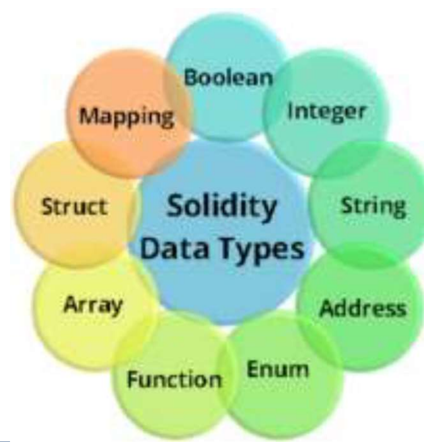
रा.इ.सू.प्रौ.सं
NIELIT



National Institute of Electronics and Information Technology

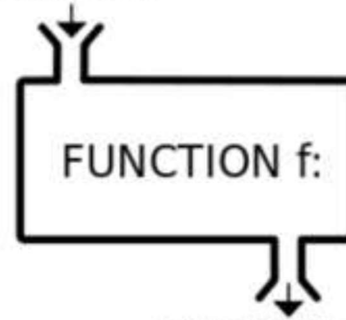
Data Types, Functions, Operators and Exception handling

Module 3-Basics of Solidity Programming

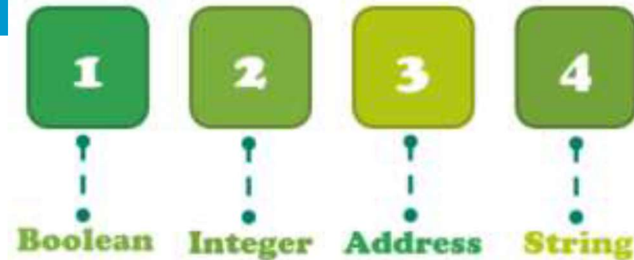
1 **Structs**

2 **Arrays**

3 **Mappings**

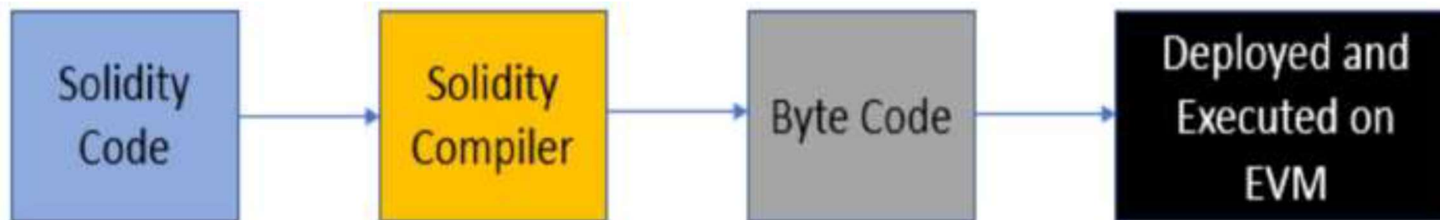


VALUE TYPES



Introduction

1. Solidity is a programming language targeting **Ethereum Virtual Machine (EVM)**. Ethereum blockchain helps extend its functionality by writing and executing code known as smart contracts.
2. The **smart contracts** are similar to **object-oriented classes**. EVM executes code that is part of smart contracts. Smart contracts are written in Solidity.
3. However, EVM does not understand the high-level constructs of Solidity. EVM understands lower-level instructions called bytecode. Solidity code needs a compiler to take its code and convert it into bytecode that is understandable by EVM.



4. Solidity comes with a compiler to do this job, known as the **Solidity compiler** or **solc**. Solidity code is written in Solidity files that have the **extension .sol**.
5. Solidity is a **statically-typed**, **case-sensitive** and **object-oriented programming** (OOP) language. Although, it has certain limitation like **variable data types** should be defined and known at compile time.
6. A Solidity file is composed of the following **four high-level constructs**:

a) Pragma **b) Comments** **c) Import** **d) Contracts/library/interface**

A Simple Smart Contract

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.8.0;
contract SimpleStorage {
uint storedData;
function set(uint x) public {
storedData = x;
}
function get() public view returns (uint) {
return storedData;
}
}
```

1. The first line tells you that the source code is **licensed under the GPL version 3.0**. **Machine-readable license specifiers** are important in a setting where publishing the source code is the default.
2. **Pragmas** are common **instructions for compilers** about **how to treat the source code**.
3. The next line specifies that the source code is written for Solidity version 0.4.16, or a newer version of the language up to, but not including version 0.8.0. This is to ensure that the contract is not compilable with a new (breaking) compiler version, where it could behave differently.
4. The line uint storedData; declares a **state variable** called storedData of type uint (unsigned integer of 256 bits). You can think of it as a single slot in a database that you can query and alter by calling functions. The contract defines the functions **set** and **get** that can be used to modify or retrieve the value of the variable.
5. To access a state variable, you do not need the prefix **this**. as is common in other languages.
6. The **statement terminator** in Solidity is the semicolon: **;**.
7. **Single-line comments** (//) and **multi-line comments** (/*...*/) are possible

Ethereum Natural Language Specification (NatSpec)

1. Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables and more. The entire specification is available at https://github.com/ethereum/wiki/wiki/Ethereum-Natural-Specification-Format
2. This documentation is segmented into developer-focused messages and end-user-facing messages. These messages may be shown to the end user (the human) at the time that they will interact with the contract (i.e. sign a transaction).
3. It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI).
4. NatSpec includes the formatting for comments that the smart contract author will use, and which are understood by the Solidity compiler.
5. Documentation is inserted above each class, interface and function using the doxygen notation format. A public state variable is equivalent to a function for the purposes of NatSpec.
6. For Solidity you may choose /// for single line comments or /** and ending with */ for multi-line comments.
7. The Solidity compiler **only interprets tags** if they are external or public. You are welcome to use similar comments for your internal and private functions, but those will not be parsed.

```
pragma solidity >0.6.10 <0.8.0;
/// @title A simulator for trees
/// @author Larry A. Gardner
/// @notice You can use this contract for only the most basic simulation
/// @dev All function calls are currently implemented without side effects
contract Tree {
```

The import statement

1. Solidity supports import statements to help modularise your code. At a global level, you can use import statements of the following form:

   import "filename";

2. This statement imports all global symbols from “filename” into the current global scope. This form is not recommended for use, because it unpredictably pollutes the namespace.
3. It is better to import specific symbols explicitly. The following example creates a new global symbol symbolName whose members are all the global symbols from "filename":

   import * as symbolName from "filename";OR
   import "filename" as symbolName;

   which results in all global symbols being available in the format **symbolName.symbol**.

4. If there is a naming collision, you can rename symbols while importing. For example, the code below creates new global symbols alias and symbol2 which reference symbol1 and symbol2 from inside "filename", respectively.

   import {symbol1 as alias, symbol2} from "filename";

5. Filename is always treated as a path with / as directory separator, and **.** as the current and **..** as the parent directory. When . or .. is followed by a character except /, it is not considered as the current or the parent directory.
6. All path names are treated as absolute paths unless they start with the current . or the parent directory ..
7. It depends on the compiler how to actually resolve the paths.

Structure of a Contract

1. Contracts in Solidity are similar to classes in object-oriented languages.
   a) Each contract can contain declarations of **State Variables**, **Functions**, **Function Modifiers**, **Events**, **Struct Types** and **Enum Types**.
   b) Furthermore, contracts can **inherit** from other contracts.
   c) There are also **special kinds of contracts** called **libraries** and **interfaces**.
2. **State Variables:** variables whose values are permanently stored in contract storage.

```
contract SimpleStorage {
uint storedData; // State variable
// ...
}
```

3. **Functions:** Functions are the executable units of code. Functions are usually defined inside a contract, but they can also be defined outside of contracts.

```
contract SimpleAuction {
function bid() public payable { // Function
// ...
}
}
// Helper function defined outside of a contract
function helper(uint x) pure returns (uint) {
return x * 2;
}
```

4. Function Calls can happen internally or externally and have **different levels of visibility** towards other contracts.
5. Functions **accept parameters** and **return variables** to pass parameters and values between them.

Structure of a Contract

4. **Function Modifiers:** Function modifiers can be used to **amend the semantics of functions** in a declarative way. **Overloading**, that is, having the same modifier name with different parameters, **is not possible**. Like functions, **modifiers can be overridden**.

