User Defined Exception

Return to Glossary

A **User Defined Exception** is an Exception designed by the system engineers to occur during runtime to reflect aberrant conditions that arise during execution, often preventing other more generic runtime conditions such as Overflow, Wrap Around, Underflow, Division by Zero (DIV/0) etc. The **User Defined Exceptions** typically provide specific diagnostic messages to help during later forensic analysis of logs. In the following example, interests are checked until n interest rate greater than 20%, and when they are, the Usury exception is raised.

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

class Usury : public exception
{ int _interestRate;
  public :
    const char* explaination()
    { return "Usury Interest Rate detected";
    } // End explaination
}; // End Exception Usury
int main()
{ float interestRate = 0;
  try
  { while(1)
    { interestRate+=interestRate+.5;
      if ( interestRate >= 20.0 )
      { Usury usery;
        throw usery;
      } // End interestRate usery check
      cout << "Interest Rate: " << interestRate <<endl;
    } // End while loop
  } // End try block
  catch ( Usury _usuryException )
  { cout  interestRate << " interest "<<  ex.explaination();
  } // End catch block
  return 0;
} // End main routine
```

When these occur, the system catches the error and a **User Defined Exception** is raised.

Deciding whether to use an **User Defined Exception** to or a return values for Control Flow is a balance of several “forces”:

- The caller can ignore a return value and continue with the next line, but the caller can not ignore an Exception. Exceptions tend to be used in cases where ignoring the problem and continuing could lead to a bigger problem.
- Exceptions impose a burden because it requires the caller to provide specific code to handle the exceptional behavior rather than just checking a return value. Since there is no way to check if the return code is checked or acted upon by the calling routine, the return code basically becomes optional and should be the decision producer of the code
- However, Exceptions relieve the immediate caller of the burden of checking the result since when Exceptions are raised, the Stack Memory is unwound
- As a general rule, Exception handling is more complex than using ordinary return codes and Control Flow, so if the **User Defined Exceptions** are thrown often there is a performance hit when thrown frequently.
- Exceptions are better when the design required a return code to be more than just a simple integer (e.g., making the method return Object so that it can return either a string or a number).

Source: Defined by the DIDO Reference Architecture