2.3.4.8 Object Data Taxonomy

Return to Data Taxonomy

In programming, an Object represents a real-world object such as a car, an employee or a task. Ultimately, as the real-world Object is defined in terms of software, it is translated to a Data Structure. The Object Data (see Figure 1 is categorized into Characteristic Data, Definition Data, Field Data and Operation Data.

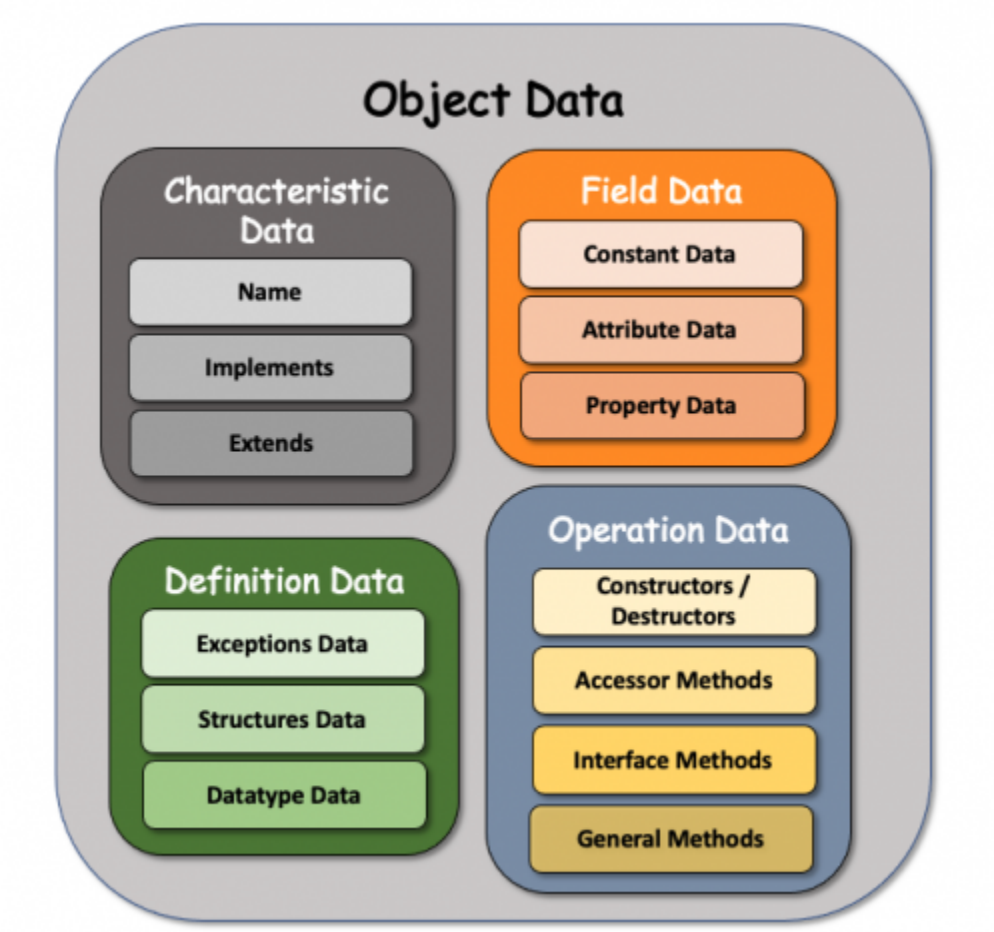

Figure 1: Components of an Object

DIDO Specifics

Return to Top

Many DIDO Platforms refer to Object Data as **Smart Contracts**. These are similar too, but not the same as classes in Object-Oriented Programming (OOP) languages such as C++, Java, C#, Python, etc. In many ways, the differences are lexical in nature (i.e., **`Smart Contract`** versus **`Contract`**, **`method`** versus **`function`**, etc. However, there are some important differences which have to do with the distributed nature of DIDOs and the immutability of the field data (i.e., **view** and **pure** designators on **methods**.

The following discussion has to do with the more generic concept of **Object Data** but does include discussions of how it relates to DIDO Platforms in general and Ethereum's Solidity more specifically.

Characteristic Data

Return to Top

Every Object has characteristics that contextually describe the object. For example, the Object's name, if it implements and interfaces or if it is an extension of another object.