```
contract Purchase {
address public seller;
modifier onlySeller() {                    // Modifier
require(msg.sender == seller, "Only seller can call this.");
_;
}
function abort() public view onlySeller {  // Modifier usage
// ...
}}
```

5. **Events:** Events are used primarily for informing the calling application about the current state of the contract by means of the logging facility of EVM. Instead of applications keep on polling the contract for certain state changes; the contract can inform them by means of events.

```
contract SimpleAuction {
event HighestBidIncreased(address bidder, uint amount); // Event
function bid() public payable {
// ...
emit HighestBidIncreased(msg.sender, msg.value); // Triggering event
}}
```

Structure of a Contract

6. **Struct Types:** Structs are custom defined types that can group several variables.

```
//structure definition
struct myStruct {
    string name; //variable fo type string
    uint myAge; // variable of unsigned integer type
    bool isMarried; // variable of boolean type
    uint[] bankAccountsNumbers; // variable - dynamic array of unsigned integer
}
```

a) To create an instance of a structure, there is no need to explicitly use the **new** keyword. The new keyword can only be used to create an **instance** of **contracts** or **arrays**.

```
human = myStruct("Ritesh",10,true,new uint[](3)); //using struct myStruct
```

b) Multiple instance of struct can be created in functions.

c) Structs can **contain array** and the **mapping variables**

d) Mappings and arrays can store values of type struct.

7. **Enum Types:** Enums can be used to create custom types with a finite set of 'constant values'

```
contract Purchase {
enum State { Created, Locked, Inactive } // Enum

State _state=State.Created;
}
```

Structure of a Contract

8. **Function Modifiers:** Function modifiers can be used to **amend the semantics of functions** in a declarative way. **Overloading**, that is, having the same modifier name with different parameters, **is not possible**. Like functions, **modifiers can be overridden**.

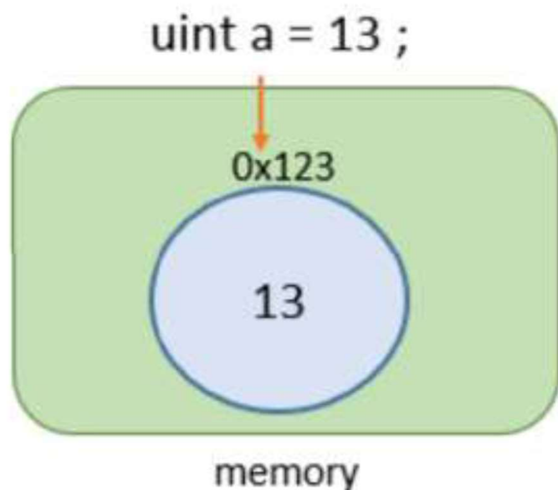
```
contract Purchase {
address public seller;
modifier onlySeller() {                    // Modifier
require(msg.sender == seller, "Only seller can call this.");
_;
}
function abort() public view onlySeller {            // Modifier usage
// ...
}
}
```

9. **Events:** Events are convenience interfaces with the EVM logging facilities.

```
contract SimpleAuction {
event HighestBidIncreased(address bidder, uint amount); // Event
function bid() public payable {
// ...
emit HighestBidIncreased(msg.sender, msg.value); // Triggering event
}
}
```

Data Types in Solidity

1. Solidity is a **statically typed language**, which means that the type of each variable (state and local) needs to be specified.
2. Solidity provides **several elementary types** which can be combined to form complex types.
3. The concept of "**undefined**" or "**null**" values does not exist in Solidity, but newly declared variables **always have a default value** dependent on its type.
4. Solidity data types can be classified in the two types: **Value types** and **Reference types**
5. **Value types** maintains independent copies of variables and changing the value in one variable does not effect value in another variable. However, changing values in **reference type** variables ensures that anybody referring to that variables gets updates value.
6. **Value types:** A type is referred as value type if it holds the data (value) directly within the memory owned by it. These types have values stored with them, instead of elsewhere.



- In this example, a variable of data type unsigned integer (uint) is declared with 13 as its data(value).
- The variable has a memory space allocated by EVM, which is referred as 0x123 and this location has the value 13 stored.
- Accessing this variable will provide us with the value 13 directly:
- **Value types are types that do not take more than 32 bytes of memory in size.**