2.3.4.8.4 Operation Data

Return to Object Data Taxonomy

Overview

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 are 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 1 provides a graphic of the difference between **Imperative** and **Procedural** Programs.

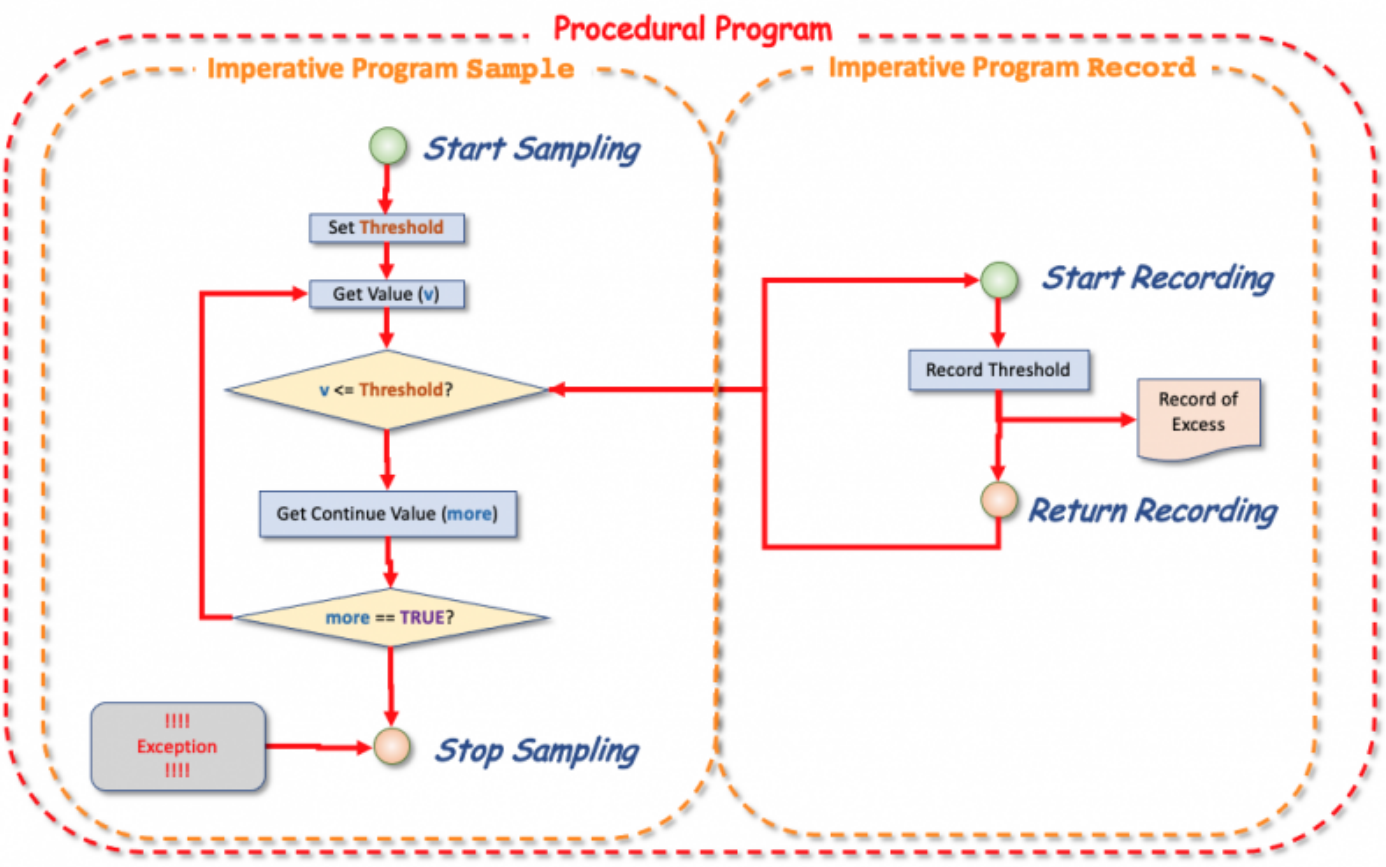

Figure 1: The difference between **Imperative** and **Procedural** Programming.

To confuse the situation, even more, there is a differentiation between the Imperative or Procedural: function, procedure, and method.

function	a named set of instructions that perform a task and return a single value to the caller of the function. Generally, all the parameters passed to the function are considered **IN** (i.e., Read-Only).
procedure	a named set of instructions that perform a task and do not return any values to the caller. The results of the procedure call are changes made to the parameters which can be declared, **IN**, **OUT** (i.e., Write-Only), or **INOUT** (i.e., Read-Write). Procedures often behave like **functions** in that they return a single value to the caller. The value can be a success flag or a return code. The caller can examine the results to help control the flow through the program. Most modern languages would rather use **Exceptions** instead.
method	a **function** or a **procedure** that is associated with an Object in OOP.

> **Note:** Regardless of the classification of the set of instructions, there should be no unintended side effects from calling the function. For example, calling a function should not affect anything else in the system. For example, calling a **square Root** function should not start a printer.

Exceptions

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, 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`**).

```
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 the middle of when an Exception was thrown. [1)]

```
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

An interesting phenomenon about many of the DIDO Platforms, is the way they treat **Operation Data**. Many of the Platforms not only distribute **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**.

- 2.3.4.8.4.2 Constructor / Destructor Methods
- 2.3.4.8.4.3 Accessor / Mutator Methods
- 2.3.4.8.4.4 Abstract/Virtual Methods
- 2.3.4.8.4.5 Special Methods
- 2.3.4.8.4.6 Pure Methods
- 2.3.4.8.4.7 Visibility

☒ [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