2.3.4.8.4.6 Pure Methods

Return to Operation Data

Overview

Return to Top

Pure methods are limited to are often referred to as **Pure Functions**. **Pure Functions** are a cornerstone in **Functional Programming** and are designed to produce no Side Effects. **Pure Functions** are characterized as follows[1)]:

- Are dependent only on
  - Declared input parameters
  - Algorithm to be implemented
  - Values within the scope of the function, therefore, it can not
    - Depend on accessing any values defined outside the function scope (i.e., another field in the same class, or global variables)
    - Modify mutable values outside the function scope (i.e., other fields in the same class, or global variables)
    - Use external input or output (I/O). It can't rely on input from files, databases, web services, UIs, etc; it can't produce output, such as writing to a file, database, or web service, writing to a screen, etc.
2. Do not modify input parameters

   Figure 1 provides a graphic representing a **Pure Function**. Basically, the **Pure FUnction** is an isolated piece of logic that given the same input always produces the same output. Its isolation means it has no unintended side effects outside of itself and only the inputs determine the processing. Another way to think of a **Pure Function** is at their center there is a Deterministic Algorithm (Also see Black Box Testing).

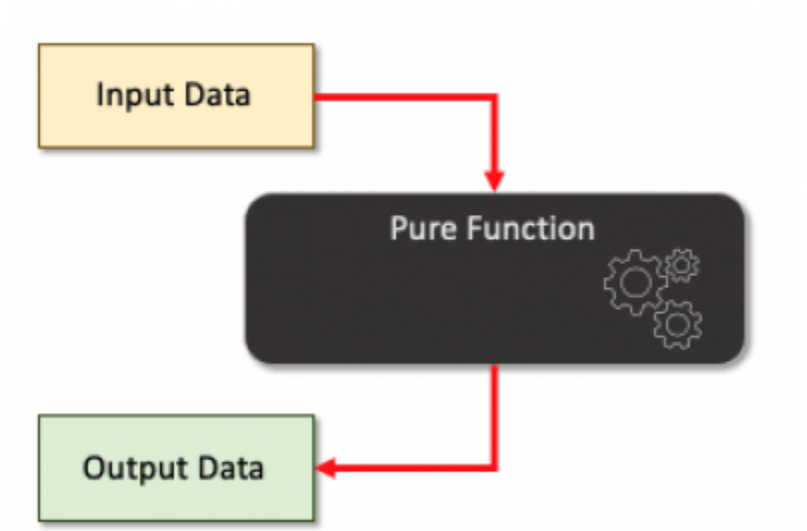

Figure 1: Pure functions

In some languages (i.e., C, C++, Rust, PHP, JavaScript/ECMAScript), it is possible to have methods (i.e., procedures or functions) existing outside the **`class`** container. Java and C# require operations to exist within a **`class`** container, and therefore do not support **General Methods**. C++ does not recommend having **General Methods** outside of a **`class`**, but because C++ is more or less an extension of C, it does support them.

Often, the architecture and design of Functional Programs depends on the identification, design and creation of pluggable, reusable functions. Many of the frameworks used in modern applications reaching across many tiers rely heavily on stateless, client-server Representational State Transfer (REST) models and Command Line Interfaces (CLIs) .

Figure 2 graphically represents **pure** functions used in a **Functional Program**.

- The **pure** functions used in the **Functional Program** are identified (See items **A**, **B**, and **C**)
- The **`functions`** are from a **reuse** repository (i.e., library), or they can be created especially for the new **Functional Program**
- The **Functional Program** is responsible for the lifecycle of each data element (i.e., **State Variables**)
- The order of the **`functions`** is established in the **Functional Program**
- The **association** is made of the **`functions`** with appropriate the **State Variables** (which are **`Input Data`** and which are **`Output Data`**)
- The **Functional Program** is executed:
    1. The **Functional Program** is started
    2. Calls are made to the **`functions`** in the desired order (i.e., steps 2-4) and the **State Variable** values are passed into or out of the **`functions`**
    3. The **Functional Program** is terminated

