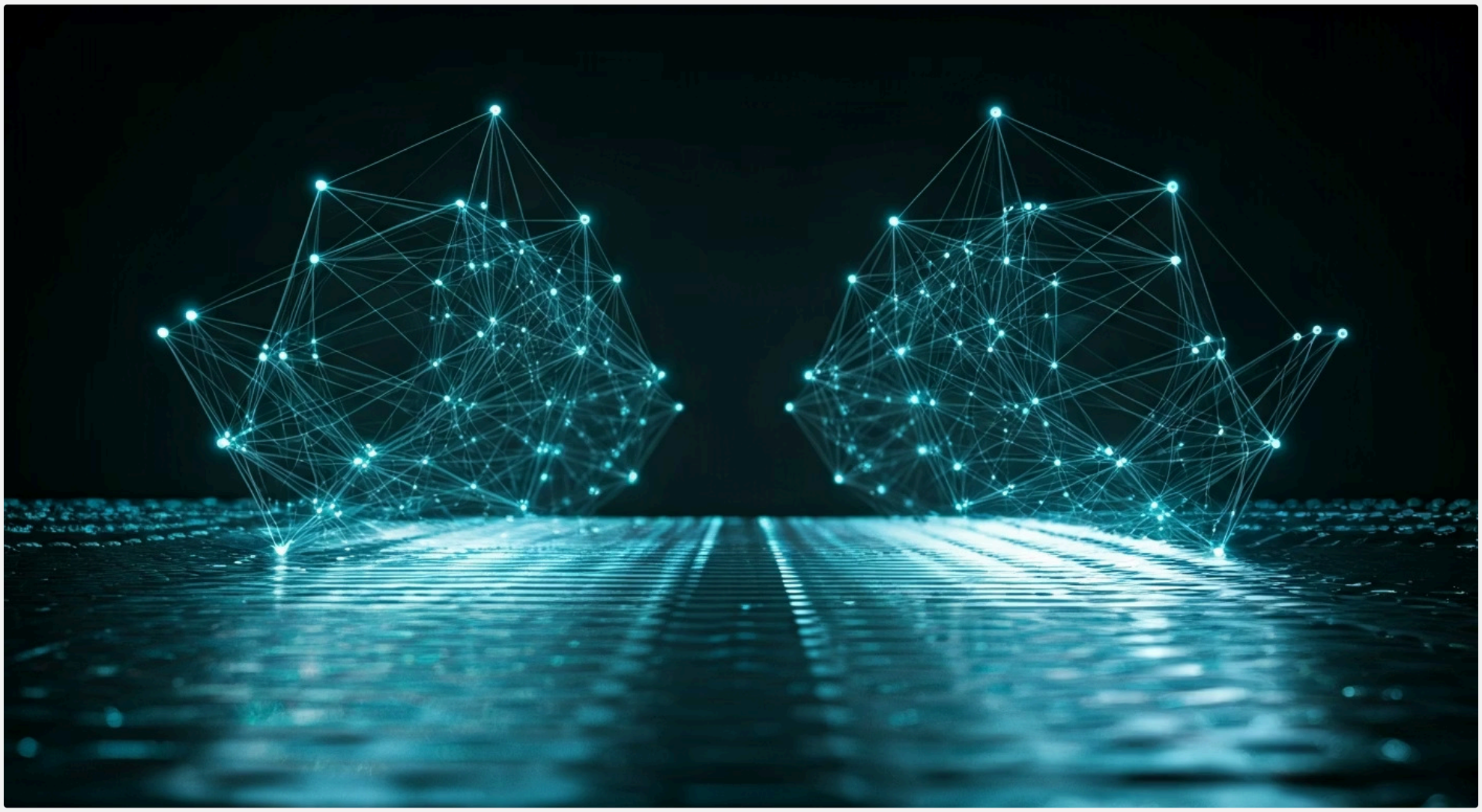
Aula 6 – Variáveis de Estado, Funções e Visibilidade



Imagine que você está construindo um aplicativo descentralizado (DApp) que precisa lembrar informações importantes, como o saldo de um usuário ou o status de uma votação. Sem uma forma de armazenar esses dados de maneira persistente, seu DApp seria como um quadro branco que se apaga a cada vez que alguém o olha. É exatamente aqui que as variáveis de estado entram em cena nos smart contracts: elas são a memória duradoura do seu contrato, o coração que guarda todas as informações cruciais que definem seu comportamento ao longo do tempo.

Mas um contrato não é apenas um repositório de dados; ele também precisa realizar ações. Pense em um caixa eletrônico: ele guarda seu dinheiro (variáveis de estado) e permite que você saque, deposite ou consulte seu saldo (funções). Da mesma forma, as funções em um smart contract são os "verbos" que permitem interagir com esses dados, alterá-los ou simplesmente consultá-los. E, assim como um caixa eletrônico tem diferentes níveis de acesso para clientes, funcionários e técnicos, um smart contract precisa de mecanismos para controlar quem pode realizar cada ação e acessar cada informação.

