Survey Exception handling Approaches

Abstract

Modern software systems had to deal with abnormal situations more than before due to ever increasing demand for performance and sophistication. Exception handling is an indispensable feature in programming language to aid programmer in structuring program's failure scenarios. Inherently it's not natural for programmers to think through failure scenarios since I life he fights to stay away from all the time. So It becomes more important for a programming language to make it easy for him to get along. Good exception handling scenarios will definitely improve the readability, reliability, maintainability of the programs. Aim of this study is to explore evolution of exception handling mechanisms from the languages in the past, then compare recent developments in popular languages today and try to reason what influences them, how they impact programs. We would concentrate on five basic aspects of exception handling, exception signature, handler matching, control flow post exception, exception specification in particular. We would use this aspects as ground to discuss what influenced them and how they impact programs. Also we would pick a resource management problem in java and discuss how getting it out of the hands of the programmer had really made a difference [1]. We would conclude with comment on recent trends on exception handling, aspectizing exceptions and automatic exception handling.

Introduction

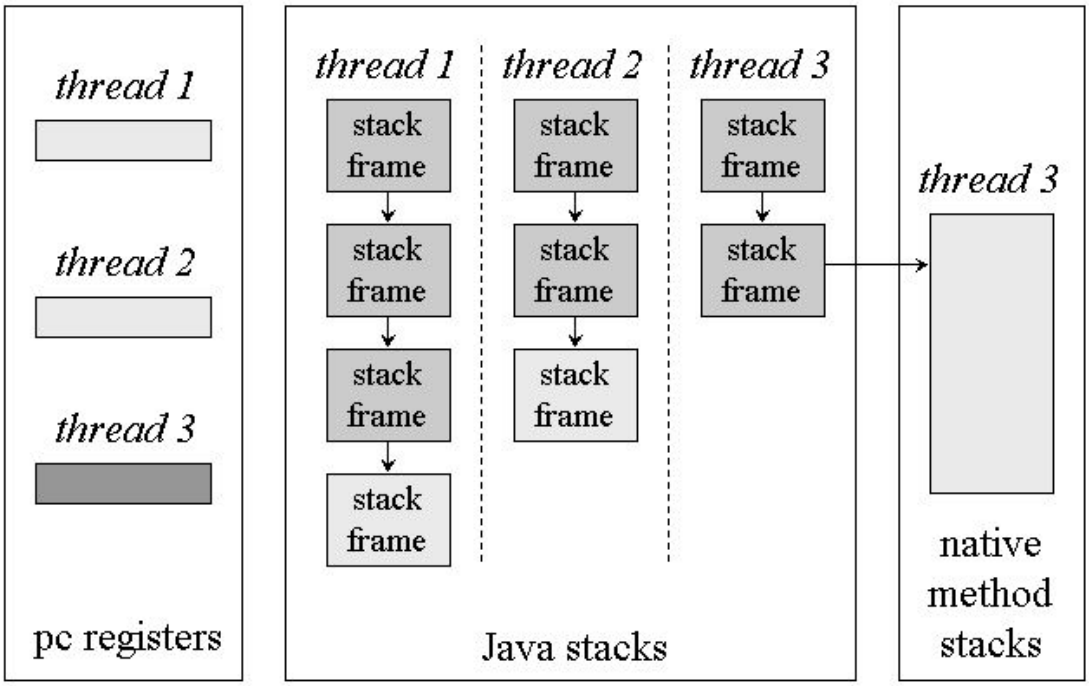
Programs get bigger and bigger day by day. May be we have found ways to see them smaller by breaking them down into smaller chunks to make it possible to comprehend, but it opened doors to difficulties in how these pieces work together, how they talk to each other. Naturally the mechanisms that programming languages had to offer to tame this takes prominence in building reliable programs for real world. In real world software programs, code around error handling and fault tolerance becomes messy, there is some truth in the fact that mentally humans are not adept dealing with more than one control path. This makes it interesting to see how exception handling evolved over the time period, how other dominating ideas in the programming language influences error handling mechanisms, how special situations like resource management, concurrency becomes much more interesting with exceptions.

Rest of the article will be structured into six sections,

1. Describe a simple exception execution model on the basis by which we can converse later on subtleties.
2. Broad classification of Exception model with relation to verification and paradigm influences on how exceptions handling mechanisms are modelled.
3. Survey famous & diverse language's exception handling mechanisms on some common aspects as ground.
4. Some concluding remarks.

