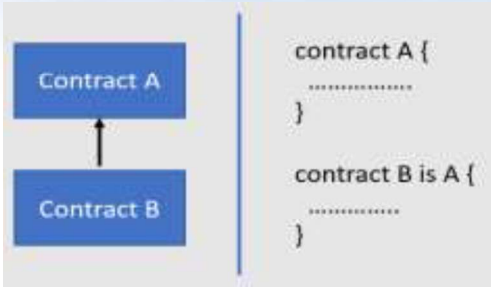
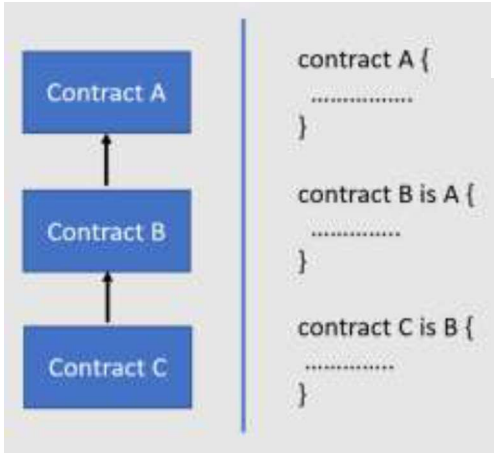


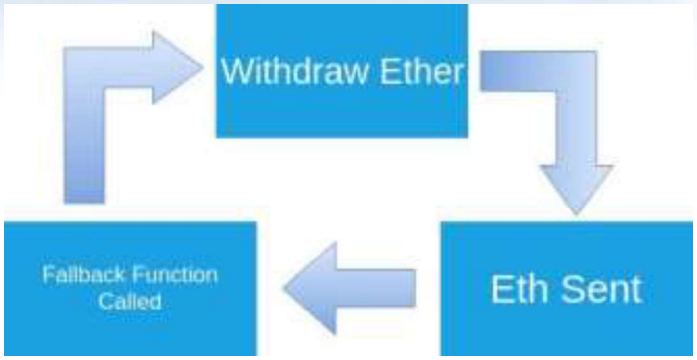

National Institute of Electronics and Information Technology

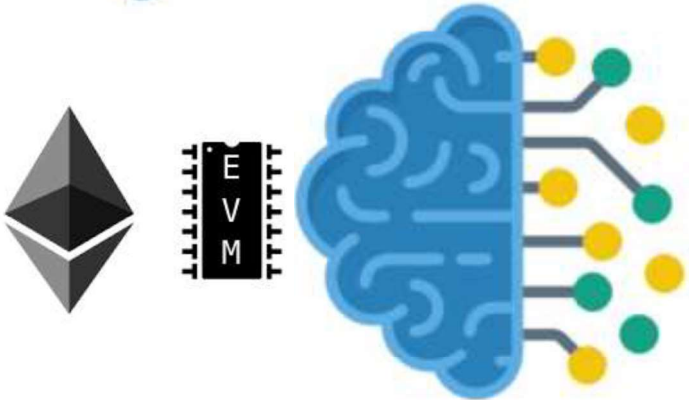
# Contracts, Inheritance, Libraries & Security Consideration
## Module 4-Mastering Solidity Programming

