

formação

Mobiup 



nearx
Innovation School



Módulo 2



APRESENTADO POR
Felipe Bravo





Fellipe Bravo



fellipebravo



0xfbravo

- ▶ UFRRJ
- ▶ Background de empreendedorismo (NFTICKIT)
- ▶ Santander, Americanas S.A., Moss.Earth, BeatStars
- ▶ Pai da Luna

Desenvolvimento de Smart Contracts

Sumário

- ▶ Alinhamento
- ▶ Ferramentas
- ▶ Solidity
- ▶ Smart Contracts



Alinhamento

- ▶ Linguagem **tipada** e **compilada**
- ▶ Virtual Machine
- ▶ EVMs (Ethereum Virtual Machines)
- ▶ Redes EVM-compatible
- ▶ Smart Contracts
- ▶ Wallets: Chaves públicas / Chaves privadas
- ▶ Gas Fee

Ferramentas

Coding

 Visual Studio Code

 Hardhat

 Remix

Wallets


 Metamask

 TrustWallet

Projeto

 Node.JS

 Solidity Compiler

 Open Zeppelin

Solidity

- ▶ Projeto open-source, criado em **2014**
- ▶ **Compilação** e execução em EVMs
- ▶ Linguagem com **tipagem estática**
- ▶ Principal linguagem utilizada para desenvolvimento de **Smart Contracts**



Mais informações em:

<https://soliditylang.org/>

<https://github.com/ethereum/solidity>

Menções honrosas

- ▶ **Vyper**: Linguagem de tipagem dinâmica, similar ao Python. Maior foco em **segurança**.
- ▶ **YUL**: Linguagem intermediária, projetada para ser low-level, que pode ser utilizada para otimizar **Gas Fee** e códigos mais eficientes, visando **performance**.

Mais informações em:

<https://docs.vyperlang.org/en/stable/index.html>

<https://docs.soliditylang.org/en/v0.8.27/yul.html>

O que são Smart Contracts?

```

1 // SPDX-License-Identifier: Apache-2.0
2 pragma solidity ^0.8.24;
3
4 import "@openzeppelin/contracts-upgradeable/utils/PausableUpgradeable.sol";
5 import "@openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol";
6 import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
7 import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Utils.sol";
8 import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
9 import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
10 import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
11 import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
12 import "./access_controllable.sol";
13 import "./crowdfunding_project.sol";
14
15 /// @title AbstractCrowdfundingManager
16 /// @notice AbstractCrowdfundingManager is a proxy contract using Transparent Proxy Pattern and it's a gateway.
17 /// It's responsible for creating, upgrading, and managing projects.
18 /// @author MOBIUP
19 /// @custom:security-contact feliipe.bravo@mobiup.com.br
20 abstract contract AbstractCrowdfundingManager is Initializable, AccessControllable, PausableUpgradeable, UUPSUpgradeable, ReentrancyGuard {
21     // Variables
22     address public companyAdmin;
23     mapping(string => address) private _projectsProxies;
24
25     // Events
26     event ManagerInitialized(address indexed managerAddress);
27     event CompanyAdminUpdated(address indexed newAdmin);
28     event ProjectRegistered(string indexed projectUUID, address projectAddress);
29     event ProjectUpgraded(string indexed projectUUID, address projectAddress);
30
31     // Errors
32     error InvalidAddress(address addr);
33     error NotCompanyAdmin(address addr);
34     error AlreadyCompanyAdmin(address addr);
35     error ProjectDoesntExist(string projectUUID);
36     error ProjectAlreadyExists(string projectUUID);
37
38     // -----
39     // Writing functions
40     // -----
41     function initialize() public virtual initializer whenNotPaused() {
42         __UUPSUpgradeable_init();
43         __Pausable_init();
44         __AccessControl_init();
45
46         grantRole(DEFAULT_ADMIN_ROLE, msgSender());

```

Smart Contract

- ▶ É um código **binário** compilado para EVMs
- ▶ É um **programa** sendo executado em uma EVM
- ▶ Após o processo de **deploy** para uma rede, todos os usuários podem executar funções de **leitura** e de **escrita** nesse programa. *

* É possível limitar acesso a funções através de **roles** ou utilizando o padrão **ownable**. Ambos disponíveis em bibliotecas terceiras, como o **OpenZeppelin**

Mais informações em:

<https://docs.soliditylang.org/en/v0.8.27/introduction-to-smart-contracts.html>

<https://docs.openzeppelin.com/contracts/2.x/access-control>

Estrutura de Smart Contract

- ▶ Um smart contract em **Solidity**, é um arquivo salvo na extensão **.sol**
- ▶ A primeira instrução em um smart contract, em Solidity, será:

pragma solidity X.Y.Z;

Onde **X.Y.Z**, representa a versão do compilador Solidity.

