2.3.4.8.4.3 Accessor / Mutator Methods

Return to Operation Data

Overview

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 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 declared 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**.

C++, Java, verbose C# Example

The following discussion uses **c++**, however, it is very similar to Java and a "verbose C#" form. C# has some shorthand ways of coding **accessor** and **mutator** operations (see the next section).

In C++, the **accessor** methods are usually implemented as **getter** operations and the **mutator** functions are usually implemented as **setter** operation. The use of the prefix **get** and **set** at the beginning of a function name is merely a programming convention, and there is nothing within the C++ language to enforce the convention, however, following the naming convention makes it easier for others to read and use the code.

Example of using **Getters** and **Setters** in C++

```
#include <iostream>
using namespace std;

class Account
{ // Private attributes
  private:
    int balance;

  // Public Accessor and Mutator operations
  public:
    // setBalance sets the balance in the account
    void setBalance
      ( int _newBalance )
    { balance = _newBalance;
    } // End setBalance
    // getBalance returns the current balance in the account
    int getBalance() {
      return balance;
    } // End getBalance
}; // End Account class

// main used as a unit tester of the Account class
int main()
{ Account account;
  account.setBalance ( 50000 );
  cout << myObj.getBalance();
  return 0;
} // End Main tester
```

Terse C# Example

C# has a shorthand way of specifying the **Accessor** and **Mutator** methods, which reduces the chance of

errors in the code since it is generated.

Example of using **Getters** and **Setters** in terse C#

```
using System;

class Account
{ public int balance { get; private set; }
} // End Account Class
```

DIDO Specifics

Return to Top

Verbose Example

The following is a verbose Ethereum example using Solidity of Smart Contract using **Accessor** and **Mutator** methods (i.e., **Getters** and **Setters**).

> **Note:** In this example, there is no method (i.e., function) attributes that designate them as **Getters** or **Setters** other than a naming convention. However, it is possible to use the **View** or **Pure** function visibility attribution to help the compiler enforce the roles of each function type.

Exaple of using Accessor / Mutator Methods in Solidity

```
pragma solidity ^0.6.0;
// A simple smart contract
contract MessageContract
{
    string private message // private is the default visibility
      = "Hello World";
    function getMessage()
      public
      returns ( string )
    { return message;
    } // End getMessage
    function setMessage
      ( string newMessage
      ) public
    { message = newMessage;
    } // End setMessage

} // End MessageContract
```

Terse Example

As with C#, the Solidity compiler automatically creates **getter** functions for all public state variables. In our contract, by changing the visibility of the state variable **`message`** to public, a function with no parameters (i.e., **`()`**) returning a **`string`** representing the current state of the **`message`** state variable.

> **Note: `Setters`** are not automatically created since the function would be visible to anyone that access the **Smart Contract**.

Exaple of using Accessor / Mutator Methods in Solidity

```
pragma solidity ^0.6.0;
// A simple smart contract
contract MessageContract
{
    string public message = "Hello World";
    function setMessage
      ( string newMessage
      ) public
    { message = newMessage;
    } // End setMessage

} // End MessageContract
```

Controlling State Access

Solidity also allows for attributing functions beyond just **public** visibility with the use of two keywords: **view** and **pure**:

- **`view`** functions only read from the contract state variables and can not modify the contract state. This is usually added to **getters**.

Example of using Accessor Method and the **view** attribute in Solidity

```
pragma solidity ^0.6.0;
// A simple smart contract
contract MessageContract
{
    string public message = "Hello World";
    function getMessage()
      public view
      returns ( string )
```

```
    { return message;
    } // End getMessage
    function setMessage
      ( string newMessage
      ) public
    { message = newMessage;
    } // End setMessage

} // End MessageContract
```

When a function is attributed as **view**, the following things are considered modifying the contract's state and the compiler gives warnings:

- *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*

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

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

- **pure** functions do not read or modify the contract state variables and are only used for computations (i.e., mathematical calculations) or table lookups.

Exaple of using a **pure** attribute in Solidity

```
pragma solidity ^0.6.0;
// A simple smart contract
contract SupportContract
{
    function add
      ( uint firstNumber,
        uint secondNumber
      ) public pure
      returns ( uint )
    { return firstNumber + secondNumber;
    } // End add

} // End SupportContract
```

pure *methods ensure no reads or modifications to the state variables occur. The following statements, if present in the function, is considered reading the state and the compiler throws warnings if they are encountered.*

- *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.*

Note: the **`revert()`** and **`require()`** operations are allowed in a ***pure*** *to ensure no state changes occur when an error occurs. :* ***Note:*** *See: https://www.tutorialspoint.com/solidity/solidity_pure_functions.htm ☐ [char]Review*