1. Simple Exception Execution Model

Let's walk through how exception executions are modelled in a real language execution environment, so that it becomes easy to visualize the subtle differences in exception mechanism with respect to implementation. Java virtual machine internals has been used as an example due to its simplicity and popularity. Other languages also do similar implementations.



Simple JVM model to visualize the exception handling implementation flow.

JVM treats each program meant for execution as an independent thread and maps it to underlying operating system's thread definition. JVM maintains a method invocation stack containing all the methods that have been invoked by the current thread in the reverse order of invocation. In other words, the first method invoked by the thread is at the bottom of the stack and the current method is at the top. Actually it is not the actual method that is present in the stack. Instead a stack frame representing the method is added to the stack. The stack frame contains the method's parameters, return value, local variables and JVM specific information. When the exception is thrown in a method at the top of the stack, code execution stops and the JVM takes over.

The JVM searches the current method for a catch clause for the exception thrown or one of the parent classes of the thrown exception. If one is not found, then the JVM pops the current stack frame and inspects the calling method (the next method in the stack), for the catch clause for the exception or its parents. The process continues until the bottom of the stack is reached. In summary, it requires considerable time and effort on the part of JVM to simulate the appropriate control flow.

```
static int divide(int dividend, int divisor)
    throws DivideByZeroException {
    try {
        return dividend % divisor;
    }
    catch (ArithmeticException e) {
        throw new DivideByZeroException();
    }
}
```
Divide method

Java compiler generates the following bytecode sequence for the divide method:

```
The main bytecode sequence for remainder:
  0 iload_0             // Push local variable 0 (arg passed as divisor)
  1 iload_1             // Push local variable 1 (arg passed as dividend)
  2 irem                // Pop divisor, pop dividend, push remainder
  3 ireturn             // Return int on top of stack (the remainder)
The bytecode sequence for the catch (ArithmeticException) clause:
  4 pop                 // Pop the reference to the ArithmeticException
                        // because it isn't used by this catch clause.
  5 new #5 <Class DivideByZeroException>
                        // Create and push reference to new object of class
                        // DivideByZeroException.
DivideByZeroException
  8 dup           // Duplicate the reference to the new
                        // object on the top of the stack because it
                        // must be both initialized
                        // and thrown. The initialization will consume
                        // the copy of the reference created by the dup.
  9 invokenonvirtual #9 <Method DivideByZeroException.<init>()V>
                        // Call the constructor for the DivideByZeroException
                        // to initialize it. This instruction
                        // will pop the top reference to the object.
  12 athrow             // Pop the reference to a Throwable object, in this
```

```
                // case the DivideByZeroException,
                // and throw the exception.
```

The bytecode sequence of the remainder method has two separate parts. The first part is the normal path of execution for the method. This part goes from pc offset zero through three. The second part is the catch clause, which goes from pc offset four through twelve.
The *irem* instruction in the main bytecode sequence may throw an ArithmeticException. If this occurs, the JVM knows to jump to the bytecode sequence that implements the catch clause by looking up and finding the exception in a table. Each method that catches exceptions is associated with an exception table that is delivered in the class file along with the bytecode sequence of the method. The exception table has one entry for each exception that is caught by each try block. Each entry has four pieces of information: the start and end points, the pc offset within the bytecode sequence to jump to, and a constant pool index of the exception class that is being caught. The exception table for the remainder method looks like this,

from	to	target	type
0	4	4	Class java.lang.ArithmeticException

The above exception table indicates that from pc offset zero through three, inclusive, ArithmeticException is caught. The try block's endpoint value, listed in the table under the label "to", is always one more than the last pc offset for which the exception is caught. In this case the endpoint value is listed as four, but the last pc offset for which the exception is caught is three. This range, zero to three inclusive, corresponds to the bytecode sequence that implements the code inside the try block of remainder. The target listed in the table is the pc offset to jump to if an ArithmeticException is thrown between the pc offsets zero and three, inclusive.

If an exception is thrown during the execution of a method, the Java virtual machine searches through the exception table for a matching entry. An exception table entry matches if the current program counter is within the range specified by the entry, and if the exception class thrown is the exception class specified by the entry (or is a subclass of the specified exception class). The Java virtual machine searches through the exception table in the order in which the entries appear in the table. When the first match is found, the Java Virtual Machine sets the program counter to the new pc offset location and continues execution there. If no match is found, the Java virtual machine pops the current stack frame and rethrows the same exception. When the Java virtual machine pops the current stack frame, it effectively aborts execution of the current method and returns to the method that called this method. But instead of continuing execution normally in the previous method, it throws the same exception in that method, which causes the Java virtual machine to go through the same process of searching through the exception table of that method.

