RUBY EXCEPTIONS

The execution and the exception always go together. If you are opening a file, which does not exist, then if you did not handle this situation properly, then your program is considered to be of bad quality.

The program stops if an exception occurs. So exceptions are used to handle various type of errors, which may occur during a program execution and take appropriate action instead of halting program completely.

Ruby provide a nice mechanism to handle exceptions. We enclose the code that could raise an exception in a *begin/end* block and use *rescue* clauses to tell Ruby the types of exceptions we want to handle.

Syntax :

```
begin
# -
rescue OneTypeOfException
# -
rescue AnotherTypeOfException
# -
else
# Other exceptions
ensure
# Always will be executed
end
```

Everything from *begin* to *rescue* is protected. If an exception occurs during the execution of this block of code, control is passed to the block between *rescue* and *end*.

For each *rescue* clause in the *begin* block, Ruby compares the raised Exception against each of the parameters in turn. The match will succeed if the exception named in the rescue clause is the same as the type of the currently thrown exception, or is a superclass of that exception.

In an event that an exception does not match any of the error types specified, we are allowed to use an *else* clause after all the *rescue* clauses.

Example:

```
#!/usr/bin/ruby

begin
   file = open("/unexistant_file")
   if file
      puts "File opened successfully"
   end
rescue
      file = STDIN
end
print file, "==", STDIN, "\n"
```

This will produce the following result. You can see that *STDIN* is substituted to *file* because *open* failed.

```
#<IO:0xb7d16f84>==#<IO:0xb7d16f84>
```

Using *retry* Statement:

You can capture an exception using *rescue* block and then use *retry* statement to execute *begin* block from the beginning.

Syntax:

```
begin
    # Exceptions raised by this code will
 # be caught by the following rescue clause
rescue
    # This block will capture all types of exceptions
    retry  # This will move control to the beginning of begin
end
```

Example:

```
#!/usr/bin/ruby

begin
   file = open("/unexistant_file")
   if file
      puts "File opened successfully"
   end
rescue
   fname = "existant_file"
   retry
end
```

The following is the flow of the process:

- an exception occurred at open
- went to rescue. fname was re-assigned
- by retry went to the beginning of the begin
- this time file opens successfully
- continued the essential process.

NOTE: Notice that if the file of re-substituted name does not exist this example code retries infinitely. Be careful if you use *retry* for an exception process.

Using *raise* Statement:

You can use *raise* statement to raise an exception. The following method raises an exception whenever it's called. It's second message will be printed. Program

Syntax:

```
raise

OR

raise "Error Message"

OR

raise ExceptionType, "Error Message"

OR

raise ExceptionType, "Error Message" condition
```

The first form simply reraises the current exception *oraRuntimeErroriftherеisnocurrentexception*. This is used in exception handlers that need to intercept an exception before passing it on.

The second form creates a new *RuntimeError* exception, setting its message to the given string. This exception is then raised up the call stack.

The third form uses the first argument to create an exception and then sets the associated message to the second argument.

The fourth form is similar to third form but you can add any conditional statement like *unless* to raise an exception.

Example:

```
#!/usr/bin/ruby

begin
    puts 'I am before the raise.'
    raise 'An error has occurred.'
    puts 'I am after the raise.'
rescue
    puts 'I am rescued.'
end
puts 'I am after the begin block.'
```

This will produce the following result:

```
I am before the raise.
I am rescued.
I am after the begin block.
```

One more example showing usage of *raise*:

```
#!/usr/bin/ruby

begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
end
```

This will produce the following result:

```
A test exception.
["main.rb:4"]
```

Using *ensure* Statement:

Sometimes, you need to guarantee that some processing is done at the end of a block of code, regardless of whether an exception was raised. For example, you may have a file open on entry to the block and you need to make sure it gets closed as the block exits.

The *ensure* clause does just this. ensure goes after the last rescue clause and contains a chunk of code that will always be executed as the block terminates. It doesn't matter if the block exits normally, if it raises and rescues an exception, or if it is terminated by an uncaught exception, the *ensure* block will get run.

Syntax:

```
begin
   #.. process
   #..raise exception
rescue
   #.. handle error
ensure
   #.. finally ensure execution
   #.. This will always execute.
end
```

Example:

```
begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
ensure
  puts "Ensuring execution"
end
```

This will produce the following result:

```
A test exception.
["main.rb:4"]
Ensuring execution
```

Using *else* Statement:

If the *else* clause is present, it goes after the *rescue* clauses and before any *ensure*.

The body of an *else* clause is executed only if no exceptions are raised by the main body of code.

Syntax:

```
begin
   #.. process
   #..raise exception
rescue
   # .. handle error
else
   #.. executes if there is no exception
ensure
   #.. finally ensure execution
   #.. This will always execute.
end
```

Example:

```
begin
 # raise 'A test exception.'
 puts "I'm not raising exception"
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
else
   puts "Congratulations-- no errors!"
ensure
  puts "Ensuring execution"
end
```

This will produce the following result:

```
I'm not raising exception
Congratulations-- no errors!
Ensuring execution
```

Raised error message can be captured using $! variable.

Catch and Throw:

While the exception mechanism of raise and rescue is great for abandoning execution when things go wrong, it's sometimes nice to be able to jump out of some deeply nested construct during normal processing. This is where catch and throw come in handy.

The *catch* defines a block that is labeled with the given name *whichmaybeaSymboloraString*. The block is executed normally until a throw is encountered.

Syntax:

```
throw :lablename
#.. this will not be executed
catch :lablename do
#.. matching catch will be executed after a throw is encountered.
end

OR

throw :lablename condition
#.. this will not be executed
catch :lablename do
#.. matching catch will be executed after a throw is encountered.
end
```

Example:

The following example uses a throw to terminate interaction with the user if '!' is typed in response to any prompt.

```
def promptAndGet(prompt)
   print prompt
   res = readline.chomp
   throw :quitRequested if res == "!"
   return res
end

catch :quitRequested do
   name = promptAndGet("Name: ")
   age = promptAndGet("Age: ")
   sex = promptAndGet("Sex: ")
   # ..
   # process information
end
promptAndGet("Name:")
```

You should try above program on your machine because it needs manual interaction. This will produce the following result:

```
Name: Ruby on Rails
Age: 3
Sex: !
Name:Just Ruby
```

Class Exception:

Ruby's standard classes and modules raise exceptions. All the exception classes form a hierarchy, with the class Exception at the top. The next level contains seven different types:

- Interrupt
- NoMemoryError
- SignalException
- ScriptError
- StandardError
- SystemExit

There is one other exception at this level, Fatal, but the Ruby interpreter only uses this internally.

Both ScriptError and StandardError have a number of subclasses, but we do not need to go into the details here. The important thing is that if we create our own exception classes, they need to be subclasses of either class Exception or one of its descendants.

Let's look at an example:

```
class FileSaveError < StandardError
   attr_reader :reason
   def initialize(reason)
      @reason = reason
   end
end
```

Now, look at the following example, which will use this exception:

```
File.open(path, "w") do |file|
begin
    # Write out the data ...
rescue
    # Something went wrong!
    raise FileSaveError.new($!)
end
end
```

The important line here is raise *FileSaveError.new$!*. We call raise to signal that an exception has occurred, passing it a new instance of FileSaveError, with the reason being that specific exception caused the writing of the data to fail.