Mais informações em:

<https://docs.soliditylang.org/en/v0.8.27/introduction-to-smart-contracts.html>

Nesse curso utilizaremos a versão

0.8.24

do compilador **Solidity**

Estrutura de Smart Contract

- ▶ Para iniciar um **SC** (Smart Contract), utilize a palavra reservada **contract**, como no exemplo:

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      // Informações do contrato
6  }
7
8  // Aqui fora podemos ter outros contratos, funções, variáveis, etc.
```

Mais informações em:

<https://docs.soliditylang.org/en/v0.8.27/introduction-to-smart-contracts.html>

Estrutura de Smart Contract

- ▶ Não se preocupe com as linhas verdes, elas são os nossos **comentários** no código e **não serão executadas**

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  ▼ contract MeuPrimeiroSC {
5      |    // Informações do contrato
6      |
7      |
8      |    // Aqui fora podemos ter outros contratos, funções, variáveis, etc.
```

Mais informações em:

<https://docs.soliditylang.org/en/v0.8.27/introduction-to-smart-contracts.html>

Conteúdo do SC

- ▶ No exemplo abaixo vemos os principais tipos de dados em Solidity: **string**, **uint**, **bool** e **address**

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string nome;
6      uint idade;
7      bool aprovado;
8      address enderecoWallet;
9  }
```

Mais informações em:

<https://docs.soliditylang.org/en/v0.8.27/types.html>

Strings

- ▶ As strings, **armazenam** ou **transferem** dados como texto corrido e são declaradas como no exemplo:

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string nome = "Fellipe Bravo";
6      uint idade;
7      bool aprovado;
8      address enderecowallet;
9  }
```

Inteiros

- ▶ Os inteiros, **armazenam** ou **transferem** dados como números de diversas grandezas.

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string nome = "Fellipe Bravo";
6      uint idade = 30;
7      int saldo = -4000;
8      bool aprovado;
9      address enderecoWallet;
10 }
```

Você sabe a diferença entre

uint / int

no compilador do **Solidity**?

Pontos flutuantes (decimais)

- ▶ Em Solidity, os pontos flutuantes ainda **não são totalmente suportados**. Eles podem ser **declarados**, mas não podem ser atribuídos **a** algum lugar ou **de** algum lugar.

Portanto, quando precisarmos representar um valor como por exemplo R\$ 40,00 devemos utilizar sua **representação em centavos**, sendo assim um número **inteiro**.

40,00 **é igual a** 4000

Booleans

- ▶ Os booleans, **armazenam** ou **transferem** dados como operações binárias: **verdadeiro** ou **falso**.

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string nome = "Fellipe Bravo";
6      uint idade = 30;
7      int saldo = -4000;
8      bool aprovado = false;
9      bool fezUmPix = true;
10     address enderecoWallet;
11 }
```

Endereços

- ▶ Os endereços, **armazenam** ou **transferem** dados como endereços públicos de wallets. Sempre iniciados por **0x**.

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string nome = "Fellipe Bravo";
6      uint idade = 30;
7      int saldo = -4000;
8      bool aprovado = false;
9      bool fezUmPix = true;
10     address enderecoWallet = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
11 }
```

Tamanho importa

- ▶ Em **2016**, um fork da rede Ethereum chamado Spurious Dragon introduziu um novo **EIP** (Ethereum Improvement Proposal)
- ▶ O **EIP-170** define um tamanho máximo de bytecodes para um **SC** na rede. Portanto, os **tipos de variáveis** importam **muito** no deploy.*

* Existem maneiras de otimizar o seu código em tempo de compilação e veremos isso nas próximas aulas.

Mais informações em:
<https://eips.ethereum.org/EIPS/eip-170>

Constants

- ▶ Vamos supor que eu quero um valor imutável no código, por exemplo uma taxa de split de pagamentos.
- ▶ Para isso, vamos utilizar a palavra reservada **constant** logo após o tipo da variável.

Exemplo de constants

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string nome = "Fellipe Bravo";
6      uint idade = 30;
7      int saldo = -4000;
8      bool aprovado = false;
9      bool fezUmPix = true;
10     address enderecoWallet = 0x5B38Da6a701c568545dCfCB03Fcb875f56beddC4;
11     uint constant taxaSplit = 10;
12 }
```

Visibilidade

Visibilidade

- ▶ Em Solidity e em outras linguagens, os atributos e funções declarados podem ou não ser acessados por outro SC e endereços.
- ▶ A visibilidade de um atributo ou função pode ser definida utilizando as palavras reservadas: **public** e **private**

Públicos

- ▶ Atributos e funções definidos como público podem ser **acessados** e **modificados** por qualquer SC ou endereço na rede.

