2.3.4.8.4.4 Abstract/Virtual Methods

Return to Operation Data

Overview

Return to Top

An **Abstract Method**, or **Virtual Method**, are methods that are declared but has 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 the Abstract Methods of the base object. The original base **Data Object** is considered as Abstract or Virtual since, without 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 on an appropriate level.

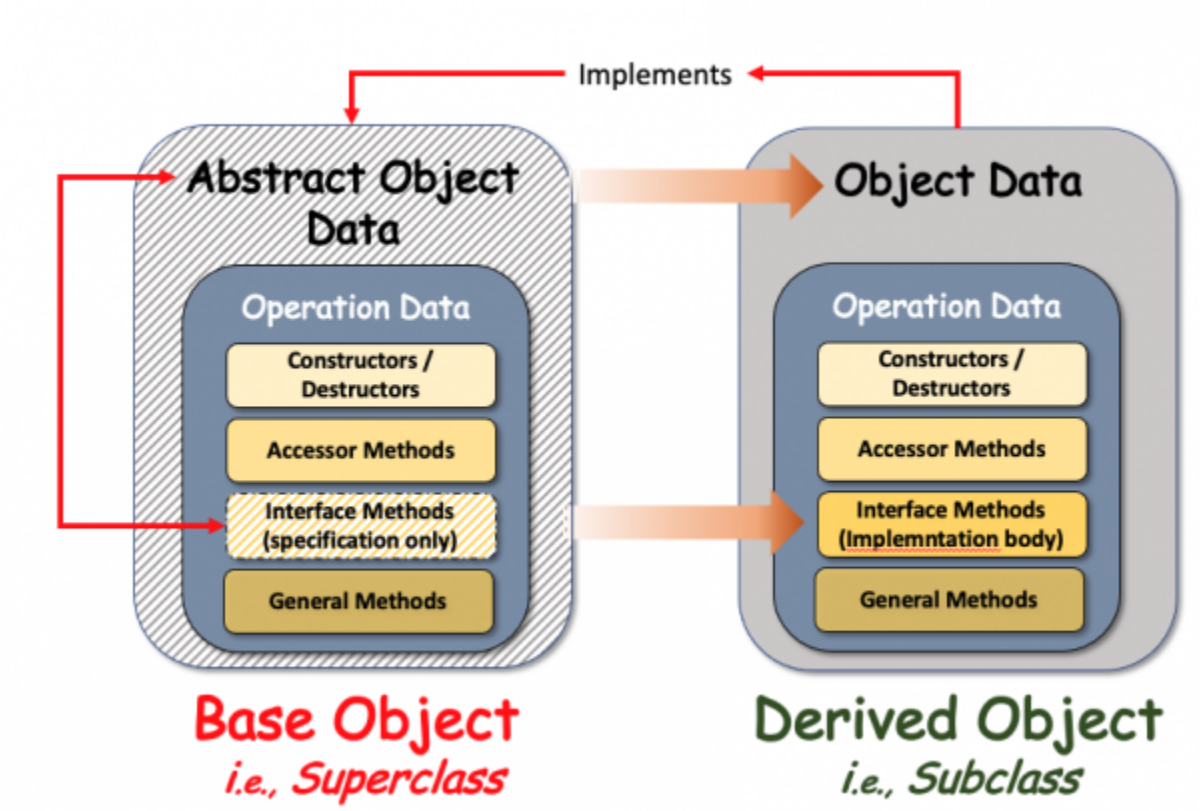

Figure 1: Relationship of Abstract Interfaces with realized Data Objects

DIDO Specifics

Return to Top

Doug Crescenzi [2)] does an excellent job in summarizing the use of Abstract Smart Contract 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"; }
}
```

☐ [char]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 November 2021, https://medium.com/upstate-interactive/solidity-how-to-know-when-to-use-abstract-contracts-vs-interfaces-874cab860c56