Nesta aula, vamos desvendar como esses três pilares – variáveis de estado, funções e visibilidade – se entrelaçam para formar a espinha dorsal de qualquer smart contract robusto e seguro. Ao final, você não apenas entenderá a teoria por trás de cada um, mas também será capaz de aplicá-los para construir contratos mais eficientes, seguros e alinhados às melhores práticas do mercado Web3. Prepare-se para dar um salto significativo na sua jornada de desenvolvimento blockchain, conectando esses conceitos fundamentais com a realidade prática da criação de DApps.

Variáveis de Estado: A Memória Permanente do Seu Contrato

Armazenamento Permanente

Dados gravados no blockchain que persistem entre transações

Livro-Razão Digital

Como registros contábeis que mantêm o histórico completo

Imutabilidade

Uma vez registrado, o dado se torna parte permanente da rede

Quando pensamos em um smart contract, é fácil focar apenas no código que ele executa. No entanto, para que um contrato seja realmente útil, ele precisa ter a capacidade de "lembrar" informações entre diferentes transações. É aqui que as variáveis de estado se tornam indispensáveis. Elas são, em essência, os dados que são armazenados permanentemente no blockchain, fazendo parte do estado do contrato e, consequentemente, do estado global da rede.

Pense nas variáveis de estado como o livro-razão principal de uma empresa. Cada entrada nesse livro é um dado vital que precisa ser persistente e imutável, refletindo o histórico e a situação atual da organização. Da mesma forma, em um smart contract, uma variável de estado pode armazenar o saldo de um token, o proprietário de um NFT, ou o resultado de uma votação. Declarar uma variável de estado significa que você está reservando um espaço na "memória" do blockchain para que essa informação seja acessível e modificável (se permitido) por todas as futuras interações com o contrato.

A declaração de uma variável de estado é bastante direta em Solidity, lembrando a sintaxe de outras linguagens de programação. Você especifica o tipo de dado (como uint para números inteiros sem sinal, address para endereços de carteiras, bool para valores booleanos) e o nome da variável. Por exemplo, `uint public totalVotes;` declara uma variável de estado chamada totalVotes que armazenará um número inteiro e será acessível publicamente. Essa simplicidade esconde um poder imenso, pois cada byte armazenado aqui tem implicações de custo e segurança que exploraremos em breve.

Persistência e Imutabilidade: O Poder das Variáveis de Estado

Persistência

A característica mais distintiva das variáveis de estado é sua persistência. Diferente das variáveis locais, que existem apenas durante a execução de uma função e são descartadas em seguida, as variáveis de estado mantêm seus valores entre chamadas de função e transações. Isso significa que, uma vez que um valor é atribuído a uma variável de estado e a transação é minerada, esse valor se torna parte do registro permanente e imutável do blockchain.