2. Broad Classification of Exception mechanism

If we try to classify exception mechanism choices from modern languages they fall under two broad terms. One being as a language feature control structure, in which control path is segregated from the actual good path and control jumps to the required code path when hit by faults and other being return values, in which . Most of the procedural languages use the formal mechanisms while the functional languages favour the later way. There is obvious reason in this choice, procedural language philosophy is all about managing the control flow. Naturally it favours dealing with exceptions as control flows. Historically if we look at the evolution of exception handling mechanisms they seem to have been inspired heavily from operating system fault tolerance mechanisms. As we move towards hardware languages need to be specific, procedural on what it need to accomplish.

On the other hand functional languages tend to treat problems in terms of pure functions, it paves way to think of faults as another input which the caller need to deal with. So when we explore more & more functional languages they tend not to build any language feature for dealing with faults but leave it to individual user program or library to model its exceptions in its convenience. More readable ones tend to get reused across applications and gain significance. There is some merit in this thinking as well, as control structures tends to get messy and not suitable in functional setting. Thinking behind such approach is, failure is another way of saying that the computation is partial, not defined for all values of arguments. For example, In a functional language like Haskell computation is always accomplished through full functions -- functions defined for all values of their arguments. so another option is to turn a partial function into a total function by changing its return type. For instance, C++ method find does this trick by returning an iterator (returns an iterator for *any* value of its argument); Haskell lookup does it by returning a Maybe.This trick of returning a different type in order to turn a non-functional computation into a pure function is used extensively in more & more functional languages like Haskell.

Exception as Control Structure [Java]	Error handling through Data Structure[Rust]

01: void read(String name){
02: try {
03: FileReader reader = new FileReader(name);
.....
04: }catch (IOException e) {
05: //do something clever
06: }
07: }
	1. fn double_number(number_str: &str) -> Result<i32, ParseIntError> {
	2.match number_str.parse::<i32>() {
	3. Ok(n) => Ok(2 * n),
	4. Err(err) => Err(err),
	5. }}
	6. fn main() {
	7. match double_number("10") {
	8. Ok(n) => assert_eq!(n, 20),
	9. Err(err) => println!("Error: {:?}", err),
	10. }}

Another thing that stands in the way of exceptions in functional languages is lazy evaluation. Having a function that executes with a different stack than the place it was queued to execute makes it difficult to determine where to fit in exception handlers.
Here's an example using a Future from [this Scala tutorial]:

```
val f: Future[List[String]] = future {
  session.getRecentPosts
}
f onFailure {
  case t => println("An error has occured: " + t.getMessage)
}
f onSuccess {
  case posts => for (post <- posts) println(post)
}
```

We can see that this has roughly the same structure as using explicit exceptions. The future block is like a try. The onFailure block is like an exception handler. In Scala, as in most functional languages, Future is implemented completely using the language itself. It doesn't require any special syntax like exceptions do. That means programmer can define their own similar constructs. Maybe add a timeout block, for example, or logging functionality. Additionally, we can also pass the future around, return it from the function, store it in a data structure, or whatever. It's a first-class value. We're not limited like exceptions which must be propagated straight up the stack.

Another interesting notion to highlight is the relation between the axiomatic semantics for a snippet and the abnormal situations. There is a direct relation between post and pre condition of a snippet to two categories of exceptional situations such as range errors and domain errors. Exceptions raised to the caller due to failures to meet pre condition are termed as range failures, example InvalidStateException, illegalArgumentException. Other kind are domain failures which can be related to invariant violation during one of its iteration or execution.

```
   Factorial
1. While X > 0 do
2. Y = X * Y
3. X = X – 1
4. Done
```

We can come up with exhaustive exceptional conditions for a snippet once we get the hoare rules for the snippet right. All preconditions inversion would amount to range faults and invariant inversion would amount to domain faults.

Preconditions { Y=1 ^ X=n ^ n >=0 }	Invariant I = (Y * X! = n! ^ X >=0)
Range Failures I. N < 0 II.Y != 1	Domain Failures Number Overflow Memory Overflow

3. Comparison of common nuances in exception handling

3a) Matching Exception handlers
In most languages the general trend is to match handlers by type, that's like we can handle all issues of particular type using a handler. However this approach provides no connection between the exception and the entity performing the operation that raises the exception. Java is one example where object responsible for raising exception is not really tied to the handler directly. This really handicaps the programmer in designing the handlers based on the entity raising it. In another object oriented language called Ada, exception handler matching has an option for the entity raising the exception.