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string public nome = "Fellipe Bravo";
6      uint public idade = 30;
7      int saldo = -4000;
8      bool aprovado = false;
9      bool fezUmPix = true;
10     address enderecoWallet = 0x5B38Da6a701c568545dCfcB03Fcb875f56beddC4;
11 }
```

Públicos

- ▶ Por padrão, em Solidity, quando um atributo é declarado como **public** funções de leitura do valor salvo são criadas automaticamente.

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string public nome = "Fellipe Bravo";
6      uint public idade = 30;
7      int saldo = -4000;
8      bool aprovado = false;
9      bool fezUmPix = true;
10     address enderecoWallet = 0x5B38Da6a701c568545dCfcB03Fcb875f56beddC4;
11 }
```


Privados

- ▶ Por outro lado, atributos definidos como **private** não poderão ser acessados. Apenas o próprio SC poderá acessar e modificá-lo.

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string public nome = "Fellipe Bravo";
6      uint public idade = 30;
7      int private saldo = -4000;
8      bool private aprovado = false;
9      bool private fezUmPix = true;
10     address private enderecoWallet = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
11 }
```

Funções

Funções

- ▶ As funções são responsáveis por execução de **pequenas tarefas**, onde você pode definir novos valores para variáveis ou até mesmo retornar novos valores para o usuário.
- ▶ As funções **podem ou não** receber atributos para a sua execução.
- ▶ As funções também tem uma **visibilidade** definida.

Funções: Atributos

- ▶ Ao declarar uma função, você deve declarar ou não seus parâmetros dentro dos parênteses, como no exemplo abaixo:

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string public nome = "Fellipe Bravo";
6      uint public idade = 30;
7      int private saldo = -4000;
8      bool private aprovado = false;
9      bool private fezUmPix = true;
10     address private enderecoWallet = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
11
12     function credito(int valor) private {
13         saldo += valor;
14     }
15
16     function debito(int valor) private {
17         saldo -= valor;
18     }
19
20 }
```

Funções: Atributos

- ▶ Para alguns tipos de atributo de funções, como Strings, alguns parâmetros especiais **precisam** ser adicionados através de palavras reservadas.
- ▶ **memory**: existência durante a execução do escopo, não persiste. **Ineficiente** para Gas Fee pois **permite alterações** durante execução do escopo.
- ▶ **calldata**: tipo de alocação de dados **especial**. Existência durante a execução do escopo, não persiste. **Eficiente** para Gas Fee dado sua **imutabilidade**.

Exemplo: string memory

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string public nome = "Fellipe Bravo";
6      uint public idade = 30;
7      int private saldo = -4000;
8      bool private aprovado = false;
9      bool private fezUmPix = true;
10     address private enderecoWallet = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
11
12     function modificarNome(string memory novoNome) public {
13         nome = novoNome;
14     }
15
16 }
```


Exemplo: string calldata

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string public nome = "Fellipe Bravo";
6      uint public idade = 30;
7      int private saldo = -4000;
8      bool private aprovado = false;
9      bool private fezUmPix = true;
10     address private enderecoWallet = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
11
12     function modificarNome(string calldata novoNome) public {
13         nome = novoNome;
14     }
15
16 }
```

Funções: Visibilidade

- Tal qual os atributos, as funções também suas visibilidades:
- **public** - executável por qualquer endereço ou contrato derivado (*herança*)
- **internal** - executável somente pelo próprio contrato ou contrato derivado (*herança*)
- **external** - executável apenas por agentes externos (wallets e outros SCs)
- **private** - executável somente pelo próprio contrato.

Funções: Retornos

- ▶ Ao declarar uma função, você pode declarar ou não seus retornos. Uma função pode retornar **nenhum** ou **vários valores**.