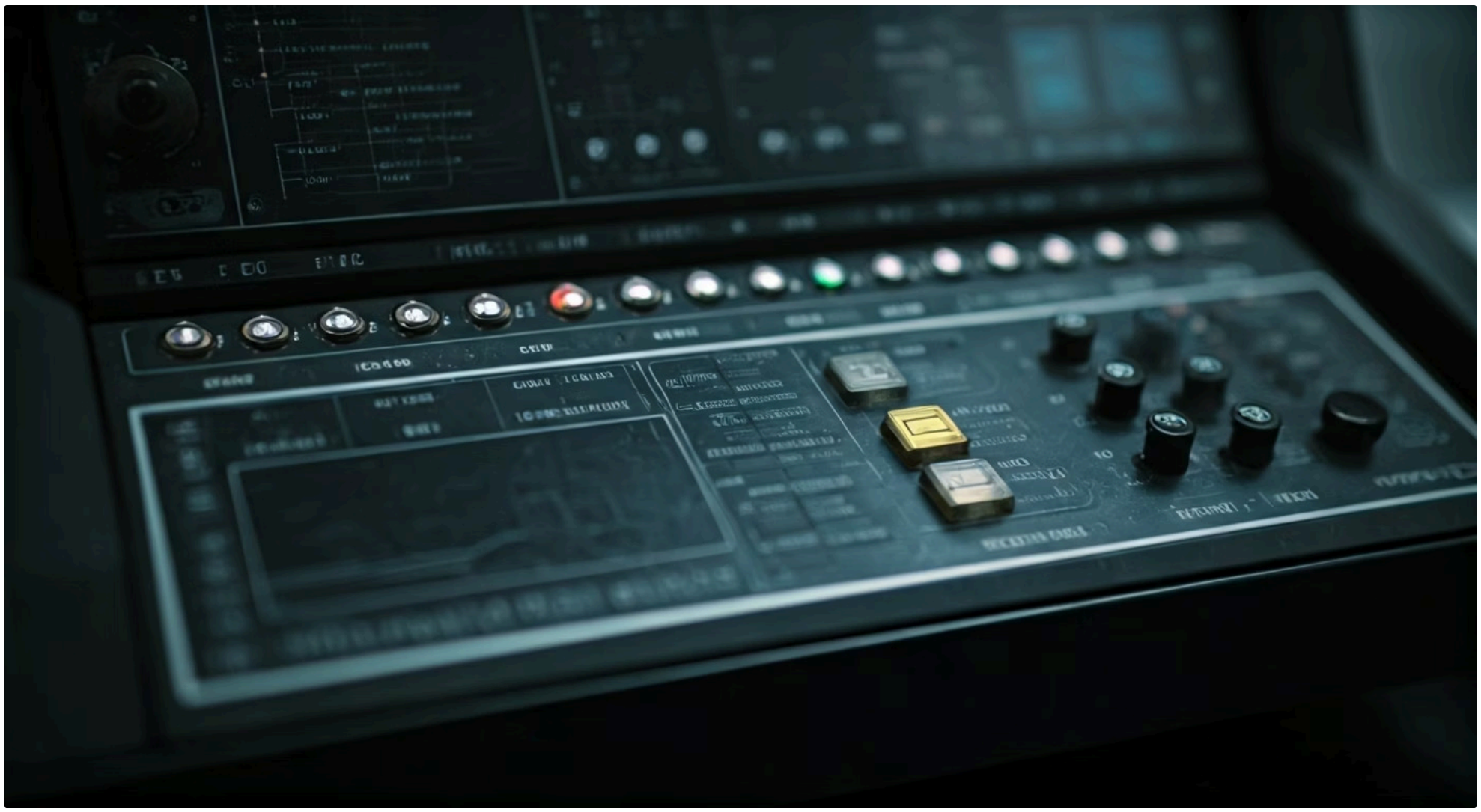
Essa persistência é o que permite que smart contracts mantenham um "estado" contínuo, agindo como sistemas autônomos que evoluem e reagem a interações. Imagine um cofre digital: ele precisa lembrar quanto dinheiro está guardado lá dentro, quem são os proprietários e quais foram as últimas transações. Cada uma dessas informações seria uma variável de estado, crucial para a funcionalidade do cofre. Sem elas, o cofre seria apenas uma estrutura vazia, incapaz de cumprir seu propósito.

Imutabilidade

No contexto do blockchain, a imutabilidade dos dados uma vez registrados é uma faca de dois gumes. Por um lado, ela garante a integridade e a auditabilidade do sistema, tornando impossível alterar registros passados. Por outro, qualquer erro na lógica de armazenamento ou na declaração de uma variável de estado pode ter consequências permanentes e caras. Por isso, a declaração e o uso de variáveis de estado exigem atenção meticulosa, pois elas são o alicerce sobre o qual toda a lógica do seu contrato será construída.

Conceito	Âmbito/Aplicação	Base/Origem	Exemplo
Variável de Estado	Armazenamento permanente no blockchain	Estado do contrato	`uint public balance;`
Variável Local	Armazenamento temporário, dentro de uma função	Memória de execução	`uint tempValue = 10;`
Persistência	Dados mantidos entre transações	Imutabilidade do blockchain	Saldo de um token após uma transferência
Imutabilidade	Registros não podem ser alterados após mineração	Natureza do blockchain	Histórico de transações confirmadas

Funções: As Ações do Seu Contrato



Se as variáveis de estado são a memória do seu contrato, as funções são os "músculos" que permitem que ele execute ações e interaja com o mundo exterior. Um smart contract sem funções seria como um banco que apenas guarda dinheiro, mas não permite saques, depósitos ou consultas. As funções são os blocos de código executáveis que definem o comportamento do contrato, permitindo que ele manipule suas variáveis de estado, interaja com outros contratos ou retorne informações.

Analogia Prática: Pense nas funções como os botões e alavancas de uma máquina complexa. Cada botão tem uma finalidade específica – ligar, desligar, ajustar velocidade. Da mesma forma, cada função em um smart contract é projetada para realizar uma tarefa bem definida, como transferir tokens, registrar um voto, adicionar um novo usuário ou verificar uma condição.

A estrutura básica de uma função em Solidity é intuitiva, lembrando muito outras linguagens de programação, mas com algumas particularidades importantes para o ambiente blockchain.

01

Palavra-chave function

Inicia a declaração da função

02

Nome da função

Identificador único para chamá-la

03

Parâmetros

Dados de entrada entre parênteses

04

Modificadores

Visibilidade e comportamento

05

Tipo de retorno

Dados que a função devolve

Por exemplo, `function deposit(uint amount) public returns (bool success)` declara uma função chamada deposit que recebe um valor amount, é pública e retorna um booleano indicando sucesso. Dentro das chaves {} é onde a lógica da função é implementada, manipulando as variáveis de estado ou realizando cálculos. É a combinação inteligente dessas funções que dá vida e utilidade a um smart contract.