- **Name** is a textual name that uniquely identifies the object with the context of the Namespace. For example, there are multiple objects with the name of **Tank**. The first is in the namespace (i.e., context) of the oil and gas industry and refers to a fuel tank. Another **Tank** object has a military namespace and refers to a military vehicle, and another **Tank** object has a clothing namespace and refers to a “tank top” tee-shirt. All the rest of the data for the object is dependent on context (i.e., namespace).
- **Implements** is a way to describe which Interfaces this object provides an implementation for. For example, the ERC-20 interface used for many Cryptocurrencies. Java defines standardized collection interfaces for things like Lists, Set, Queue, Maps and Iterators[1)]. C++ has similar interfaces to those describe for Java [2)]. The exact way that interfaces are implemented differs from language to language, and can even differ from versions of the same language.
- **Extends** is a way to describe a hierarchical tree for an object (i.e., its Inheritance). For example, a graphics program ma have a base object called a **`GeographicShape`**'. Other objects within the graphics program such as **`Square`**, **`Circle`** and **`Triangle`** are said to **extend** the original definition of **`GeographicShape`**.

DIDO Specifics

Return to Top

Smart Contracts are akin to classes in Object-Oriented Programming (OOP) languages, and they can have Inheritance and Interfaces. There are two very well-known examples of **interfaces** in use within Ethereum:

- EIP 20: ERC-20 Token Standard
- EIP 721: ERC-721 Non-Fungible Token Standard

However, all many of the Ethereum: Ethereum Improvement Proposals (EIPs) define interfaces.

Interfaces are similar to **abstract contracts**, but they are limited to what the contract’s ABI can represent. In other words, you could convert an ABI into an interface, or vice versa, and no information would be lost. According to the Solidity docs they have a few additional restrictions. For example, Doug Crescenzi, 13 June 2018, Accessed 3 Novemebr 2021, https://medium.com/upstate-interactive/solidity-how-to-know-when-to-use-abstract-contracts-vs-interface

s-874cab860c56)) outlines the following restrictions:

- *Interfaces cannot have any functions implemented*
- *Interfaces cannot inherit other contracts or interfaces (contracts can however inherit interfaces just as they would inherit other contracts)*
- *Interfaces cannot define a constructor*
- *Interfaces cannot define variables*
- *Interfaces cannot define structs*
- *Interfaces cannot define enums*

Interfaces are expressed using the interface keyword. Here's an example:

```
pragma solidity ^0.4.24;

interface token
  { function totalSupply() public view returns (uint256);
    function balanceOf(address who) public view returns (uint256);
    function transfer(address to, uint256 value) public returns (bool);
  } // End Token
```

Definition Data

Return to Top

An Object has the ability to **Define Data** that can be accessed throughout the rest of the Object and depending on the public/private attribution of the definitions outside the Object Definition. The kinds of components that can be defined are:

- **Exceptions**, or faults, are abnormal or unprecedented events occurring during or after the execution of an operation which adversely affects the flow of the operation or the flow of the operations that use the object. Although must runtime systems define some standard exceptions such as Array-out-of-bounds, divide-by-zero, or Runtime, each Data object may define its own exceptions based upon the unique characteristics of the operations. For example, a Thermostat object might have a Too-Cold or Too-Hot exception. It is a runtime error of an undesired result or event affecting normal program flow.
- **Structures** or **Objects** are closely related. The easiest way to think of the difference between **Structures** and **Objects** is that structures usually only define the actual data, and object (i.e., Class) defines the data structures as well as operations that can be performed on that data. Object Data is a recursive definition. However in C++, the main difference between a structure and a class (i.e., object) is the assumption of public/private attribution. Structures assume that all components are by default **`public`** while classes (i.e., objects) assume that all components are **`private`**. For example, a Thermostat object might define a TemperatureReading structure that is uses to report results.
- **Data Type** are definitions of type information associated with the object. Data Type information and be numeric that include ranges, enumerations, scalars or collections. For example, the Thermostat might include a type describing the legal range of numeric values, it might define an

enumeration type for Fahrenheit, Celsius or Calvin. It might also define a type for array of temperature readings.

Note: These components are not required by any Object and the names used to describe these components can be different depending on the programming language.

Field Data

Return to Top

Field Data, or **Element Data**, are the values stored within the Object and in many ways represent the reason for the Object (i.e., Class in C++ or Java) to exist. In Procedural Language) **Field Data** can be thought of as a Variable with some variables having their values set at compile time (i.e., constants) and those during the execution of the procedure. **Note:** Methods or Operations in Objects are procedural in nature, generally represented by a block of code that executes from the top to the bottom, but can be circuitous within the block of code having logical expressions, looping and exceptions to control the flow within the block.

