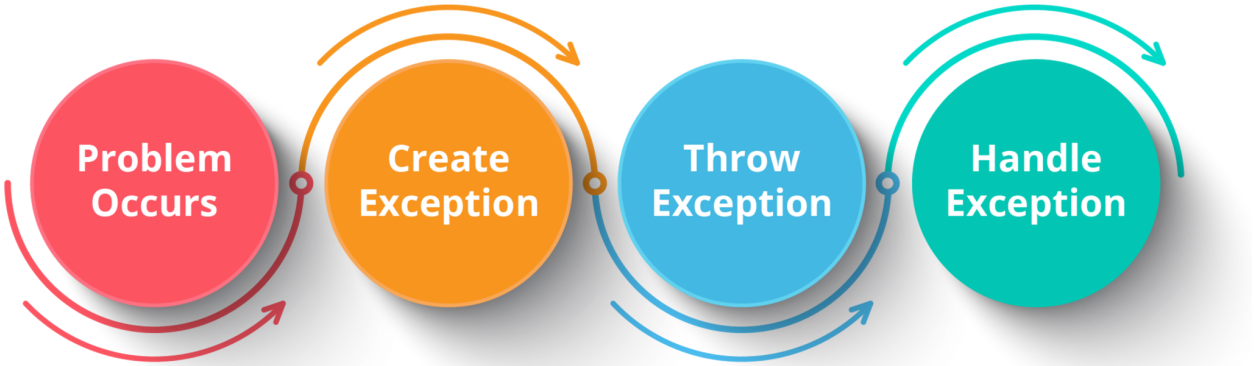

Exceptions and Error Handling

Ryan Hessler & Brandon Mitchell

17 November 2021

CSCI 305: Concepts of Programming Languages

Fall 2021

Why Exceptions and Error Handling?

The graceful handling of exceptions is critical to writing solid and robust code. If your program terminates from an unhandled exception unexpectedly here in college, you might only suffer from a lower grade. Out in the real world, a program that repeatedly crashes due to unhandled exceptions can lead to distrust amongst consumers and damage the reputation of your company. Improper exception handling has also been linked to several aviation disasters. Proper exception handling can protect a program from abnormal inputs and allow it to elegantly handle exceptional situations by passing control over to a section of code that knows what is needed to recover.

Overview

At a high level, exceptions and error handling allow programmers to debug their code and make it as secure as possible. Error handling gives the programmer more insight into problems that occur within the code by providing tracebacks of where the error came from, the line of which it occurred, and sometimes things like the exact error code/name and highlights that show the problematic portion of the sentence. Exceptions, on the other hand, don't directly help debug anything. Exceptions are intended to allow a program to handle unexpected scenarios that may occur during runtime, or even during programming. At runtime, an exception could be used to cast a wide net over the program to capture (and handle) any unexpected inputs from the user. However, they can also be used to keep programmers from creating unintentionally bad code by preventing certain actions from taking place. The main difference between an error and an exception is that an error is a programming language-defined problem,

while an exception is a programmer-defined problem. That being said, almost all programming languages also have pre-defined exceptions built into the language.

Why is it Necessary?

Handling unexpected behaviors is essential to creating good software. In software development, the concept of "defensive programming" best describes the need for such features. Defensive programming is the idea that a piece of software should always be able to continue running in some capacity during unforeseen circumstances. One of the most common ways exceptions can be used is to verify the quality/integrity of data by ensuring data is in the correct format. As we know, there are many ways that data may potentially need to be checked - it needs to be the correct data type, within some type of constraint of possible values, and in the correct format. Exceptions can be used to check for any one of the requirements not being met, and handle it appropriately without crashing the program.

There are still dangers to exception handling, however. Similar to over-commenting, over-handling of exceptions can be detrimental to a program. A programmer must decide what they would like to handle. If there are too many errors/exceptions that are being needlessly listened for, the program's performance and code readability may suffer.

Exceptions at the Low Level

The general flow of an exception is as follows. When a method throws an exception, the first thing the program does is try to handle it. This is done by tracing back up, or unwinding, the call stack of methods that were called to get to the current state to search for a method that contains some code that can handle the exception. This is called the exception handler, which

takes the exception from the runtime system and handles it appropriately. If an exception handler cannot be found, however, the program terminates.

History of Exception Handling

The first occurrence of error handling within programming languages was with Lisp. In Lisp 1.5 (1962), exceptions were caught by the `ERRSET` keyword which returned `NIL` instead of terminating the program or entering the debugger. In 1972, MacLisp included the `CATCH` and `THROW` keywords, which introduced our modern idea of user-defined exception handling. Also in the seventies, NIL, or New Implementation of Lisp, introduced `UNWIND-PROTECT`, which behaved similarly to the cleanup behavior known as `finally`. Exception handling became more widely adopted by many programming languages in the eighties and onward.

