**Error Handling**

**Table of Contents**

* **Assert condition**
* **Require condition**
* **Revert condition**
* **Try/catch block**

**Error Handling in Solidity**

Solidity includes a variety of functions for dealing with errors. Errors can arise during the compilation process or during the execution process. Solidity is compiled to byte code, and a syntactic error check is performed at build time. Runtime mistakes, on the other hand, are difficult to detect and arise mostly during the execution of contract statements. Some of the runtime errors include the following: out-of-gas error, data type overflow error, divide by zero error, array-out-of-index error, and so on. Until version 0.4.10, there was only a single throw statement in Solidity to handle errors, which meant that to manage errors, one had to implement numerous if...else statements for checking the values and throwing errors, which consumed more gas than before. New error handling constructs assert, require, and revert statements were introduced with the release of version 0.4.10. The throw statement was made absolute with the release of version 0.4.10.

Listed below are some of the most important methods that are utilized in error handling:

assert(bool condition)- This method call results in an invalid opcode and any changes made to the state are undone. If the condition is not met, the method call returns false. This technique should only be used in the case of internal errors.

require(bool condition)- In case a condition is not met, this method call returns the system to its previous state. This technique is intended to be utilized in the event of input or external component failures.

require(bool condition, string memory message) - This reverts the state of the object to its original state if the condition is not met by this method call. - This technique is intended to be utilized in the event of input or external component failures. It gives you the option of sending a personalized message.

revert()- This function aborts the execution and undoes any modifications that have been made to the state of the object.

revert(string memory reason) - This method aborts the execution and undoes any modifications that have been made to the state of the application. It gives you the option of sending a personalized message.

The Require Statement

The 'require' statements specify the conditions that must be met before the function can be executed, i.e., they specify the constraints that must be met before the code can be executed. This function only receives a single input and evaluates it to return a boolean value; it also provides an option for a custom string message. If this is the case, an exception is thrown and the program is terminated. This results in the caller receiving any unused gas, as well as the state being reversed back to its previous condition. The following are some examples of situations in which the required sort of exception is triggered:

* When the function require() is invoked with the specified arguments, the outcome is false.
* When a function that was called by a message does not correctly terminate.
* It occurs when a contract is made using the new keyword and the procedure does not complete successfully.
* Any time a codeless contract is directed at an external function
* Ethers are sent to the contract via the public getter method when this method is called.
* When the .transfer() method does not work.
* In the case of an assert that is called with a condition that results in a false result,
* It occurs when a function that has a zero-initialized variable is called.
* It occurs when a process through which a big or a negative value is turned to an enumeration.

| // Solidity program to demonstrate require statement pragma solidity ^0.8.0;   // Creating a contract contract requireStatement {    // Defining function to check input  function checkInput(uint \_input) public view returns(string memory){  require(\_input >= 0, "invalid uint8");  require(\_input <= 255, "invalid uint8");    return "Input is Uint8";  }    // Defining function to use require statement  function Odd(uint \_input) public view returns(bool){  require(\_input % 2 != 0);  return true;  } } |
| --- |

The Assert Statement

Its syntax is very similar to that of the require statement. Following the execution of the condition, it returns a boolean value to the user. It will depend on the return value whether the application will continue to run or whether it will throw an exception. Instead of returning the unused gas, the assert statement uses the entire gas supply, causing the state to be reverted back to its initial condition. Before executing the contract, Assert is used to verify that the current state and function circumstances are as expected. The following are some examples of assert type exceptions:

* When an assert is called with a condition that results in false.
* When a zero-initialized variable of a function is called.
* When a large or a negative value is converted to an enum.
* When a value is divided or modulo by zero.
* When accessing an array in an index which is too big or negative.

| pragma solidity ^0.8.0;   // Creating a contract contract assertStatement {    // Defining a state variable   bool result;    // Defining a function to check condition  function checkOverflow(uint \_num1, uint \_num2) public {  uint sum = \_num1 + \_num2;  assert(sum<=255);  result = true;  }    // Defining a function to print result of assert statement  function getResult() public view returns(string memory){  if(result == true){  return "No Overflow";  }  else{  return "Overflow exist";  }  }  } |
| --- |

The Revert Statement

Similar to the require statement, this one specifies the condition. It does not assess any conditions and does not rely on any state or statement in order to function properly. It can be used to generate exceptions, display errors, and revert the function call, amongst other things. This statement comprises a string message that describes the problem with the information provided by the exception. It is implied by the usage of the revert statement that an exception is raised, the used gas is returned, and the state is restored to its initial state. Revert is used to handle the same types of exceptions as require does, but with a little bit more advanced logic than require does.

| pragma solidity ^0.8.0;   // Creating a contract contract revertStatement {    // Defining a function to check condition  function checkOverflow(uint \_num1, uint \_num2) public view returns(string memory, uint){  uint sum = \_num1 + \_num2;  if(sum < 0 || sum > 255){  revert(" Overflow Exist");  }  else{  return ("No Overflow", sum);  }    }   }   } |
| --- |

Some more examples

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract Error {  function testRequire(uint \_i) public pure {  // Require should be used to validate conditions such as:  // - inputs  // - conditions before execution  // - return values from calls to other functions  require(\_i > 10, "Input must be greater than 10");  }   function testRevert(uint \_i) public pure {  // Revert is useful when the condition to check is complex.  // This code does the exact same thing as the example above  if (\_i <= 10) {  revert("Input must be greater than 10");  }  }   uint public num;   function testAssert() public view {  // Assert should only be used to test for internal errors,  // and to check invariants.   // Here we assert that num is always equal to 0  // since it is impossible to update the value of num  assert(num == 0);  }   // custom error  error InsufficientBalance(uint balance, uint withdrawAmount);   function testCustomError(uint \_withdrawAmount) public view {  uint bal = address(this).balance;  if (bal < \_withdrawAmount) {  revert InsufficientBalance({balance: bal, withdrawAmount: \_withdrawAmount});  }  } } |
| --- |

| // SPDX-License-Identifier: MIT pragma solidity ^0.8.0;  contract Account {  uint public balance;  uint public constant MAX\_UINT = 2\*\*256 - 1;   function deposit(uint \_amount) public {  uint oldBalance = balance;  uint newBalance = balance + \_amount;   // balance + \_amount does not overflow if balance + \_amount >= balance  require(newBalance >= oldBalance, "Overflow");   balance = newBalance;   assert(balance >= oldBalance);  }   function withdraw(uint \_amount) public {  uint oldBalance = balance;   // balance - \_amount does not underflow if balance >= \_amount  require(balance >= \_amount, "Underflow");   if (balance < \_amount) {  revert("Underflow");  }   balance -= \_amount;   assert(balance <= oldBalance);  } } |
| --- |

Try/Catch block

pragma solidity ^0.8.0;

contract errorContract{

function errorFunction() public payable returns(bool){

revert();

}

}

contract tryCatch{

bool public success = false;

uint256 public errorCount = 0;

errorContract eC;

function testTryCatch(errorContract \_errorContractAddress) public{

eC = \_errorContractAddress;

try eC.errorFunction(){

success = true;

}catch{

success = false;

errorCount++;

}

}

}