## **🔹 What are Access Modifiers?**

Access modifiers in Solidity control **who can access functions and variables** in a smart contract. This helps **protect sensitive data** and **prevent unauthorized actions**.

### **🔹 Types of Access Modifiers & When to Use Them**

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Who Can Access?** | **When to Use?** |
| public | **Anyone** (external users & other contracts) | For functions or variables that should be visible to everyone (e.g., getBalance()) |
| private | **Only inside the contract** | For internal data that should **never** be accessed outside (e.g., sensitive mappings like invoiceExists) |
| internal | **Only inside the contract & derived contracts** | When a variable or function should be accessible within the contract **and its child contracts** |
| external | **Only external accounts or contracts** | For functions that should only be called **from outside** (not from inside the contract) |

## **🔹 Key Points to Consider When Choosing Access Modifiers**

### **✅ 1. Protect Sensitive Data**

* **Use private for mappings & variables** that should not be exposed.
* Example: mapping (uint256 => Invoice) private invoices; // Prevents exposing invoice details

### **✅ 2. Minimize Function Exposure**

* If a function should only be used inside the contract, **use internal or private** instead of public.

Example: function \_calculateFee(uint256 amount) private pure returns (uint256) {  
 return amount \* 2 / 100; // Internal fee calculation  
}

* **Why private?** No need for external access.

### **✅ 3. Use external for Gas Optimization**

* external functions **use less gas** than public because Solidity does not store extra data for internal calls.
* **Use external if the function is never called inside the contract.**

Example: function processPayment(uint256 invoiceId) external payable {  
 // Payment logic here  
}

* **Why external?** Only external users call this function; no need for internal calls.

### **✅ 4. Use public Wisely**

* Only make a function **public** if you want **both external users & internal functions** to access it.

Example: function getInvoiceDetails(uint256 invoiceId) public view returns (uint256, address, bool) {  
 require(invoiceExists[invoiceId], "Invoice does not exist");  
 Invoice memory inv = invoices[invoiceId];  
 return (inv.amount, inv.payer, inv.isPaid);  
}

* **Why public?** The contract itself & external users may need to check invoice details.

## **🔹 Summary: When to Use Each Modifier**

|  |  |
| --- | --- |
| **Modifier** | **Use Case** |
| public | For getter functions and data that can be visible to everyone |
| private | For internal calculations, mappings, or variables that should not be exposed |
| internal | For functions and variables that should be inherited by child contracts |
| external | For functions that are **only** called from outside the contract (saves gas) |

### **🔹 Best Practices**

✅ **Follow the Principle of Least Privilege** – Only give access to what’s needed.

✅ **Use private & internal wherever possible** to prevent unauthorized access.

✅ **Use external instead of public for gas savings** if a function is never called internally.

✅ **Avoid public state variables** unless necessary, as they generate automatic getter functions.

Now, let’s go over some **common mistakes developers make with access modifiers** and how to **avoid them**.

## **🔴 1. Exposing Sensitive Data with public State Variables**

### **❌ Mistake:**

Making state variables public **when they contain sensitive data**.

contract InvoiceContract {  
 mapping(uint256 => Invoice) public invoices; // ❌ BAD: Anyone can see all invoices!  
}

### **✅ Fix:**

Make it **private** and create a controlled getter function if needed.

contract InvoiceContract {  
 mapping(uint256 => Invoice) private invoices; // ✅ GOOD: Restricted access  
}

📌 **Why?**

* If invoices is public, anyone can see **all invoice details**, which might contain **confidential financial data**!

## **🔴 2. Making Functions public When They Should Be internal or private**

### **❌ Mistake:**

If a function is **only needed within the contract**, making it public is **a security risk**.

contract InvoiceContract {  
 function \_calculateTax(uint256 amount) public pure returns (uint256) { // ❌ BAD: No need for public  
 return amount \* 10 / 100;  
 }  
}

### **✅ Fix:**

Use private or internal for **helper functions** that should not be exposed.

contract InvoiceContract {  
 function \_calculateTax(uint256 amount) private pure returns (uint256) { // ✅ GOOD: Private function  
 return amount \* 10 / 100;  
 }  
}

📌 **Why?**

* \_calculateTax() is an **internal calculation function**. There’s **no reason for external users** to call it.

## **🔴 3. Calling external Functions Inside the Contract**

### **❌ Mistake:**

If a function is marked external, **it cannot be called from within the contract**.

contract InvoiceContract {  
 function processPayment(uint256 invoiceId) external payable {  
 // Payment logic  
 }  
  
 function anotherFunction() public {  
 processPayment(1); // ❌ ERROR: Cannot call external functions internally!  
 }  
}

### **✅ Fix:**

Use public or internal if it needs to be called inside the contract.

contract InvoiceContract {  
 function processPayment(uint256 invoiceId) public payable { // ✅ GOOD: Now can be called internally  
 // Payment logic  
 }  
}

📌 **Why?**

* external is meant **only for external calls** to **save gas**. If a function needs to be called **inside the contract**, use public.

## **🔴 4. Forgetting to Restrict Critical Functions (Using onlyOwner)**

### **❌ Mistake:**

Leaving functions open to **anyone**, when only the **contract owner** should be able to execute them.

contract InvoiceContract {  
 function withdrawFunds() public {  
 // ❌ BAD: Anyone can withdraw money!  
 }  
}

### **✅ Fix:**

Use **Access Control** (like onlyOwner) to **restrict sensitive functions**.

contract InvoiceContract {  
 address private owner;  
  
 constructor() {  
 owner = msg.sender; // Set contract deployer as owner  
 }  
  
 modifier onlyOwner() {  
 require(msg.sender == owner, "Not the contract owner!");  
 \_;  
 }  
  
 function withdrawFunds() public onlyOwner { // ✅ GOOD: Only the owner can withdraw money  
 // Withdraw logic }}

📌 **Why?**

* Without onlyOwner, **anyone** could call withdrawFunds() and **drain the contract**!

## **🔴 5. Making State Variables public Instead of Using Getter Functions**

### **❌ Mistake:**

Declaring state variables public instead of using controlled **getter functions**.

contract InvoiceContract {  
 uint256 public invoiceCount; // ❌ BAD: Uncontrolled access  
}

### **✅ Fix:**

Keep it private and use **getter functions for controlled access**.

contract InvoiceContract {  
 uint256 private invoiceCount; // ✅ GOOD: Controlled access  
  
 function getInvoiceCount() public view returns (uint256) {  
 return invoiceCount;  
 }  
}

📌 **Why?**

* Public state variables **automatically generate getter functions**, but **you lose control** over how data is accessed.

## **💡 Summary of Mistakes to Avoid**

|  |  |
| --- | --- |
| **❌ Mistake** | **✅ Fix** |
| Making sensitive state variables public | Use private and create getter functions |
| Making helper functions public | Use private or internal |
| Calling external functions inside the contract | Use public instead |
| Not restricting critical functions (withdrawFunds) | Use onlyOwner modifier |
| Exposing state variables directly | Use getter functions |

### **🚀 Best Practices for Secure Smart Contracts**

✔ \*\*Always follow the **principle of least privilege** – expose only what is necessary.

✔ **Use private and internal wherever possible** to prevent unauthorized access.

✔ **Use external functions for gas optimization**, but never call them inside the contract.

✔ **Restrict access to critical functions** (like withdrawals, admin tasks) using onlyOwner or similar mechanisms.

✔ **Review smart contract security best practices** to avoid common vulnerabilities.