Constant Data is a quality of the data to not be modifiable during the execution of the procedure. There are two broad categories of constants: **Named Constants** and **Literal COnstants**. **Note: Constants** can be scoped to the particular procedure, the package of procedures or during the execution of the program.

Often the two terms: **Attribute** and **Property** are used interchangeable in Computer Science because we often use them to describe the fields (i.e., elements) of an Object in an Object-Oriented Programming (OOP). However, it is useful to understand the difference between the two grammatically:

Attribute Data is a quality or object that we attribute to someone or something. For example, intelligence is an attribute of a person. You can not go to the store and buy intelligence, it is an attribute of the individual. Another example would be a planet belonging to a solar system. Planets do not usually exist outside a solar system.

Property Data is a quality that exists without any attribution. Geometric shapes are properties in their own right. They can be used to independently describe many things. For example, a planet or a ball can be described as a sphere[3]. A definition of the sphere exists without the attribution to a planet or a ball. Therefore, we say that a planet or a ball have spherical properties.

In Unified Modeling Language (UML), an **Attribute** and a **Property** both represent an structural association between two entities. **Attributes** are most often represented as **Composition**[4], while **Property** is best represented by **Aggregation** [5].

According to Lenny Delligatti[6], “UML Attributes are Data Types (e.g., Strings, Integers, etc.) whereas Systems Modeling Language (SysML) properties are ValueTypes (e.g., can be assigned computed values).”

There are different ways that **Field Data** can be stored. In most programming paradigms, the **Field Data** is **Mutable**, meaning the values simply replaced or overwritten with new values. The Object Accessor Methods and responsible for making sure the new values are syntactically and semantically correct. Figure 2 graphically illustrates the concept of **`mutable`**. Each time the **`setter`** is called, the contents of the **Field Data** within the **Data Object** is changed. **Note:** There can be **`setters`** for **Attribute** and **Property** data. In the example, the first time the **`setter`** is called, the value of the **attribute** is set to **2**, the next time it is **4**.

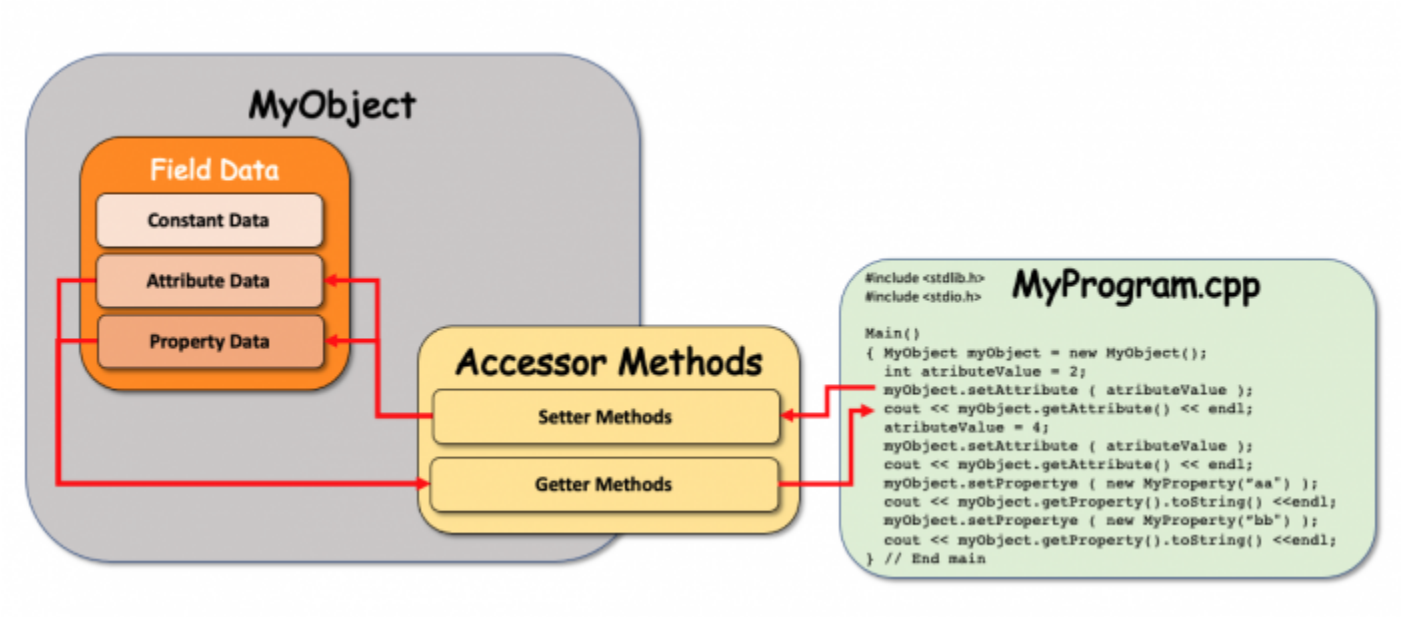

Figure 2: Each time the **`setter`** is called, the value of the **`Filed Data`** is changed.