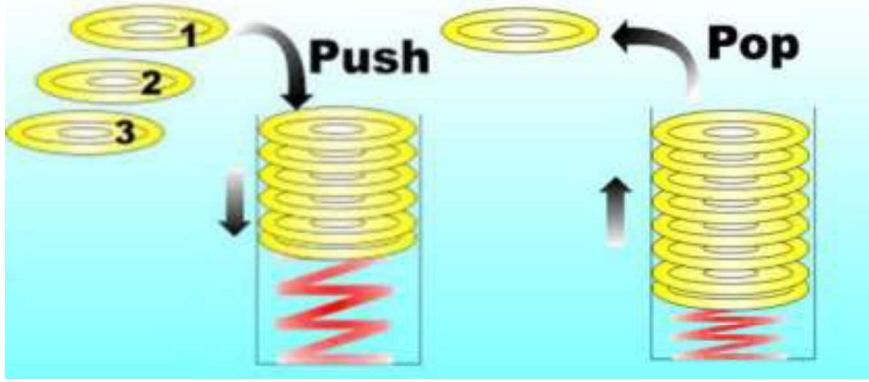






# Creating Contracts

1. Contracts in Solidity are similar to classes in object-oriented languages. They contain persistent data in state variables and functions that can modify these variables.
2. A contract and its functions need to be called for anything to happen. There is no **"cron" concept** in Ethereum **to call a function at a particular event automatically**.
3. Contracts can be created **"from outside"** via **Ethereum transactions** or from **within Solidity contracts**.
4. One way to **create contracts programmatically** on Ethereum is via the **JavaScript API web3.js**. It has a function called **web3.eth.Contract** to facilitate contract creation.
5. When a contract is created, its **constructor** is executed once.
   a) A constructor is **optional**.
   b) Only **one constructor is allowed**, which means overloading is not supported.
   c) After executing constructor **final code of the contract** is stored on the blockchain.
   d) This code includes all public and external functions and all functions that are reachable from there through function calls.
   e) The **deployed code** does not include the **constructor code** or **internal functions** only called from the constructor.
6. Internally, **constructor arguments** are passed **ABI encoded** after the code of the contract itself, but you do not have to care about this if you use **web3.js**.
7. If a contract wants to create another contract, the source code (and the binary) of the created contract has to be known to the creator.

# Creating Contracts

```
contract OwnedToken {
    // `TokenCreator` is a contract type that is defined below. It is fine to reference it
    //  as long as it is not used  to create a new contract.
    TokenCreator creator;
    address owner;
    bytes32 name;
// This is the constructor which registers the creator and the assigned name.
    constructor(bytes32 _name) {
    // State variables are accessed via their name and not via e.g. `this.owner`. Functions can be
    // accessed directly or through `this.f`, but the latter provides an external view to the function.
    // In constructor, you should not access functions externally, because the function does not exist yet.
   owner = msg.sender;
    // We perform an explicit type conversion from `address` to `TokenCreator` and assume that the type of
    // calling contract is `TokenCreator`, there is no real way to verify that.
        creator = TokenCreator(msg.sender);
        name = _name;
    }

    function changeName(bytes32 newName) public {
        // Only the creator can alter the name. We compare the contract based on its address which can
        // be retrieved by explicit conversion to address.
        if (msg.sender == address(creator))
            name = newName;
     }
```

# Creating Contracts

```
function transfer(address newOwner) public {
        // Only the current owner can transfer the token.
        if (msg.sender != owner) return;

        // We ask the creator contract if the transfer should proceed. If  the call fails the execution also fails here.
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }
}

contract TokenCreator {
    function createToken(bytes32 name) public returns (OwnedToken tokenAddress)   {
        // Create a new `Token` contract and return its address. From the JavaScript side, the return type of this function is
       // `address`,  as this is closest type available in the ABI.
        return new OwnedToken(name);
    }

    function changeName(OwnedToken tokenAddress, bytes32 name) public {  tokenAddress.changeName(name);  }

    // Perform checks to determine if transferring a token to the `OwnedToken` contract should proceed
    function isTokenTransferOK(address currentOwner, address newOwner)
        public pure returns (bool ok)    {
        // Check an arbitrary condition to see if transfer should proceed
        return keccak256(abi.encodePacked(currentOwner, newOwner))[0] == 0x7f;
    }
}
```

# Getter Functions

1. The compiler automatically creates getter functions for all public state variables. State variables can be initialized when they are declared.

```
contract C { uint public data = 42; }

contract Caller {
    C c = new C();
    function f() public view returns (uint) { return c.data(); }
}
```

2. The getter functions have external visibility. If symbol is accessed **internally** (**without this**), it **evaluates** to a **state variable**. If it is accessed **externally** (**with this**), it evaluates to **a function**.

```
contract C {     uint public data;
    function x() public returns (uint) {
        data = 3;                  // internal access
        return this.data();        // external access
    }
}
```

3. If you have a public state variable of array type, then you can only retrieve single elements of the array via the generated getter function. This mechanism exists to avoid high gas costs when returning an entire array.
4. If you want to return an entire array in one call, then you need to write a function.

