2.3.4.8.4.1 Exceptions

Return to Operational Data

Overview

Return to Top

An important aspect of executing any set of instructions is **`What happens if there is an error?`**.

The error condition could arise from system errors such as:

- **Access denied Segmentation Fault (SEGFAULT)** occurs when the user does not have privileges to a resource, such as a file, or when the resources are locked by some program or user. Other examples would be trying to access memory that is outside the current program space or trying to execute a null instruction.
- **Device not ready** occurs when an external device is not ready or available. For example, a disk drive that has not been mounted or accessing a monitor that has been disconnected. The resource may have been powered down, disconnected, damaged, moved, deleted, or had a fault. Alternatively, the resource as specified may simply not exist (i.e., typographical errors in its name).
- **Low Disk Space** occurs when the disk system is full or nearly full.
- **Out of memory** occurs when the system runs out of **Heap** and/or **Stack Memory** memory. The system can not load new programs or functions (i.e., **Stack**) or execute the existing ones (i.e., **Heap** limit).

The error conditions could arise from runtime errors such as:

- **Dividing by zero** occurs when, during execution, an attempt is made to divide a number by zero. Because these types of errors can not be c=found at compile-time, they occur at runtime.
- **Out of Bound** occurs when a program tries to access an element in an array that is out of range of the array. Many languages dimension arrays 0 to +number), some allow negative indexing and others do not.
- **Type Mismatch** occurs when an attempt is made to store a value into an array that is not compatible (i.e., trying to store an integer in an array of strings).
- **Range Error** occurs when a value does not meet the allowable range of values specified by the code. For example, trying to set a value of 200 into a value described by a percent type that ranges from 0 to 100.
- **Number Format Error** occurs when an attempt is made to convert a string to a number, but the string does not match the described format. For Example, trying to convert the String **“one”** to a number.

The error conditions could arise from syntax or semantic errors. **Note:** strongly typed systems avoid these errors) such as

- **`Value must be an Unsigned Integer`** occurs when the basic types of parameters do not

match. Usually, this occurs when the code has changed or the correct Shared Library can not be found.
- **`Too Few Parameters`** occurs when the code tries to call a procedure, function or method that requires more parameters than can be found in the existing libraries. This can occur due to changes in the calling signature or when a Shared Library is not found.
- **`Class Not Found`** occurs when the code is trying to instantiate a class whose definition is not included in the current program or any of the other Shared Library.
- **`Interrupted`** occurs when one thread or process interrupts another thread or process.

In most modern Object-Oriented (OO) languages (i.e., Java, C#, C++, Python, etc.) these **Error Conditions** are processed through the use of **Exceptions**. When an error occurs, regardless of where an Exception is thrown which terminates the current process. Terminating the process amounts to "popping" the **Stack Memory** for the most recent call and (if possible) releasing memory from the **Heap**. This process of unwinding the **Stack** and the **Heap** until the **Exception** is caught by an Exception Handler programmed to handle the particular kind of error thrown. If an **Exception Handler** is encountered that does not handle the error, the exception is rethrown until an **Exception Handler** is found that does handle the error or the program **Stack** is empty (nothing left to pop), which terminates the program and control reverts to the Operating System (OS).

Exception Handlers can be quite specific in which **Exceptions** they can handle, or they can be very generic.

To catch specific **Exceptions** that can arise from a method, many OO languages allow for a method to declare what known exceptions can be thrown from the method. For example, if a method allocates memory from the **Heap**, it may need to indicate it can throw an **`Out of Memory`** exception. The following example of a writeMessage routing demonstrates how a method can inform a caller of the potential exceptions it can throw (i.e., **`IOEception`**).

Example of throwing an exception in Java.

```
public void writeMessage
  ( char[] message
  ) throws IOException;
```

To generically catch all **Exceptions** in most OO languages, using a simple **`Catch (Exception e)`** block within the **Exception Handler** is all it takes. It is considered good programming practice to do this at most component, subsystem or system boundaries and to log these conditions for later review. These log entries also usually include a **Stack Trace**, which in simple terms is a list of the method calls that the application was in middle of when an Exception was thrown. [1)]

Example of defining an exception in Java.

```


public class IncorrectValueExcpetion
  extends Exception
{  public IncorrectValueExcpetion
    ( String errorMessage )
  { super(errorMessage);
```

```
  } // End IncorrectFileNameException
} // End IncorrectValueExcpetion

void myProcedure()
  throws IncorrectValueExcpetion
{ throw new IncorrectValueExcpetion
    ( "Opps! The vaue is incorrect");
} // End myProcedure

void tester()
{ try
  { myProcedure();
  } // End try
  catch ( IncorrectValueExcpetion incorrectValueExcpetion )
  { // Prints what exception has been thrown
    System.out.println ( incorrectValueExcpetion );
  } // End MyException block
  catch ( Exception exception )
  { exception.printStackTrace();
  } // End generic catch block
} // End tester
```