However, in DIDOs, most of the **Field Data** is considered **immutable**, meaning that updates and modifications to the **Field Data** are made while not destroying the previous data, but a new entry is made for the Field Data that points back to the original value. See Section 2.1.6 Immutable Data Objects and Figure See Figure 3. **Note:** The concepts of **mutable** and **immutable** are different from those built into C/C++, which describes **immutable** more like constants.

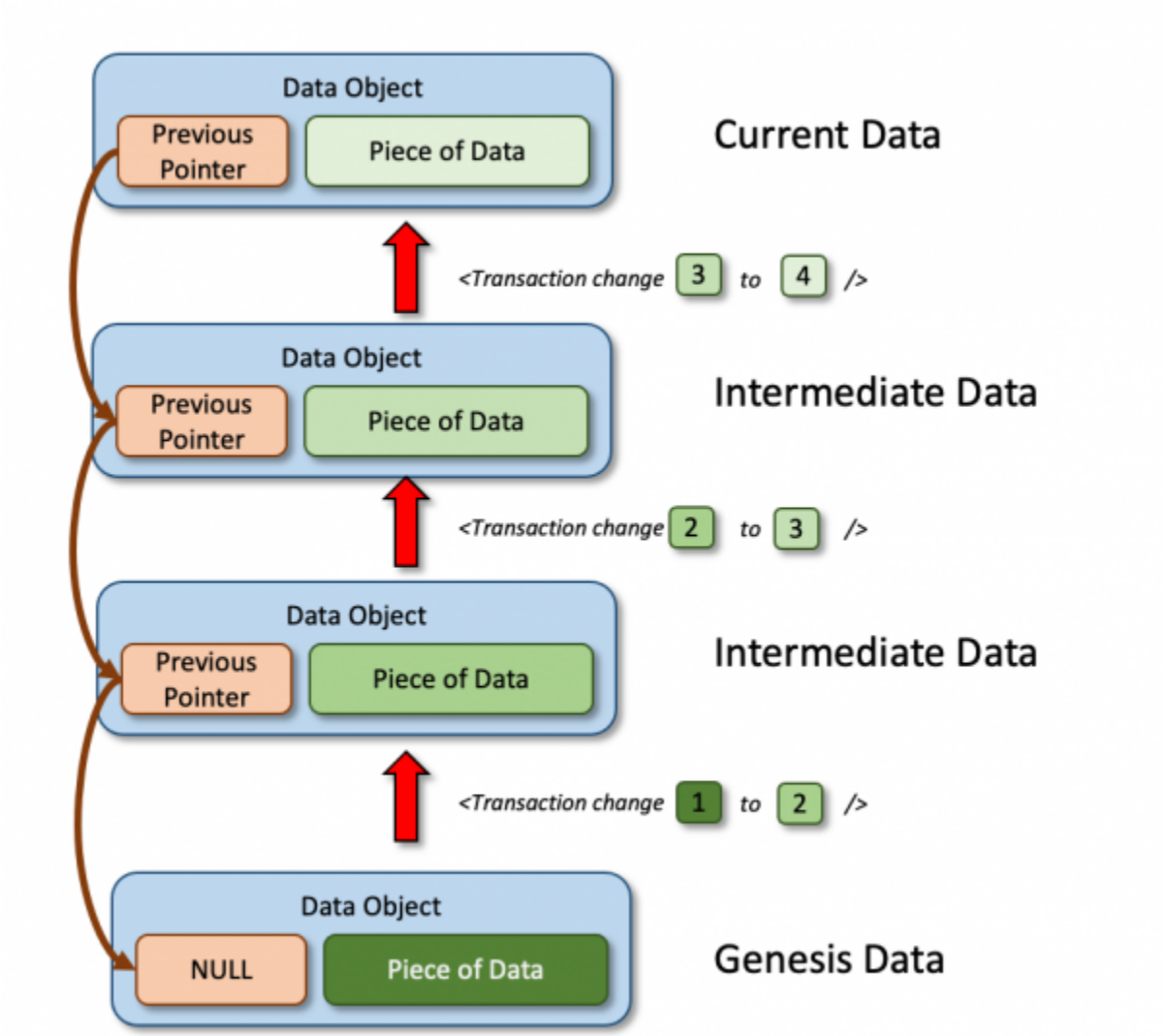

Figure 3: The Immutable Data Chain where the current value of a **Field Data** points to the previous value.

