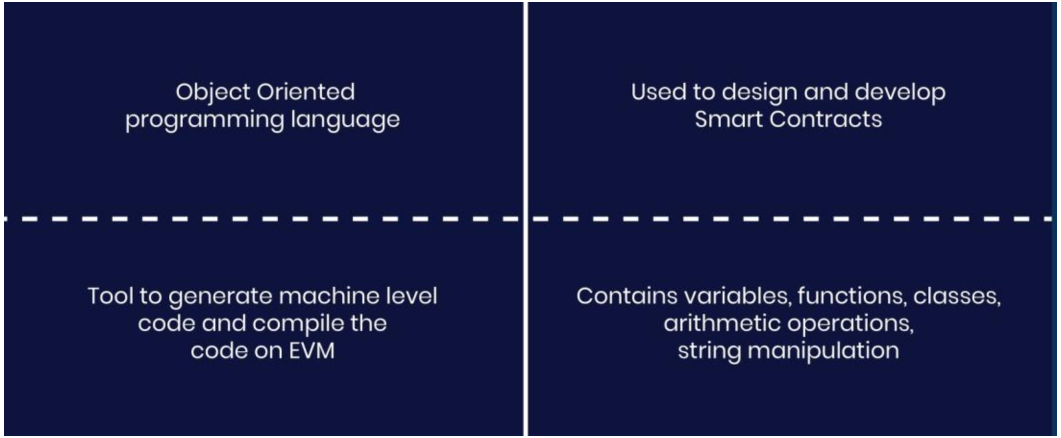
Solidity for Smart Contracts

- Solidity is an object-oriented, high-level language to implement smart contracts on the Ethereum Virtual Machine.
- Smart contracts are programs which govern the behaviour of accounts within the Ethereum state.



https://docs.soliditylang.org/en/v0.8.10/

Solidity Programming

- It is an object-oriented programming language that was specially built by the Ethereum team itself to enable developing and designing smart contracts on Blockchain platforms.
- Solidity programming language is used to write smart contracts to implement business logic in the system to generate a chain of transaction records.
- Solidity is a curly-bracket language. It is influenced by C++, Python and JavaScript, and is designed to target the Ethereum Virtual Machine.
- Not only a language, but Solidity programming is also a tool that generates machine-level code and compiles the code on the EVM.

Solidity Programming

- Having similarities with C and C++, Solidity programming is simple to learn and understand.
- Any developer with knowledge of C, C++ or Python for that matter, can quickly familiarize with the concepts and syntax of Solidity, which is very similar to those languages. For example, a “main” in C is equivalent to a “contract” in Solidity.
- Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features.
- Like other languages, Solidity programming has variables, functions, classes, arithmetic operations, string manipulation and many other concepts that are available in other modern programming languages.

https://docs.soliditylang.org/en/v0.8.10/

Layout of a Solidity Source File

Source files can contain an arbitrary number of

- **contract definitions**,
- **import directives**,
- **pragma directives,**
- **struct**, **enum**, **function**, **error** and
- **constant variable definitions**.

Layout of a Solidity Source File

SPDX License Identifier

- Making source code available always touches on legal problems with regards to copyright, the Solidity compiler encourages the use of machine-readable **SPDX license identifiers**.
- Every source file should start with a comment indicating its license:

// SPDX-License-Identifier: MIT

- The compiler does not validate that the license is part of the **list allowed by SPDX**, but it does include the supplied string in the **bytecode metadata**.
- If you do not want to specify a license or if the source code is not open-source, please use the special value **UNLICENSED**.
    - Supplying this comment of course does not free you from other obligations related to licensing like having to mention a specific license header in each source file or the original copyright holder.
    - The comment is recognized by the compiler anywhere in the file at the file level, but it is recommended to put it at the top of the file.

Layout of a Solidity Source File

Pragmas

- The **pragma** keyword is used to enable certain compiler features or checks.
- A pragma directive is always local to a source file, so you have to add the pragma to all your files if you want to enable it in your whole project.
- If you **import** another file, the pragma from that file does not automatically apply to the importing file.

