Exception handling

From Wikipedia, the free encyclopedia

Exception handling is the process of responding to the occurrence, during computation, of *exceptions* – anomalous or exceptional conditions requiring special processing – often changing the normal flow of program execution. It is provided by specialized programming language constructs or computer hardware mechanisms.

In general, an exception is *handled* (resolved) by saving the current state of execution in a predefined place and switching the execution to a specific subroutine known as an *exception handler*. If exceptions are *continuable*, the handler may later resume the execution at the original location using the saved information. For example, a floating point divide by zero exception will typically, by default, allow the program to be resumed, while an out of memory condition might not be resolvable transparently.

Alternative approaches to exception handling in software are error checking, which maintains normal program flow with later explicit checks for contingencies reported using special return values or some auxiliary global variable such as C's `errno` or floating point status flags; or input validation to preemptively filter exceptional cases.

Contents

Exception handling in hardware

Hardware exception mechanisms are processed by the CPU. It is intended to support error detection and redirects the program flow to error handling service routines. The state before the exception is saved on the stack.[1]

Hardware exception handling/traps: IEEE 754 floating point

Exception handling in the IEEE 754 floating point hardware standard refers in general to exceptional conditions and defines an exception as "an event that occurs when an operation on some particular operands has no outcome suitable for every reasonable application. That operation might signal one or more exceptions by invoking the default or, if explicitly requested, a language-defined alternate handling."

By default, an IEEE 754 exception is resumable and is handled by substituting a predefined value for different exceptions, e.g. infinity for a divide by zero exception, and providing status flags for later checking of whether the exception occurred (see C99 programming language for a typical example of handling of IEEE 754 exceptions). An exception-handling style enabled by the use of status flags involves: first computing an expression using a fast, direct implementation; checking whether it failed by testing status flags; and then, if necessary, calling a slower, more numerically robust, implementation.[2]

The IEEE 754 standard uses the term "trapping" to refer to the calling of a user-supplied exception-handling routine on exceptional conditions, and is an optional feature of the standard. The standard recommends several usage scenarios for this, including the implementation of non-default pre-substitution of a value followed by resumption, to concisely handle removable singularities.[2][3][4]

The default IEEE 754 exception handling behaviour of resumption following pre-substitution of a default value avoids the risks inherent in changing flow of program control on numerical exceptions. For example, in 1996 the maiden flight of the Ariane 5 (Flight 501) ended in a catastrophic explosion due in part to the Ada programming language exception handling policy of aborting computation on arithmetic error, which in this case was a 64-bit floating point to 16-bit integer conversion overflow.[3] In the Ariane Flight 501 case, the programmers protected only four out of seven critical variables against overflow due to concerns about the computational constraints of the on-board computer and relied on what turned out to be incorrect assumptions about the possible range of values for the three unprotected variables because they reused code from the Ariane 4, for which their assumptions were correct.[5] According to William Kahan, the loss of Flight 501 would have been avoided if the IEEE 754 exception-handling policy of default substitution had been used because the overflowing 64-bit to 16-bit conversion that caused the software to abort occurred in a piece of code that turned out to be completely unnecessary on the Ariane 5.[3] The official report on the crash (conducted by an inquiry board headed by Jacques-Louis Lions) noted that "An underlying theme in the development of Ariane 5 is the bias towards the mitigation of random failure. The supplier of the SRI was only following the specification given to it, which stipulated that in the event of any detected exception the processor was to be stopped. The exception which occurred was not due to random failure but a design error. The exception was detected, but inappropriately handled because the view had been taken that software should be considered correct until it is shown to be at fault. [...] Although the failure was due to a systematic software design error, mechanisms can be introduced to mitigate this type of problem. For example the computers within the SRIs could have continued to provide their best estimates of the required attitude information. There is reason for concern that a software exception should be allowed, or even required, to cause a processor to halt while handling mission-critical equipment. Indeed, the loss of a proper software function is hazardous because the same software runs in both SRI units. In the case of Ariane 501, this resulted in the switch-off of two still healthy critical units of equipment."[6]

From the processing point of view, hardware interrupts are similar to resumable exceptions, though they are typically unrelated to the user program's control flow.

