exibirá todas as opções que o Truffle suporta. Alternativamente, o repositório está disponível em <https://github.com/trufflesuite/truffle>, o qual pode ser clonado localmente para instalar o Truffle. O Git pode ser usado para clonar o repositório usando o seguinte comando:

$ git clone <https://github.com/trufflesuite/truffle.git>

Usaremos o Truffle mais adiante no **Capítulo 12, Desenvolvimento Web3 Usando Ethereum**, para testar e implantar contratos inteligentes na blockchain Ethereum. Por ora, continuaremos explorando alguns dos frameworks usados para desenvolvimento na blockchain Ethereum.

**Drizzle**

**Drizzle** é uma coleção de bibliotecas de frontend que permite o desenvolvimento fácil de interfaces web para aplicações descentralizadas. É baseado no **Redux store** e permite a sincronização contínua dos dados do contrato e da transação.

O Drizzle é instalado com o seguinte comando:

$ npm install --save @drizzle/store

O desenvolvimento de interface de usuário Web (UI) é uma parte importante do desenvolvimento de dApps. Sendo assim, muitas técnicas e ferramentas web, que vão desde HTML e JavaScript simples até frameworks avançados como **Redux** e **React**, são usadas para desenvolver interfaces web para dApps.

**Outras ferramentas**

Existem muitas outras ferramentas e frameworks disponíveis. Informações sobre essas ferramentas estão disponíveis aqui:  
<https://ethereum.org/en/developers/local-environment/>. No entanto, discutiremos algumas aqui:

* **Embark** é uma plataforma de desenvolvimento completa e poderosa para construção e implantação de aplicações descentralizadas. É usada para desenvolvimento, configuração, testes e implantação de contratos inteligentes. Também se integra com **Swarm**, **IPFS** e **Whisper**. Há também uma interface web chamada **Cockpit** disponível com o Embark, que fornece um ambiente de desenvolvimento integrado para desenvolvimento e depuração fáceis de aplicações descentralizadas.
* **Brownie** é um framework baseado em Python para desenvolvimento e testes de contratos inteligentes Ethereum. Ele tem suporte completo ao Solidity e Vyper com ferramentas relevantes de testes e depuração. Mais informações estão disponíveis em:  
  <https://eth-brownie.readthedocs.io/en/stable/>
* **Waffle** é um framework para desenvolvimento e testes de contratos inteligentes. Alega ser mais rápido que o Truffle. Este framework permite desenvolvimento, depuração e testes de dApps em Solidity e Vyper. É baseado em **Ethers.js**. Mais detalhes estão disponíveis no site oficial:  
  <https://getwaffle.io>
* O **toolkit da OpenZeppelin** possui um conjunto robusto de ferramentas que permitem o desenvolvimento fácil de contratos inteligentes. Ele oferece suporte para compilação, implantação, atualização e interação com contratos inteligentes. Mais informações estão disponíveis em:  
  <https://openzeppelin.com/sdk/>

Nesta seção, cobrimos alguns dos frameworks principais que são usados no ecossistema Ethereum para desenvolvimento. Na próxima seção, exploraremos quais ferramentas estão disponíveis para escrever e implantar contratos inteligentes.

**Desenvolvimento e implantação de contratos**

Existem várias etapas que precisam ser seguidas para desenvolver e implantar contratos. De forma ampla, essas etapas podem ser divididas em três: **escrever**, **testar** e **implantar**. Após a implantação, o próximo passo opcional é criar a interface de usuário (UI) e apresentá-la aos usuários finais por meio de um servidor web. Cobriremos isso no próximo capítulo. Uma interface web às vezes não é necessária em contratos onde nenhuma entrada humana ou monitoramento é exigido, mas geralmente há uma necessidade de criar uma interface web para que os usuários finais possam interagir com o contrato usando interfaces web familiares.

**Escrevendo contratos inteligentes**

A etapa de escrita diz respeito à escrita do código-fonte do contrato em Solidity. Isso pode ser feito em qualquer editor de texto. Existem vários plugins e complementos disponíveis para o Vim no Linux, Atom e outros editores que fornecem realce de sintaxe e formatação para código-fonte Solidity.

O **Visual Studio Code** tornou-se bastante popular e é comumente usado para desenvolvimento em Solidity. Existem plugins Solidity disponíveis que permitem realce de sintaxe, formatação e IntelliSense. Além disso, o Truffle está disponível como um plugin, o que melhora consideravelmente a experiência do desenvolvedor.

Ambos podem ser instalados via a opção Extensions no Visual Studio Code.

**Testando contratos inteligentes**

O teste geralmente é realizado por meios automatizados. Mais cedo neste capítulo, você foi apresentado ao Truffle, que usa o framework **Mocha** para testar contratos. No entanto, testes funcionais manuais também podem ser realizados usando o **Remix IDE**, que foi discutido no **Capítulo 10, Ethereum na Prática**, e executando funções manualmente e validando os resultados. Cobriremos isso no **Capítulo 12, Desenvolvimento Web3 Usando Ethereum**.