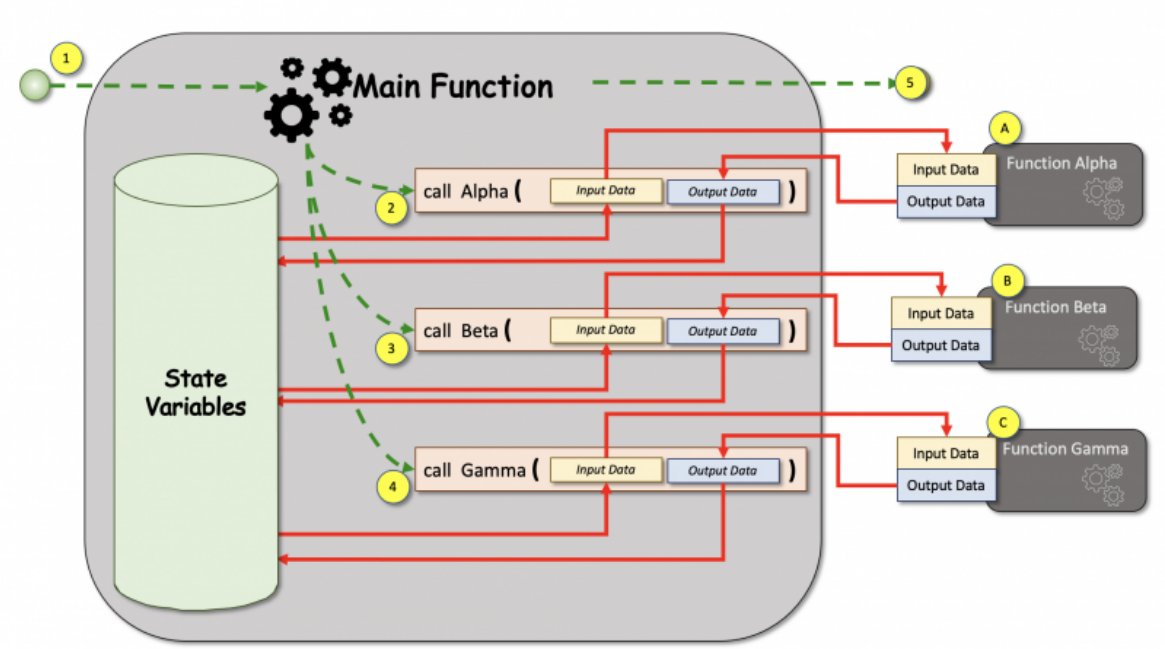

Figure 2: The use of **pure** Functions in a **Functional Program**

DIDO Specifics

Return to Top

Ethereum's Solidity is an Object-Oriented Programming (OOP) supporting four closely related object container types (Java and C++ have just one **`class`**):

contract	**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. Calling a function on a different contract (instance) will perform an EVM function call and thus switch the context such that state variables in the calling contract are inaccessible. 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.

interface

Interfaces
are similar to **abstract contracts**[2], but they cannot have any **`functions`** implemented. There are further restrictions:

- They cannot inherit from other **`contracts`**, but they can inherit from other **`interfaces`**.
- All declared **`functions`** must be **`external`**.
- They cannot declare a constructor.
- They cannot declare **State Variables**.
- They cannot declare **`function modifiers`** modifiers.

Some of these restrictions might be lifted in the future.
Interfaces are basically limited to what the Contract ABI can represent, and the conversion between the ABI and an interface should be possible without any information loss.

- **Note:** The difference between an **Interface** and an **Abstract Contract** is the interface has no implementations for the function. And **abstract contract** can have some **`functions`** with implementations and some without implementations.