Exception handling in software

Software exception handling and the support provided by software tools differs somewhat from what is understood under exception in hardware, but similar concepts are involved. In programming language mechanisms for exception handling, the term *exception* is typically used in a specific sense to denote a data structure storing information about an exceptional condition. One mechanism to transfer control, or *raise* an exception, is known as a *throw*. The exception is said to be *thrown*. Execution is transferred to a "catch".

From the point of view of the author of a routine, raising an exception is a useful way to signal that a routine could not execute normally - for example, when an input argument is invalid (e.g. value is outside of the domain of a function) or when a resource it relies on is unavailable (like a missing file, a hard disk error, or out-of-memory errors). In systems without exceptions, routines would need to return some special error code. However, this is sometimes complicated by the semipredicate problem, in which users of the routine need to write extra code to distinguish normal return values from erroneous ones.

According to a 2006 comparative paper by Joseph R. Kiniry, programming languages differ substantially in their notion of what is an exception. According to Kiniry, the contemporary languages can be roughly divided in two groups:[7]

- those languages in which exceptions "are designed to be used as flow control structures"; according to this paper, Ada, C++, Java, Modula-3, ML, OCaml, Python, and Ruby fall in this category
- those languages in which exceptions "are designed to represent and handle abnormal, unpredictable, erroneous situations"; according to the paper these include: C#, Common Lisp, Eiffel, and Modula-2

Kiniry also notes that "Language design only partially influences the use of exceptions, and consequently, the manner in which one handles partial and total failures during system execution. The other major influence is examples of use, typically in core libraries and code examples in technical books, magazine articles, and online discussion forums, and in an organization's code standards."[7]

Contemporary applications face many design challenges when considering exception handling strategies. Particularly in modern enterprise level applications, exceptions must often cross process boundaries and machine boundaries. Part of designing a solid exception handling strategy is recognizing when a process has failed to the point where it cannot be economically handled by the software portion of the process.[8]

Exception handling is often not handled correctly in software, especially when there are multiple sources of exceptions; data flow analysis of 5 million lines of Java code found over 1300 exception handling defects.[9]

History

Software exception handling developed in Lisp in the 1960s and 1970s. This originated in LISP 1.5 (1962), where exceptions were *caught* by the `ERRSET` keyword, which returned `NIL` in case of an error, instead of terminating the program or entering the debugger.[10] Error *raising* was introduced in MacLisp in the late 1960s via the `ERR` keyword.[10] This was rapidly used not only for error raising, but for non-local control flow, and thus was augmented by two new keywords, `CATCH` and `THROW` (MacLisp June

1972), reserving `ERRSET` and `ERR` for error handling. The cleanup behavior now generally called "finally" was introduced in NIL (New Implementation of LISP) in the mid- to late-1970s as `UNWIND-PROTECT`.[11] This was then adopted by Common Lisp. Contemporary with this was `dynamic-wind` in Scheme, which handled exceptions in closures. The first papers on structured exception handling were Goodenough (1975a) and Goodenough (1975b).[12] Exception handling was subsequently widely adopted by many programming languages from the 1980s onward.

Originally software exception handling included both resumable exceptions (resumption semantics), like most hardware exceptions, and non-resumable exceptions (termination semantics). However, resumption semantics proved ineffective in practice in the 1970s and 1980s, and are no longer in common use.