DIDO Specifics

Return to Top

Ethereum's Solidity is considered the “standard” for creating smart contracts. The language supports multiple mechanisms (language constructs) that allow for raising of exceptions:

throw

Originally, Solidity had a **`throw`** expression which was analogous to the **`throw`** in other languages such as Ada, C++, C#, Java, etc. However, there is a major difference: in the Solidity **`throw`**, all the state changes made in a smart contract are undone automatically when the **`throw`** occurs, similarly to a database performing a rollback within a transaction. The Solidity keyword **`throw`** has been deprecated in Solidity version 0.4.13 and removed in version 0.5.0. It has been replaced with a **`revert()`** expression with the same semantics as the original **`throw`** expression. However, **`throw`**. It has been reported that the **`revert`** was more readable and semantically also implied the rollback functionality. In addition, the original **`throw`** did not refund any unused gas while the **`revert`** expression does.

revert

Does not evaluate any condition and does not depend on any state or statement. It is used to generate

exceptions, display errors, and revert the function call. This statement contains a string message which indicates the issue related to the information of the exception. Calling a revert statement implies an exception is thrown, the unused gas is returned, and the state reverts to its original state.

Example of using the **revert** exception mechanism

```
pragma solidity ^0.5.11

contract Innventory
{ unit public quantityInStock;
  function checkInventory() external view
    { if ( quantityInStock <= 0 )
      { revert ( 'quantityInStock must be greater than or equal to 0');
      } // End if
    } // End checkInventory
} // End contract Inventroy
```

Note: The reason displayed for the error in the console is quantityInStock must be greater than or equal to 0.

require

Evaluates a boolean expression representing prerequisites for running the function (i.e., it declares the constraints which should be satisfied before executing the code). The **require** accepts a single argument and returns a boolean value after evaluation, it supports another optional argument a custom string message option. If **false**, an exception is raised and execution is terminated. The unused gas is returned to the caller and the state is reversed to its original state.

Example of using the **require** exception mechanism

```
pragma solidity ^0.5.11

contract Innventory
{ unit public quantityInStock;
  function checkInventory() external view
    { require ( quantityInStock <= 0 , 'quantityInStock must be greater or
equal to than 0');
    } // End checkInventory
} // End contract Inventroy
```

Note: The reason displayed for the error in the console is quantityInStock must be greater than or equal to 0.

assert

Evaluates a boolean expression and either the program will continue its execution (**`true`**) or when (**`false`**) throws an exception. Instead of returning the unused gas, the assert statement consumes the entire gas supply and the state is then reversed to the original state. **`Assert`** is used to check the current state and function conditions before the execution of the contract. The **`assert`** should only be used during testing and should check for conditions that should never occur in a finished **Smart Contract**.

Example of using the **`assert`** exception mechanism

```
pragma solidity ^0.5.11

contract Inventory
{ unit public quantityInStock;
  function checkInventory() external view
    { assert ( quantityInStock <= 0 );
    } // End checkInventory
} // End contract Inventroy
```

Note: The reason displayed for the error in the console is Invalid Opcode.

propagated exceptions

Occurs when an exception (**`throw`**, **`revert`**, **`require`**, **`assert`**, **`illegal op-code`**) occurs somewhere within the call stack. The call stack is unwound (i.e., popped) until the exception is caught, or the main program is terminated because of the exception. Within Solidity, this occurs when one smart contract calls another smart contract or library function and an exception occurs.

No Error Recover

In the following example:

1. There are two contracts, **`Inventory`** and **`ShoppingCart`** defined (lines 3 and 12 respectively).
2. The **`ShoppingCart`** contract makes a call to the **`Inventory`** contract by creating a pointer (line 17) to a new instance of the **`Inventory`** contract.
3. The **`ShoppingCart`** then makes a call to the **`Inventory`** the operation (i.e., function) **`checkInventory`** (line 18).
4. Since we have never changed the **`quantityInStock`** from its original zero value, an error condition is detected at (line 6).
5. An exception is called using the **`revert`** statement (line 7).
6. Any changes made to **`Inventory`** are unwound and the stack is popped, returning control to **`ShoppingCart`** with an exception which causes the **`ShoppingCart`** contract to **`undo`** all the state changes made and terminates the **`ShoppingCart`**.

Note: This process is repeated until the stack is empty, or the exception is caught. (See next sections).