Parâmetros e Retornos: A Comunicação das Funções

Parâmetros: As Entradas

Para que as funções sejam verdadeiramente interativas e flexíveis, elas precisam de mecanismos para receber informações de quem as chama e para fornecer resultados de volta. É aí que entram os parâmetros e os valores de retorno. Os parâmetros são as entradas que uma função espera receber, agindo como os argumentos que você passa para um comando.

Imagine que você está pedindo uma pizza online. Você precisa informar o sabor, o tamanho e o endereço de entrega – esses são os parâmetros da sua "função de pedido". Em um smart contract, se você tem uma função para transferir tokens, ela precisará de parâmetros como o endereço do destinatário e a quantidade de tokens a serem transferidos.

Retornos: As Saídas

Já os valores de retorno são as saídas que a função produz após sua execução, informando o resultado da operação. Depois de processar seu pedido, a pizzaria pode te retornar uma confirmação com o tempo estimado de entrega – esse é o valor de retorno.

Em Solidity, os parâmetros são declarados dentro dos parênteses após o nome da função, especificando o tipo e o nome de cada entrada, como em `function transfer(address _to, uint _amount)`. Os valores de retorno são especificados após a palavra-chave returns, também com seus tipos e nomes, como em `returns (bool success)`.

Essa capacidade de passar dados e receber resultados é fundamental para construir contratos complexos que podem se comunicar de forma eficaz tanto com usuários externos quanto com outros contratos na rede.

```
// Exemplo de função com parâmetros e retorno
contract SimpleCounter {
 uint public count; // Variável de estado

 function incrementBy(uint _value) public returns (uint newCount) {
 require(_value > 0, "O valor deve ser positivo."); // Validação
 count += _value; // Modifica a variável de estado
 return count; // Retorna o novo valor
 }

 function getCount() public view returns (uint currentCount) {
 return count; // Retorna o valor atual sem modificá-lo
 }
}
```

Modificadores de Função: Controlando o Comportamento

view

Lê o estado do contrato, mas não o modifica

pure

Não lê nem modifica o estado do contrato

Sem modificador

Pode ler e modificar o estado do contrato

Nem todas as funções de um smart contract são criadas para alterar o estado do blockchain. Algumas são apenas para leitura de dados, outras para cálculos que não dependem do estado atual do contrato. Para otimizar o uso de gás e, mais importante, para garantir a segurança e a clareza do contrato, Solidity oferece modificadores de função que especificam como uma função interage com o estado do contrato. Os mais comuns são **view** e **pure**.

Pense em um painel de controle de um sistema. Alguns botões são para "ação" (ligar/desligar), enquanto outros são apenas para "informação" (mostrar temperatura atual) ou "cálculo" (converter unidades). Os modificadores de função agem como rótulos nesses botões, indicando claramente o que cada função pode ou não fazer. Isso é crucial não só para quem desenvolve, mas também para quem audita ou interage com o contrato, pois sinaliza as intenções da função.

> **Importante:** A distinção entre funções que modificam o estado e as que não modificam é vital no blockchain. Funções que alteram o estado (como incrementBy no exemplo anterior) custam gás, pois precisam ser mineradas e registradas na rede. Já as funções que apenas leem o estado (view) ou que nem sequer interagem com ele (pure) podem ser executadas localmente por um nó da rede sem custo de gás, pois não geram uma transação que precise ser minerada.

Entender e aplicar corretamente esses modificadores é um passo fundamental para escrever contratos eficientes e seguros.

view e pure em Detalhe

Modificador view

Aprofundando nos modificadores de função, **view** e **pure** são ferramentas poderosas para otimizar e securitizar seus smart contracts. Uma função marcada como **view** promete que ela não modificará o estado do contrato. Ela pode ler variáveis de estado, mas não pode alterá-las. Isso significa que, ao chamar uma função view de fora do contrato (por exemplo, de uma aplicação front-end), você não precisa enviar uma transação para o blockchain, economizando gás e tempo.

Imagine que você está consultando o saldo da sua conta bancária. Essa é uma operação de "leitura" que não altera o seu saldo. Em um smart contract, uma função `getBalance()` seria um exemplo perfeito para usar o modificador view. Ela acessaria a variável de estado que armazena o saldo, mas não faria nenhuma alteração nela. Isso garante aos usuários que a chamada dessa função é segura e não terá efeitos colaterais indesejados no estado do contrato.

Modificador pure

Por outro lado, uma função marcada como **pure** é ainda mais restritiva. Ela promete que não modificará *nem lerá* o estado do contrato. Funções pure são ideais para cálculos puramente matemáticos ou lógicos que dependem apenas dos parâmetros de entrada da função. Por exemplo, uma função que calcula o dobro de um número (`function double(uint x) public pure returns (uint) { return x * 2; }`) não precisa acessar nenhuma variável de estado.

Assim como as funções view, as funções pure também podem ser chamadas sem custo de gás quando invocadas externamente, tornando-as extremamente eficientes para operações computacionais isoladas.