Layout of a Solidity Source File

Version Pragma

- Source files can (and should) be annotated with a version pragma to reject compilation with future compiler versions that might bring incompatible changes.
- It is always a good idea to read through the changelog at least for releases that contain breaking changes.
- These releases always have versions of the form **0.x.0** or **x.0.0**.
- The version pragma is used as follows: **pragma solidity ^0.5.2**;
  - A source file with the line above does not compile with a compiler earlier than version 0.5.2, and it also does not work on a compiler starting from version 0.6.0 (this second condition is added by using ^). Because there will be no breaking changes until version 0.6.0, you can be sure that your code compiles the way you intended.

Layout of a Solidity Source File

Importing other Source Files

- Solidity supports import statements to help modularise your code that are similar to those available in JavaScript.
- However, Solidity does not support the concept of a **default export**.
- At a global level, you can use import statements of the following form:

```
import "filename";
```

- The filename part is called an import path.
- This statement imports all global symbols from “filename” (and symbols imported there) into the current global scope so 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";
```

Layout of a Solidity Source File

Comments

- Single-line comments (**//**) and multi-line comments (**/*...*/**) are possible.

```
// This is a single-line comment.

/*
This is a
multi-line comment.
*/
```

Smart Contacts in Solidity

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

- A simple example

Smart Contacts in Solidity

Pragmas are common instructions for compilers about how to treat the source code.

source code is licensed under the GPL version 3.0

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

source code is written for Solidity version 0.4.16, or a newer version of the language up to, but not including version 0.9.0.

A contract in the sense of Solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain.

Smart Contacts in Solidity

Pragmas are common instructions for compilers about how to treat the source code.

source code is licensed under the GPL version 3.0

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

source code is written for Solidity version 0.4.16, or a newer version of the language up to, but not including version 0.9.0.

The line uint **storedData**; declares a state variable called **storedData** of type **uint** (unsigned integer of 256 bits).

the contract defines the functions **set** and **get** that can be used to modify or retrieve the value of the variable.

Smart Contacts in Solidity

- A detailed example

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping (address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() {
        minter = msg.sender;
    }

    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Errors allow you to provide information about
    // why an operation failed. They are returned
    // to the caller of the function.
    error InsufficientBalance(uint requested, uint available);

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        if (amount > balances[msg.sender])
            revert InsufficientBalance({
                requested: amount,
                available: balances[msg.sender]
            });

        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

Smart Contacts in Solidity

- The line address **public minter**; declares a state variable of type address.
    - It is a 160-bit value that does not allow any arithmetic operations.
    - It is suitable for storing addresses of contracts, or a hash of the public half of a keypair belonging to external accounts.
- Keyword **public** automatically generates a **function** that allows you to access the current value of the state variable from outside of the contract.
    - Without this, other contracts have no way to access the variable.
- The next line, **mapping (address => uint) public balances**; also creates a public state variable, but it is a more complex datatype.
    - The **mapping** type maps addresses to **unsigned integers**.
    - Mappings can be seen as hash tables which are virtually initialised such that every possible key exists from the start and is mapped to a value whose byte-representation is all zeros.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping (address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() {
        minter = msg.sender;
    }

    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Errors allow you to provide information about
    // why an operation failed. They are returned
    // to the caller of the function.
    error InsufficientBalance(uint requested, uint available);

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        if (amount > balances[msg.sender])
            revert InsufficientBalance({
                requested: amount,
                available: balances[msg.sender]
            });

        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

Smart Contacts in Solidity

- The line **event Sent(address from, address to, uint amount);** declares an "**event**", which is emitted in the last line of the function send.
- Ethereum clients such as web applications can listen for these events emitted on the blockchain without much cost.
- As soon as it is emitted, the listener receives the arguments **from, to** and **amount**, which makes it possible to track transactions.

- The **constructor** is a special function that is executed during the creation of the contract and cannot be called afterwards.
- In this case, it permanently stores the address of the person creating the contract.
- The **msg** variable (together with **tx** and **block**) is a **special global variable** that contains properties which allow access to the blockchain.
- **msg.sender** is always the address where the current (external) function call came from.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping (address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() {
        minter = msg.sender;
    }