Operation Data

Return to Top

Any particular operation within an Object is generally Imperative in nature. This means that the Operations are composed of a set of statements. As the program steps through the Operation and executes each statement (i.e., instruction) the state of the Object is changed. Some statements change the values of the **Field Data** while others change the flow of execution through Control Flow operations (i.e., **`IF / ELSE IF / ELSE / ENDIF`**, or **`FOR LOOPS | WHILE LOOPS /`**, etc.). As the statements are executed, the **`Program Counter`** for the Operation is also modified. **Note:** Rarely all the statements contained within a single operation, but relies on calling and executing other Operations. Therefore, most Operations are also considered a Procedural Language. Figure 4 provides a graphic of the difference between **Imperative** and **Procedural** Programs.

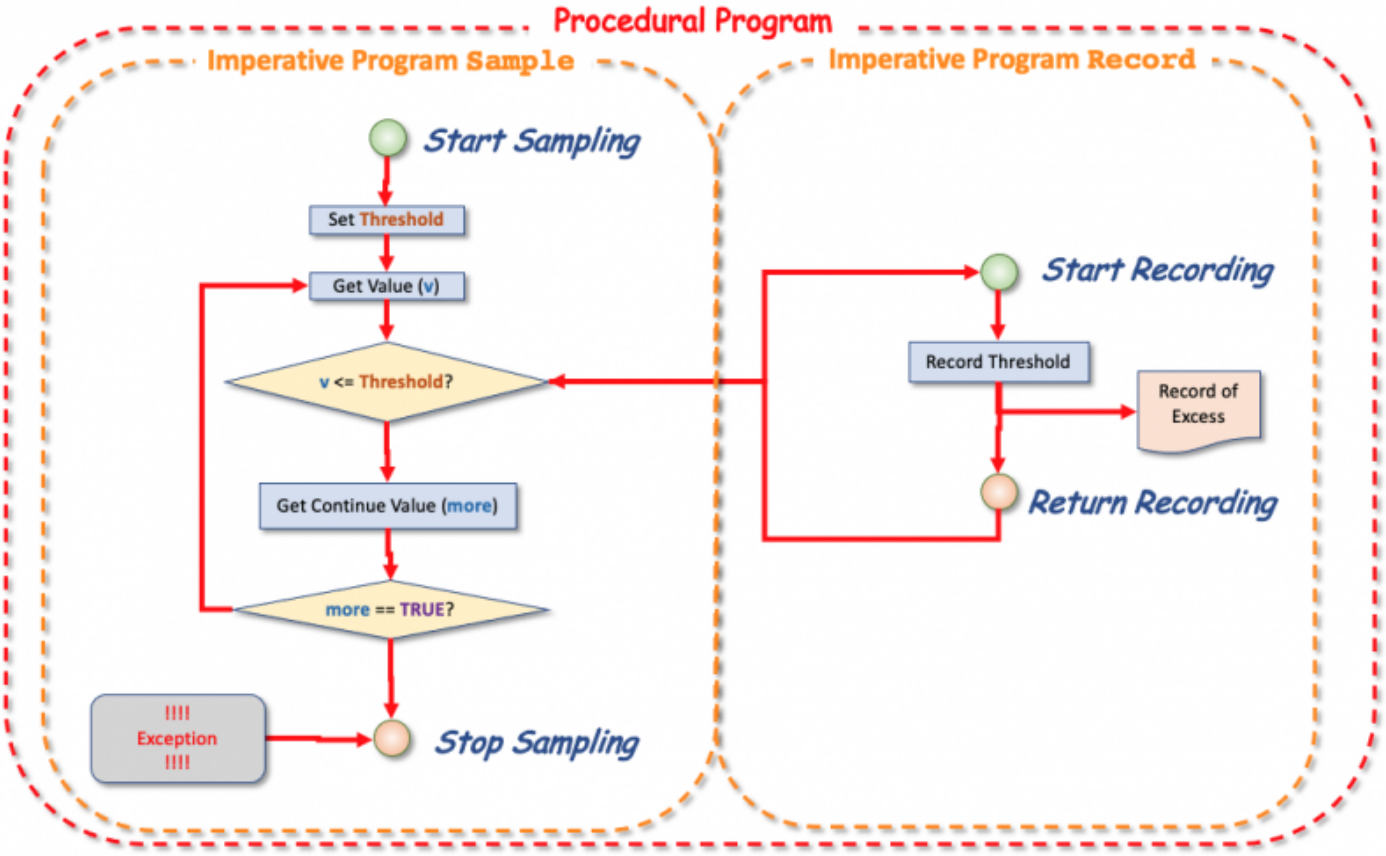

Figure 4: The difference between **Imperative** and **Procedural** Programming.

DIDO Specifics

One of the interesting phenomenons about many of the DIDO Platforms, is the way they treat **Operation Data**. Many of the Platforms not only distribute the **Field Data** using **Distributed Ledger** or **Journal**, they also distribute the **Operation Data** in the same way (in essence, treating the software just as any other data). When changes are made to the **Operation Data**, a **Transaction** is created to be pushed to the **Node Network**.

Constructor / Destructor Methods

Return to Top

Constructor

A **Constructor** is a special group of Operators that are called when an **Object** is first created. In most Object-Oriented Programming (OOP) languages (i.e., C++, Java and C#, e.g.), the **Constructor** has the same name as the **Object**. It is the initial stage in the Lifecycle of Object Data (see 05_lifecycle). 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**, 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.

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

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

The contract is initialized using the **Constructor** method named: **`constructor()`**

> ***Constructor*** *is a special function declared using 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.*

- *A contract can have only one constructor.*
- *A constructor code is executed once when a contract is created, and it is used to initialize contract state.*
- *After a constructor code executed, the final code is deployed to blockchain. This code include public functions and code reachable through public functions. Constructor code or any internal method used only by constructor are not included in final code*
- *A constructor can be 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*

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

Destructor

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 05_lifecycle)