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string public nome = "Fellipe Bravo";
6      uint public idade = 30;
7      int private saldo = -4000;
8      bool private aprovado = false;
9      bool private fezUmPix = true;
10     address private enderecoWallet = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;
11
12     function trocaNome(string calldata novoNome) public returns(string memory, uint) {
13         nome = novoNome;
14         return (nome, idade);
15     }
16
17 }
```

Funções de visualização

- ▶ Em Solidity temos a opção de criar funções que apenas retornam dados. Para declarar esse tipo de função, usamos a palavra reservada **view**.

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string public nome = "Fellipe Bravo";
6      uint public idade = 30;
7      int private saldo = -4000;
8      bool private aprovado = false;
9      bool private fezUmPix = true;
10     address private enderecoWallet = 0x5B38Da6a701c568545dcFcb03FcB875f56beddC4;
11
12     function apenasVisualizacao() public view returns(address) {
13         return enderecoWallet;
14     }
15
16 }
```

Funções de visualização

- ▶ Funções declaradas como **view** **não podem fazer alterações** em variáveis declaradas anteriormente no escopo do contrato. Caso você faça uma alteração, o seu contrato **não compilará**.

Função construtora

- ▶ Ao iniciar um contrato, temos uma função reservada chamada **constructor**. Ela é executada **apenas uma vez**, no momento do deploy do SC e pode receber parâmetros assim como as demais.
- ▶ **Não pode** retornar dados.
- ▶ Normalmente utilizada para definir valores padrões ao inicializar um contrato.
- ▶ Em um construtor, parâmetros do tipo **string** **devem** ser marcados como **memory**


```

1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5      string public nome;
6      uint public idade;
7      int private saldo;
8      bool private aprovado;
9      bool private fezUmPix;
10     address private enderecoWallet;
11
12     constructor(
13         string memory novoNome,
14         uint novaIdade,
15         address novoEnderecoWallet
16     ) {
17         nome = novoNome;
18         idade = novaIdade;
19         saldo = 0;
20         aprovado = false;
21         fezUmPix = false;
22         enderecoWallet = novoEnderecoWallet;
23     }
24
25 }

```

Função construtora

Operações aritméticas

Operações aritméticas

- ▶ Assim como na matemática básica conseguimos **somar, subtrair, dividir, multiplicar e etc.** em **inteiros**
- ▶ Porém, as variáveis devem respeitar o **mesmo tipo** nos dois lados da operação.
- ▶ O Solidity dá suporte para operações aritméticas com atribuição.

Exemplo de operações aritméticas

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5
6      uint public saldo = 0;
7
8      function operacoesAritmeticas(uint valor) public {
9          saldo = saldo + valor; // Soma
10         saldo = saldo - valor; // Subtração
11         saldo = saldo * valor; // Multiplicação
12         saldo = saldo / valor; // Divisão
13         saldo = saldo % valor; // Resto
14         saldo = saldo ** valor; // Exponenciação
15     }
16
17 }
```

Operações lógicas

Operações lógicas

- ▶ O Solidity obviamente dá suporte para operações lógicas para uma movimentação mais “*hardcore*” dos valores.
- ▶ Essas operações lógicas normalmente são utilizadas mais em algoritmos específicos e operações que requisitam uma validação de segurança sobre o valor

Exemplo de operações lógicas

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5
6      uint public saldo = 0;
7
8      function operacoesLogicas(uint valor) public {
9          saldo = saldo << valor; // Deslocamento para a esquerda (Shift left)
10         saldo = saldo >> valor; // Deslocamento para a direita (Shift right)
11         saldo = saldo & valor; // E bit a bit (Operação booleana AND)
12         saldo = saldo | valor; // Ou bit a bit (Operação booleana OR)
13         saldo = saldo ^ valor; // Ou exclusivo bit a bit (Operação booleana XOR)
14         saldo = ~saldo; // Negação bit a bit (Operação booleana NOT)
15     }
16
17 }
```


Operações com atribuição

Operações com atribuição

- ▶ É uma boa prática de programação evitar que as variáveis sejam repetidas durante uma operação lógica ou aritmética
- ▶ Nos exemplos anteriores utilizamos:

variávelA = variávelA (operador) variávelB

- ▶ Podemos **simplificar** a operação usando as operações com atribuição, conforme exemplo:

variávelA (operador)= variávelB

Exemplo de operações c/ atribuição

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  contract MeuPrimeiroSC {
5
6      uint public saldo = 0;
7
8      function operacoesComAtribuicao(uint valor) public {
9          saldo -= valor; // Subtração com atribuição
10         saldo += valor; // Soma com atribuição
11         saldo *= valor; // Multiplicação com atribuição
12         saldo /= valor; // Divisão com atribuição
13         saldo %= valor; // Resto com atribuição
14         saldo <<= valor; // Deslocamento para a esquerda com atribuição
15         saldo >>= valor; // Deslocamento para a direita com atribuição
16     }
17
18 }
```

Estrutura de dados avançadas

Structs

- ▶ É uma boa prática de programação agrupar conjunto de dados dentro de um modelo (estrutura)
- ▶ Facilita na ingestão de dados por funções
- ▶ Facilita no retorno de dados por funções
- ▶ Padroniza **comportamento** entre múltiplas funções
- ▶ São definidas a partir da palavra reservada **struct**