O plugin Solidity para Visual Studio está disponível no Marketplace do Visual Studio em:  
<https://marketplace.visualstudio.com/items?itemName=JuanBlanco.solidity>

O Truffle para VS Code está disponível aqui:  
<https://marketplace.visualstudio.com/items?itemName=trufflesuite-csi.truffle-vscode>

**Implantando contratos inteligentes**

Uma vez que o contrato esteja verificado, funcionando e testado em um ambiente simulado (por exemplo, Ganache) ou em uma rede privada, ele pode ser implantado em uma testnet pública como a **Ropsten** e, eventualmente, na blockchain ao vivo (mainnet). Cobriremos todas essas etapas, incluindo verificação, desenvolvimento e criação de uma interface web, no próximo capítulo.

Agora que cobrimos quais ferramentas podem ser usadas para escrever contratos inteligentes Solidity, vamos apresentar a linguagem Solidity. Esta será uma breve introdução ao Solidity, que deve fornecer o conhecimento base necessário para escrever contratos inteligentes. A sintaxe da linguagem é muito semelhante ao **C** e **JavaScript**, e é bastante fácil de programar. Começaremos explorando como é um contrato inteligente escrito na linguagem Solidity.

**A linguagem Solidity**

Solidity é a linguagem específica de domínio escolhida para programar contratos na Ethereum. Sua sintaxe é próxima tanto do JavaScript quanto do C. A Solidity evoluiu para uma linguagem madura nos últimos anos e é bastante fácil de usar, mas ainda tem um longo caminho a percorrer antes que possa se tornar avançada, padronizada e rica em recursos, como outras linguagens bem estabelecidas, como Java, C e C#. No entanto, é a linguagem mais amplamente utilizada atualmente disponível para programação de contratos.

É uma linguagem **estaticamente tipada**, o que significa que a verificação de tipo de variável na Solidity é realizada em tempo de compilação. Cada variável, seja de estado ou local, deve ser especificada com um tipo em tempo de compilação. Isso é benéfico no sentido de que qualquer validação e verificação são concluídas em tempo de compilação e certos tipos de bugs, como a interpretação de tipos de dados, podem ser capturados mais cedo no ciclo de desenvolvimento em vez de em tempo de execução, o que poderia ser custoso, especialmente no paradigma de blockchain/contratos inteligentes.

Outros recursos da linguagem incluem herança, bibliotecas e a capacidade de definir tipos de dados compostos. A Solidity também é chamada de linguagem orientada a contratos. Em Solidity, os contratos são equivalentes ao conceito de classes em outras linguagens de programação orientadas a objetos.

Nas subseções seguintes, veremos os componentes de um arquivo de código-fonte Solidity, o que é importante cobrir antes de passarmos para a escrita de contratos inteligentes na próxima seção.

Para lidar com problemas de compatibilidade que podem surgir de versões futuras do **solc**, **pragma** pode ser usado para especificar a versão do compilador compatível, como no exemplo a seguir:

pragma solidity ^0.8.0

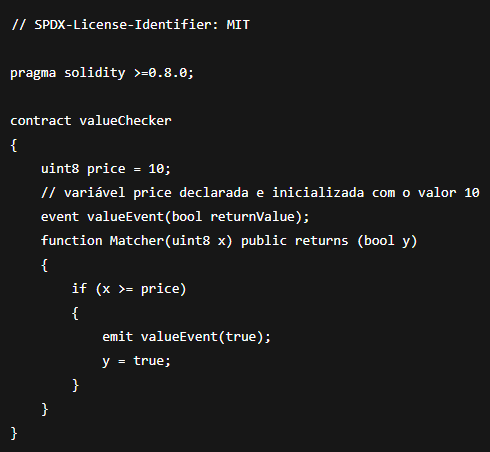
Isso garantirá que o arquivo-fonte não seja compilado com versões inferiores a 0.8.0.

**import** em Solidity permite a importação de símbolos de arquivos Solidity existentes para o escopo global atual. Isso é semelhante às instruções de importação disponíveis em JavaScript, como no exemplo a seguir:

import "module-name";

Comentários podem ser adicionados ao arquivo de código-fonte Solidity de maneira semelhante à linguagem C. Comentários de várias linhas são delimitados por /\* e \*/, enquanto comentários de uma única linha começam com //.

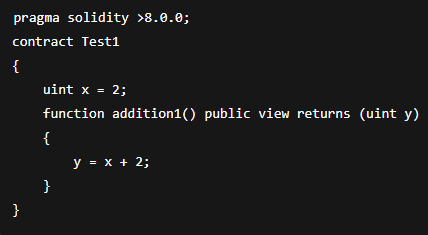
Um exemplo de programa Solidity é o seguinte, mostrando o uso de pragma, import e comentários:



Nesta seção, examinamos como é o código Solidity de um contrato inteligente. Agora é hora de aprender sobre a linguagem Solidity.

