The layout of a Solidity source code file

In order to address compatibility issues that may arise from future versions of the `solc` version, `pragma` can be used to specify the version of the compatible compiler as in the following example:

```
pragma solidity ^0.5.0
```

This will ensure that the source file does not compile with versions lower than `0.5.0` and versions starting from `0.6.0`.

Import

Import in Solidity allows the importing of symbols from the existing Solidity files into the current global scope. This is similar to `import` statements available in JavaScript, as in the following:, for example:

```
import "module-name";
```

Comments

Comments can be added to the Solidity source code file in a manner similar to the C language. Multiple-line comments are enclosed in `/*` and `*/`, whereas single-line comments start with `//`.

```
1  pragma solidity ^0.5.0; //specify the solidity compiler version
2  /*
3  this is a simple value checker contract that checks the value provided
4  and returns boolean value (true or false) based on the condition expression
5  evaluation
6  */
7  import "./mapping.sol"; //import a file
8  contract valuechecker {
9      uint price = 10;
10     //price variable declared and initialized with a value of 10
11     event valueEvent(bool returnValue);
12     function Matcher (uint8 x) public returns (bool) {
13         if (x >= price )
14         {
15             emit valueEvent(true);
16             return true;
17         }
18     }
19 }
```

The Solidity language

Solidity is a domain-specific language of choice for programming contracts in Ethereum. There are other languages that can be used, such as Serpent, Mutan, and LLL, but Solidity is the most popular. Its syntax is closer to both JavaScript and C.

Variables

Just like any programming language, variables in Solidity are the named memory locations that hold values in a program. There are three types of variables in Solidity: **local variables**, **global variables**, and **state variables**.

1. Local variables

These variables have a scope limited to only within the function they are declared in. In other words, their values are present only during the execution of the function in which they are declared.

2. Global variables

These variables are available globally as they exist in the global namespace. They are used for performing various functions such as ABI encoding, cryptographic functions, and querying blockchain and transaction information.

Solidity provides a number of global variables that are always available in the global namespace.

This function is used to compute the Keccak-256 hash of the argument provided to the function:

```
keccak256(...) returns (bytes32)
```

This function returns the associated address of the public key from the elliptic curve signature:

```
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)
```

This returns the current block number:

```
block.number
```

This returns the gas price of the transaction:

```
tx.gasprice (uint)
```

3. State variables

State variables have their values permanently stored in smart contract storage. State variables are declared outside the body of a function, and they remain available throughout the contract depending on the accessibility assigned to them and as long as the contract persists:

```
pragma solidity >=0.5.0;
contract Addition {
    uint x; // State variable
}
```
Here, x is a state variable whose value will be stored in contract storage.

There are three types of state variables, based on their visibility scope

- **Private**: These variables are only accessible internally from within the contract that they are originally defined in. They are also not accessible from any derived contract from the original contract.
- **Public**: These variables are part of the contract interface. In simple words, anyone is able to get the value of these variables. They are accessible within the contract internally by using the `this` keyword. They can also be called from other contracts and transactions. A `getter` function is automatically created for all public variables.
- **Internal**: These variables are only accessible internally within the contract that they are defined in. In contrast to private state variables, they are also accessible from any derived contract from the original (parent) contract

Data types

Solidity has two categories of data types: **value types** and **reference types**.

Value types are variables that are always passed by a value. This means that value types hold their value or data directly, allowing a variable's value held in memory to be directly accessible by accessing the variable.

Reference types store the address of the memory location where the value is stored. This is in contrast with value types, which store the actual value of a variable directly with the variable itself. When using reference types, it is essential to explicitly specify the storage area where the type is stored, for example, *memory*, *storage*, or *calldata*.

Value types

Value types mainly include **Booleans**, **integers**, **addresses**, and **literals**, which are explained in detail here.

Boolean

This data type has two possible values, `true` or `false`, for example:

```
bool v = true;
```
or

```
bool v = false;
```
This statement assigns the value `true` or `false` to `v` depending on the assignment.

Integers

This data type represents integers. The following table shows various keywords used to declare integer data types

- Logs: This is the area where the output of the events emitting from the smart contracts is stored.