- 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 05_lifecycle).

DIDO Specifics

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

- A contract can be destructed via the **`selfdestruct()`** method or the deprecated **`suicide()`** method
- **Note:** See https://www.tutorialspoint.com/solidity/solidity_constructors.htm

Accessor / Mutator Methods

Return to Top

An **Accessor Method** is an Operation defined by Object-Oriented Programming (OOP) providing access to **Field Data**. There are generally two **Accessor Methods** defined: **`get`** and **`set`**. In some Object-Oriented (OO) paradigms, the **Accessor** methods are further defined as:

- **Accessor Methods** provide Read-Only access to the **Field Data**. **Accessor Methods** provide a way to obtain the state of an object's **Field Data** from outside the **Object**.
- **Mutator Methods** provide a Write-Only access to the **Field Data**. A Mutator not only does simple checking of the data being set such as type and bound checking; it can also check the semantics of the **Field Data** to ensure they are valid and consistent. For example, using a **Mutex Method** to deduct a payment, might also set the date of the payment at the same time even though they are separate **Fields** within the **Object**.

  Note: There may or may not be any MUTual EXclusion (mutex) or Resource Lock associated with the **Field Data**.
  Note: Often both the **Accessor** and the **Mutator** are simply referred to as the **Accessors**. " **Note:** Both the **Accessor** and the **Mutator** are designated as **public**, but they can also be declares as **private** or **protected**. Those that are designated **Private** can only be used from within the **Object**. **Public** designations allow for the methods to be used from inside and outside the **Object**. **Protected** allows the internal **Object Methods** to access the methods, but also allows any **Objects** derived from the **Object** to access the methods.

Some benefits of using a **Mutator Methods** include:

- The prevention of data corruption by directly accessing the private **Field Data** of an **Object**

- The flexibility in modifying the internal representation of the **Data Fields** of an **Object** without breaking the Interface and consequently code that depends on the **Interface**.
- The ability to include additional processing logic such as validation of a value set, triggering of events, etc. during mutation of the **Field Data** or adding Data Logging of acccesses.
- The ability to add MUTual EXclusion (mutex) or Resource Lock for synchronizing multithreading or multiprocessor scenarios.
- The ability to define **Accessor** and **Mutator** methods in derived **Data Objects** from the base **Data Object**.

DIDO Secifics

Return to Top

Ethereum's Solidity **View** can attribute Methods as **View**. methods ensure they do not modify the Data Object state.