Programming Languages with Exception Handling

Basically, any modern programming language today has some capability of using exceptions. The major difference, however, is in how they are used. Some languages (like Ada, OCaml, and Python) are designed for exceptions to be used as control flow structures (as described in a later section). Other languages are designed to have exceptions used specifically to handle unexpected behaviors that may arise at runtime.

There are only a couple modern languages that do not have runtime exceptions in the language. Most notably, Go and Rust do not have exceptions. Instead, they both simply use return values to signal errors. Rust also additionally does have a “panic!” keyword that can be used to intentionally crash the program, but that can hardly be considered an exception.

Exception Handling in Different Languages

Generally, all programming languages handle exceptions in a similar way to that described in the above section titled *Exceptions at the Low Level*. The only major differences are in the keywords used to trigger and handle exceptions. In Python, this would be the `try`, `except`, and `finally` keywords to handle exceptions and the `raise` keyword to throw an exception. Python also allows the use of an `else` keyword that only triggers should the try block execute without error. Java is fairly similar with the keywords `try`, `catch`, `finally`, and `throw`. JavaScript features these keywords, but also has the `.then()`, `.catch()` and `.finally()` methods when working with promises, which are too complicated of a subject to explain in a couple lines. Scheme, or specifically the Racket dialect, has several different ways to both throw and handle exceptions. There is the general `raise` function but also several other functions for throwing specific exceptions like `raise-result-error` and `raise-arguments-error`. Likewise, Scheme has several functions for handling exceptions such as `call-with-exception-handler` and `with-handlers`.

Exception Handling in Modern Languages

Many modern languages implement exceptions and exception handling in similar ways. Typically, if one expects an expression or function to possibly throw (raise) an exception and wishes to detect and handle it, they can wrap it in a `try-catch` construct. The `try` block holds the code that could possibly throw the exception, and it should have as little code as possible to ensure the exception is from that particular expression or function. The optional `catch` block holds the code that will be executed if and only if an exception is thrown in the `try` block. Exceptions can be thrown in `catch` statements, and they will unwind the stack until

caught by another `try-catch` construct. Multiple catch blocks may be allowed in order to respond differently to different exceptions, but a catch block with an `if-else` chain or switch statement can also be used if multiple `catch` statements are not supported. Some languages also include a `finally` block. This `finally` block, if included, will always execute whether or not an exception was thrown, and its purpose is to free resources and perform clean-up. An example of the `try-catch-finally` construct in C# can be seen below.

```
try
{
     file.ReadBlock(buffer, index, buffer.Length);
}
catch (System.IO.IOException e)
{
     Console.WriteLine("Error reading from {0}. Message = {1}",
          path, e.Message);
}
finally
{
     if (file != null)
     {
          file.Close();
     }
}
```

(Microsoft)

Since there are many different types of exceptions, languages with object-orientated capabilities typically implement an exception hierarchy in which all exceptions inherit from a general exception class. The programmer can easily, through inheritance, create their own exceptions from the most general, top-most exception or from one of its more specific children. Polymorphism is at play when specifying the type of exception to catch in the `catch` block and is important to keep in mind. When using multiple `catch` statements, the most specific

exceptions (lower in the hierarchy) should be listed first and then the most general (higher in the hierarchy) last.

When Exceptions Should be Thrown

Exceptions are typically thrown by the programmer when there is no better alternative to indicate that something went wrong or that the values given are invalid. An alternative to throwing an exception could be to simply return a bool, or `true` / `false`, value that the client can check. This works well for functions with a void or no return type such as setters or functions that write to a file or port. However, this is not always applicable for functions that return a value. Some languages allow the use of option types that get around this issue, though, and C# allows nullable types which could also work around this issue.

Another alternative is to return an invalid value that can be checked by the programmer. Since this value should only be returned when an issue is encountered, it is important that it is not a valid or possible value. A good example of this strategy would be with an index function. Should the value passed not be in the iterable object, the invalid index `-1` could be returned or even a `false` value as with Scheme. Unfortunately, this strategy fails when there is no invalid value that can be returned to signify an issue. For example, the programmer should not return `false` when popping from an empty stack as the stack could contain bool variables and the `false` value could be misunderstood to be a valid value.

A final alternative is to return or set an error code that can be reviewed by the client, and this is actually how error handling is done in C. In C, there is the `errno.h` header file that contains the `errno` (error number) variable that can be examined after a function call to determine if something went wrong. A typical use of it would be when working with files to

make sure opening, reading, writing, and other such activities succeeded. Another way to implement this idea is to have an argument that a function takes. This argument is then updated should a problem occur, and it may be passed by reference or just returned from the function. Unfortunately, it can be too easy for a lazy programmer to ignore these values, which could cause potential issues later on or even terminating errors. Throwing an exception forces the client to respond and perform any actions needed to continue.

