Armstrong State University
Engineering Studies
MATLAB Marina – Exception Handling Primer

Prerequisites
The Exception Handling Primer assumes knowledge of the MATLAB IDE, MATLAB help, arithmetic operations, built in functions, scripts, variables, arrays, logic expressions, conditional structures, iteration, and functions. Material on these topics is covered in the MATLAB Marina Introduction to MATLAB module, MATLAB Marina Variables module, MATLAB Marina Arrays module, MATLAB Marina Logic Expressions module, MATLAB Marina Conditional Structures module, MATLAB Marina Iteration module, and MATLAB Marina functions module.

Learning Objectives
1. Be able to use inline error handling to avoid run-time errors in program code segments and functions.
2. Be able to use MATLAB's exception handling (try-catch, error) to handle run-time errors in program code segments and functions.

Terms
Important terms you will encounter in this module include: inline error handling, exception, exception handling

MATLAB keywords and functions
try, catch, error

Exception Handling
An exception is an unexpected error. Exception (error) handling code can be interspersed with program code and thus handle errors where they occur (for example asking a user for a positive integer and then verifying that the integer entered is positive). Some common errors: failing to obtain requested memory, array subscript out of bounds, invalid function parameters, arithmetic overflow, and divide by zero.

Program generated errors (often from bad data) can be dealt with by: ignoring the error and either generating a bad result or letting program terminate, set some indicator that an error occurred and let another part of the program deal with it, or dealing with the error where it occurs. To deal with errors, inline error handling code is often used. The inline error handling code is usually a conditional statement to check for the error either by itself or as part of a try catch block. For functions, error handling can be done with a wrapper function. Inline error handling has the advantage that it is relatively easy to determine that proper error checking is being done but the inline error handling code typically makes the program code more difficult to understand and maintain.

The MATLAB code of Figure 1a shows the use of an inline if-else to detect a divide by zero error.

```
% read in two numbers
num1 = input('Enter number 1: ');
num2 = input('Enter number 2 (not 0): ');

% perform division
if (num2 == 0)
   num3 = NaN;
   disp('divide by zero');
else
    num3 = num1/num2;
end

disp(num3);
```

Figure 1a, Exception Handling Using Inline if-else for Divide by Zero

MATLAB has a `try catch` construct for exception handling. Code that might generate errors is enclosed in a `try` block. When an error is detected, an exception can be thrown using the `error` function. When an exception is generated, the try block is halted and depending on how the program or function has been designed to respond to the error, MATLAB either enters a `catch` block to handle the exception or exits the program. MATLAB does not distinguish between types of exceptions but the `error` function allows one to pass on information about the type of exception thrown. The simplest form of the `error` function is the one parameter version, `error('msgString')`. This version throws an exception and sets the exception message to the string argument of the `error` function.

The catch block can then either generically handle all exceptions or have specific operations for different types of exceptions. The exception in the catch statement is a variable name that will be assigned the exception object that was last thrown. One can also use the `try catch` construct to handle exceptions generated by MATLAB built functions.

The MATLAB code of Figure 1b shows the use of an inline if-else to detect a divide by zero error and throw an exception if an error is detected. The `error` function will throw an exception but without a `catch` block the program is exited and the error message is displayed.

The MATLAB code of Figure 1c shows the use of exception handling (try-catch) to detect and handle a divide by zero error. In the MATLAB program of Figure 1c, if num2 is zero, an exception with the error message "divide by zero" is thrown by the `error` function. Since there is a catch block, the program enters the catch block rather than exiting the program. The thrown exception is saved in the variable of the catch statement, in this example `err`. In the catch block, the result num3 is set to not a number (NaN) and the exception message is extracted and displayed.

```
% read in two numbers
num1 = input('Enter number 1: ');
num2 = input('Enter number 2 (not 0): ');

% perform division
if (num2 == 0)
    error('divide by zero');
else
    num3 = num1/num2;
end

disp(num3)
```

Figure 1b, Error Handling Using Inline if-else for Divide by Zero

```
% read in two numbers
num1 = input('Enter number 1: ');
num2 = input('Enter number 2 (not 0): ');

% perform division
try
    if (num2 == 0)
        error('divide by zero');
    end
    num3 = num1/num2;
    disp(num3)
catch err
    num3 = NaN;
    disp(err.message);
end
```

Figure 1c, Exception Handling for Divide by Zero

Wrapper Functions

Since recursive functions call clones of themselves, if the error handling is inline code, it will be executed every time the recursive function is called. This degrades performance especially when there is a large number of recursive calls. A wrapper function allows the error handling to be done once during the first call. The wrapper function then calls a local function that actually performs the function operation.

Figure 2 shows the recursive Fibonacci function with an added wrapper function to handle values of n less than zero. One might also want to ensure that n is an integer.

```
function result = fibonacciRecursive(n)
% wrapper function for error handling
if (n >= 0)
    result = localFibonacci(n);
else
    result = [];
end
end        % end wrapper function

% local Fibonacci function
function localResult = localFibonacci(n)

if (n == 0)
    localResult = 0;
elseif (n == 1)
    localResult = 1;
else
    localResult = localFibonacci(n-1) + localFibonacci(n-2);
end
end        % end localFibonacci
```

Figure 2, Recursive Solution to Fibonacci Problem with Wrapper Function

The function filename must match the wrapper function name. The local function `localFibonacci` can only be called by the wrapper function and is not accessible outside of the `fibonacciRecursive` function.

Last modified Thursday, March 05, 2015