2.3.4.8.4.2 Constructor / Destructor Methods

Return to Operational Data

Constructor

Return to Top

Overview

Return to Top

A **Constructor** is a special group of Operators that are called when an **Object** is first created. In most Object-Oriented Programming (OOP) languages (e.g., C++, Java and C#), the **Constructor** has the same name as the **Object**. It is the initial stage in the Lifecycle of Object Data (see 2.3.4.5 Data Lifecycle Taxonomy). For example, for an **Object** called **`Vehicle`**, the **Constructor** would also be called **`Vehicle`**. The constructor is responsible for the setup of the **Object**, the initialization of **Field Data**, and the allocation of memory from the **Heap**. For example, if there are dynamic Field Data, then the memory needed for those Fields is usually allocated from the **Heap**. Although **Constants** are generally managed and allocated by the compiler and are from the **Stack**, **Static Field Data** can be allocated from the **Heap** and can be set during construction. For example, the values of **Field Data** initialized by using initialization parameters on the **Constructor** or read from initialization or setup files.

Generally, there are three kinds of constructors available in OO:

- **Default Constructor** is a constructor which does not take any arguments (i.e.parameters) when it is called.
- **Parametrized Constructor** is a constructor which is generally called after the **Default Constructor** and has arguments (i.e., parameters) that it can use during the construction process.
- **Copy Constructor** is a constructor that is called to create a replica of the first object. The designer of the **Copy Constructor** needs to understand the object to determine what needs to be part of a Deep Copy and what can remain as part of a Shallow Copy.

There is always a default **Constructor** that required no parameters, however, there can be other **Constructors** allowing for the passing of values to be used during initialization. For example, the minimum or maximum values used for the Field Data.

Regardless of the number of **Constructors**, there is always a **Constructor** that is called when an object is created. Often the calling of a **Constructor** is automatic and used the default **Constructor**, but the programmer can use any of the **Constructors** defined for the **Object**. See: https://www.tutorialspoint.com/solidity/solidity_constructors.htm

DIDO Specifics

Return to Top

When Data Object is deployed in Ethereum, the following occurs:

The contract is initialized using the optional **Constructor** method named: **`constructor()`**. A **Constructor** is a special function declared using the **`constructor`** keyword. It is an ***optional*** function and is used to initialize state variables of a contract. Following are the key characteristics of a constructor[1)]

- A **contract** can have only one **constructor**
- A **constructor** code is executed once when a contract is created, and it is used to initialize the contract state. **Note:** When the **constructor** does not explicitly set any state variables, they are set to zero
- After the **constructor** code is executed, the final code is deployed to the blockchain. This code include **public** functions and code reachable through **public** functions. **Note: Constructor** code or any internal method used only by **constructor** is not included in the final code
- A **constructor** is either **public** or **internal**
- A **internal constructor** marks the contract as **abstract**
- In case, no **constructor** is defined, a default constructor is present in the contract which initializes all values to zero

Simple Constructor

Example of a simple Constructor

```
pragma Solidity ^0.6.0;

contract Inventory
{ uint public quantityInStock;
  constructor () public
  { quantityInStock = 0;
  } // End Inventory constructor
  function checkInventory() external view
    { if ( quantityInStock < 0 )
      { revert ( "quantityInStock must be greater than 0");
      } // End if
    } // End checkInventory
} // end Inventroy contract
```

Constructor with Arguments

Example of a Constructor with arguments that initialize the state variables

```
pragma Solidity ^0.6.0;
```

```
contract Inventory
{ uint public quantityInStock;
  constructor ( uint _initialQuantity ) public
  { quantityInStock = _initialQuantity;
  } // End Inventory constructor
  function checkInventory() external view
    { if ( quantityInStock < 0 )
      { revert ( "quantityInStock must be greater than 0");
      } // End if
    } // End checkInventory
} // end Inventroy contract
```

Destructor

Return to Top

Overview

Return to Top

Destructor is a special method called automatically during the destruction of an object. Actions executed in the destructor include the following:

- Recovering Heap space allocated during the lifetime of an object (see 2.3.4.5 Data Lifecycle Taxonomy)
- Releasing Shared or Network Resources such as printers, faxes, scanners or outside Voice over Internet Protocol (VOIP) resources
- Releasing Resource Lock, for example, Web Servers, banking systems read versus update, travel reservations, etc
- Closing connections such as database, file or service
- Other housekeeping tasks

- **Note:** Often the first use of the **Destructor**, Recovering **Heap** memory, is automated in many modern systems (i.e., Java, .NET, Python, JavaScript, etc.) but the remaining reasons for a destructor remain in place, and it is up to the architect to determine what resources need to be freed upon the end of a Data Object (see 2.3.4.5 Data Lifecycle Taxonomy).

DIDO Specifics

Return to Top

Although the original intent of a DIDO is built around the concept of the immutability of the data, why is there a need for destruction of the data (see 2.3.4.5 Data Lifecycle Taxonomy). For this discussion, Ethereum's Solidity as a rubric.

> **Note:** Even when a contract is removed using the **`selfdestruct`**, it remains as part of the history of the DIDO and is probably retained by most nodes. Therefore, using Solidity **`selfdestruct`** is not equivalent to deleting it from a computer's hard drive or even from the cloud.