    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Errors allow you to provide information about
    // why an operation failed. They are returned
    // to the caller of the function.
    error InsufficientBalance(uint requested, uint available);

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        if (amount > balances[msg.sender])
            revert InsufficientBalance({
                requested: amount,
                available: balances[msg.sender]
            });

        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

Smart Contacts in Solidity

- The functions that make up the contract, and that users and contracts can call are **mint** and **send**.
- The **mint** function sends an amount of newly created coins to another address.
- The **require** function call defines conditions that reverts all changes if not met.
- In this example, **require(msg.sender == minter)**; ensures that only the creator of the contract can call **mint**.
- In general, the creator can mint as many tokens as they like, but at some point, this will lead to a phenomenon called “overflow”.

- **Errors** allow you to provide more information to the caller about why a condition or operation failed. Errors are used together with the **revert statement**.
- The revert statement unconditionally aborts and reverts all changes similar to the require function, but it also allows you to provide the name of an error and additional data which will be supplied to the caller so that a failure can more easily be debugged or reacted upon.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping (address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() {
        minter = msg.sender;
    }

    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Errors allow you to provide information about
    // why an operation failed. They are returned
    // to the caller of the function.
    error InsufficientBalance(uint requested, uint available);

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        if (amount > balances[msg.sender])
            revert InsufficientBalance({
                requested: amount,
                available: balances[msg.sender]
            });

        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

Smart Contacts in Solidity

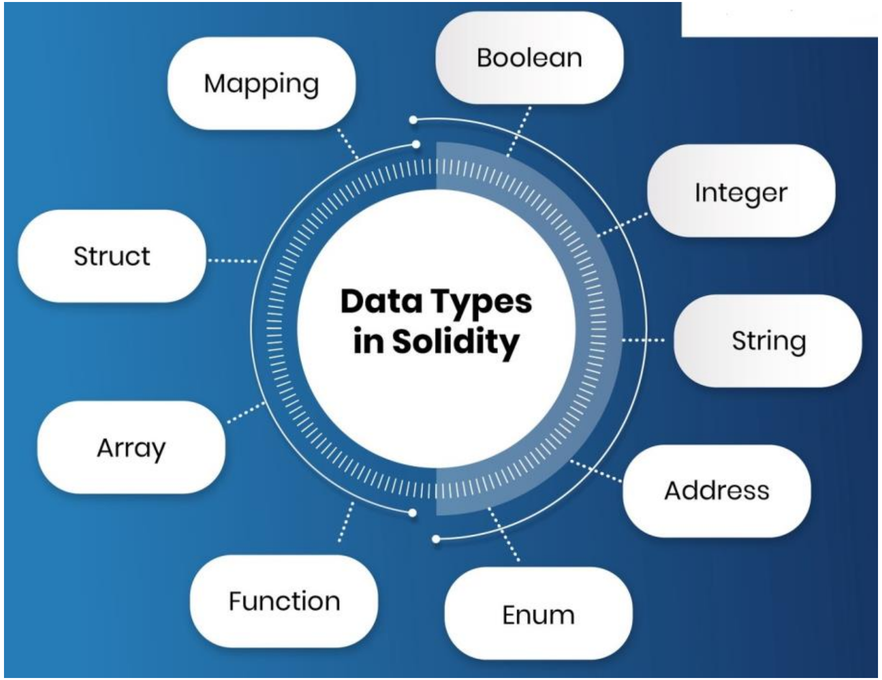
- The **send** function can be used by anyone (who already has some of these coins) to send coins to anyone else.
- If the sender does not have enough coins to send, the **if** condition evaluates to true.
- As a result, the **revert** will cause the operation to fail while providing the sender with error details using the **InsufficientBalance** error.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping (address => uint) public balances;

    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() {
        minter = msg.sender;
    }

    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // Errors allow you to provide information about
    // why an operation failed. They are returned
    // to the caller of the function.
    error InsufficientBalance(uint requested, uint available);

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        if (amount > balances[msg.sender])
            revert InsufficientBalance({
                requested: amount,
                available: balances[msg.sender]
            });

        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

Basics of Solidity Programming

Types

- Solidity is a statically typed language, which means that the type of each variable (state and local) needs to be specified.
- Solidity provides several elementary types which can be combined to form complex types.



https://docs.soliditylang.org/en/v0.8.10/cheatsheet.html#order

Value Types

The following types are also called value types because variables of these types will always be passed by value, i.e., they are always copied when they are used as function arguments or in assignments,

- **Booleans**
- **Integers**
- **Fixed Point Numbers**
- **Contract Types**
- **Arrays**
- **Enums**
- **Functions**

Value Types

Booleans

- Depending on the state of the condition, Boolean data type returns ‘1’ when true and ‘0’ when false. The possible values are constants, i.e., true and false.
- **bool**: The possible values are constants **true** and **false**.
  - Operators:
    - `!` (logical negation)
    - `&&` (logical conjunction, “and”)
    - `||` (logical disjunction, “or”)
    - `==` (equality)
    - `!=` (inequality)

Value Types

Integers

- **int/uint**: Signed and unsigned integers of various sizes.
  Keywords **uint8** to **uint256** in steps of **8** (unsigned of 8 up to 256 bits) and **int8** to **int256**.
- **uint** and **int** are aliases for **uint256** and **int256**, respectively.
  - Operators:
    - Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
    - Bit operators: `&`, `|`, `^` (bitwise exclusive or), `~` (bitwise negation)
    - Shift operators: `<<` (left shift), `>>` (right shift)
    - Arithmetic operators: `+`, `-`, unary `-` (only for signed integers), `*`, `/`, `%` (modulo), `**` (exponentiation)

For an integer type **X**, you can use **type(X).min** and **type(X).max** to access the minimum and maximum value representable by the type.

Value Types

Integers

- Comparisons
- Bit Operations
- ShiftsAddition, Subtraction and Multiplication
- Division
- Modulo
- Exponentiation

https://docs.soliditylang.org/en/v0.8.10/types.html

Value Types

Fixed Point Numbers

- **fixed / ufixed:** Signed and unsigned fixed-point number of various sizes.
- Fixed point numbers are not fully supported by Solidity yet. They can be declared but cannot be assigned to or from.
  - Operators:
    - Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
    - Arithmetic operators: `+`, `-`, unary `-`, `*`, `/`, `%` (modulo)

Value Types

Address

- The address type comes in two flavors, which are largely identical:
  - **address**: Holds a 20-byte value (size of an Ethereum address).
  - **address payable**: Same as address, but with the additional members **transfer** and **send**.

The idea behind this distinction is that **address payable** is an address you can send Ether to, while a plain **address** cannot be sent Ether.

For example:

```
address payable x = address(0x123);
address myAddress = address(this);
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

Value Types

Address

- **Members of Address-**
  - **balance** and **transfer**

It is possible to query the balance of an address using the property **balance** and to send Ether (in units of wei) to a payable address using the **transfer** function.

For example:

```
address payable x = address(0x123);
address myAddress = address(this);
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

Value Types

Address

- **Members of Address-**
  - **send**

Send is the low-level counterpart of **transfer**. If the execution fails, the current contract will not stop with an exception, but **send** will return **false**.

  - **Call, delegatecall** and **staticall**

In order to interface with contracts that do not adhere to the Application Binary Interface (ABI), or to get more direct control over the encoding, the functions call, delegatecall and staticcall are provided. They all take a single byte's memory parameter and return the success condition (as a bool) and the returned data (bytes memory).

Value Types

Contract Types

- Every **contract** defines its own type.
- You can implicitly convert contracts to contracts they inherit from.
- Contracts can be explicitly converted to and from the **address** type.

Value Types

Arrays

Solidity programming supports single as well as multidimensional arrays and the syntax is similar to other OOP languages.

- Fixed-size byte arrays
- Dynamically-sized byte arrays.

Value Types

Arrays

- **Fixed sized**- The value types **bytes1**, **bytes2**, **bytes3**, ..., **bytes32** hold a sequence of bytes from one to up to 32.
  - byte is an alias for bytes1.
  - Operators:
    - Comparisons: `<=`, `<`, `==`, `!=`, `>=`, `>` (evaluate to `bool`)
    - Bit operators: `&`, `|`, `^` (bitwise exclusive or), `~` (bitwise negation)
    - Shift operators: `<<` (left shift), `>>` (right shift)
    - Index access: If `x` is of type `bytesI`, then `x[k]` for `0 <= k < I` returns the `k` th byte (read-only).

For example:

```
contract FixedArrays{
    bytes2 public x; //fixed-size array of 2 octets
    bytes3 public y; //fixed-size array of 3 bytes
}
```

Value Types

Arrays

- **Dynamic:** In dynamic array, **bytes** represents the dynamically-sized byte array whereas **string** represents the dynamically-sized UTF-8-encoded string.

For example:

```
Contract dynamic array{
    uint[] public myArray = [1, 2, 3, 4];
}
```

Value Types

Enum

- It is one way to create a user-defined type in Solidity.
- They are explicitly convertible to and from all integer types, but implicit conversion is not allowed.
- The explicit conversion from integer checks at runtime that the value lies inside the range of the enum and causes a **failing assert** otherwise.
- Enums needs at least one member, and its default value when declared is the first member.
- Enums cannot have more than 256 members.

For example:

enum *ActionChoices { GoLeft, GoRight, GoStraight, SitStill }*
ActionChoices choice;

Value Types

Functions

- Function types are the types of functions.
- Variables of function type can be assigned from functions and function parameters of function type can be used to pass functions to and return functions from function calls.
- Function types come in two flavors - internal and external functions.

For example:

function *(<parameter types>)* *{**internal**|**external**}* *[**pure**|**view**|**payable**]* *[**returns** (<**return** types>)]*

- **pure** functions can be declared in which case they promise not to read from or modify the state.
- **view** functions can be declared in which case they promise not to modify the state.
- **payable** functions can be declared in which case they promise to transfer.

Value Types

Functions

- Functions must be specified as being external, public, internal or private. For state variables, external is not possible.
  - **external:** External functions are part of the contract interface, which means they can be called from other contracts and via transactions.
  - **public:** These functions are part of the contract interface and can be either called internally or via messages. For public state variables, an automatic getter function is generated.
  - **internal:** Those functions and state variables can only be accessed internally (i.e., from within the current contract or contracts deriving from it, without using this.
  - **private:** These functions and state variables are only visible for the contract they are defined in and not in derived contracts.

Value Types

Modifier

- Typically, a **modifier** ensures the logicality of any condition before executing the code for a smart contract.

For example:

```
modifier priceGreaterThan10000{
        require(price > 10000);
        _;}
function setPrice(uint _price) public onlyOwner priceGreaterThan10000{
        require(_price > price);
        price = _price;
    }
```

Value Types

Mapping

- mapping types with the syntax mapping **(_KeyType => _ValueType)**.
- The **_KeyType** can be any elementary type.
- This means it can be any of the built-in value types plus bytes and string.

For example:

```
pragma solidity >=0.4.0 <0.6.0;
contract MappingExample {
mapping(address => uint) public balances;
function update(uint newBalance) public {
balances[msg.sender] = newBalance; } }
```

Basic Data Types

State variables

- The variables whose values are permanently stored in contract storage.

For example:

```
pragma solidity >=0.4.0 <0.6.0;
contract SimpleStorage {
uint storedData; // State variable //
... }
```

Basic Data Types

Events

- They are convenience interfaces with the EVM logging facilities.

For example:

```
pragma solidity >=0.4.21 <0.6.0;
contract SimpleAuction {
event HighestBidIncreased(address bidder, uint amount); // Event
function bid() public payable {
// ...
 emit HighestBidIncreased(msg.sender, msg.value ); // Triggering event } }
```

Basic Data Types

Structures

- Structs are custom defined types that can group several variables.

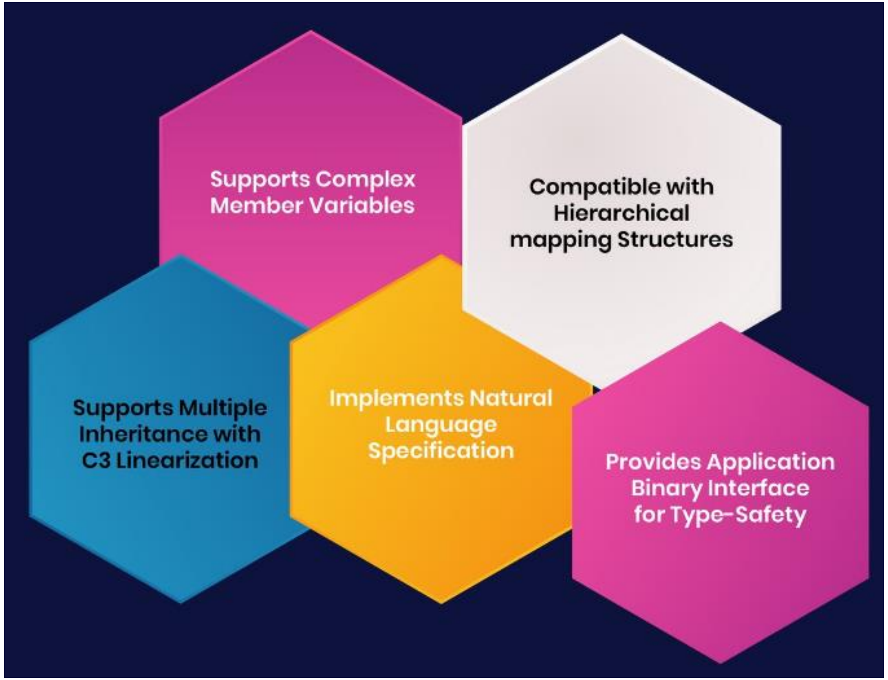
For example:

```
pragma solidity >=0.4.0 <0.6.0;
contract Ballot {
 struct Voter { // Struct
uint weight;
bool voted;
address delegate;
uint vote; }
 }
```

Advantages of Solidity Programming

- Solidity programming, apart from the basic data types, also supports complex data types and member variables. As read above, data structures like mapping are compatible with Solidity programming.
- To enable type-safety, Solidity programming provides an ABI. The ABI issues an error if the compiler encounters a data type mismatch for any variable.
- Solidity supports multiple inheritance with C3 linearization which follows an algorithm to determine the method that should be used in case of multiple inheritance.
- Solidity programming refers to the 'Natural Language Specification,' for converting user-centric specifications to machine understandable language.

Advantages of Solidity Programming



Solidity Programming

- Create account with metamask chrome extension (Ethereum wallet)
- Switch to Ropsten test network
- Get coins from ropsten faucet

- Visit https://remix.ethereum.org for an online IDE

How to develop a custom blockchain?

Define the block

Libraries required

- **Datetime** (This module supplies classes for manipulating dates and times.)
- **Hashlib** (Secure hashes and message digests)

Import these libraries

```
import datetime
import hashlib
```

Step 1: Define the structure of a block

- **Define the structure for the block as a class:**

```
class Block:
    def __init__(self, previous_block_hash, data, timestamp):
        self.previous_block_hash = previous_block_hash
        self.data = data
        self.timestamp = timestamp
        self.hash = self.get_hash()
```

- #function to create our own hash this hash will be take all the data in the block header and run SHA 256 two time (inner and outer encryption)
- #Only considered the previous block hash, data and timestamp to be part of block for simplicity.

Step 1: Define the structure of a block

- **Define a class:**

```
class Block:
    def __init__(self, previous_block_hash, data, timestamp):
        self.previous_block_hash = previous_block_hash
        self.data = data
        self.timestamp = timestamp
        self.hash = self.get_hash()
```

Step 2: Define the function to perform the hashing of block components

```
def get_hash(self):                    #will take only object self as it has all we need for this function
#generating binary representation of the header
    header_bin = (str(self.previous_block_hash) +
                  str(self.data) +
                  str(self.timestamp)).encode()      #encode function
#we convert all the data in strings to feed it as input for hash function SHA256 (in this function two-level
hashing is illustrated in the from of inner hash and outer hash)
#encode() is used to encode the code into unicode
    inner_hash = hashlib.sha256(header_bin.encode()).hexdigest().encode()
#hexdigest() convert the data in hexadecimal format
#hashlib.sha256 is used for hashing using SHA256 from library hashlib
    outer_hash = hashlib.sha256(inner_hash).hexdigest()
    return outer_hash
```

Step 2: Define the function to perform the hashing of block components

```python
def get_hash(self):
    header_bin = (str(self.previous_block_hash) +
                  str(self.data) +
                  str(self.timestamp))

    inner_hash = hashlib.sha256(header_bin.encode()).hexdigest().encode()
    outer_hash = hashlib.sha256(inner_hash).hexdigest()
    return outer_hash
```

Step 3: Creating a genesis block (Static Coding Method)

```
def create_genesis_block():
    return Block("0", "0", datetime.datetime.now())
```

A static method is bound to a class rather than the objects for that class. This means that a static method can be called without an object for that class. This also means that static methods cannot modify the state of an object as they are not bound to it.

Step 3: Creating a genesis block (Static Coding Method)

