#5 Solidity II: Contracts in Practice

Lecture Notes for CS190N: Blockchain Technologies and Security October 13, 2025

This lecture moves from single-contract code to modular, reusable, and safe Solidity systems. We use inheritance to factor shared policy, applying `virtual`/`override`, constructors, `super`, and predictable multiple inheritance via C3. We show how events communicate with off-chain code and how indexed topics enable fast queries, then package utilities as libraries with `using for`, explaining when calls are inlined versus `delegatecall` and what that implies for storage and `msg.sender`. We finish with Checks–Effects–Interactions and disciplined error handling to reduce reentrancy risk, so you can design multi-contract architectures with clear behavior and robust instrumentation.

1 STRUCTURING CONTRACTS WITH INHERITANCE

Inheritance lets one contract specialize another. It clarifies intent when the relationship is truly "is a", for example a token that is an ownable asset. Keeping base contracts small makes it safer for children to combine them.

1.1 The `is` Relationship

A derived contract declared with `is` inherits the base contract's `public` and `internal` members. This is ideal for cross-cutting policies such as ownership. The key mechanism is the `modifier`, which runs its check before the function body and resumes at the placeholder `_`.

Example: Ownable access control. Read the modifier first; it calls `require` to ensure the caller is the owner, then yields control back to the function where `onlyOwner` is attached. The constructor sets the initial owner to the deployer, so any child can immediately gate sensitive actions without duplicating checks.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;
contract Ownable {
    address public owner;
    modifier onlyOwner() {
        require(msg.sender == owner, "Caller is not the owner");
        _;
    }
    constructor() {
        owner = msg.sender;
    }
}
contract MyContract is Ownable {
    function doSomethingCritical() public onlyOwner {
        // restricted action
    }
}
```

1.2 Overriding with `virtual` and `override`

Sometimes the interface should stay stable while the behavior changes. The base marks a function `virtual` to permit overriding, and the child marks its replacement `override`. Matching the signature keeps polymorphism explicit; if you want the child to remain overridable, use `override virtual`.

Example: Overriding `transfer`. Compare the two `transfer` implementations. The base checks balance and performs the state updates in two assignments. The child keeps those semantics but adds a recipient check so that `address(0)` cannot receive funds. Because the signature is identical, callers do not need to change, yet the policy is stricter.

```
contract Token {
    mapping(address => uint256) public balances;
    function transfer(address to, uint256 amount) public virtual {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        balances[to] += amount;
    }
}
contract AuditedToken is Token {
    function transfer(address to, uint256 amount) public override {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        require(to != address(0), "Invalid recipient");
        balances[msg.sender] -= amount;
        balances[to] += amount;
    }
}
```

1.3 Extending Parent Logic with super

Replacing tested code can be risky when you only need to add small steps. The `super` keyword calls the next implementation in Solidity's linearized order. This lets you add pre or post actions around the trusted core behavior.

Example: Extend and then call parent with `super`. The function emits an audit log before delegating updates to the parent. Notice that the event is fired with the same arguments the parent will use to mutate state; this ensures the log reflects exactly what happened when `super.transfer` runs.

```
contract AuditedToken2 is Token {
    event TransferAudited(address indexed from, address indexed to, uint256 amount);
    function transfer(address to, uint256 amount) public override {
        emit TransferAudited(msg.sender, to, amount);
        super.transfer(to, amount);
    }
}
```

1.4 Constructors in an Inheritance Chain

When a parent requires parameters, the child must supply them. You can pass arguments inline in the inheritance list, which is concise, or in the child's constructor head, which is clearer when values come from parameters.

Example: Passing constructor arguments. The first child hardcodes the name at compile time, while the second forwards a runtime value. Both initialize the parent before the child's body runs, so `name` is always defined.