Termination semantics

Exception handling mechanisms in contemporary 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; these were extensively debated during C++ standardization discussions 1989–1991, which resulted in a definitive decision for termination semantics.[13] On the rationale for such a design for the C++ mechanism, Stroustrup notes:

> [A]t the Palo Alto [C++ standardization] meeting in November 1991, we heard a brilliant summary of the arguments for termination semantics backed with both personal experience and data from Jim Mitchell (from Sun, formerly from Xerox PARC). Jim had used exception handling in half a dozen languages over a period of 20 years and was an early proponent of resumption semantics as one of the main designers and implementers of Xerox's Cedar/Mesa system. His message was
>
>> “termination is preferred over resumption; this is not a matter of opinion but a matter of years of experience. Resumption is seductive, but not valid.”
>
> He backed this statement with experience from several operating systems. The key example was Cedar/Mesa: It was written by people who liked and used resumption, but after ten years of use, there was only one use of resumption left in the half million line system – and that was a context inquiry. Because resumption wasn't actually necessary for such a context inquiry, they removed it and found a significant speed increase in that part of the system. In each and every case where resumption had been used it had – over the ten years – become a problem and a more appropriate design had replaced it. Basically, every use of resumption had represented a failure to keep separate levels of abstraction disjoint.[12]

Criticism

A contrasting view on the safety of exception handling was given by C.A.R Hoare in 1980, described the Ada programming language as having "...a plethora of features and notational conventions, many of them unnecessary and some of them, like exception handling, even dangerous. [...] Do not allow this language in its present state to be used in applications where reliability is critical[...]. The next rocket to go astray as a result of a programming language error may not be an exploratory space rocket on a harmless trip to Venus: It may be a nuclear warhead exploding over one of our own cities." [14]

Citing multiple prior studies by others (1999–2004) and their own results, Weimer and Necula wrote that a significant problem with exceptions is that they "create hidden control-flow paths that are difficult for programmers to reason about".[9]:8:27

Exception support in programming languages

Many computer languages have built-in support for exceptions and exception handling. This includes ActionScript, Ada, BlitzMax, C++, C#, D, ECMAScript, Eiffel, Java, ML, Object Pascal (e.g. Delphi, Free Pascal, and the like), PowerBuilder, Objective-C, OCaml, PHP (as of version 5), PL/1, PL/SQL, Prolog, Python, REALbasic, Ruby, Scala, Seed7, Tcl, Visual Prolog and most .NET languages. Exception handling is commonly not resumable in those languages, and when an exception is thrown, the program searches back through the stack of function calls until an exception handler is found.

Some languages call for unwinding the stack as this search progresses. That is, if function `f`, containing a handler `H` for exception `E`, calls function `g`, which in turn calls function `h`, and an exception `E` occurs in `h`, then functions `h` and `g` may be terminated, and `H` in `f` will handle `E`.

An exception-handling language without this unwinding is Common Lisp with its Condition System. Common Lisp calls the exception handler and does not unwind the stack. This allows the program to continue the computation at exactly the same place where the error occurred (for example when a previously missing file has become available). The stackless implementation of the *Mythryl* programming language supports constant-time exception handling without stack unwinding.

Excluding minor syntactic differences, there are only a couple of exception handling styles in use. In the most popular style, an exception is initiated by a special statement (`throw` or `raise`) with an exception object (e.g. with Java or Object Pascal) or a value of a special extendable enumerated type (e.g. with Ada). The scope for exception handlers starts with a marker clause (`try` or the language's block starter such as `begin`) and ends in the start of the first handler clause (`catch`, `except`, `rescue`). Several handler clauses can follow, and each can specify which exception types it handles and what name it uses for the exception object.

A few languages also permit a clause (`else`) that is used in case no exception occurred before the end of the handler's scope was reached.

More common is a related clause (`finally` or `ensure`) that is executed whether an exception occurred or not, typically to release resources acquired within the body of the exception-handling block. Notably, C++ does not provide this construct, since it encourages the Resource Acquisition Is Initialization (RAII) technique which frees resources using destructors.

In its whole, exception handling code might look like this (in Java-like pseudocode; note that an exception type called `EmptyLineException` would need to be declared somewhere):

```
try {
    line = console.readLine();

    if (line.length() == 0) {
        throw new EmptyLineException("The line read from console was empty!");
    }

    console.printLine("Hello %s!" % line);
    console.printLine("The program ran successfully");
}
catch (EmptyLineException e) {
    console.printLine("Hello!");
}
catch (Exception e) {
    console.printLine("Error: " + e.message());
}
finally {
    console.printLine("The program terminates now");
}
```

As a minor variation, some languages use a single handler clause, which deals with the class of the exception internally.