Unbound Exception - Java	Bould Exception - Ada

LogFileType logFile;
FileType dataFile;
CsvFileType csvFile;
try {
  logFile.write();
  dataFile.write();
  csvFile.write();
} catch(FileError) {. . .}
	package dataFile is new FileType;
	package logFile is new LogFileType;
	begin
	  dataFile.write;
	  logFile.write;
	exception
	when dataFile.FileError =>
	when logFile.FileError =>

The ability to associate exceptions with objects strengthens the relationship between an exception and the object responsible for it.

3b) Control Flow - Resumption or Termination Model
Exception handling mechanisms in recent languages are typically non-resumable ("termination semantics") as opposed to hardware exceptions, which are typically resumable. This is based on experience of using both, as there are theoretical and design arguments in favor of either

decision. Using resumption semantics means that the exception handler is expected to do something to rectify the situation, and then the faulting code is automatically retried, presuming success the second time.
Historically, programmers using operating systems that supported resumptive exception handling eventually ended up using termination-like code and skipping resumption. Although resumption sounds attractive at first, it seems it isn't quite so useful in practice. Reason being the distance that can occur between the exception and its handler. It is one thing to terminate to a handler that's far away, but to jump to that handler and then back again may be too conceptually difficult for large systems where the exception is generated from many points.

All recent languages like java have termination model, whereas older languages like Cedar/Mesa has option for resumption semantics.

3c) Exception Specifications
Another interesting argument nowadays is whether it's a good thing to make exception handling compile time verifiable. Java does it while other more recent languages such as C# shy away from it. The argument in favour of it is that they are useful when programmer wants to force the user of the method to think how to handle the exceptional situation (if it is recoverable). It's just that checked exceptions are overused in the Java platform, which makes people hate them. It provides safety as quick as compilation phase which is quite effective.
Recent arguments against it are that most of the time calle cannot decide its importance to its caller. Forcing the user to do something most probably will annoy him which leads him to get away with it using empty handlers. There is some truth in the recent developments against checking exceptions, mostly important thing in exception handling is cleaning up the resources which are not needed after termination, one way or the other we end up doing nothing than informing the user that we can't do it.

3d) Exception Propagation

All languages agree in this aspect, how should we propagate the exception, through the call hierarchy. Look up for handlers up the chain of invokers. A problem occurs where exceptions have scope; an exception may be propagated outside its scope, thereby making it impossible for a handler to be found. An unhandled exception causes a sequential program to be aborted. If the program contains more than one process and a particular process does not handle an exception it has raised, then usually that process is aborted.

3e) Granularity of domain

Another aspect to consider is the granularity of the block which can be guarded with exception check. By expanding the domain to block, handler cannot make sense out of the issue to rectify. Most languages overcome this by providing option to enrich the exception type being shared with handler, like in java it's an object so that it can carry additional information required for the handler to deal with the situation.

Summary

Language	Domain	Propagation	Model	Parameters
Java	Block	Yes	Termination	Yes
C++	Block	Yes	Termination	Yes
Mesa	Block	Yes	Hybrid	Yes
Ada	Block	Yes	Termination	Yes
CHILL	Statement	No	Termination	No

Conclusion:

Exception handling mechanisms are becoming important features of programming languages.They provide support at the programming language level for structuring fault tolerance activities. We have reviewed and evaluated the exception handling models used in five existing exception mechanisms for varied languages. It is not unanimously accepted that exception handling facilities should be provided in a language. Language features and their corresponding mechanisms for exception handling continue to evolve in both experimental and commercial languages.

References:

Why Do Developers Neglect Exception Handling? - http://www.cc.gatech.edu/grads/h/hinashah/documents/WEH-shah.pdf
Exception handling - issues and proposed notation - https://www.cs.virginia.edu/~weimer/615/reading/goodenough-exceptions.pdf
Exception Safety: Concepts and Techniques - http://www.stroustrup.com/except.pdf
Unchecked Exceptions: Can the Programmer be Trusted to Document Exceptions? - https://eden.dei.uc.pt/~bcabral/ivnet06.pdf
Implementing zero overhead Exception handling - https://raptor.cs.arizona.edu/~collberg/Teaching/553/2011/Resources/exceptions-drew.pdf
http://blog.jamesdbloom.com/JVMInternals.html
http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/