Modificador	Leitura de Estado	Modificação de Estado	Custo de Gás (chamada externa)	Exemplo de Uso
Nenhum	Sim	Sim	Sim	`function deposit(uint amount)`
view	Sim	Não	Não	`function getBalance(address user) view returns (uint)`
pure	Não	Não	Não	`function calculateTax(uint amount) pure returns (uint)`

Visibilidade: Quem Pode Ver e Interagir?

A segurança é a pedra angular do desenvolvimento de smart contracts, e um dos pilares para construí-la é o controle de visibilidade. A visibilidade define quem pode chamar uma função ou acessar uma variável de estado: se é apenas o próprio contrato, contratos herdados, ou qualquer entidade externa. Sem um controle de visibilidade adequado, funções sensíveis poderiam ser chamadas por qualquer pessoa, levando a vulnerabilidades e perdas financeiras.

public

Acessível por todos: o próprio contrato, contratos externos e usuários

private

Acessível apenas pelo próprio contrato onde foi definida

internal

Acessível pelo contrato e por contratos que herdam dele

external

Acessível apenas de fora do contrato, não internamente

Pense em uma casa com diferentes portas e janelas. Algumas portas são públicas (a entrada principal), outras são internas (entre cômodos), e algumas são privadas (um cofre). A visibilidade em Solidity funciona de maneira análoga, controlando o "acesso" a diferentes partes do seu contrato. Definir a visibilidade correta para cada função e variável de estado é uma decisão de design crucial que impacta diretamente a segurança, a modularidade e a capacidade de interação do seu contrato.

Regra de Ouro: Sempre comece com a visibilidade mais restritiva e só expanda-a se houver uma necessidade clara e justificada.

public e private: O Básico do Controle de Acesso

public: Acesso Total

Começando pelos níveis de visibilidade mais fundamentais, **public** e **private** estabelecem a base para o controle de acesso em seus smart contracts. Entender a diferença entre eles é essencial para evitar exposições indevidas e garantir que seu contrato se comporte como esperado.

Uma função ou variável de estado declarada como **public** é a mais aberta de todas. Ela pode ser acessada e chamada por qualquer pessoa, tanto de dentro do próprio contrato quanto de outros contratos ou de contas externas (como uma carteira de usuário). É como a porta da frente de uma loja: qualquer um pode entrar e interagir. Variáveis de estado public automaticamente geram uma função "getter" (leitora) que permite a qualquer um ler seu valor. Isso é útil para dados que precisam ser transparentes e amplamente acessíveis, como o saldo total de um token ou o nome de um contrato.

private: Acesso Restrito

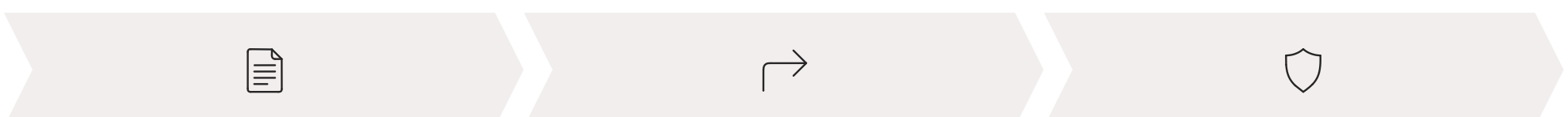
Por outro lado, uma função ou variável de estado **private** é a mais restritiva. Ela só pode ser acessada de dentro do próprio contrato onde foi definida. É como um diário pessoal: apenas o proprietário pode lê-lo. Funções private são usadas para lógica interna que não deve ser exposta ao mundo exterior, como funções auxiliares que realizam cálculos intermediários ou validam dados antes de uma operação principal.

Variáveis de estado private não geram automaticamente uma função getter, mantendo seu valor inacessível diretamente de fora. Usar private para dados e funções sensíveis é uma prática de segurança fundamental para proteger seu contrato de interações não autorizadas.

```
contract AccessControlExample {
 uint private _secretNumber; // Variável privada
 uint public publicData; // Variável pública

 constructor() {
 _secretNumber = 42;
 publicData = 100;
 }

 function getSecretNumber() public view returns (uint) {
 // Uma função pública pode retornar um dado privado, mas o dado em si é privado
 return _secretNumber;
 }

 function _internalHelper() private pure returns (string memory) {
 // Função privada, só pode ser chamada por outras funções dentro deste contrato
 return "Esta é uma função auxiliar interna.";
 }

 function callHelper() public view returns (string memory) {
 return _internalHelper(); // Chamando a função privada internamente
 }
}
```

internal: Compartilhando Lógica Dentro da Família

A visibilidade **internal** preenche uma lacuna importante entre public e private, especialmente quando se trata de contratos que utilizam herança. Uma função ou variável de estado declarada como **internal** pode ser acessada de dentro do contrato em que foi definida e também por contratos que herdam dela. É como um segredo de família: é compartilhado entre os membros da mesma linhagem, mas não com estranhos.