**Funções**

Funções são blocos de código dentro de um contrato inteligente. Por exemplo, observe o seguinte bloco de código:



Observe que um identificador de licença SPDX precisa ser adicionado também como primeira linha, caso contrário o compilador Solidity gerará um aviso. Se tudo estiver em ordem, o programa será compilado com sucesso.

No exemplo de código anterior, com o contrato Test1, definimos uma função chamada addition1(), que retorna um número inteiro sem sinal após adicionar 2 ao valor fornecido por meio da variável x, inicializada logo antes da função.

Neste caso, 2 é fornecido via a variável x, e a função retornará 4 somando 2 ao valor de x.  
É uma função simples, mas demonstra como funções funcionam e quais são seus diferentes elementos.

Existem dois tipos de funções – **internas** e **externas**:

* **Funções internas** só podem ser usadas dentro do contexto do contrato atual.
* **Funções externas** podem ser chamadas via chamadas de função externas.

Uma função em Solidity pode ser marcada como **constante**. Funções constantes não podem alterar nada no contrato; elas apenas retornam valores quando são invocadas e não consomem gás. Esta é a implementação prática do conceito de chamada (*call*), como discutido no capítulo anterior.

Funções em Solidity são módulos de código associados a um contrato. Funções são declaradas com um nome, parâmetros opcionais, modificadores de acesso, modificadores de mutabilidade de estado e um tipo de retorno opcional.

A sintaxe para definir uma função é mostrada a seguir:

function <nome da função>(<parâmetros>) <especificador de visibilidade> <modificador de mutabilidade de estado> returns (<tipo de dado de retorno> <nome da variável>)

{

<corpo da função>

}

Ela contém os seguintes elementos:

* **Assinatura da função**: Funções em Solidity são identificadas por sua assinatura, que são os quatro primeiros bytes do hash Keccak-256 de sua string de assinatura completa. Isso também é visível no Remix IDE. Por exemplo, a assinatura da função Matcher que vimos anteriormente são os quatro primeiros bytes do hash Keccak-256 de 32 bytes da função:

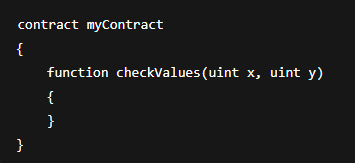
{

"f9d55e21": "Matcher(uint8)"

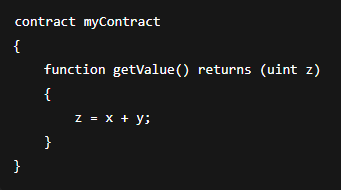
}

Neste exemplo, a função Matcher tem o hash de assinatura f9d55e21. Esta informação é útil para construir interfaces.

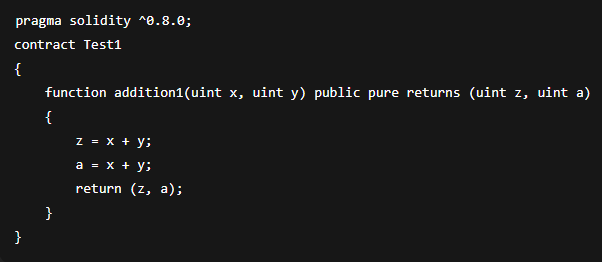
* **Parâmetros de entrada de uma função**: Os parâmetros de entrada de uma função são declarados na forma <tipo de dado> <nome do parâmetro>. Este exemplo esclarece o conceito, onde uint x e uint y são parâmetros de entrada da função checkValues:



**Parâmetros de saída de uma função**: Os parâmetros de saída de uma função são declarados na forma <tipo de dado> <nome do parâmetro>. Este exemplo mostra uma função simples que retorna um valor uint:

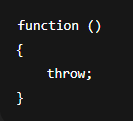


Uma função pode retornar vários valores, assim como pode receber múltiplas entradas. No exemplo anterior, a função getValue retorna apenas um valor, mas uma função pode retornar até 14 valores de diferentes tipos de dados. Os nomes dos parâmetros de retorno não utilizados podem ser opcionalmente omitidos. Um exemplo de tal função poderia ser:



Aqui, quando o código for executado, ele receberá dois parâmetros como entrada, x e y, somará ambos, atribuindo-os a z e a, e finalmente retornará z e a. Por exemplo, se fornecermos 1 e 1 para x e y, respectivamente, então, quando as variáveis z e a forem retornadas pela função, ambas conterão 2 como resultado.