The specific location used for storing values of a variable depends on the data type of the variable and where the variable has been declared. For example, function parameter variables are stored in *memory*, whereas state variables are stored in *storage.*

Keyword	Types	Details
`int`	Signed integer	`int8` to `int256`, which means that keywords are available from `int8` up to `int256` in increments of 8, for example, `int8`, `int16`, and `int24`.
`uint`	Unsigned integer	`uint8`, `uint16`, ... to `uint256`, unsigned integer from 8 bits to 256 bits. Usage is dependent on how many bits are required to be stored in the variable.

For example, in this code, note that `uint` is an alias for `uint256`:

```
uint256 x;
uint y;
uint256 z;
```

These types can also be declared with the `constant` keyword, which means that no storage slot will be reserved by the compiler for these variables. In this case, each occurrence will be replaced with the actual value:

```
uint constant z=10+10;
```

Address

This data type holds a 160-bit long (20 byte) value. This type has several members that can be used to interact with and query the contracts. These members are described here:

- **Balance**: The `balance` member returns the balance of the address in Wei.
- **Send**: This member is used to send an amount of ether to an address (Ethereum's 160-bit address) and returns `true` or `false` depending on the result of the transaction, for example, the following:

```
address to = 0x6414cc08d148dce9ebf5a2d0b7c220ed2d3203da;
address from = this;
if (to.balance < 10 && from.balance > 50) to.send(20);
```

- **Call functions**: The `call`, `callcode`, and `delegatecall` functions are provided in order to interact with functions that do not have an ABI. These functions should be used with caution as they are not safe to use due to the impact on type safety and security of the contracts.
- **Array value types (fixed-size and dynamically sized byte arrays)**: Solidity has fixed-size and dynamically sized byte arrays. Fixed-size keywords range from `bytes1` to `bytes32`, whereas dynamically sized keywords include `bytes` and `string`. The `bytes` keyword is used for raw byte data, and `string` is used for strings encoded in UTF-8. As these arrays are returned by the value, calling them will incur a gas cost. `length` is a member of array value types and returns the length of the byte array.
    - An example of a static (fixed size) array is as follows:
      ```
      bytes32[10] bankAccounts;
      ```
    - An example of a dynamically sized array is as follows:
      ```
      bytes32[] trades;
      ```
    - Get the length of trades by using the following code:
      ```
      trades.length;
      ```

Literals

These are used to represent a fixed value. There are different types of literals that are described as follows:

Integer literals: These are a sequence of decimal numbers in the range of 0–9. An example is shown as follows:

```
uint8 x = 2;
```

String literals: This type specifies a set of characters written with double or single quotes. An example is shown as follows:

```
'packt' "packt"
```

Hexadecimal literals: These are prefixed with the keyword `hex` and specified within double or single quotation marks. An example is shown as follows:

```
(hex'AABBCC');
```

Enums: This allows the creation of user-defined types. An example is shown as follows:

```
enum Order {Filled, Placed, Expired };
Order private ord;
ord=Order.Filled;
```

Reference types

```
pragma solidity ^0.4.0;
contract TestStruct {
  struct Trade
  {
    uint tradeid;
    uint quantity;
    uint price;
    string trader;
  }
  //This struct can be initialized and used as below
  Trade tStruct = Trade({tradeid:123, quantity:1, price:1, trader:"equinox"});
}
```

In the preceding code, we declared a `struct` named `Trade` that has four fields. `tradeid`, `quantity`, and `price` are of the `uint` type, whereas `trader` is of the `string` type. Once the `struct` is declared, we can initialize and use it. We initialize it by using `Trade tStruct` and assigning `123` to `tradeid`, `1` to `quantity`, and `"equinox"` to `trader`.

Data location

For example, in the preceding structs example, if we want to use only memory (temporarily) we can do that by using the `memory` keyword when using the structure and assigning values to fields in the `struct`, as shown here:

```
Trade memory tStruct;
tStruct.tradeid = 123;
```

Mappings

Mappings are used for a key-to-value mapping. This is a way to associate a value with a key. All values in this map are already initialized with all zeroes, as in the following for example:

```
mapping (address => uint) offers;
```

This example shows that `offers` is declared as a mapping. Another example makes this clearer:

```
mapping (string => uint) bids;
bids["packt"] = 10;
```

This is basically a dictionary or a hash table, where string values are mapped to integer values. The mapping named `bids` has the string `packt` mapped to value `10`.