```
@staticmethod
def create_genesis_block():
    return Block("0", "0", datetime.datetime.now())
```

Resultant of Step1, 2 and 3 (block.py)

[ ]:
```python
import datetime
import hashlib


class Block:
    def __init__(self, previous_block_hash, data, timestamp):
        self.previous_block_hash = previous_block_hash
        self.data = data
        self.timestamp = timestamp
        self.hash = self.get_hash()

    @staticmethod
    def create_genesis_block():
        return Block("0", "0", datetime.datetime.now())

    def get_hash(self):
        header_bin = (str(self.previous_block_hash) +
                      str(self.data) +
                      str(self.timestamp))

        inner_hash = hashlib.sha256(header_bin.encode()).hexdigest().encode()
        outer_hash = hashlib.sha256(inner_hash).hexdigest()
        return outer_hash
```

- save this as block.py

Create a Blockchain

Creating Blockchain

- Create a file named Blockchain.py
- Import the structure of block defined in block.py file

from block import Block

```python
from block import Block
```

Creating Blockchain

- Generate the Genesis block and print its hash on screen

```
b1 = Block.create_genesis_block()
Print(b1.hash)
```

Creating Blockchain

- Generate the Genesis block and print its hash on screen

```
from block import Block

block_chain = [Block.create_genesis_block()]

print("The genesis block has been created.")
print("Hash: %s" % block_chain[0].hash)
```

Creating Blockchain

- Lets add some elements to our blockchain
  - For simplicity, lets fix the number of blocks

num_blocks_to_add = 10

```
num_blocks_to_add = 10
```

Creating Blockchain

- Lets add some elements to our blockchain
  - Initialize the loop and append the blocks

```
for i in range(1, num_blocks_to_add):
    block_chain.append(Block(block_chain[i-1].hash,
                             "Block number %d" % i,
                             datetime.datetime.now()))
```

Creating Blockchain

- Lets add some elements to our blockchain
  - Initialize the loop and append the blocks

```
for i in range(1, num_blocks_to_add):
    block_chain.append(Block(block_chain[i-1].hash,
                             "Block number %d" % i,
                             datetime.datetime.now()))
```

Adding more blocks to Blockchain

```
from block import Block
import datetime

num_blocks_to_add = 10

block_chain = [Block.create_genesis_block()]

print("The genesis block has been created.")
print("Hash: %s" % block_chain[0].hash)

for i in range(1, num_blocks_to_add):
    block_chain.append(Block(block_chain[i-1].hash,
                             "Block number %d" % i,
                             datetime.datetime.now()))
    print("Block #%d created." % i)
    print("Hash: %s" % block_chain[-1].hash)
```

Specifying the number of blocks to add into blockchain

Adding more blocks to Blockchain

```
from block import Block
import datetime

num_blocks_to_add = 10

block_chain = [Block.create_genesis_block()]

print("The genesis block has been created.")
print("Hash: %s" % block_chain[0].hash)

for i in range(1, num_blocks_to_add):
    block_chain.append(Block(block_chain[i-1].hash,
                             "Block number %d" % i,
                             datetime.datetime.now()))
    print("Block #%d created." % i)
    print("Hash: %s" % block_chain[-1].hash)
```

Creating the genesis block and adding into blockchain

Adding more blocks to Blockchain

```
from block import Block
import datetime

num_blocks_to_add = 10

block_chain = [Block.create_genesis_block()]

print("The genesis block has been created.")
print("Hash: %s" % block_chain[0].hash)

for i in range(1, num_blocks_to_add):
    block_chain.append(Block(block_chain[i-1].hash,
                             "Block number %d" % i,
                             datetime.datetime.now()))
    print("Block #%d created." % i)
    print("Hash: %s" % block_chain[-1].hash)
```

Printing the value of hashed genesis block

Adding more blocks to Blockchain

```
from block import Block
import datetime

num_blocks_to_add = 10

block_chain = [Block.create_genesis_block()]

print("The genesis block has been created.")
print("Hash: %s" % block_chain[0].hash)

for i in range(1, num_blocks_to_add):
    block_chain.append(Block(block_chain[i-1].hash,
                             "Block number %d" % i,
                             datetime.datetime.now()))
    print("Block #%d created." % i)
    print("Hash: %s" % block_chain[-1].hash)
```

Running a loop to generate the given number of blocks

Adding more blocks to Blockchain