* **Chamadas internas de função**: Funções dentro do contexto do contrato atual podem ser chamadas internamente de maneira direta. Essas chamadas resultam em chamadas JUMP simples no nível de bytecode da EVM.
* **Chamadas externas de função**: Chamadas de função externas são feitas via chamadas de mensagem de um contrato para outro contrato. Nesse caso, todos os parâmetros da função são copiados para a memória. Se uma chamada a uma função interna for feita usando a palavra-chave this, ela também é considerada uma chamada externa. A variável this é um ponteiro que se refere ao contrato atual. Ela é explicitamente convertível para um endereço, e todos os membros de um contrato são herdados do endereço.
* **Funções fallback**: Esta é uma função sem nome em um contrato, sem argumentos e sem dados de retorno. Esta função é executada toda vez que ether é recebido. É necessário que seja implementada dentro de um contrato se este contrato for projetado para receber ether; caso contrário, uma exceção será lançada e o ether será devolvido. Esta função também é executada se nenhuma outra assinatura de função corresponder no contrato. Se o contrato for esperado para receber ether, então a função fallback deve ser declarada com o modificador payable.
* O payable é obrigatório; caso contrário, esta função não poderá receber nenhum ether. Esta função pode ser chamada usando o método address.call(), como, por exemplo:



Neste caso, se a função fallback for chamada de acordo com as condições descritas anteriormente, ela chamará throw, o que reverterá o estado para o que era antes da chamada. Também pode ser alguma outra construção em vez de throw; por exemplo, pode registrar um evento que pode ser usado como um alerta para retornar o resultado da chamada ao aplicativo chamador.

* **Funções modificadoras**: Essas funções são usadas para alterar o comportamento de uma função e podem ser chamadas antes de outras funções. Normalmente, elas são usadas para verificar algumas condições ou validações antes de executar a função. O caractere \_ (sublinhado) é usado nas funções modificadoras e será substituído pelo corpo real da função quando o modificador for chamado. Basicamente, ele simboliza a função que precisa ser protegida. Este conceito é semelhante às funções guardas em outras linguagens.
* **Função construtora (constructor)**: Esta é uma função opcional que tem o mesmo nome do contrato e é executada uma vez quando um contrato é criado. Funções construtoras não podem ser chamadas posteriormente pelos usuários, e apenas um construtor é permitido em um contrato. Isso implica que nenhuma funcionalidade de sobrecarga está disponível.
* **Especificadores de visibilidade de função (modificadores de acesso/níveis de acesso)**: Funções podem ser definidas com quatro especificadores de acesso, conforme a seguir:
  + **external**: Essas funções são acessíveis a partir de outros contratos e transações. Elas não podem ser chamadas internamente, a menos que se use a palavra-chave this.
  + **public**: Por padrão, as funções são públicas. Elas podem ser chamadas internamente ou por meio de mensagens.
  + **internal**: Funções internas são visíveis para outros contratos derivados do contrato pai.
  + **private**: Funções privadas são visíveis apenas para o mesmo contrato em que foram declaradas.
* **Modificadores de função**:
  + pure: Este modificador proíbe acesso ou modificação ao estado.
  + view: Este modificador impede qualquer modificação no estado.
  + payable: Este modificador permite o envio de ether junto com uma chamada.
  + virtual: Permite que o comportamento da função ou do modificador seja alterado em contratos derivados.
  + override: Indica que esta função, modificador ou variável de estado pública altera o comportamento de uma função ou modificador em um contrato base.

Podemos ver esses elementos em um exemplo mostrado abaixo:

function orderMatcher (uint x)

private view returns (bool returnValue)

No exemplo de código acima, function é a palavra-chave usada para declarar a função. orderMatcher é o nome da função, uint x é um parâmetro opcional, private é o modificador de acesso que controla o acesso à função a partir de contratos externos, view é uma palavra-chave opcional usada para especificar que esta função não altera nada no contrato, sendo usada apenas para recuperar valores, e returns (bool returnValue) é o tipo de retorno opcional da função.

**Variáveis**

Assim como em qualquer linguagem de programação, variáveis em Solidity são locais de memória nomeados que armazenam valores em um programa. Existem três tipos de variáveis em Solidity: **variáveis locais**, **variáveis globais** e **variáveis de estado**.

**Variáveis locais**

Essas variáveis têm escopo limitado apenas à função em que são declaradas. Em outras palavras, seus valores estão presentes apenas durante a execução da função na qual foram declaradas.

**Variáveis globais**

Essas variáveis estão disponíveis globalmente, pois existem no namespace global. Elas são usadas para realizar várias funções como codificação ABI, funções criptográficas e consulta de informações da blockchain e das transações.

A Solidity fornece várias variáveis globais que estão sempre disponíveis no namespace global. Essas variáveis fornecem informações sobre blocos e transações. Adicionalmente, funções criptográficas, codificação/decodificação ABI e variáveis relacionadas a endereços estão disponíveis.