Control structures

The control structures available in the Solidity language are `if...else`, `do`, `while`, `for`, `break`, `continue`, and `return`. They work exactly the same as other languages, such as the C language or JavaScript.

Some examples are shown here:

- `if`: If `x` is equal to `0`, then assign value `0` to `y`, else assign `1` to `z`:

```
if (x == 0)
    y = 0;
else
    z = 1;
```

- `do`: Increment `x` while `z` is greater than `1`:

```
do{
    x++;
} (while z>1);
```

- `while`: Increment `z` while `x` is greater than `0`:

```
while(x > 0){
    z++;
}
```

- `for`, `break`, and `continue`: Perform some work until `x` is less than or equal to `10`. This `for` loop will run `10` times; if `z` is `5`, then break the `for` loop:

```
for(uint8 x=0; x<=10; x++)
{
  //perform some work
    z++
    if(z == 5) break;
}
```

  It will continue the work in a similar vein, but when the condition is met, the loop will start again.

- `return`: `return` is used to stop the execution of a function and returns an optional value. For example:

```
return 0;
```

  It will stop the execution and return a value of `0`.

Events

Events in Solidity can be used to log certain events in EVM logs. These are quite useful when external interfaces are required to be notified of any change or event in the contract. These logs are stored on the blockchain in transaction logs. Logs cannot be accessed from the contracts but are used as a mechanism to notify change of state or the occurrence of an event (meeting a condition) in the contract.

In a simple example here, the `valueEvent` event will return `true` if the `x` parameter passed to the function `Matcher` is equal to or greater than `10`:

```
pragma solidity >=0.6.0;
contract valueChecker
{
    uint8 price=10;
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) public returns (bool)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            return true;
        }
    }
}
```

Inheritance

Inheritance is supported in Solidity. The `is` keyword is used to derive a contract from another contract. In the following example, `valueChecker2` is derived from the `valueChecker` contract. The derived contract has access to all non-private members of the parent contract:

```
pragma solidity >=0.6.0;
contract valueChecker
{
    uint8 price = 20;
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) public returns (bool)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            return true;
        }
    }
}
contract valueChecker2 is valueChecker
{
    function Matcher2() public view returns (uint)
    {
        return price+10;
    }

}
```

In the preceding example, if the `uint8 price = 20` is changed to `uint8 private price = 20`, then it will not be accessible by the `valueChecker2` contract. This is because now the member is declared as `private`, and thus it is not allowed to be accessed by any other contract.

Libraries

Libraries are deployed only once at a specific address and their code is called via the `CALLCODE` or `DELEGATECALL` opcode of the EVM. The key idea behind libraries is code reusability. They are similar to contracts and act as base contracts to the calling contracts. A library can be declared as shown in the following example:

```
library Addition
{
    function Add(uint x,uint y) returns (uint z)
    {
        return x + y;
    }
}
```

This library can then be called in the contract, as shown here. First, it needs to be imported and then it can be used anywhere in the code. A simple example is shown as follows:

```
import "Addition.sol"
function Addtwovalues() returns(uint)
{
    return Addition.Add(100,100);
}
```

There are a few limitations with libraries; for example, they cannot have state variables and cannot inherit or be inherited. Moreover, they cannot receive ether either; this is in contrast to contracts, which can receive ether

Functions

Functions are pieces of code within a smart contract. For example, look at the following code block:

```
pragma solidity >5.0.0;
contract Test1
{
    uint x=2;
    function addition1() public view returns (uint y)
    {
        y=x+2;
    }
}
```

code example, with `contract Test1`, we have defined a function called `addition1()`, which returns an unsigned integer after adding `2` to the value supplied via the variable `x`, initialized just before the function.

In this case, `2` is supplied via variable `x`, and the function will return `4` by adding `2` to the value of `x`. It is a simple function, but demonstrates how functions work and what their different elements are.

There are two function types: *internal* and *external* functions.

- **Internal functions** can be used only within the context of the current contract.
- **External functions** can be called via external function calls.

  A **function** in Solidity can be marked as a constant. Constant functions cannot change anything in the contract; they only return values when they are invoked and do not cost any gas. This is the practical implementation of the concept of *call* as discussed in the previous chapter.

