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.

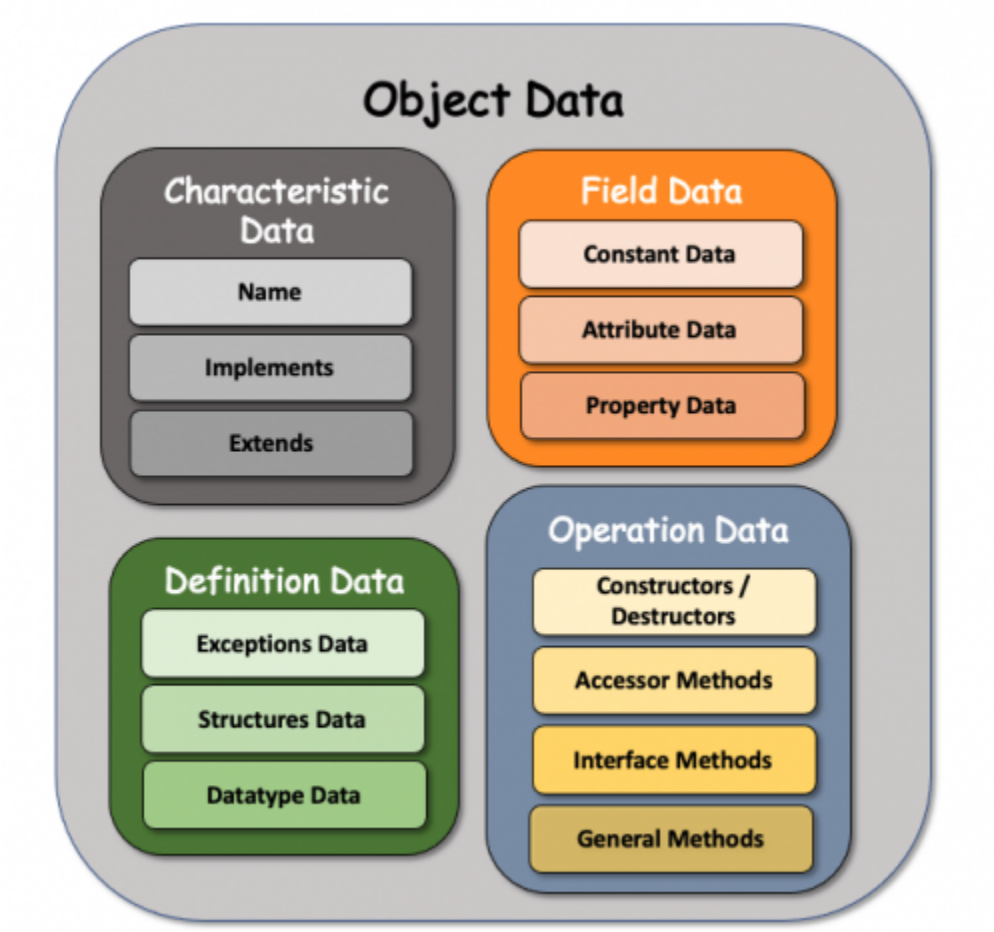

Figure 1: Components of an Object

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.

- 01_char
- 03_def
- 05_field
- 07_opers

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 2 provides a graphic of the difference between **Imperative** and **Procedural** Programs.