Um subconjunto de variáveis globais disponíveis é mostrado a seguir:

* Retorna o número atual do bloco:

block.number

* Retorna o preço do gás da transação:

tx.gasprice // (uint)

* Retorna o endereço do minerador do bloco atual:

block.coinbase // (address payable)

* Retorna o timestamp atual do bloco:

now // (uint)

* Retorna a dificuldade atual do bloco:

block.difficulty // (uint)

Um subconjunto de funções disponíveis é mostrado a seguir:

* A função abaixo é usada para calcular o hash Keccak-256 do argumento fornecido:

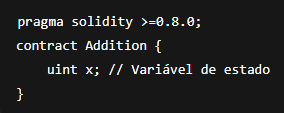
keccak256(...) returns (bytes32)

* Esta função retorna o endereço associado à chave pública da assinatura da curva elíptica:

ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)

**Variáveis de estado**

Variáveis de estado têm seus valores armazenados permanentemente no armazenamento do contrato inteligente. Variáveis de estado são declaradas fora do corpo de uma função, e permanecem disponíveis ao longo do contrato, dependendo da acessibilidade atribuída a elas e enquanto o contrato persistir:



Aqui, x é uma variável de estado cujo valor será armazenado no armazenamento do contrato.

Existem três tipos de variáveis de estado, com base no escopo de visibilidade:

* **Private**: Estas variáveis são acessíveis apenas internamente, dentro do contrato no qual foram originalmente definidas. Elas também não são acessíveis por nenhum contrato derivado do contrato original.
* **Public**: Estas variáveis fazem parte da interface do contrato. Em palavras simples, qualquer um pode obter o valor dessas variáveis. Elas são acessíveis dentro do contrato internamente usando a palavra-chave this. Também podem ser chamadas a partir de outros contratos e transações. Uma função "getter" é automaticamente criada para todas as variáveis públicas.
* **Internal**: Estas variáveis são acessíveis apenas internamente dentro do contrato no qual são definidas. Em contraste com variáveis de estado private, elas também são acessíveis a partir de qualquer contrato derivado do contrato original (pai).

Existem várias outras variáveis globais disponíveis. Uma lista abrangente e seus detalhes podem ser encontrados na documentação oficial da Solidity:  
<https://solidity.readthedocs.io/en/latest/units-and-global-variables.html>

Existem dois modificadores para variáveis de estado:

* **constant**: Impede atribuição, exceto durante a inicialização.
* **immutable**: Permite exatamente uma atribuição no momento da construção e é constante posteriormente.

**Tipos de dados**

Solidity possui duas categorias de tipos de dados — **tipos de valor** (*value types*) e **tipos de referência** (*reference types*):

* **Tipos de valor** são variáveis que são sempre passadas por valor. Isso significa que os tipos de valor armazenam seu valor ou dado diretamente, permitindo que o valor de uma variável armazenado na memória seja acessado diretamente acessando a variável.
* **Tipos de referência** armazenam o endereço do local de memória onde o valor é armazenado. Isso é diferente dos tipos de valor, que armazenam o valor real da variável diretamente com a variável.

Lembre-se do Capítulo 9, *Arquitetura Ethereum*, onde vimos que a EVM pode ler e escrever dados em diferentes locais. O local específico usado para armazenar valores de uma variável depende do tipo de dado da variável e onde ela foi declarada. Por exemplo, variáveis de parâmetros de função são armazenadas na memória, enquanto variáveis de estado são armazenadas no armazenamento.

Agora descreveremos os **tipos de valor** em detalhes.

**Tipos de valor**

Os tipos de valor incluem principalmente Booleanos, inteiros, endereços e literais, os quais são explicados detalhadamente aqui.

**Booleano**

Este tipo de dado tem dois valores possíveis: true ou false, por exemplo:

bool v = true;

bool v = false;

Esta instrução atribui o valor true ou false à variável v, dependendo da atribuição.

**Inteiros**

Este tipo de dado representa inteiros. Várias palavras-chave são usadas para declarar tipos inteiros:

* **int**: Inteiro com sinal. int8 a int256, o que significa que palavras-chave estão disponíveis de int8 até int256 em incrementos de 8, por exemplo, int8, int16, int24.
* **uint**: Inteiro sem sinal. uint8, uint16 até uint256. O uso depende de quantos bits precisam ser armazenados na variável.

Por exemplo, neste código, observe que uint é um apelido para uint256:

uint256 x;

uint y;

uint256 z;

Esses tipos também podem ser declarados com a palavra-chave constant, o que significa que nenhum slot de armazenamento será reservado pelo compilador para essas variáveis. Neste caso, cada ocorrência será substituída pelo valor real:

uint constant z = 10 + 10;

**Endereço**

Este tipo de dado armazena um valor de 160 bits (20 bytes). Este tipo possui vários membros que podem ser usados para interagir com e consultar contratos. Esses membros são descritos aqui:

* **balance**: O membro balance retorna o saldo do endereço em Wei.
* **send**: Este membro é usado para enviar uma quantidade de ether a um endereço (endereço Ethereum de 160 bits) e retorna true ou false dependendo do resultado da transação, por exemplo:

address to = 0x6414cc08d148dce9ebf5a2d0b7c220ed2d3203da;

address from = this;

if (to.balance < 10 && from.balance > 50) to.send(20);

* **Funções call**: As funções call, callcode e delegatecall são fornecidas para interagir com funções que não possuem uma ABI. Estas funções devem ser usadas com cautela, pois não são seguras devido ao impacto na segurança e na tipagem dos contratos.