The following is an examle **Interface** called EIP 20: ERC-20 Token Standard[3].

```
pragma solidity ^0.8.7;
// SPDX-License-Identifier: MIT

interface IERC20 {
    function totalSupply()
      external
      view
      returns (uint256);
    function balanceOf
      ( address account
      )
      external
      view
      returns (uint256);
    function transfer
      ( address recipient,
        uint256 amount
      )
      external
      returns (bool);
    function allowance
      ( address owner,
        address spender
      )
      external
      view
      returns (uint256);
    function approve
      ( address spender,
        uint256 amount
      )
      external
      returns (bool);
    function transferFrom
      ( address sender,
        address recipient,
        uint256 amount
      )
      external
      returns (bool);
    event Transfer
      ( address indexed from,
        address indexed to,
        uint256 value
      );
    event Approval
      ( address indexed owner,
        address indexed spender,
        uint256 value
     );
} // End  IERC20 interface
```

library

Libraries, in Solidity, are similar to contracts, but they are deployed only once at a specific address and their code is reused using the **`delegatecall`** operation[4] is a low level function similar to **`call`**.
When **`contract A`** executes **`delegatecall`** to contract **B**, **B's** code is excuted with contract **A's** storage, **`msg.sender`** and **`msg.value`**. Solidity By Example, Versoin 0.8.10, Delegatecall, Accessed: 2 January 2022)) feature of the Ethereum Virtual Machine (EVM).
Therefore, when a library **`functions`** are called, the code is executed in the context of the calling **`contract`**, i.e. this points to the calling contract, and especially the storage from the calling contract can be accessed. A **`library`** is an isolated source code and only has access **State Variables** defined by the calling **`contract`** when the variables are explicitly supplied. Library functions are only called directly (i.e. without the use of **`delegatecall`**) when the **`function`** does not modify the state (i.e. in other words, the functions are **`view`** or **`pure functions`**).
To **`contracts`** that use **Libraries**, the **`library`** behaves as base **`contracts`**. The **libraries** are not explicitly visible within the inheritance hierarchy of the derived contract, but calls made to the library **`functions`** appear calls to **`functions`** of explicit **base contracts** and require the use of qualified names such as **`Library.function()`**. Naturally, **`calls`** to internal **`functions`** follow the usual internal calling convention allowing all internal types to be stored stored in memory raather than passed by reference (i.e., not copied).
An example:

```
pragma solidity ^0.7.1;
pragma abicoder v2;
// SPDX-License-Identifier: MIT

library Geolocation
{ struct Coordinate
  { uint Latitude;
    uint Longitude;
  } // End Location structure
  struct DistanceMeasurement
  { uint Distance;
    string UnitsOfMeasurement;
  } // End Distance Struct
  function distanceBetweenCoordinates
    ( Coordinate  memory _originalLocation,
      Coordinate memory _nextLocation
    )
    public
    pure
    returns ( DistanceMeasurement memory )
   { // actually do the calculation here
   } // End distanceBetweenCoordinates function
} // End geolocation library
```

struct

Struct*ures*, in Solidity, is not unlike a simple **`struct`** in C, C++ or C#. Sometimes, **`structs`** are thought of as a **`contract`** that does not allow the inclusion of **`functions`** in the definition. In other words, it is a datatype that is a collection of other datatypes referenced by a single name.
•
Structs are defined as stand-alone entities in Solidity, or they can be encapsulated within a **`library`** or a **`contract`**. **Structs** declared outside a **`contract`** facilitates reuse.
•
Structs are generally contiguous memory and can be used in collections such as arrays.
An example:

```
pragma solidity ^0.7.1;
// SPDX-License-Identifier: MIT

struct coordinate
{ uint Latitude;
  uint Longitude;
} // End Location structure

contract MyContract
{ // An array of locations tracking movements
  coordinate[] public movementTrace;
} // End MyContract Contract
```

does have support defining and using General Methods, those **`functions`** not directly pertaining to the **Smart Contract**, but can be used by the smart contracts but it does support being able to create libraries of reusable **`functions`** that support a way to provide operators for specific types.

