Exercise IV: Exception Handling

1. Exception Semantics (Refer to ECMA 8.26.8 for complete description)

An exception triggered during an execution of a feature $f$ causes the effect of the following sequence of events.

1. A direct/indirect instance of EXCEPTION is created and made available via `last_exception`; (See next page for accessing `last_exception`)
2. The rest instructions in the body of $f$ are skipped;
3. If the recipient of the exception is $g$, execute the rescue block of $g$. ($g$ can be the same as $f$)
4. If case 3 applies and the rescue block executes a Retry, this terminates the processing of the exception. Execution continues with a new execution of the feature body of $g$.
5. If neither case 3 nor case 4 applies (in particular in case 3 if the rescue block executes to the end without executing a Retry), this terminates the processing of the current exception and the current execution of $g$, causing a failure of that execution. This failure will further cause an exception of type ROUTINE_FAILURE in the calling routine of $g$, or cause the program to fail if $g$ is the root procedure.

2. Exception Cases (Refer to ECMA 8.26.10 for complete description)

Concerning the exception object instantiated in step 1 above, two properties about the exception will be set, as per the following table, where:

- The "Recipient" is either $f$ or $f$'s caller
- "Type" indicates the type of the exception (a descendant of EXCEPTION).
- If $f$ is the root procedure, executed during the original system creation call, the value of caller as given below does not apply.

<table>
<thead>
<tr>
<th>Exception cases</th>
<th>Recipient</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception during evaluation of invariant on entry</td>
<td>caller</td>
<td>[Type of exception as triggered]</td>
</tr>
<tr>
<td>Invariant violation on entry</td>
<td>caller</td>
<td>INVARIANT_ENTRY_VIOLATION</td>
</tr>
<tr>
<td>Exception during evaluation of precondition</td>
<td>caller</td>
<td>[Type of exception as triggered]</td>
</tr>
<tr>
<td>Exception during evaluation of Old expression on entry</td>
<td>See Old Expression Semantics</td>
<td></td>
</tr>
<tr>
<td>Precondition violation</td>
<td>caller</td>
<td>PRECONDITION_VIOLATION</td>
</tr>
<tr>
<td>Exception in body</td>
<td>$f$</td>
<td>[Type of exception as triggered]</td>
</tr>
<tr>
<td>Exception during evaluation $f$ of invariant on exit</td>
<td>$f$</td>
<td>[Type of exception as triggered]</td>
</tr>
<tr>
<td>Invariant violation on exit</td>
<td>$f$</td>
<td>INVARIANT_EXIT_VIOLATION</td>
</tr>
<tr>
<td>Exception during evaluation $f$ of postcondition on exit</td>
<td>$f$</td>
<td>[Type of exception as triggered]</td>
</tr>
<tr>
<td>Postcondition violation</td>
<td>$f$</td>
<td>POSTCONDITION_VIOLATION</td>
</tr>
</tbody>
</table>
3. More Exception Handling in Eiffel

- Defining and raising your own exceptions: DEVELOPER_EXCEPTION

```eiffel
class UNICODE_MESSAGE_EXCEPTION inherit DEVELOPER_EXCEPTION
end
```

```eiffel
foo
local l_unicode_ex: UNICODE_MESSAGE_EXCEPTION
...
do
  create l_unicode_ex
  ...
  l_unicode_ex.raise
end
```

- Fine-tuned exception handling: EXCEPTIONS

```eiffel
class MY_TASK inherit EXCEPTIONS
feature -- task
  process ( ... )
    -- process the task
    do
      ...
    rescue
      if exception = Developer_exception then
        -- if is_developer_exception then
        ...
      end
      retry
    end
...
end
```

- Exception type codes: EXCEP_CONST
- Accessing the exception object: EXCEPTION_MANAGER

```eiffel
class MY_TASK inherit EXCEPTIONS
feature -- task
  process ( ... )
    -- process the task
    do
      ...
    rescue
      if attached {UNICODE_MESSAGE_EXCEPTION}
        exception_manager.last_exception.original as lt_ex then
          -- do something with lt_ex
        else
          -- do something else
        end
      retry
    end
...
end
```

4. Exercise

In class INTEGER_8 from EiffelBase, the algebra operations return quietly incorrect values when overflow happens. In this exercise, you need to write a class SAFE_INTEGER_8 to represent the 8-bit signed integers with “safe” algebra operations, i.e. they would raise exceptions upon errors rather than return bad results.

Design the SAFE_INTEGER_8 class and the necessary exception class such that an exception would be raised if an operation cannot return successfully, and that the exception object should contain information about the failing operation name.