# Function Modifiers

1. Modifiers can be used to change the **behaviour of functions** in a declarative way. You can use a modifier to **automatically check a condition** prior to **executing the function**.
2. Modifiers are **inheritable properties** of contracts and may be **overridden** by derived contracts, but only **if they are marked virtual**.
3. Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented.
4. Explicit returns from a modifier or function body only leave the current modifier or function body.
5. Arbitrary expressions are allowed for modifier arguments and in this context, all symbols visible from the function are visible in the modifier.
6. Symbols introduced in the modifier are not visible in the function.
7. The **function body is inserted** where the **special symbol "_;"** appears in the **definition of a modifier**. Thus, if condition of modifier is satisfied while calling this function, the function is executed and otherwise, an exception is thrown.

# Function Modifiers

```
contract Owner {
  address owner;
  constructor() public {  owner = msg.sender;  }
  modifier onlyOwner {
    require(msg.sender == owner);
    _;
  }
  modifier costs(uint price) {
    if (msg.value >= price) {
      _;
    }
  }
}

 // This contract inherits modifiers from `owner`
contract Register is Owner {
  mapping (address => bool) registeredAddresses;
  uint price;
  constructor(uint initialPrice) public { price = initialPrice; }

  function register() public payable costs(price) { registeredAddresses[msg.sender] = true; }
  function changePrice(uint _price) public onlyOwner { price = _price; }
}
```

# Constant & Immutable State Variables

1. **State variables** can be declared as constant or immutable. In both cases, the variables cannot be modified after the contract has been constructed.
   a) For constant variables, the value has to be fixed at compile-time
   b) For immutable, value can be assigned at construction time.
   c) No storage slot is reserved for such variables and their occurrence is replaced by their value.
2. Compared to regular state variables, the gas costs of constant & immutable variables are much lower.
   a) For a **constant variable**, the expression assigned to it is **copied** to all the places **where it is accessed** and also **re-evaluated each time**. This allows for local optimizations.
   **b) Immutable variables** are **evaluated once at construction time** and their value is copied to all the places in the code where they are accessed. For these values, 32 bytes are reserved.
   c) Due to this, constant values can sometimes be cheaper than immutable values.
3. **Constant:** For constant variables, the value has to be a constant at compile time and it has to be assigned where the variable is declared.
   a) Any expression that accesses storage, blockchain data or execution data or makes calls to external contracts is disallowed.
   b) Expressions that might have a side-effect on memory allocation are allowed, but those that might have a side-effect on other memory objects are not.
4. **Immutable:** Variables declared as immutable are a bit less restricted than those declared as constant.
   a) Immutable variables can be assigned an arbitrary value in the constructor of the contract or at the point of their declaration.
   b) They cannot be read during construction time and can only be assigned once.
   c) Contract creation code generated by compiler will modify the contract's runtime code before it is returned by replacing **all references to immutables** by the **values assigned to the them**.

# Constant & Immutable State Variables

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.7.2;

uint constant X = 32**22 + 8;

contract C {
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
    uint immutable decimals;
    uint immutable maxBalance;
    address immutable
    Functions

    constructor(uint _decimals, address _reference) {
        decimals = _decimals;
        // Assignments to immutables can even access the environment.
        maxBalance = _reference.balance;
    }

    function isBalanceTooHigh(address _other) public view returns (bool) {
        return _other.balance > maxBalance;
    }
}
```

# Functions

1. Functions can be defined inside and outside of contracts.
2. Code of functions outside of a contract, also called **"free functions"** is included in all contracts that call them, similar to internal library functions.

```
function sum(uint[] memory _arr) pure returns (uint s) {
    for (uint i = 0; i < _arr.length; i++)
        s += _arr[i];
}

contract ArrayExample {
    bool found;
    function f(uint[] memory _arr) public {
        // This calls the free function internally. The compiler will add its code to the contract.
        uint s = sum(_arr);
        require(s >= 10);
        found = true;
    }
}
```

3. Functions take **typed parameters as input** and unlike in many other languages, may also return an **arbitrary number of values** as output.
4. Function parameters are declared the same way as variables and the **name of unused parameters can be omitted**.