Pure Functions

Return to Top

Ethereum's Solidity has a special label to identify **`functions`** that qualify for the **`pure`** classification meeting the definition of **Pure Function** given earlier. Not surprisingly, they are identified with **`pure`** label. The purity rules are enforced by the Solidity compiler. The **`pure function`** in solidity can be thought of as a **black box** and only relies on the data pased into it for processing. It can use local variables defined within the **`pure function`**. The compiler throws an error if the **`pure function`** tries to[5]:

- read or update **State Variables**
- access the contract's **`address`**
- acccess the contract's **`balance`**
- access any **global variable block**
- access the **`msg`**
- call a **`function`** that is not **`pure`**

In the below example, the contract Test defines a pure function to calculate the product and sum of two numbers.

```
pragma solidity ^0.7.1;
// SPDX-License-Identifier: MIT

contract TestContract
{
   function getResult
     ( uint _leftSide,
       unit _rightSide
     )
     public
     pure
     returns
       ( uint product,
         uint sum
       )
   {  product = _leftSide * _rightSide;
      sum = _leftSide + _rightSide;
   } // End getResult function
} // End TestContract contract
```

Libraries

Return to Top

As of Solidity 8.1, it is possible to defines a **`library`**. A **`library`** is a kind of **`contract`**, that has no Ethereum Storage associated with it and in addition, it cannot hold **`ether`**. One way to think bout a solidity **`library`** is as a Singleton in the Ethereum Virtual Machine (EVM). In other words, it is a piece of code callable from any contract without the need to redeploy it. [6)]

Libraries in Solidity **`contracts`** are blocks of reusable code containing **`functions`** usable by other **`contracts`** on the blockchain network. When used correctly, **`libraries`** support modular, Object-Oriented Programming (OOP) designs.

The main advantage of using **`library`** is code reusability across multiple **`contracts`** preventing duplication of code and the reuse of testing snd documentation of the code. In addition, **`libraries`** save on **`gas`** by not deploying the code multiple times on the blockchain.

Libraries are a special form of **`contracts`** with the following restrictions:

- Are singletons
- Not allowed any **storage** or **state variables** that change
- Cannot have **fallback** functions
- Have no **event logs**
- Do not hold **`ether`**
- Are stateless
- Cannot use **`destroy`**
- Cannot inherit or be inherited

Libraries allow for the addition of functionality to the basic types (i.e., **`uint`**) or complex user defined types (i.e., **`struct`**). **Libraries** are isolated from other blocks of code (i.e., **`contracts`**) that have no rely on the **storage** (i.e., **state variables**) from the calling **`contract`** and supplied to the **`functions`**.[7)]

Libraries support different Data Types:

- User defined **`strut`**
- User derinde **`enum`**
- User defined immutable variables (i.e., **`constant`**)

> **Note:** Library constants become part of the Bytecode rather than in storage (i.e., not as variables on the blockchain itself)

Example of Defining a Library

Return to Top

The following code provides examples for:

- Creates a **`library`** called **`StudentRecord`** (Line 4)