According to a 2008 paper by Westley Wiemer and George Necula, the syntax of the `try...finally` blocks in Java is a contributing factor to software defects. When a method needs to handle the acquisition and release of 3–5 resources, programmers are apparently unwilling to nest enough blocks due to readability concerns, even when this would be a correct solution. It is possible to use a single `try...finally` block even when dealing with multiple resources, but that requires a correct use of sentinel values, which is another common source of bugs for this type of problem.[9]:8:6–8:7 Regarding the semantics of the `try...catch...finally` construct in general, Wiemer and Necula write that "While try-catch-finally is conceptually simple, it has the most complicated execution description in the language specification [Gosling et al. 1996] and requires four levels of nested “if”s in its official English description. In short, it contains a large number of corner cases that programmers often overlook."[9]:8:13–8:14

C supports various means of error checking, but generally is not considered to support "exception handling." Perl has optional support for structured exception handling.

By contrast Python's support for exception handling is pervasive and consistent. It's difficult to write a robust Python program without using its `try` and `except` keywords.

Exception handling implementation

The implementation of exception handling in programming languages typically involves a fair amount of support from both a code generator and the runtime system accompanying a compiler. (It was the addition of exception handling to C++ that ended the useful lifetime of the original C++ compiler, Cfront.[15]) Two schemes are most common. The first, *dynamic registration*, generates code that continually updates structures about the program state in terms of exception handling.[16] Typically, this adds a new element to the stack frame layout that knows what handlers are available for the function or method associated with that frame; if an exception is thrown, a pointer in the layout directs the runtime to the appropriate handler code. This approach is compact in terms of space, but adds execution overhead on frame entry and exit. It was commonly used in many Ada implementations, for example, where complex generation and runtime support was already needed for many other language features. Dynamic registration, being fairly straightforward to define, is amenable to proof of correctness.[17]

The second scheme, and the one implemented in many production-quality C++ compilers, is a *table-driven* approach. This creates static tables at compile time and link time that relate ranges of the program counter to the program state with respect to exception handling.[18] Then, if an exception is thrown, the runtime system looks up the current instruction location in the tables and determines what handlers are in play and what needs to be done. This approach minimizes executive overhead for the case where an exception is not thrown. This happens at the cost of some space, but this space can be allocated into read-only, special-purpose data sections that are not loaded or relocated until an exception is actually thrown.[19] This second approach is also superior in terms of achieving thread safety.

Other definitional and implementation schemes have been proposed as well.[20] For languages that support metaprogramming, approaches that involve no overhead at all have been advanced.[21]

Exception handling based on design by contract

A different view of exceptions is based on the principles of design by contract and is supported in particular by the Eiffel language. The idea is to provide a more rigorous basis for exception handling by defining precisely what is "normal" and "abnormal" behavior. Specifically, the approach is based on two concepts:

- **Failure**: the inability of an operation to fulfill its contract. For example an addition may produce an arithmetic overflow (it does not fulfill its contract of computing a good approximation to the mathematical sum); or a routine may fail to meet its postcondition.
- **Exception**: an abnormal event occurring during the execution of a routine (that routine is the "*recipient*" of the exception) during its execution. Such an abnormal event results from the *failure* of an operation called by the routine.

The "Safe Exception Handling principle" as introduced by Bertrand Meyer in Object-Oriented Software Construction then holds that there are only two meaningful ways a routine can react when an exception occurs:

- Failure, or "organized panic": The routine fixes the object's state by re-establishing the invariant (this is the "organized" part), and then fails (panics), triggering an exception in its caller (so that the abnormal event is not ignored).
- Retry: The routine tries the algorithm again, usually after changing some values so that the next attempt will have a better chance to succeed.

In particular, simply ignoring an exception is not permitted; a block must either be retried and successfully complete, or propagate the exception to its caller.