Example of a propagated exceptions.

```
pragma solidity ^0.5.11;

contract Inventory
{ uint external public quantityInStock;
  function checkInventory() external view
    { if ( quantityInStock <= 0 )
      { revert ( "quantityInStock must be greater than 0");
      } // End if
    } // End checkInventory
} // End Inventroy contract

contract ShoppingCart
{ uint public numberToOrder;
  function addItemToCart
   ( uint desiredQuantity
   ) external
  { Inventory inventoryPointer = new Inventory();
    inventoryPointer.checkInventory();
    if ( desiredQuantity < inventoryPointer.quantityInStock() )
    { revert ( "Not enough inventory for the order");
    } // End if
  } // End addItemToCart operation
} // End ShoppingCart contract
```

With Error Recover

An alternative to having no error recovery during the execution of a subordinate smart contract (i.e., **`Inventory`** from inside **`ShoppingCart`**), is to:

1. Use the low-level **`call`** mechanism to invoke **`checkInventory()`** which returns a pointer to the new **`Inventory`** contract. (Line 17).
2. Save the address of the pointer (i.e., **`inventoryPointer`**) to **`inventoryAddress`** by casting the pointer to an **`address`** (Line 18)
3. Determine if there is any inventory by calling the **`checkInventory()`** method of the **`Inventory`** contract (Line 29).
4. If there is inventory (i.e., **`hasInventory`** is true) (Line 20)
5. Then determine if there is enough inventory to fulfill the request (line 21)
6. If there is not enough inventory to fill the request, use the **`revert`** error (Line 22)
7. Else, do something to fulfill the order and decrement the quantity in the **`Inventory`** as appropriate (Line 23)

Alternate of catching an Exception without propagating up the stack.

```
contract ShoppingCart
{ uint public numberToOrder;
  function addItemToCart
   ( uint desiredQuantity
   ) external
  { Inventory inventoryPointer = new Inventory();
    address inventoryAddress = address(inventoryPointer);
    (bool hasInventory,  )
       = inventoryAddress.call ( abi.encodePacked ( "checkInventory()" ) );
    if ( hasInventory )
    {  if ( desiredQuantity < inventoryAddress.quantityInStock )
       { revert ( "Not enough inventory for the order");
       } // End if
       // do something here!!
    } // End if
  } // end addItemToCart operation
} // end ShoppingCart contract
```

Note: The **`call`** is a low-level mechanism used to invoke smart contract operations, but it is subject to re-entrance attacks:
Computer scientists say that a procedure is re-entrant if its execution can be interrupted in the middle, initiated over (re-entered), and both runs can complete without any errors in execution. In the context of Ethereum smart contracts, re-entrancy can lead to serious vulnerabilities. [2)].

With Error Catching

Catching errors is easy and efficient in Solidity after version 0.6.0 with its introduction of **`try/catch`** blocks. The previous **Error Recovery** method might be acceptable when there are one to a few smart contracts executed within a smart contract. However, when there are many subordinate contracts called from within a smart contract, the control flow can become very complex and hard to read (see Lines 19-25 above). The **`try/catch`** blocks allow a section of code to be surrounded with an exception handler that can covert a single subordinate contract call (i.e., **`Inventory`**) or multiple contract invocations l using the same exception handling software.

1. An attempt is made to call **`checkInventory`** (Line 19).
2. Block is executed if the call to **`checkInventory`** did not raise an exception (Line 20-22).
    1. The **`successFlag`** is set to **`true`** (Line 21).
3. Block is executed in case a **`revert`** with a reason string was called inside **`Inventory`** contract (Line 23-26).
    1. The **`successFlag`** is set to **`false`** (Line 24).
4. Block is executed in case a **`revert()`** was used or there was a failing assertion, **`division by`**

zero, etc. inside **Inventory** contract (Line 27-30).
    1. The **successFlag** is set to **false** (Line 28).
5. There is only one place where the program returns to the caller (Line 31).

Alternate of catching an Exception with a **try/catch** block.

```
contract ShoppingCart3
{ uint public numberToOrder;
  function addItemToCart
   ( uint desiredQuantity
   ) external returns ( bool success )
  { Inventory inventoryPointer = new Inventory();
    bool successFlag = false;
    try inventoryPointer.checkInventory()
    { // Do something here!
      successFlag = true;
    } // End try
    catch Error ( string memory )  // Reason
    { // do something here and log the error here
      succsuccessFlagess = false;
    } // End Error Block
    catch ( bytes memory ) //lowLevelData
    { // do something here and log the error here
      successFlag = false;
    } // End generic catch Block
    return successFlag;
  } // end addItemToCart operation
} // end ShoppingCart contract3
```

☒ [char][✓ char, 2022-03-22]~~Review~~

1)

Stackoverflow, What is a stack trace, and how can I use it to debug my application errors?, 2010, Accessed: 6 November 2021, https://stackoverflow.com/questions/3988788/what-is-a-stack-trace-and-how-can-i-use-it-to-debug-my-application-errors

2)

Martin Derka, What is a Re-Entrancy Attack?, Quantstamp, 19 August 2019, Accessed: 30 November 2021, https://quantstamp.com/blog/what-is-a-re-entrancy-attack