- Creates a user defined type (i.e., **`struct`**) defining a **`StudentRecord`** concept adding the following fields: (Lines 5-9)
    - **`name`** (Line 6)
    - **`studentNumber`** (Line 7)
    - **`totalClassPoints`** (Line 8)
3. Defines a **`function`** named **`addPoints`** that accepts two parameters (Lines 11-17):
    - An instance of a **`StudentRecord`** in **`storage`** (Line 12)
    - The **`earnedPoints`** to add to the students record (Line 13)
    - The actual calculation of the students **`totalClassPoints`** (Line 16)
    - The end of the definition of **`addPoints function`**
4. End of the definition of the **`library`** (Line 17)

```
pragma solidity ^0.8.1;
// SPDX-License-Identifier: MIT

library StudentLibrary
{ struct StudentRecord
    { string name;
      uint studentNumber;
      uint totalClassPoints;
    } // End StudentRecord structure
  function addPoints
    ( StudentRecord storage _studentRecord,
      uint _earnedPoints
    )
    public
    { _studentRecord.totalClassPoints += _earnedPoints;
    } // End addPoints function
} // End StudentRecord library

contract MyClass
{ // Uses the newly created StudentLibrary
  mapping ( uint => StudentLibrary.StudentRecord )
studentRoster;
  function addQuizResults() external
  { // Add points for each student from latest quiz
    StudentLibrary.addPoints ( studentRoster[0], 10 );
    StudentLibrary.addPoints ( studentRoster[1],  5 );
    StudentLibrary.addPoints ( studentRoster[2],  8 );
  } // End addQuizResults function
} // End MyClass contract
```

Example of Importing and Using a Library

Return to Top

In the example, the library code is saved iin the same file as **`contract MyClass`**. It could be stored in a separte file and then imported iinto the **`contract MyClass`** file. If the **`StudentLibrary`** file is kept in its own file in the same directory as the **`contract MyClass`** file **`StudentLibrary.sol`**.

In the folowing exaple, both the **`import`** and the **`using`** are used:

- The entirity of the **`library StudentLibrary`** is replaced by an **`import`** statement (Line 4)
- The **Using** statements extends the instances of **`StudentRecord`** with the operations iin the **`StudentLibrary`** (Line 7)
- The code is modified to use the cleaner, more easily read **`StudentLibrary`** defined **`function`** (Lines 11-13)

This form of defining and using a **`library`** facilitates the reuse of the library by multiple **Smart Contracts**, helps with the maintenance by only having the code defined once, and helps with creating Object-Oriented (OO) architectures and designs.

```
pragma solidity ^0.8.1;
// SPDX-License-Identifier: MIT

import StudentLibrary from "./StudentLibrary.sol";

contract MyClass
{ using StudentLibrary for
StudentLibrary.StudentRecord;
  mapping ( uint => StudentLibrary.StudentRecord )
studentRoster;
  function addQuizResults() external
  { // Add points for each student from latest quiz
    studentRoster[0].addPoints ( 10);
    studentRoster[1].addPoints (  5);
    studentRoster[2].addPoints (  8);
  } // End addQuizResults function
} // End MyClass contract
```

☐ [char]Review

1)

Alvin Alexander, AlvinAlexander.com, The Definition of “Pure Function”, Accessed: 30 December 2021, https://alvinalexander.com/scala/fp-book/definition-of-pure-function/

2)

Abstract contracts are contracts that have at least one function without

its implementation. An instance of an abstract cannot be created. Abstract contracts are used as base contracts so that the child contract can inherit and utilize its functions. GeeksForGeeks, Solidity – Abstract Contract, 13 July 200, Accessed: 2 January 2022, https://www.geeksforgeeks.org/solidity-abstract-contract/
3)

Crypto Market Pool, Blockchain Engineer Resource, Interface in Solidity smart contracts, Accessed: 2 January 2022, https://cryptomarketpool.com/interface-in-solidity-smart-contracts/
4)

delegatecall
5)

GeeksForGeeks, Solidity – View and Pure Functions, 13 July 2020, Accessed: 2 January 2022, https://www.geeksforgeeks.org/solidity-view-and-pure-functions/
6)

Jorge Izquierdo, Aragon Association, Library Driven Development in Solidity, 13 February 2017, Accessed: 28 December 2021, https://aragon.org/blog/library-driven-development-in-solidity-2bebcaf88736
7)

Crypto Market Pool, Blockcahin Engineer Resource, Libraries in Solidity smart contracts, Accessed 28 December 2021, https://cryptomarketpool.com/libraries-in-solidity/