```
from block import Block
import datetime

num_blocks_to_add = 10

block_chain = [Block.create_genesis_block()]

print("The genesis block has been created.")
print("Hash: %s" % block_chain[0].hash)

for i in range(1, num_blocks_to_add):
    block_chain.append(Block(block_chain[i-1].hash,
                             "Block number %d" % i,
                             datetime.datetime.now()))
    print("Block #%d created." % i)
    print("Hash: %s" % block_chain[-1].hash)
```

Appending new blocks

Adding more blocks to Blockchain

```
from block import Block
import datetime

num_blocks_to_add = 10

block_chain = [Block.create_genesis_block()]

print("The genesis block has been created.")
print("Hash: %s" % block_chain[0].hash)

for i in range(1, num_blocks_to_add):
    block_chain.append(Block(block_chain[i-1].hash,
                             "Block number %d" % i,
                             datetime.datetime.now()))
    print("Block #%d created." % i)
    print("Hash: %s" % block_chain[-1].hash)
```

Printing the hash of the generated block

Adding more blocks to Blockchain

```
The genesis block has been created.
Hash: e1b8183133ecd5b65a29c8472b1524faecab6c8c166a58c7b151c943ed7bfd59
Block #1 created.
Hash: ec602252bfa4c52dcc0871819755d9eee68aae05aa4ef8e60c69bd41a2328ba1
Block #2 created.
Hash: 9a699414bc833af9f193ea86bee92c32fe6e3d3524239e76b2ceaeceb575fa63
Block #3 created.
Hash: c8683d36d4bdb0fd234d4d92ced23ca6a02c1b662fb2ea12f2693c287010443e
Block #4 created.
Hash: 7a1a7e9a7cedfc42ebaff5d69c4afa8bf57dc671496fd86824b219b2c0e5d2b8
Block #5 created.
Hash: 90a58958a67f0d9681b60a8a50e6ed61b0ab51d72d635adf372643cb925f7924
Block #6 created.
Hash: a5c454e16887021f50dff9c02aa13b74d3d33a2016bf886774e59790096dafff
Block #7 created.
Hash: 4928894e0e0578c4191520c0104b3fd8dd341a4d3b418d450087e76fc99afb2f
Block #8 created.
Hash: 7c07d699b10fa6d69e2288683a3d14acf2fa9050a00e83b228fc315985c418ce
Block #9 created.
Hash: 05188f4b17f7affc0d329c372afe43180974e58c2a7bca40982f979b1d710d1b
>>>
```

A single file

```python
import datetime
import hashlib

class Block:
    def __init__(self, previous_block_hash, data, timestamp):
        self.previous_block_hash = previous_block_hash
        self.data = data
        self.timestamp = timestamp
        self.hash = self.get_hash()

    @staticmethod
    def create_genesis_block():
        return Block("0", "0", datetime.datetime.now())

    def get_hash(self):
        header_bin = (str(self.previous_block_hash) +
                      str(self.data) +
                      str(self.timestamp))

        inner_hash = hashlib.sha256(header_bin.encode()).hexdigest().encode()
        outer_hash = hashlib.sha256(inner_hash).hexdigest()
        return outer_hash

num_blocks_to_add = 10

block_chain = [Block.create_genesis_block()]

print("The genesis block has been created.")
print("Hash: %s" % block_chain[0].hash)

for i in range(1, num_blocks_to_add):
    block_chain.append(Block(block_chain[i-1].hash,
                             "Block number %d" % i,
                             datetime.datetime.now()))
    print("Block #%d created." % i)
    print("Hash: %s" % block_chain[-1].hash)
```

```
The genesis block has been created.
Hash: 4391be3b5972249da184e04f56ee07f9b0c2054d3c45c0035b5855c490bcaae6
Block #1 created.
Hash: 5a7c2cb24ab9e84b8370e801dfbe5326ee1112d4dc57ea05ef9df5017240da3d
Block #2 created.
Hash: 4b96f37f105c42872f0cd9c1757fe759c562660acf401302bdeb6ca1583d4ab8
Block #3 created.
Hash: 22f62a333059f50f761cd4530617ed7993df7ee1c72d51a307896755fe42336b
```