Here is an example expressed in Eiffel syntax. It assumes that a routine `send_fast` is normally the better way to send a message, but it may fail, triggering an exception; if so, the algorithm next uses `send_slow`, which will fail less often. If `send_slow` fails, the routine `send` as a whole should fail, causing the caller to get an exception.

```
send (m: MESSAGE) is
  -- Send m through fast link, if possible, otherwise through slow link.
local
  tried_fast, tried_slow: BOOLEAN
do
  if tried_fast then
     tried_slow := True
     send_slow (m)
  else
     tried_fast := True
     send_fast (m)
  end
rescue
  if not tried_slow then
     retry
  end
end
```

The boolean local variables are initialized to False at the start. If `send_fast` fails, the body (`do` clause) will be executed again, causing execution of `send_slow`. If this execution of `send_slow` fails, the `rescue` clause will execute to the end with no `retry` (no `else` clause in the final `if`), causing the routine execution as a whole to fail.

This approach has the merit of defining clearly what "normal" and "abnormal" cases are: an abnormal case, causing an exception, is one in which the routine is unable to fulfill its contract. It defines a clear distribution of roles: the `do` clause (normal body) is in charge of achieving, or attempting to achieve, the routine's contract; the `rescue` clause is in charge of reestablishing the context and restarting the process, if this has a chance of succeeding, but not of performing any actual computation.

Although exceptions in Eiffel have a fairly clear philosophy, Kiniry (2006) criticizes their implementation because "Exceptions that are part of the language definition are represented by INTEGER values, developer-defined exceptions by STRING values. [...] Additionally, because they are basic values and not objects, they have no inherent semantics beyond that which is expressed in a helper routine which necessarily cannot be foolproof because of the representation overloading in effect (e.g., one cannot differentiate two integers of the same value)."[7]

Static checking of exceptions

Checked exceptions

The designers of Java devised[22][23] checked exceptions,[24] which are a special set of exceptions. The checked exceptions that a method may raise are part of the method's signature. For instance, if a method might throw an `IOException`, it must declare this fact explicitly in its method signature. Failure to do so raises a compile-time error.

Kiniry (2006) notes however that Java's libraries (as they were in 2006) were often inconsistent in their approach to error reporting, because "Not all erroneous situations in Java are represented by exceptions though. Many methods return special values which indicate failure encoded as constant field of related classes."[7]

Checked exceptions are related to exception checkers that exist for the OCaml programming language.[25] The external tool for OCaml is both invisible (i.e. it does not require any syntactic annotations) and optional (i.e. it is possible to compile and run a program without having checked the exceptions, although this is not recommended for production code).

The CLU programming language had a feature with the interface closer to what Java has introduced later. A function could raise only exceptions listed in its type, but any leaking exceptions from called functions would automatically be turned into the sole runtime exception, `failure`, instead of resulting in compile-time error. Later, Modula-3 had a similar feature.[26] These features don't include the compile time checking that is central in the concept of checked exceptions, and hasn't (as of 2006) been incorporated into major programming languages other than Java.[27]

The C++ programming language introduces an optional mechanism for checked exceptions, called *exception specifications*. By default any function can throw any exception, but this can be limited by a `throw` clause added to the function signature, that specifies which exceptions the function may throw. Exception specifications are not enforced at compile-time. Violations result in the global function `std::unexpected` being called.[28] An empty exception specification may be given, which indicates that the function will throw no exception. This was not made the default when exception handling was added to the language because it would require too much modification of existing code, would impede interaction with code written in another language, and would tempt programmers into writing too many handlers at the local level.[28] Explicit use of empty exception specifications can, however, allow C++ compilers to perform significant code and stack layout optimizations that generally have to be suppressed when exception handling may take place in a function.[19] Some analysts view the proper use of exception specifications in C++ as difficult to achieve.[29] In the recent C++ language standard (C++11), the use of exception specifications as specified in the preceding version of the standard (C++03) is deprecated.[30]