Imagine que você está construindo uma biblioteca de contratos base que fornecem funcionalidades comuns, como controle de acesso ou gerenciamento de tokens. Você quer que os contratos "filhos" (derivados) possam usar essas funcionalidades, mas não quer que elas sejam chamadas diretamente por usuários externos ou outros contratos não relacionados. Nesses casos, **internal** é a escolha perfeita. Ele promove a reutilização de código e a modularidade, mantendo a segurança.

Por exemplo, um contrato base pode ter uma função `_onlyOwner()` marcada como internal que verifica se o chamador é o proprietário do contrato. Contratos que herdam desse base podem então usar `_onlyOwner()` em seus próprios modificadores de função, sem precisar reimplementar a lógica de verificação. Isso não só economiza gás, mas também garante consistência e reduz a chance de erros. Variáveis de estado internal se comportam de maneira similar, permitindo que dados sejam compartilhados entre contratos relacionados na hierarquia de herança.

```
// Contrato Base
contract BaseContract {
 uint internal _internalValue; // Variável interna

 constructor() {
 _internalValue = 10;
 }

 function _incrementInternalValue(uint amount) internal {
 _internalValue += amount;
 }
}

// Contrato Derivado
contract DerivedContract is BaseContract {
 function modifyAndGetInternalValue(uint amount) public view returns (uint) {
 // Podemos chamar a função interna do contrato base
 // _incrementInternalValue(amount); // Não podemos chamar uma função view para modificar estado
 return _internalValue + amount; // Podemos ler a variável interna
 }

 function callInternalIncrement(uint amount) public {
 _incrementInternalValue(amount); // Podemos chamar a função interna para modificar estado
 }
}
```

external: A Interface Externa do Seu Contrato

O modificador de visibilidade **external** é um pouco peculiar e merece atenção especial, pois otimiza a forma como seu contrato interage com o mundo exterior. Uma função declarada como **external** só pode ser chamada de fora do contrato. Ela não pode ser chamada internamente por outras funções dentro do mesmo contrato. É como uma API pública de um serviço: você pode acessá-la de fora, mas o próprio serviço não a usa para suas operações internas.

Otimização de Gás

A principal vantagem de external sobre public para funções que *apenas* serão chamadas externamente é a otimização de gás. Quando uma função é external, os parâmetros são passados de uma forma mais eficiente, o que pode resultar em economia de gás, especialmente para funções com muitos parâmetros ou tipos de dados complexos.

Interface Clara

Se você tem uma função que sabe que nunca será chamada por outra função dentro do seu contrato, external é geralmente a escolha preferencial. Ela sinaliza claramente que a função é parte da interface pública do contrato.

Casos de Uso

Por exemplo, uma função para receber pagamentos em Ether (`function receiveFunds() external payable`) ou uma função para registrar um novo usuário (`function registerUser(string memory _name) external`) são candidatas ideais para external.

Importante: É importante notar que variáveis de estado não podem ser external; elas são sempre public, private ou internal. A escolha entre public e external para funções é uma decisão de design que reflete a intenção de uso da função e pode impactar a eficiência do seu contrato.

Visibilidade	Chamada Interna (mesmo contrato)	Chamada Externa (outros contratos/EOAs)	Chamada por Contratos Derivados	Uso Típico
public	Sim	Sim	Sim	Funções de interface geral, getters de estado
private	Sim	Não	Não	Funções auxiliares internas, dados sensíveis
internal	Sim	Não	Sim	Funções e dados para herança
external	Não	Sim	Não	Funções de interface externa, otimização de gás

Escolhendo a Visibilidade Certa: Melhores Práticas e Segurança

A escolha do modificador de visibilidade para cada função e variável de estado não é apenas uma questão de funcionalidade, mas uma decisão crítica de segurança. Uma visibilidade mal aplicada pode abrir portas para ataques, resultando em perda de fundos ou controle do contrato. É por isso que a compreensão profunda e a aplicação cuidadosa desses conceitos são tão importantes no desenvolvimento de smart contracts.

1

Princípio do Menor Privilégio

Uma das melhores práticas mais difundidas é o princípio do "menor privilégio". Isso significa que você deve sempre começar com a visibilidade mais restritiva (private ou internal) e só torná-la mais aberta (public ou external) se houver uma necessidade explícita e bem justificada.

2

Proteja a Lógica Interna

Por exemplo, se uma função é apenas um passo intermediário em uma lógica complexa e nunca deve ser chamada diretamente por um usuário externo, ela deve ser private ou internal. Expor essa função publicamente poderia permitir que um atacante a chamasse fora da sequência esperada, explorando vulnerabilidades.

3

Segurança como Prioridade

A segurança como prioridade é um tema recorrente na Web3. Bibliotecas auditadas como a OpenZeppelin, por exemplo, utilizam esses princípios de visibilidade extensivamente em seus contratos para garantir que apenas as funções e dados necessários sejam expostos.

