# Forward

## About the tutorial

Every useful library will, sooner or later, encounter an abnormal condition, like a missing file, that the library itself can’t handle. The usual reaction of the program is often to somehow display an error message. Simply printing unstructured text out stderr is inconvenient for library users who may need to handle the condition programmatically, translate the message, or display the message elsewhere, like a GUI or web page.

This tutorial is about handling such conditions properly, especially in libraries, in SWI-Prolog, using a system borrowed from Quintus Prolog and enshrined in the ISO Prolog standard. It is particularly important for authors of packs to follow this method, and not simply use format/3. Doing so ensures that libraries can be embedded in diverse environments.

It should take most programmers well under an hour.

This tutorial should be accessible to any programmer who understands the basics of Prolog. It would also be useful to understand the basic idea of an execution stack, and to have seen the common throw/catch mechanism in some computer language.

# Motivation

Is there really a lot to say about error messages?

Yes, there is. That most languages provide little help handling error messages is no reason we can't do better.

Error messages often result from an exception, so our

To start the discussion, let's first look at exception handling in the procedural language Java. I'll go quickly through this material, as I assume most readers will have encountered exceptions in some language.

ANNIE - YOURE OFF TOPIC. TOPIC IS MESSAGE PRINTING, NOT EXCEPTIONS. Cover ISO exceptions and go on.

## Exception handling in Java

What should a piece of code do when it finds itself undable to perform it's normal function? When the log\_user\_on method discovers the user's credentials are invalid or the server is down, most languages provide an exception mechanism, and Java does so as well.

When a method wishes to signal that an abnormal condition has occurred, it performs the special form +throw+, passing it a subclass of +Throwable+ (the exception). The interpreter looks for an enclosing block, possibly looking in higher stack frames, 'unwinding' the stack until it encounters a matching +catch+ block.

At this point the catch block is executed. Usually the catch block will print an error message, but sometimes it will perform someother function, eg. error recovery or orderly shutdown.

It's easy to forget about exceptions when coding. Java enforces some handling of exceptions that are likely to occur by dividing +Throwable+ into two classes, checked and unchecked exceptions.

When an exception is checked, methods calling the throwing method will have to either declare that they throw the exception in their signature, or catch the exception.

The exception itself is semantic. It's an object. Handling it is procedural - it's bound to a variable in the catch clause, and the catch clause must do something with it

Beyond this, there's little help. +Throwable+ has a few methods like +printStackTrace()+, but mostly the programmer is on her or his own.

## Issues

While Java's exception handling

### Semantic exceptions are good

### Catch-print antipattern is a massive DRY violation

### Decorating the exception

### No help to convert semantic content to output string

### 'Printing' could be some other sort of output

### Libraries that print aren't reusable

### Libraries that don't print are foisting a job on the application programmer, and lose encapsulation.

### The application programmer needs to be able to override the library's default conversion behavior

## Review

### Semantic representation- good

### Direct output - bad

### Error message printing is awkward

### Libraries need to handle this reusably

## Exercise - code review some curly brace language code

# Simplified pipeline

## Handle the exception (semantic term)

## Produce list of tokens ("lines")

## Hooks to output or print\_message\_lines

# Producing a message term

## Semantic messages

### Describe semantic message and why it's better

### Format of semantic message term

### ISO exception format

### The ISO exceptions

### The format/2 format - and why not to use it

### The debug/2 format - its for debug/3

## Producing messages

### Directly calling

### Debug library (discussed later)

### Uncaught exceptions

## Exception Handling

### Basics of exceptions and catching

### Rethrowing

### Decorating the exception with prolog\_exception\_hook

### Uncaught exceptions - the top level exception handler

### Transition to print\_message/2

## Review

### Use the ISO messages

### Make your own terms

# Converting messages to lines of strings

## Print\_message/2

## prolog:message//1

## The 'lines' format

# Producing output

## print\_message\_lines/3

## message\_hook

## thread\_message\_hook

## message\_to\_string/2

# The debug library

## Debug messages with debug/3

## The debug message window

## Prolog:debug\_print\_hook