> **Note:** Even when a contract's code does not explicity contain a call to **`selfdestruct`**, it can still perform the functionality by using **`delegatecall`** or **`callcode`**.

Since the software (i.e., Smart Contracts) are also stored on the DIDO and are self-executing, they too cannot be modified after they are deployed, not even by the creator of the contract. This is particularly true in Ethereum, which is a permission-less network of nodes meaning the software on the network (i.e., smart contracts) are executed by everyone who can access the network, which includes nefarious actors (i.e., attackers). In addition, the entire contents of the network including constants, state variables, transactions, and the smart contract byte code are completely visible to anyone having access to the DIDO making it an ideal target for "bad actors".

> *In 2016, attackers utilized a vulnerability (reentrancy [40]) to attack a smart contract owned by an organization named DAO (Decentralized Autonomous Organization). This attack made the organization lose 3.6 million Ethers1. People usually call this attack a DAO attack [1]. Actually, the attack continued for several days, and the organization even noticed that their contract had been attacked at that time. However, they could not stop the attack or transfer the Ethers because of the immutability feature of smart contracts.*[2)]

The 2016 attack known as the **reentrancy attack** or **DAO attack** drew the attention of both academia and industry as various schemes were introduced to prevent such attacks in the future. Part of the solution is to specify requirements, and develop and test Smart Contracts rigorously before they are deployed. Although this is always best, it is not always possible to predict all the possible ways a Smart Contract is vulnerable, especially in the future. Therefore, another part of the solution is to add some mechanisms to stop the contracts and/or transfer the tokens when emergency situations arise (e.g., a contract is under attack). The only option left for the owners of the Smart Contract is to reduce the impact of financial loss. In response, Ethereum's Solidity provides a **`Selfdestruct`** function which allows the Smart Contract to transfer all remaining tokens to a different Smart Contract and to remove the errant Smart Contract from the Ethereum network.

When a Data Object is destroyed in Ethereum[3)], the following occurs:

- The remaining tokens are transferred to a new address
- The contract is destructed via the **`selfdestruct()`** method or the deprecated **`suicide()`** method

> **Note:** See https://www.tutorialspoint.com/solidity/solidity_constructors.htm

> **Note:** It is recommended that the deactivation of contracts should use a disabling mechanism, such as changing an internal state variable that causes most functions to stop working, making it impossible to use the contract.
> https://docs.soliditylang.org/en/v0.8.10/introduction-to-smart-contracts.html?highlight=destruct#d

eactivate-and-self-destruct

Example of Self Destruction

1. A state variable containing the address of the **owner** of the **contract** (Line 4)
2. A state variable indicating if the **contract** is active or **paused** (i.e., paused ⇒ false means it is working) (Line 5)
3. The contract is constructed is executed once (Line 7-9)
    1. The **owner** is set to the message sender of the that created the **contract** (Line 8)
4. A function **setPaused** allowing the **owner** to pause or resume the **contract** (Lines 16-21)
    1. If **setPaused** is not the owner, an error occurs (Line 19)
    2. State variable **paused** is set to the argument **_paused** (Line 20)
5. A function **withdrawAllMoney** allows the owner to withdraw all the remaining balance to a new payable address (Line 23-29)
    1. Only the owner of the contract can trnsfer the balance, otherwise an error (Line 26)
    2. Only transfer the money if the contract is not paused (Line 27)
    3. Transfer the balance to the address specified (Line 28)
6. A function **destroySmartContract** allowing the owner of the contract to selfdestrct the contract (Line 31-36)
    1. Specify the **_to** address to transfer the remaining balance to (Line 32)
    2. Verify the **owner** made the request tp destroy the contract (Line 34)
    3. Transfer the remaining balance to the **_to** address (Line 35)

Example of using the **selfdestruct** operation

```
pragma Solidity ^0.6.0;

contract StartStopUpdateExample
{ address public owner;
  bool public paused;

  constructor()
  { owner = msg.sender;
  } // End constructor function

  function sendMoney()
  public payable
  {
  } // End sendMoney function

  function setPaused
    ( bool _paused )
    public
  { require(msg.sender == owner, "You are not the owner");
    paused = _paused;
  } // End sendMoney function
```

```
  function withdrawAllMoney
    ( address payable _to )
    public
  { require(owner == msg.sender, "You cannot withdraw.");
    require(paused == false, "Contract Paused");
    _to.transfer(address(this).balance);
  } // End withdrawAllMoney function

  function destroySmartContract
    ( address payable _to )
    public
  { require(msg.sender == owner, "You are not the owner");
    selfdestruct(_to);
  } // End destroySmartContract function

} // End StartStopUpdateExample contract
```

☐ [char]Review

1)

https://www.tutorialspoint.com/solidity/solidity_constructors.htm

2) , 3)

Jiachi Chen, Xin Xia, David Lo, John Grundy, Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum, May 2020, Accessed: 5 December 2021, https://www.researchgate.net/publication/341478354_Why_Do_Smart_Contracts_Self-Destruct_Investigating_the_Selfdestruct_Function_on_Ethereum