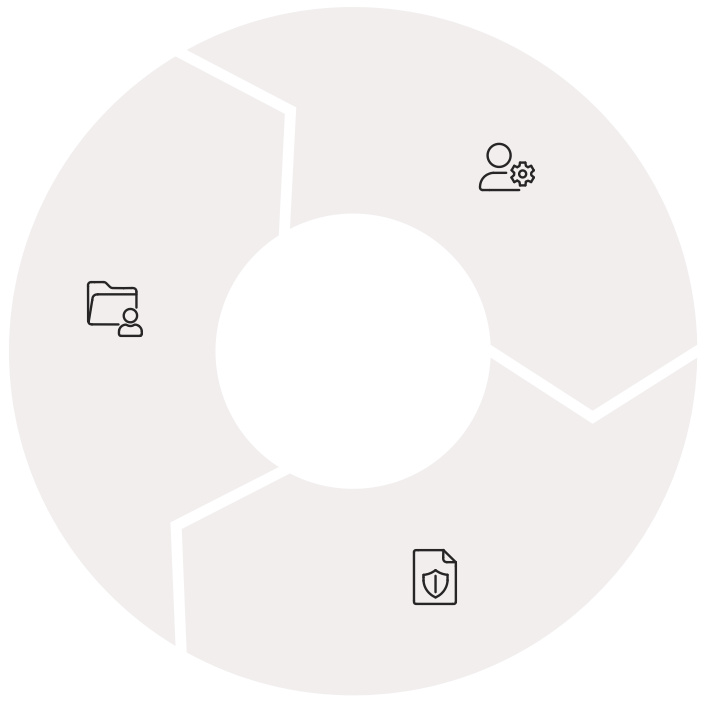
4

Perguntas-Chave

Ao desenvolver, sempre se pergunte: "Quem *precisa* acessar esta função ou variável? E quem *não deve*?". Responder a essas perguntas guiará suas escolhas de visibilidade e ajudará a construir contratos mais robustos e resistentes a ataques.

Combinando Conceitos: Um Contrato Completo

Até agora, exploramos as variáveis de estado, as funções e os níveis de visibilidade de forma individual. No entanto, o verdadeiro poder dos smart contracts reside na forma como esses elementos se combinam para criar lógica complexa e interativa. Um contrato bem projetado é uma orquestra onde cada componente desempenha seu papel em harmonia, garantindo funcionalidade e segurança.



Variáveis de Estado

Armazenam dados persistentes

Funções

Executam ações e lógica

Visibilidade

Controla acesso e segurança

Cenário Prático: Contrato de Votação

Vamos considerar um cenário prático: um contrato de votação simples. Ele precisa de uma variável de estado para armazenar o número total de votos, outra para registrar quem já votou (para evitar votos duplicados) e talvez uma para o nome do item em votação. Ele também precisará de funções: uma para permitir que os usuários votem, outra para consultar o número atual de votos e, possivelmente, uma função para o administrador iniciar ou encerrar a votação.

A visibilidade será crucial aqui. A função de votação deve ser public ou external para que qualquer um possa participar. A variável que registra quem já votou pode ser private ou internal se for apenas para uso interno do contrato. A função do administrador para iniciar/encerrar a votação deve ter um controle de acesso rigoroso, talvez usando um modificador de acesso como onlyOwner (que veremos em aulas futuras) e ser public para ser chamada pelo administrador. Essa interconexão de variáveis de estado, funções e visibilidade é o que transforma um pedaço de código em um DApp funcional e seguro.

```
// Exemplo de Contrato de Votação Simples
contract SimpleVoting {
 string public proposalName; // Variável de estado pública para o nome da proposta
 uint public totalVotes; // Variável de estado pública para o total de votos
 address public owner; // Variável de estado pública para o proprietário do contrato

 // Mapeamento para registrar quem já votou (evita votos duplicados)
 mapping(address => bool) private hasVoted; // Variável de estado privada

 // Construtor: executado apenas uma vez na implantação
 constructor(string memory _proposalName) {
 proposalName = _proposalName;
 owner = msg.sender; // Define o deployer como proprietário
 totalVotes = 0;
 }

 // Modificador para restringir acesso ao proprietário
 modifier onlyOwner() {
 require(msg.sender == owner, "Apenas o proprietário pode chamar esta funcao.");
 _;
 }

 // Função para votar (pública)
 function vote() public {
 require(!hasVoted[msg.sender], "Voce ja votou."); // Verifica se o usuario ja votou
 totalVotes++; // Incrementa o total de votos
 hasVoted[msg.sender] = true; // Marca o usuario como votante
 }

 // Função para obter o nome da proposta (view)
 function getProposalName() public view returns (string memory) {
 return proposalName;
 }

 // Função para obter o total de votos (view)
 function getTotalVotes() public view returns (uint) {
 return totalVotes;
 }

 // Função para resetar a votação (apenas proprietário)
 function resetVoting() public onlyOwner {
 totalVotes = 0;
 // Resetar o mapeamento hasVoted seria mais complexo e custoso,
 // em um contrato real, usaria-se um novo mapeamento ou um sistema de "sessões".
 // Para este exemplo, vamos simplificar e apenas resetar os votos.
 }
}
```

Revisão e Aplicações Avançadas

Chegamos a um ponto crucial onde os conceitos de variáveis de estado, funções e visibilidade se solidificam como os pilares fundamentais da programação de smart contracts. Entender como cada um funciona isoladamente é importante, mas a maestria vem da capacidade de integrá-los de forma coesa e segura. As variáveis de estado fornecem a memória persistente, as funções definem as ações e a visibilidade controla quem pode acessar e modificar esses elementos.