```
contract Named {
    string public name;
    constructor(string memory _name) { name = _name; }
}
```

```
contract MyToken is Named("My First Token") {}
contract AnotherToken is Named {
    constructor(string memory _tokenName) Named(_tokenName) {}
}
```

1.5 Multiple Inheritance and Linearization

Composing small policies from multiple parents is common. Solidity resolves calls with C3 linearization so that `super` has a single, deterministic next stop. For `Final is M1, M2`, a call to `super.foo()` from `Final` proceeds to `M2.foo()` next. When overriding a function present in multiple parents, list all of those parents in the `override` specifier so the compiler knows which branches you merged.

Example: Overriding from multiple parents. The `override(Mixin1, Mixin2)` clause makes the intent explicit. The `super.foo()` call then walks to `Mixin2.foo()` because `M2` is the rightmost parent in the declaration, which is the next hop in the linearized order.

```
contract Base { function foo() public virtual {} }
contract Mixin1 is Base { function foo() public virtual override {} }
contract Mixin2 is Base { function foo() public virtual override {} }
contract Final is Mixin1, Mixin2 {
    function foo() public override(Mixin1, Mixin2) {
        super.foo(); // next is Mixin2.foo() by the linearized order
    }
}
```

2 EVENTS AND LOGGING

Contracts cannot push notifications to users, and storing long histories in state is expensive. Events solve this by writing structured data into transaction logs that off-chain consumers can subscribe to. Emit events after state updates so that logs match the new state. When you plan to filter by a field, mark it `indexed` so clients can query efficiently.

2.1 Declaring and Emitting

An event declaration is a schema for what you want to record. Emitting the event takes concrete values from the execution and stores them in the log. Contracts cannot read logs, so the primary consumers are front ends and indexers.

Example: A `Transfer` event. The function updates balances and then emits a log with the sender, recipient, and amount. A front end can subscribe to this event and update the displayed balances without re-reading storage for all accounts.

```
contract MyToken {
    event Transfer(address from, address to, uint256 amount);
    mapping(address => uint256) public balances;
    function transfer(address to, uint256 amount) public {
        // ... update balances ...
        emit Transfer(msg.sender, to, amount);
    }
}
```

2.2 Filtering with `indexed` Topics

Topics act like database indexes. In a non-anonymous event you can index up to three parameters because the signature occupies the first topic. In an anonymous event you can index up to four. If

an indexed field is dynamic, the topic stores its keccak256 hash while the full value remains in the data section. This makes "all transfers to address X" queries fast without scanning every log entry.

Example: Indexed parameters for efficient queries. Marking `from` and `to` as `indexed` enables clients to filter for senders or recipients directly from topics without decoding every log.

```
event Transfer(address indexed from, address indexed to, uint256 amount);
```

3 LIBRARIES

Libraries package helpers for situations where there is no meaningful "is a" relationship. When every function is `internal`, the compiler inlines them into callers, which saves deployment cost and avoids separate addresses. If any function is `public` or `external`, the library is deployed once and calls use `delegatecall`, which executes the library's code in the caller's context.

3.1 Using `using for`

Attaching a library to a type with `using for` gives method-like syntax without changing storage. The call still compiles to the same logic; it is a readability aid that helps keep arithmetic and bounds checks close to the data they affect.

Example: Minimal helper with `using for`. The method-style call `balances[msg.sender].add(amount)` is just syntax sugar for `SafeMath.add(balances[msg.sender], amount)`; both evaluate to a new `uint256` that we assign back into the mapping. Since Solidity 0.8 adds overflow checks by default, this example is mainly pedagogical.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;
library SafeMath {
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        return a + b;
    }
}
contract MyToken {
    using SafeMath for uint256;
    mapping(address => uint256) public balances;
    function mint(uint256 amount) public {
        balances[msg.sender] = balances[msg.sender].add(amount);
    }
}
```

3.2 Call Semantics

A public or external library call uses `delegatecall`. The code executes from the library but reads and writes the caller's storage, and it preserves `msg.sender` and `msg.value`. This is why upgradeable proxies work, and it is also why storage layout discipline is essential. The three call styles in Table 1 summarize the differences.

3.3 Storage Layout Collisions

A `delegatecall` writes into the caller's storage slots. If an implementation assumes a different order or types, assignments can overwrite unrelated data. In upgradeable designs, keep variable order stable or use unstructured storage with fixed hash-based slots. The high-level tradeoffs between inheritance and libraries in Table 2 can help you decide when composition is safer than delegation.