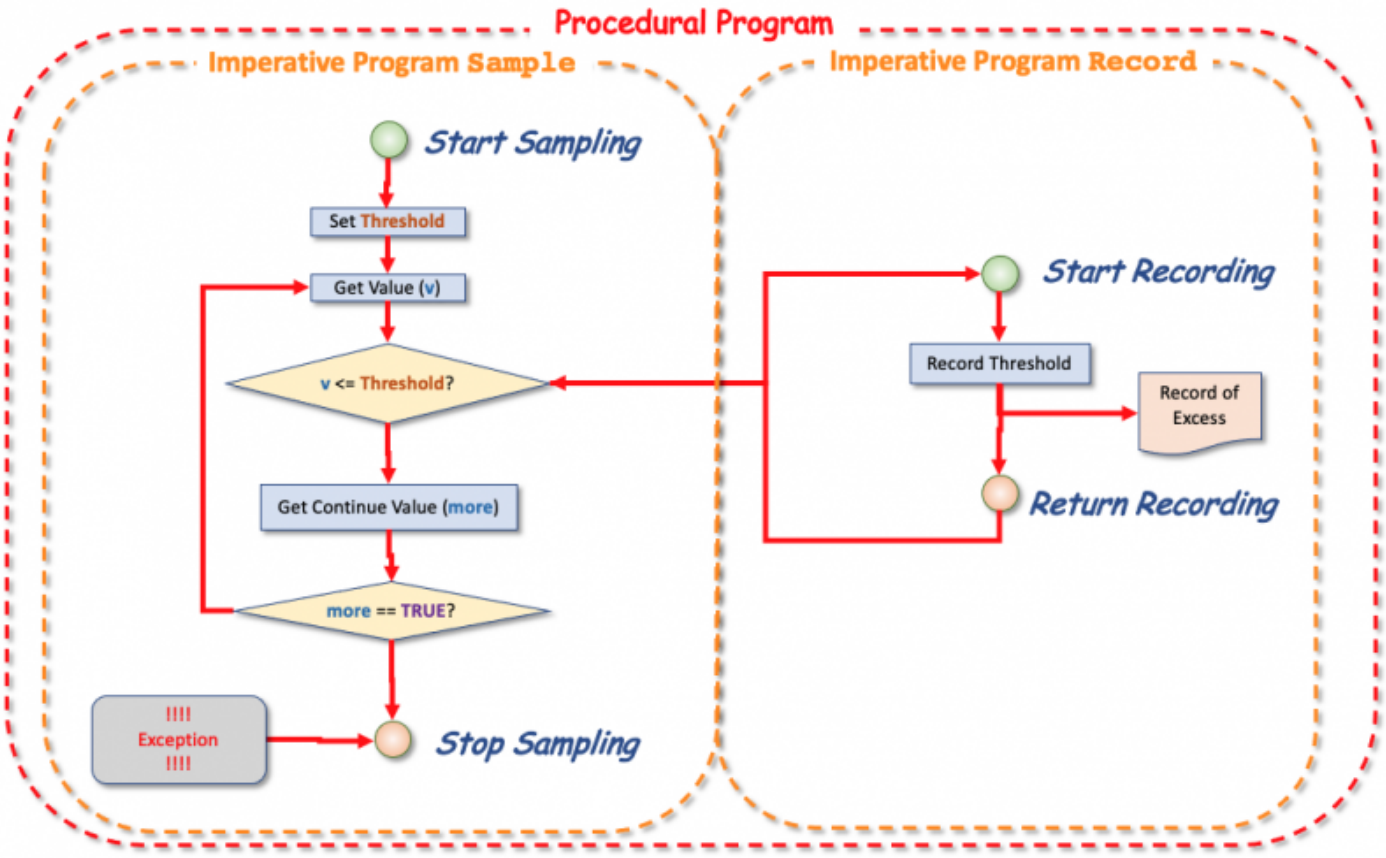

Figure 2: The difference between **Imperative** and **Procedural** Programming.

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**[1)]:

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.

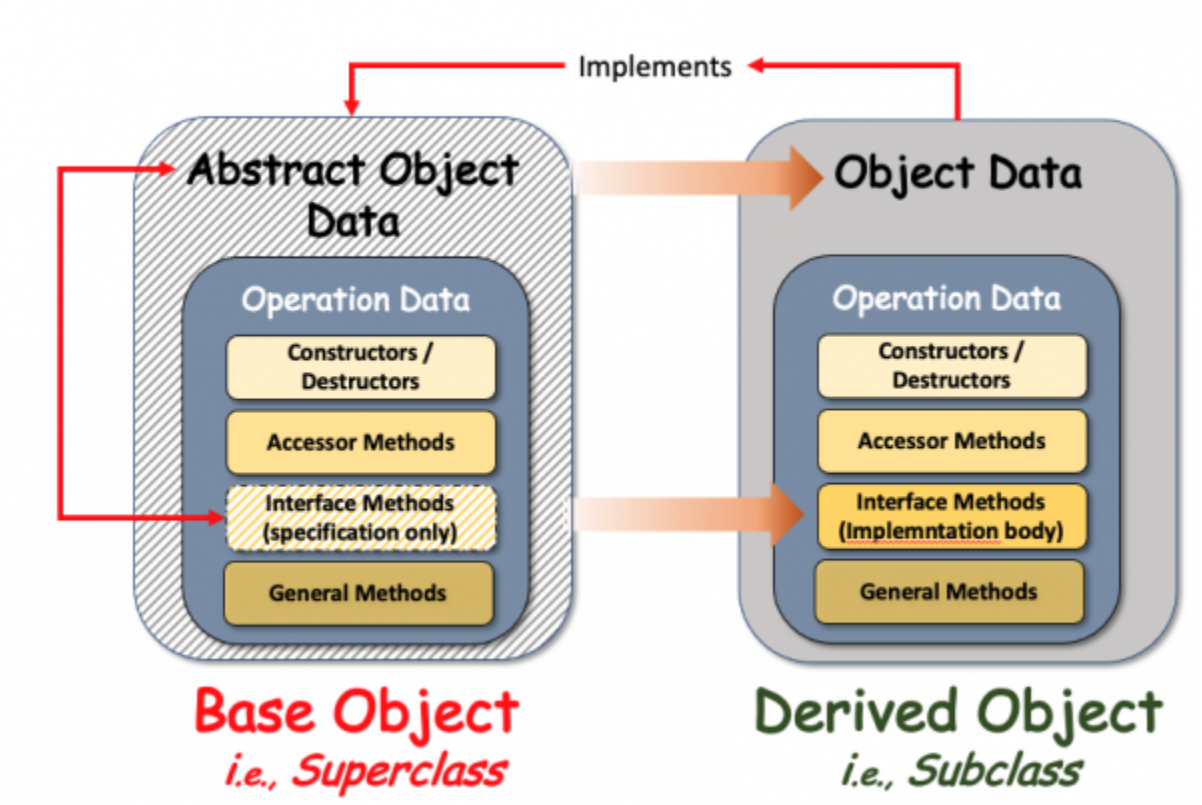

Figure 3: Relationship of Abstract Interfaces with realized Data Objects

DIDO Specifics

Return to Top

Doug Crescenzi [2)] 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)
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
2)
_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