Exceptions as Control Flow

Although it is typically frowned upon and even considered to be an anti-pattern by many, exceptions can be used for control flow. A small example below shows using exceptions to break out of a for loop in C++, and other similar uses include terminating a search once the solution has been found by throwing an exception.

```
try
{
     for (int i = 0; /* no test */ ; i++)
          array[i]++;
}
catch (ArrayIndexOutOfBoundsException e) {}
```

(Don't use exceptions for Flow Control)

Despite the fact that some languages, like Python, were designed with exceptions as control flow in mind, using exceptions in this way can make the code hard to read. Debugging may be made difficult as well as many debuggers are set up to pause execution whenever an exception is triggered, which makes telling a real exception from a control flow exception harder. An exception, when used as control flow, is essentially a non-local `goto`, and it can be confusing as to where control will resume. Furthermore, exceptions, depending on the language, can have overhead associated with them that can cause performance issues when used

excessively. To add to the performance issue, some compilers will produce far less efficient code when exceptions are used in this way. Finally, exceptions are meant for exceptional circumstances. If there is a better way to handle the control flow, such as checking the length of the array as in the above example, or to prevent the exception in the first place by validating the data or using a special function like C#'s `TryParse`, then that should be preferred.

Terminating Exceptions and Uncaught Exceptions

Although it will vary from language to language, there are some exceptions that, when thrown, can not be caught. Typically, these are for situations in which recovering from is impossible or simply too difficult. An example of a terminating exception would be a stack overflow. The program must terminate as there is no realistic way to recover without causing other issues in the program. It may be beneficial for the programmer to throw terminating exceptions or throw exceptions that they do not later catch in situations in which they are unable to deal with the problem in a reasonable manner. In such a case, it is important to collect enough information so the issue can be diagnosed and fixed.

An uncaught exception will be handled by the runtime and lead to the termination of the program, but the programmer has the ability in some languages to override the uncaught exception handler. It is of note that exceptions that are not caught in a multi-threaded application will typically terminate the thread of origin, but not the entire process.

Conclusion

Errors and exceptions: two of the most useful tools for a programmer, even though they may be difficult at times. A language that gives proper insights with its errors is

going to be invaluable for debugging code, and using a language with exceptions implemented allows the programmer to handle an even wider variety of errors manually. One of the oldest of the "modern" programming language features, error handling has become an integral part of any programming language. Exception handling has been around for much less time, but has quickly been adopted. Even across languages, the syntax for exception handling has converged on very similar keywords - often some variation of `try`, `catch`, and `throw`. While frowned upon at times, exceptions can even be used to control the flow of a program. At the end of the day, error and exception handling have become just another useful part of the modern programmer's toolbox.

Works Cited

Don't use exceptions for Flow Control. (1n.d.). Retrieved November 16, 2021, from http://c2.com/cgi-bin/wiki?DontUseExceptionsForFlowControl.

IBM. (2019, December 16). Which is better, a return code or an exception? IBM. Retrieved November 16, 2021, from https://www.ibm.com/support/pages/which-better-return-code-or-exception.

Microsoft. (2021, September 15). Try-catch-finally - C# reference. try-catch-finally - C# Reference | Microsoft Docs. Retrieved November 16, 2021, from https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch-finally.

Stack Exchange. (2012, August 18). Why are errors named as "exception" but not as "error" in programming languages? Software Engineering Stack Exchange. Retrieved November 16, 2021, from https://softwareengineering.stackexchange.com/questions/161488/why-are-errors-named-as-exception-but-not-as-error-in-programming-languages.

Stack Exchange. (2013, March 4). Are exceptions as control flow considered a serious antipattern? if so, why? Software Engineering Stack Exchange. Retrieved November 16, 2021, from https://softwareengineering.stackexchange.com/questions/189222/are-exceptions-as-control-flow-considered-a-serious-antipattern-if-so-why.

Wikimedia Foundation. (2021, April 23). Errno.h. Wikipedia. Retrieved November 16, 2021, from https://en.wikipedia.org/wiki/Errno.h.

Wikimedia Foundation. (2021, November 13). Defensive Programming. Wikipedia. Retrieved November 16, 2021, from https://en.wikipedia.org/wiki/Defensive_programming.

Wikimedia Foundation. (2021, November 15). Data validation. Wikipedia. Retrieved November 16, 2021, from https://en.wikipedia.org/wiki/Data_validation.

Wikimedia Foundation. (2021, October 16). Exception handling. Wikipedia. Retrieved November 16, 2021, from https://en.wikipedia.org/wiki/Exception_handling.