Base Sólida

Domínio dos conceitos fundamentais

Herança

Compartilhamento de lógica entre contratos

Interfaces

Definição de como contratos interagem

Bibliotecas

Código reutilizável e modular

Essa base sólida é o trampolim para conceitos mais avançados. Por exemplo, a herança, onde um contrato pode estender a funcionalidade de outro, depende fortemente do modificador internal para compartilhar lógica e dados entre contratos relacionados. Interfaces, que definem como contratos podem interagir entre si, utilizam funções external e public para expor suas capacidades. Até mesmo o desenvolvimento de bibliotecas, que encapsulam código reutilizável, se beneficia da correta aplicação da visibilidade.

Segurança em Foco: A segurança, como sempre, é um tema central. Muitos dos ataques a smart contracts, como ataques de reentrância ou acesso não autorizado, são mitigados ou prevenidos pela correta aplicação dos modificadores de visibilidade e de estado. Ao planejar seu contrato, sempre visualize o fluxo de dados e as interações, e questione-se sobre o nível de acesso apropriado para cada componente. Essa mentalidade proativa é o que diferencia um desenvolvedor de DApps competente de um mestre na arte da programação blockchain.

Consolidação e Próximos Passos

Nesta aula, desvendamos os mistérios por trás das variáveis de estado, que são a memória persistente do seu contrato; das funções, que são as ações que ele pode executar; e dos modificadores de visibilidade (public, private, internal, external), que controlam o acesso a esses elementos. Compreender e aplicar esses conceitos é fundamental para construir smart contracts seguros, eficientes e funcionais. Eles formam a base sobre a qual toda a lógica e interação do seu DApp serão construídas.

> **Em prática**
>
> Ao desenvolver seus próprios contratos, sempre comece definindo quais dados precisam ser persistentes (variáveis de estado). Em seguida, pense nas ações que seu contrato deve realizar (funções). Por fim, e crucialmente, determine quem deve ter acesso a cada dado e função, aplicando o modificador de visibilidade mais restritivo possível para garantir a segurança.

Autoavaliação

1. Qual modificador de visibilidade permite que uma função seja chamada apenas de dentro do próprio contrato e por contratos derivados?
   a) public
   b) private
   c) internal
   d) external
2. Uma função marcada com o modificador pure pode:
   a) Ler e modificar variáveis de estado.
   b) Apenas ler variáveis de estado, mas não modificá-las.
   c) Nem ler nem modificar variáveis de estado.
   d) Modificar variáveis de estado, mas não lê-las.
3. Qual é a principal vantagem de usar external em vez de public para uma função que *apenas* será chamada de fora do contrato?
   a) Permite que a função seja chamada internamente.
   b) Garante que a função não modificará o estado.
   c) Oferece otimização de gás para chamadas externas.
   d) Torna a função inacessível para outros contratos.
4. Qual das seguintes afirmações sobre variáveis de estado é **correta**?
   a) Elas são temporárias e existem apenas durante a execução de uma função.
   b) Elas são armazenadas permanentemente no blockchain e fazem parte do estado do contrato.
   c) Elas não podem ser modificadas após sua declaração inicial.
   d) Elas não podem ser declaradas como public.
5. Explique a importância da escolha correta da visibilidade para a segurança de um smart contract, utilizando um exemplo prático de uma vulnerabilidade que poderia surgir de uma visibilidade mal aplicada.

Gabarito e Recursos

1	2	3	4
Resposta	**Resposta**	**Resposta**	**Resposta**
c) internal	c) Nem ler nem modificar variáveis de estado	c) Oferece otimização de gás para chamadas externas	b) Elas são armazenadas permanentemente no blockchain e fazem parte do estado do contrato

---

Próxima Aula

Na Aula 7, vamos aprofundar ainda mais no armazenamento de dados, explorando **Mapeamentos e Arrays: Armazenando Dados em Contratos**. Você aprenderá a lidar com coleções de dados de forma eficiente e segura, um passo essencial para contratos mais complexos.

Recursos Adicionais

- **Documentação Oficial do Solidity:** Para detalhes técnicos e exemplos de sintaxe.
- **OpenZeppelin Contracts:** Para estudar implementações seguras e padrões de design.
- **Hardhat Documentation:** Para configurar um ambiente de desenvolvimento e testar seus contratos.

> **NOTA IMPORTANTE:** As informações regulatórias/legais/técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais para verificar alterações.