> *A method can be declared as view. The following statements if present in the method are considered modifying the state and the compiler throws warning in such cases.*

- *Modifying state variables*
- *Emitting events*
- *Creating other contracts*
- *Using* ***`selfdestruct`***
- *Sending* ***`via calls`***
- *Calling any method which is not marked* ***view*** *or* ***pure***
- *Using low-level calls*
- *Using inline assembly containing certain opcodes*

Getter Method (i.e., **Accessor Methods** are by default **view** methods.

- **Note:** See: https://www.tutorialspoint.com/solidity/solidity_view_functions.htm

Ethereum's Solidity can attribute Methods as **Pure**.

> *Pure methods ensure that they not read or modify the state. A method can be declared as pure. The following statements if present in the method are considered reading the state and compiler will throw warning in such cases.*

- *Reading state variables.*
- *Accessing address(this).balance or <address>.balance.*
- *Accessing any of the special variable of* ***`block`***, ***`tx`***, ***`msg`*** *(****`msg.sig`*** *and* ***`msg.data`*** *can be read).*
- *Calling any method not marked pure.*
- *Using inline assembly that contains certain opcodes.*

> ***Pure*** *methods can use the* ***`revert()`*** *and* ***`require()`*** *functions to revert potential state changes*

if an error occurs.

- **Note:** See: https://www.tutorialspoint.com/solidity/solidity_pure_functions.htm

Abstract/Virtual Methods

Return to Top

An **Abstract Method**, or **Virtual Method**, are methods that are declared but have no underlying implementation (i.e., no body). The primary purpose of these methods is to form a base, template or guide for subsequent Data Objects derived from this **Data Object**. The derived Data Objects all have the same behavior defined by Abstract Methods of the base object. The original base **Data Object** is considered as Abstract or Virtual since without an implementation for these methods, the **Data Object** remains a "concept".

Almost all Object-Oriented Programming (OOP) languages provide for the concept of **Abstract** or **Virtual** methods. In recent years, the concept of **Virtual** or **Abstract Data Objects** have lost favor to the use of **Generics** or **Templates** which can be instantiated for a particular type. For example, a double-linked-lst template, or a name-value pair template.

There are benefits of using Abstract **Data Objects**[7)]:

Template	The abstract **Data Object** enables the best way to execute the process of data abstraction by providing the developers with the option of hiding the code implementation. It also presents the end-user with a template that explains the methods involved.
Loose Coupling	Data abstraction in **Data Object** enables loose coupling, by reducing the dependencies at an exponential level. See 4.3.5.3 System Manageability Issues
Code Reusability	Using an abstract **Data Object** in the code saves time. Abstract Data Objects avoid the process of writing the same code again.
Abstraction	Data abstraction in **Data Objects** helps systems hide code complications and implementation details from **Derived Classes** (i.e., subclasses) aiding developers of **Base Object** and **Derived Object** to focus their attention to appropriate level.