**Tipos de valor de array (arrays de bytes de tamanho fixo e dinâmico)**

Solidity possui arrays de bytes de tamanho fixo e de tamanho dinâmico. As palavras-chave de tamanho fixo vão de bytes1 a bytes32, enquanto palavras-chave de tamanho dinâmico incluem bytes e string. A palavra-chave bytes é usada para dados brutos de bytes, e string é usada para strings codificadas em UTF-8. Como esses arrays são retornados por valor, sua chamada implicará em custo de gás.

Um exemplo de array estático (tamanho fixo) é:

bytes32[10] bankAccounts;

Um exemplo de array de tamanho dinâmico é:

bytes32[] trades;

length é um membro dos tipos de array e retorna o comprimento do array de bytes:

trades.length;

**Literais**

Literais são usados para representar um valor fixo. Existem diferentes tipos de literais, descritos a seguir:

* **Literais inteiros**: São uma sequência de números decimais no intervalo de 0–9. Um exemplo:

uint8 x = 2;

**Literais de string**: Este tipo especifica um conjunto de caracteres escritos com aspas duplas ou simples. Exemplos:

'packt'

"packt"

**Literais hexadecimais**: São prefixados com a palavra-chave hex e especificados entre aspas duplas ou simples. Exemplo:

(hex'AABBCC');

**Enums**: Permitem a criação de tipos definidos pelo usuário. Exemplo:

enum Order {Filled, Placed, Expired };

Order private ord;

ord = Order.Filled;

A conversão explícita para e a partir de todos os tipos inteiros é permitida com enums.

**Tipos de referência**

Como o nome sugere, esses tipos são passados por referência e são discutidos na seção a seguir. Também são conhecidos como **tipos complexos**. Tipos de referência incluem arrays, structs e mappings.

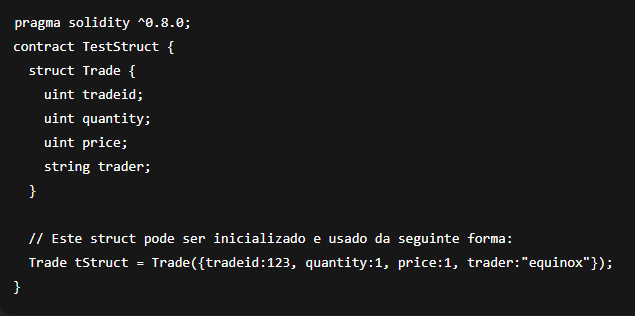
Ao usar tipos de referência, é essencial especificar explicitamente a área de armazenamento onde o tipo será armazenado, por exemplo, **memory**, **storage** ou **calldata**.

**Arrays**

Arrays representam um conjunto contíguo de elementos de mesmo tamanho e tipo dispostos em um local de memória. O conceito é o mesmo de qualquer outra linguagem de programação. Arrays têm dois membros, chamados length e push.

**Structs**

Esses constructos podem ser usados para agrupar um conjunto de tipos de dados diferentes sob um grupo lógico. Eles podem ser usados para definir novos tipos personalizados, conforme mostrado no exemplo a seguir:



No código anterior, declaramos um struct chamado Trade que possui quatro campos. tradeid, quantity e price são do tipo uint, enquanto trader é do tipo string. Uma vez que o struct é declarado, podemos inicializá-lo e usá-lo. Inicializamos utilizando Trade tStruct e atribuímos 123 ao tradeid, 1 ao quantity, e "equinox" ao trader.

Às vezes, é desejável escolher o local de armazenamento dos dados da variável. Essa escolha permite melhor gerenciamento do gasto de gás. Podemos usar o nome do local de dados para especificar onde um determinado tipo de dado complexo será armazenado. Dependendo da anotação padrão ou especificada, o local pode ser storage, memory ou calldata. Isso é aplicável a arrays e structs e pode ser especificado usando as palavras-chave storage ou memory. calldata se comporta quase como memory. É uma área temporária e não modificável que pode ser usada para armazenar argumentos de função.

Por exemplo, no exemplo de structs anterior, se quisermos usar apenas memória (temporariamente), podemos fazer isso usando a palavra-chave memory ao utilizar a estrutura e atribuir valores aos campos do struct, como mostrado aqui:

Trade memory tStruct;

tStruct.tradeid = 123;

Como a cópia entre memory e storage pode ser bastante custosa, especificar um local pode ser útil para controlar o gasto com gás em alguns momentos.