The syntax to declare a function is shown as follows:

```
function <nameofthefunction> (<parameter types> <name of the variable>)
{internal|external} [state mutability modifier] [returns (<return types> <name of the variable>)]
```

For example:

```
function addition1() public view returns (uint y)
```

Here, `function` is the keyword used to declare the function, `addition` is the name of the function, `public` is the visibility modifier, `view` is the state mutability modifier, `returns` is the keyword to specify what is returned from the function, and `uint` is the return type with the variable name `y`.

Functions in Solidity are modules of code that are associated with a contract. Functions are declared with a name, optional parameters, access modifiers, state mutability modifiers, and an optional return type. This is shown in the following example

```
function orderMatcher (uint x)
private view returns (bool return value)
```

In the preceding code, `function` is the keyword used to declare the function. `orderMatcher` is the function name, `uint x` is an optional parameter, `private` is the **access modifier** or **specifier** that controls access to the function from external contracts, `view` is an optional keyword used to specify that this function does not change anything in the contract but is used only to retrieve values from the contract, and `returns (bool return value)` is the optional return type of the function. After this introduction to functions in Solidity, let's consider some of their key elements, parameters, and modifiers:

- **How to define a function**: The syntax of defining a function is shown as follows:

```
  function <name of the function>(<parameters>) <visibility specifier> <state mutability
modifier> returns
  (<return data type> <name of the variable>)
  {
      <function body>
  }
```

- **Function signature**: Functions in Solidity are identified by their **signature**, which is the first four bytes of the Keccak-256 hash of its full signature string. This is also visible in

the Remix IDE, as shown in the following screenshot. `f9d55e21` is the first four bytes of the 32-byte Keccak-256 hash of the function named `Matcher`

this example function, `Matcher` has the signature hash of `d99c89cb`. This information is useful in order to build interfaces.

- **Input parameters of a function**: Input parameters of a function are declared in the form of `<data type> <parameter name>`. This example clarifies the concept where `uint x` and `uint y` are input parameters of the `checkValues` function:

```
contract myContract
{
    function checkValues(uint x, uint y)
    {
    }
}
```

Output parameters of a function: Output parameters of a function are declared in the form of `<data type> <parameter name>`. This example shows a simple function returning a `uint` value:

```
contract myContract
{
    function getValue() returns (uint z)
    {
        z=x+y;
    }
}
```

A function can return multiple values as well as take multiple inputs. In the preceding example function, `getValue` only returns one value, but a function can return up to 14 values of different data types. The names of the unused return parameters can optionally be omitted. An example of such a function could be:

```
pragma solidity ^0.5.0;
contract Test1
{

    function addition1(uint x, uint y) public pure returns (uint z, uint a)
    {
        z= x+y ;
        a=x+y;
        return (z,a);
```

```
    }
}
```

Here when the code runs, it will take two parameters as input `x` and `y`, add both, and then assign them to `z` and `a`, and finally return `z` and `a`. For example, if we provide 1 and 1 for `x` and `y`, respectively, then when the variables `z` and `a` are returned by the function, both will contain 2 as the result.

- **Internal function calls**: Functions within the context of the current contract can be called internally in a direct manner. These calls are made to call the functions that exist within the same contract. These calls result in simple `JUMP` calls at the EVM bytecode level.

- **External function calls**: External function calls are made via message calls from a contract to another contract. In this case, all function parameters are copied to the memory. If a call to an internal function is made using the `this` keyword, it is also considered an external call. The `this` variable is a pointer that refers to the current contract. It is explicitly convertible to an address and all members of a contract are inherited from the address.

- **Fallback functions**: This is an unnamed function in a contract with no arguments and return data. This function executes every time ether is received. It is required to be implemented within a contract if the contract is intended to receive ether; otherwise, an exception will be thrown and ether will be returned. This function also executes if no other function signatures match in the contract. If the contract is expected to receive ether, then the fallback function should be declared with the payable **modifier**.

  The payable is required; otherwise, this function will not be able to receive any ether. This function can be called using the `address.call()` method as, for example, in the following:

```
function ()
{
    throw;
}
```

  In this case, if the fallback function is called according to the conditions described earlier; it will call `throw`, which will roll back the state to what it was before making the call. It can also be some other construct than `throw`; for example, it can log an event that can be used as an alert to feed back the outcome of the call to the calling application.