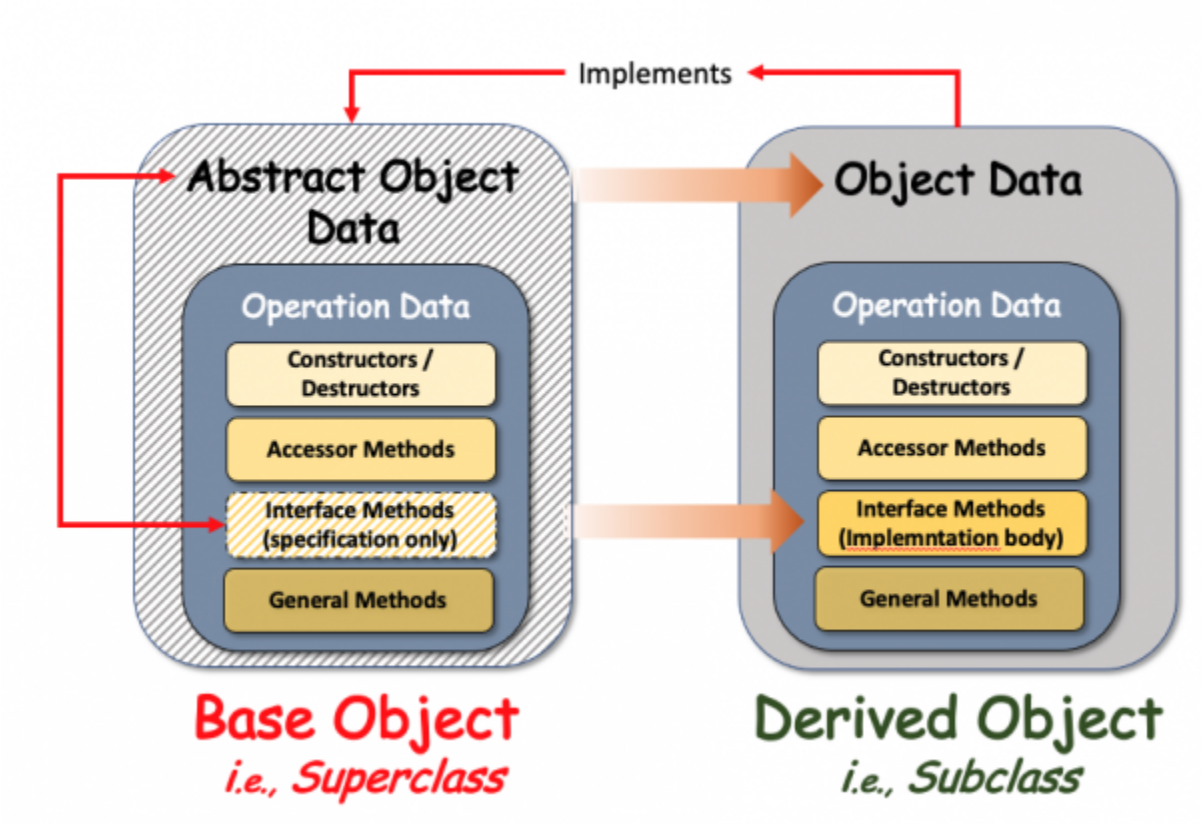

Figure 5: Relationship of Abstract Interfaces with realized Data Objects

DIDO Specifics

Return to Top

Doug Crescenzi [8] does an excellent job in summarizing the use of Abstract smart_contracts and Interfaces.

Ethereum allows for both Interfaces and for Abstract Contracts (i.e., Data Objects). An **Abstract Smart Contract** is any Smart Contract that has at least one method specified that does not have a corresponding body (implementation). This means that the **Abstract Data Object** acts like a **Base Class**, then the **Derived Class** MUST provide an implementation for the abstract method(s).

```
pragma solidity ^0.4.24;

contract Person
{ function gender() public returns (bytes32);
}

contract Employee is Person
{ function gender() public returns (bytes32)
    { return "female"; }
}
```

General Methods

Return to Top

☐ [char]New Section -- review

1)
Programiz.com Java Collection Frameworks, Accessed: 27 October 2021, https://www.programiz.com/java-programming/collections
2)
Geeksforgeeks.com, The C++ Standard Template Library (STL), Accessed: Accessed: 27 October 2021, https://www.geeksforgeeks.org/the-c-standard-template-library-stl/
3)
Sphere, In geometry, the set of all points in three-dimensional space lying the same distance (the radius) from a given point (the center), or the result of rotating a circle about one of its diameters. The components and properties of a sphere are analogous to those of a circle.https://www.britannica.com/science/sphere
4)
The **Composition** is a part of aggregation, and it portrays the whole-part relationship. It depicts dependency between a composite (parent) and its parts (children), which means that if the composite is discarded, so will its parts get deleted. It exists between similar objects. https://www.javatpoint.com/uml-association-vs-aggregation-vs-composition
5)
Aggregation is a subset of association, is a collection of different things. It represents has a relationship.

- It is more specific than an association
- It describes a part-whole or part-of relationship
- It is a binary association, i.e., it only involves two classes
- It is a kind of relationship in which the child is independent of its parent.

https://www.javatpoint.com/uml-association-vs-aggregation-vs-composition
6)
Lenny Delligatti, SysML Distilled: A Brief Guide to the Systems Modeling Language, FIrst Edition, Addison-Wesley Professional , 8 November 2013, ISBN-13: 978-0321927866
7)
Simplilearn, What is an Abstract Class in Java and How to Implement It?, Accessed: 3 November 2021, https://www.simplilearn.com/tutorials/java-tutorial/abstract-class-in-java
8)
_Solidity: How to know when to use Abstract Contracts vs Interfaces__, Doug Crescenzi, 13 June 2018, Accessed 3 Novemebr 2021, https://medium.com/upstate-interactive/solidity-how-to-know-when-to-use-abstract-contracts-vs-interfaces-874cab860c56