Parâmetros de funções externas usam memória calldata. Por padrão, parâmetros de funções são armazenados em memory, enquanto todas as outras variáveis locais usam storage. Variáveis de estado, por outro lado, são obrigadas a usar storage.

**Mappings**

Mappings são usados para mapeamento de chave-valor. Essa é uma forma de associar um valor a uma chave. Todos os valores neste mapa já estão inicializados com zero, como no exemplo a seguir:

mapping (address => uint) offers;

Este exemplo mostra que offers foi declarado como um mapping. Outro exemplo torna isso ainda mais claro:

mapping (string => uint) bids;

bids["packt"] = 10;

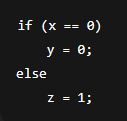
Isto é basicamente um dicionário ou tabela de hash, onde valores do tipo string são mapeados para valores inteiros. O mapping chamado bids tem a string "packt" mapeada para o valor 10.

**Estruturas de controle**

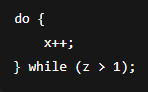
As estruturas de controle disponíveis na linguagem Solidity são: if...else, do, while, for, break, continue e return. Elas funcionam exatamente da mesma maneira que em outras linguagens, como C ou JavaScript.

Alguns exemplos são mostrados aqui:

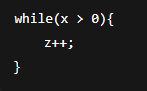
* **if**: Se x for igual a 0, então atribuir o valor 0 a y, senão atribuir 1 a z:



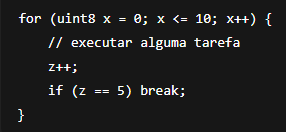
**do**: Incrementar x enquanto z for maior que 1:



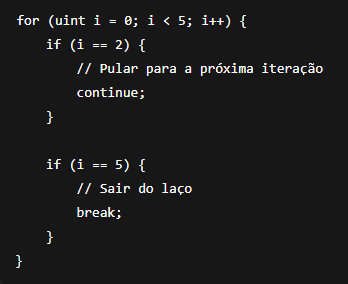
**while**: Incrementar z enquanto x for maior que 0:



**for**, **break** e **continue**: Executar algum trabalho enquanto x for menor ou igual a 10. Este laço for será executado 10 vezes; se z for 5, então o laço será interrompido com break:



continue pode ser usado em situações onde queremos iniciar imediatamente a próxima iteração do laço sem executar o restante do código. Por exemplo, observe o código a seguir:



Esse código continuará a execução normalmente, mas quando a condição i == 2 for satisfeita, ele irá imediatamente para a próxima iteração do laço, sem executar o restante do bloco.

* **return**: return é usado para parar a execução de uma função e retornar um valor opcional. Por exemplo:

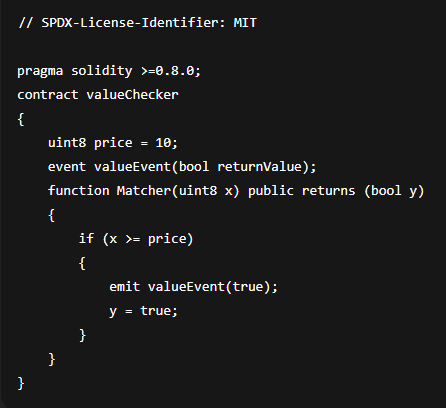
return 0;

Este comando encerrará a execução e retornará o valor 0.

**Eventos**

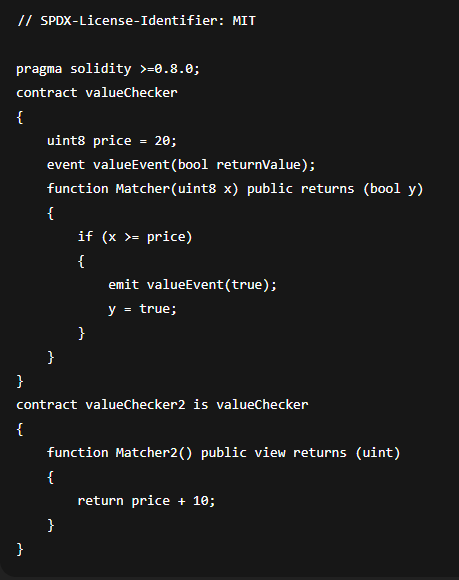
Eventos em Solidity podem ser usados para registrar certos eventos nos logs da EVM. Eles são bastante úteis quando interfaces externas precisam ser notificadas de qualquer mudança ou evento no contrato. Esses logs são armazenados na blockchain nos registros de transações. Os logs não podem ser acessados pelos contratos, mas são usados como um mecanismo para notificar mudanças de estado ou a ocorrência de um evento (atendimento de uma condição) no contrato.