```
1 // SPDX-License-Identifier: Unlicense
2 pragma solidity 0.8.24;
3
4 // Definindo um modelo de dados / estrutura
5 struct Pessoa {
6     string nome;
7     uint idade;
8     bool aprovado;
9     address enderecoWallet;
10 }
11
12 // Definindo um contrato
13 contract MeuPrimeiroSC {
14
15     Pessoa private pessoa;
16
17     constructor() {
18         pessoa.nome = "Fellipe Bravo";
19         pessoa.idade = 30;
20         pessoa.aprovado = false;
21         pessoa.enderecoWallet = msg.sender;
22     }
23
24     function quemSouEu() public view returns (Pessoa memory) {
25         return pessoa;
26     }
27
28 }
```

Exemplo de struct

Enum

- ▶ Conforme você vai evoluindo em código começa a perceber padrões, por exemplo, enumerações.
- ▶ O Solidity possui um tipo de dado chamado **enum**
- ▶ Facilita na padronização de enumerações como por exemplo, um status de compra Aprovada, Recusada e Pendente
- ▶ São definidas a partir da palavra reservada **enum**


```

1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  // Definindo enumeração
5  enum StatusCompra { Pendente, Aprovada, Recusada }
6
7  // Definindo um contrato
8  contract MeuPrimeiroSC {
9
10     StatusCompra private status;
11
12     constructor() {
13         status = StatusCompra.Pendente;
14     }
15
16     function setStatus(StatusCompra _status) public {
17         status = _status;
18     }
19
20     function getStatus() public view returns (StatusCompra) {
21         return status;
22     }
23
24 }

```

Exemplo de enum

Array

- ▶ Você já deve ter pensado em como agrupar uma lista de pessoas. Isso pode ser feito usando arrays.
- ▶ O Solidity possui um tipo de dado chamado **array**
- ▶ Os arrays podem ter tamanho **fixo** ou tamanho **dinâmico**
- ▶ São definidos a partir da utilização de colchetes **[]**
- ▶ Podem ser **multidimensionais** (matrizes) **[] []**

```

1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  // Definindo um modelo de dados / estrutura
5  struct Pessoa {
6      string nome;
7      uint idade;
8      bool aprovado;
9      address enderecoWallet;
10 }
11
12 // Definindo um contrato
13 contract MeuPrimeiroSC {
14
15     // Array dinâmico de pessoas
16     Pessoa[] private pessoas;
17     // Array estático de pessoas
18     Pessoa[5] private melhoresAmigos;
19
20     function adicionarPessoa(Pessoa calldata novaPessoa) public {
21         pessoas.push(novaPessoa);
22     }
23
24 }

```

Exemplo de array

Mapping

- ▶ O Solidity possui também uma estrutura de dados que permite o armazenamento **chave-valor**
- ▶ Dada uma **chaveA**, qual o valor que essa chave guarda?
- ▶ Os mappings são definidos a partir da palavra reservada **mapping(tipoChave => tipoValor)**

```
1  // SPDX-License-Identifier: Unlicense
2  pragma solidity 0.8.24;
3
4  // Definindo um contrato
5  contract MeuPrimeiroSC {
6
7      mapping(address => uint) public saldos;
8
9      function depositar(int valor) public {
10         |   saldos[msg.sender] += uint(valor);
11         |   }
12
13     }
```

Exemplo de **map**

msg

- ▶ Em Solidity, temos uma estrutura global (especial) que armazena os dados das transações ou mensagem que está sendo enviada ao SC.
- ▶ Essa estrutura contém atributos **pré-definidos** e podem ser acessados de qualquer local do smart contract.
- ▶ Para acessar esses dados, utilize a palavra reservada **msg**

```
1 // SPDX-License-Identifier: Unlicense
2 pragma solidity 0.8.24;
3
4 // Definindo um contrato
5 contract MeuPrimeiroSC {
6
7     address donoDoContrato;
8
9     constructor() {
10         donoDoContrato = msg.sender;
11         msg.sender;
12     }
13
14 }
```

Exemplo msg

Time to CODE!

Time to CODE!

- ▶ Acesse <https://remix.ethereum.org/>
- ▶ Vamos colocar algumas teorias em prática!

OBRIGADO!

formação

Mobiup 



nearx

Innovation School