Table 1. Comparison of EVM call opcodes

Opcode	Executes Code In	State Modified In	`msg.sender`/`msg.value`
`call`	Target contract	Target contract	New values
`delegatecall`	Target contract	Calling contract	Preserved
`staticcall`	Target contract	Read-only, writes revert	Preserved

Example: Mismatched storage order causes corruption. The two contracts declare the same fields in different orders. When the proxy delegates to the logic contract and runs `setOwner`, the assignment that should land in slot 1 for `owner` instead writes to slot 1 in the proxy, which is `implementation`. This corrupts the address the proxy will delegate to next time.

```
contract Logic {
    address public implementation; // slot 0
    address public owner; // slot 1
    function setOwner(address _newOwner) public { owner = _newOwner; }
}
contract Proxy {
    address public owner; // slot 0
    address public implementation; // slot 1
    // fallback that delegatecalls to 'implementation' ...
}
```

Table 2. Inheritance vs. Libraries

Feature	Inheritance	Libraries
Relationship	is-a specialization	uses-a toolbox shared across contracts
State	Inheritable state allowed	No mutable state, constants allowed
Deployment	Code included in child	Internal-only functions are inlined; public or external functions require a deployed library
Call	Internal calls compile to jumps	Public or external library calls use `delegatecall`

4 CHECKS–EFFECTS–INTERACTIONS (CEI)

External calls are risky when your internal state still reflects the pre-call world. CEI reduces that risk by validating inputs first, updating your own storage second, and contacting other contracts last. This sequencing helps ensure that any attempt to reenter observes the already-updated state.

Example: Vulnerable withdraw. The function sends value before clearing the balance. If the recipient is a contract that implements a fallback function, it can reenter `withdraw` during the external call. Because the balance is not yet zero, the repeated call passes the initial check and drains funds.

```
contract VulnerableVault {
    mapping(address => uint) public balances;
    function withdraw() public {
        uint amount = balances[msg.sender];
        require(amount > 0, "No balance to withdraw");
        // Interaction before effect: vulnerable
        (bool ok, ) = msg.sender.call{value: amount}("");
        require(ok, "Transfer failed");
```

```
        // Effect too late
        balances[msg.sender] = 0;
    }
}
```

Example: CEI-compliant withdraw. Here we clear the balance before sending funds. A reentry attempt reads a zero balance at the very start of the function and fails immediately, which closes the attack path without adding complex guards.

```
contract SecureVault {
    mapping(address => uint) public balances;
    function withdraw() public {
        uint amount = balances[msg.sender];
        // Checks
        require(amount > 0, "No balance to withdraw");
        // Effects
        balances[msg.sender] = 0;
        // Interactions
        (bool ok, ) = msg.sender.call{value: amount}("");
        require(ok, "Transfer failed");
    }
}
```

Error handling supports this style. `require` communicates user-facing validation failures, `revert` is convenient inside branches or helper functions, and `assert` expresses invariants that should never be false in correct code. A side-by-side summary appears in Table 3.

Table 3. Comparison of Solidity error-handling primitives (transposed)

Aspect	**require**	**revert**	**assert**
Primary Use	Validate inputs or external state	Same as `require`, flexible placement	Check internal invariants
Gas on Failure	Unused gas refunded	Unused gas refunded	Unused gas refunded (0.8+)
Error Type	`Error(string)`	`Error(string)` or custom error	`Panic(uint256)`
Signals	Expected, recoverable error	Expected, recoverable error	Bug or impossible state

REFERENCES

A USEFUL RESOURCES

A.0.1 ConsenSys Smart Contract Best Practices. A comprehensive guide written by industry leaders, covering a wide range of security considerations, known attack patterns, and recommended development practices. https://consensysdiligence.github.io/smart-contract-best-practices/

A.0.2 Smart Contract Vulnerability Dataset. A dataset that categorizes known smart contract vulnerabilities. https://solodit.cyfrin.io/