Em um exemplo simples aqui, o evento valueEvent retornará true se o parâmetro x passado para a função Matcher for igual ou maior que 10:

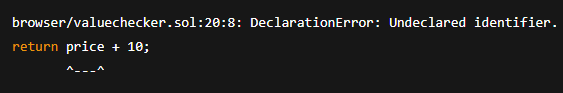


**Herança**

Herança é suportada em Solidity. A palavra-chave is é usada para derivar um contrato a partir de outro contrato. No exemplo a seguir, valueChecker2 é derivado do contrato valueChecker. O contrato derivado tem acesso a todos os membros não privados do contrato pai:



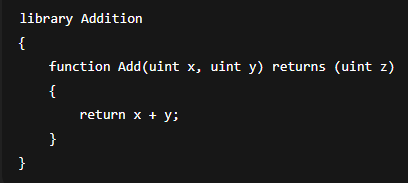
No exemplo acima, se uint8 price = 20 for alterado para uint8 private price = 20, então ele não será acessível pelo contrato valueChecker2. Isso ocorre porque agora o membro foi declarado como private e, portanto, não pode ser acessado por nenhum outro contrato. A mensagem de erro que você verá ao tentar compilar esse contrato será a seguinte:



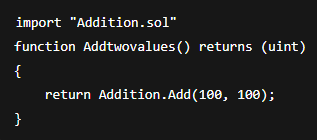
**Bibliotecas**

Bibliotecas são implantadas apenas uma vez em um endereço específico e seu código é chamado por meio dos opcodes **CALLCODE** ou **DELEGATECALL** da EVM. A ideia principal por trás das bibliotecas é a **reutilização de código**. Elas são semelhantes a contratos e atuam como contratos base para os contratos que as chamam.

Uma biblioteca pode ser declarada como mostrado no exemplo a seguir:



Essa biblioteca pode então ser chamada dentro de um contrato, como mostrado aqui. Primeiro, ela precisa ser importada e, em seguida, pode ser usada em qualquer lugar do código. Um exemplo simples é o seguinte:



Existem algumas limitações com bibliotecas; por exemplo:

* Elas **não podem ter variáveis de estado**;
* **Não podem herdar nem ser herdadas**;
* **Não podem receber ether**;

— o que é diferente dos contratos, que podem receber ether.

**Tratamento de erros**

A Solidity fornece várias funções para tratamento de erros. Por padrão, na Solidity, sempre que ocorre um erro, o estado não muda e **reverte** para o estado original.

Algumas construções e funções auxiliares disponíveis para tratamento de erros em Solidity são apresentadas a seguir:

* **assert**: Usada para verificar condições e lançar uma exceção se a condição não for satisfeita. assert é destinada para ser usada em erros internos e verificação de invariantes. Quando chamada, essa função resulta em um opcode inválido e quaisquer alterações no estado são revertidas.
* **require**: Semelhante ao assert, é usada para verificar condições e lança uma exceção se a condição não for atendida. A diferença é que require é usada para validar entradas, valores de retorno ou chamadas a contratos externos. O método também resulta na reversão ao estado original. Pode também receber um parâmetro opcional para fornecer uma mensagem de erro personalizada.
* **revert**: Este método **aborta a execução** e reverte a chamada atual. Também pode receber um parâmetro opcional para retornar uma mensagem de erro personalizada ao chamador.
* **try/catch**: Esta construção é usada para lidar com falhas em chamadas externas.
* **throw**: throw é usado para interromper a execução. Como resultado, todas as alterações de estado são revertidas. Neste caso, nenhum gás é retornado ao originador da transação porque **todo o gás restante é consumido**.

**Isso conclui uma breve introdução à linguagem Solidity.**

A linguagem é bastante rica e está em constante evolução. Documentação detalhada e diretrizes de codificação estão disponíveis online em:  
<http://solidity.readthedocs.io/en/latest/>

**Resumo**

Este capítulo começou com a introdução às ferramentas de desenvolvimento para Ethereum, como o **Ganache CLI**. A instalação do **Node.js** também foi apresentada, já que a maioria das ferramentas são baseadas em **JavaScript** e **Node.js**. Em seguida, discutimos alguns frameworks como o **Truffle**, junto com soluções locais de blockchain para desenvolvimento e testes, como **Ganache** e **Drizzle**. Também introduzimos a linguagem **Solidity** neste capítulo e exploramos diferentes conceitos, como **tipos de valor**, **tipos de referência**, **funções** e conceitos de **tratamento de erros**. Também aprendemos como escrever contratos usando Solidity.

No próximo capítulo, exploraremos o tema **Web3**, uma API JavaScript que é usada para se comunicar com a blockchain Ethereum.

**Junte-se a nós no Discord!**  
Para entrar na comunidade do Discord deste livro – onde você pode compartilhar feedback, fazer perguntas ao autor e saber sobre novos lançamentos – siga o código QR abaixo:

<https://packt.link/ips2H>