- **Modifier functions**: These functions are used to change the behavior of a function and can be called before other functions. Usually, they are used to check some conditions or verification before executing the function. `_` (underscore) is used in the modifier functions that will be replaced with the actual body of the function when the modifier is called. Basically, it symbolizes the function that needs to be *guarded*. This concept is similar to guard functions in other languages.

- **Constructor function**: This is an optional function that has the same name as the contract and is executed once a contract is created. Constructor functions cannot be called later on by users, and there is only one constructor allowed in a contract. This implies that no overloading functionality is available.
- **Function visibility specifiers** (access modifiers/access levels): Functions can be defined with four access specifiers as follows:
    - **External**: These functions are accessible from other contracts and transactions. They cannot be called internally unless the `this` keyword is used.
    - **Public**: By default, functions are public. They can be called either internally or by using messages.
    - **Internal**: Internal functions are visible to other derived contracts from the parent contract.
    - **Private**: Private functions are only visible to the same contract they are declared in.
- **Function modifiers**:
    - **pure**: This modifier prohibits access or modification to state.
    - **view**: This modifier disables any modification to state.
    - **payable**: This modifier allows payment of ether with a call.
    - **constant**: This modifier disallows access or modification to state. This is available before version 0.5.0 of Solidity.

Error handling

Solidity provides various functions for error handling. By default, in Solidity, whenever an error occurs, the state does not change and reverts back to the original state.

Some constructs and convenience functions that are available for error handling in Solidity are introduced as follows:

- **Assert**: This is used to check for conditions and throw an exception if the condition is not met. Assert is intended to be used for internal errors and invariant checking. When called, this method results in an invalid opcode and any changes in the state are reverted back.
- **Require**: Similar to *assert*, this is used for checking conditions and throws an exception if the condition is not met. The difference is that *require* is used for validating inputs, return values, or calls to external contracts. The method also results in reverting back to

the original state. It can also take an optional parameter to provide a custom error message.

- **Revert**: This method aborts the execution and reverts the current call. It can also optionally take a parameter to return a custom error message to the caller.
- **Try/Catch**: This construct is used to handle a failure in an external call.
- **Throw**: Throw is used to stop execution. As a result, all state changes are reverted. In this case, no gas is returned to the transaction originator because all the remaining gas is consumed.

Structure of a smart contract

- A smart contract is a self-executing program that resides on the blockchain and performs certain functions when certain conditions are met. Solidity is a popular programming language used for writing smart contracts on the Ethereum blockchain.
- Here's an example of a simple Solidity smart contract that stores and retrieves a single string

  // SPDX-License-Identifier:

  pragma solidity ^0.8.0;

  contract MyContract {

  string public myString;

  constructor(string memory initialString) {

  myString = initialString;

  } function setString(string memory newString) public {

  myString = newString;

  }

  function getString() public view returns (string memory) {

  return myString;

  }

  }

- SPDX-License-Identifier: This is an SPDX license identifier that specifies the license under which the contract is released. This is optional, but it is good practice to include it.
- Pragma statement: This specifies the minimum version of the Solidity compiler that the contract is compatible with. In this case, we are using Solidity version 0.8.0.

- Contract definition: This is the contract keyword followed by the name of the contract. In this example, the contract is named MyContract.
- State variables: These are variables that define the state of the contract. They can be accessed and modified by the functions in the contract. In this example, we have one state variable called myString which is of type string. It is marked as public so that it can be accessed from outside the contract.
- Constructor: This is a special function that is called when the contract is deployed. It is used to initialize the state variables of the contract. In this example, the constructor takes a string parameter and initializes the myString variable to that value.
- Public functions: These are functions that can be called from outside the contract. They can read and modify the state variables of the contract. In this example, we have two public functions: setString and getString.
- setString is a function that takes a string parameter and sets the value of myString to that value.
- getString is a function that returns the current value of myString.
- Memory keyword: In Solidity, string parameters need to be marked with the memory keyword, which indicates that the string data is stored in memory rather than on the storage of the contract.

Case Study

Voting

Auction