In contrast to Java, languages like C# do not enforce that exceptions have to be caught. According to Hanspeter Mössenböck, not distinguishing between to-be-called (checked) exceptions and not-to-be-called (unchecked) exceptions makes the written program more convenient, but less robust, as an uncaught exception results in an abort with a stack trace.[31] Kiniry (2006) notes however that Java's JDK (version 1.4.1) throws a large number of unchecked exceptions: one for every 140 lines of code, whereas Eiffel uses them much more sparingly, with one thrown every 4,600 lines of code. Kiniry also writes that "As any Java programmer knows, the volume of try/catch code in a typical Java application is sometimes larger than the comparable code necessary for explicit formal parameter and return value checking in other languages that do not have checked exceptions. In fact, the general consensus among in-the-trenches Java programmers is that dealing with checked exceptions is nearly as unpleasant a task as writing documentation. Thus, many programmers report that they “resent” checked exceptions. This leads to an abundance of checked-but-ignored exceptions".[7] Kiniry also notes that the developers of C# apparently were influenced by this kind of user experiences, with the following quote being attributed to them (via Eric Gunnerson): "Examination of small programs leads to the conclusion that requiring exception specifications could both enhance developer productivity and enhance code quality, but experience with large software projects suggests a different result – decreased productivity and little or no increase in code quality."[7] According to Anders Hejlsberg there was fairly broad agreement in their design group to not have checked exceptions as a language feature in C#. Hejlsberg explained in an interview that “The throws clause, at least the way it's implemented in Java, doesn't necessarily force you to handle the exceptions, but if you don't handle them, it forces you to acknowledge precisely which exceptions might pass through. It requires you to either catch declared exceptions or put them in your own throws clause. To work around this requirement, people do ridiculous things. For example, they decorate every method with, "throws Exception." That just completely defeats the feature, and you just made the programmer write more gobbledy gunk. That doesn't help anybody.”[32]

Views on usage

Checked exceptions can, at compile time, reduce the incidence of unhandled exceptions surfacing at runtime in a given application. Unchecked exceptions (such as the Java objects `RuntimeException` and `Error`) remain unhandled.

However, checked exceptions can either require extensive **`throws`** declarations, revealing implementation details and reducing encapsulation, or encourage coding poorly considered **`try`**/**`catch`** blocks that can hide legitimate exceptions from their appropriate handlers. Consider a growing codebase over time. An interface may be declared to throw exceptions X & Y. In a later version of the code, if one wants to throw exception Z, it would make the new code incompatible with the earlier uses. Furthermore, with the adapter pattern, where one body of code declares an interface that is then implemented by a different body of code so that code can be plugged in and called by the first, the adapter code may have a rich set of exceptions to describe problems, but is forced to use the exception types declared in the interface.

It is possible to reduce the number of declared exceptions either by declaring a superclass of all potentially thrown exceptions, or by defining and declaring exception types that are suitable for the level of abstraction of the called method[33] and mapping lower level exceptions to these types, preferably wrapped using exception chaining in order to preserve the root cause. In addition, it's very possible that in the example above of the changing interface that the calling code would need to be modified as well, since in some sense the exceptions a method may throw are part of the method's implicit interface anyway.

Using a `throws Exception` declaration or `catch (Exception e)` is usually sufficient for satisfying the checking in Java. While this may have some use, it essentially circumvents the checked exception mechanism, which Oracle discourages.[34]

Unchecked exception types should generally not be handled, except possibly at the outermost levels of scope. These often represent scenarios that do not allow for recovery: `RuntimeException`s frequently reflect programming defects,[35] and `Error`s generally represent unrecoverable JVM failures. Even in a language that supports checked exceptions, there are cases where the use of checked exceptions is not appropriate.[36]

However, it is important to note that in multi-threaded applications running as services, if the standard error stream is redirected to `/dev/null` or otherwise unobservable, the server and/or the application(s) are responsible for catching and logging exceptions in threads where appropriate, at or close to the top level. Java provides the `Thread.setUncaughtExceptionHandler` method to set a very last resort exception handler, for situations where nothing in the thread itself catches an exception, or the top-level exception handler in the thread itself throws an exception.

Dynamic checking of exceptions

The point of exception handling routines is to ensure that the code can handle error conditions. In order to establish that exception handling routines are sufficiently robust, it is necessary to present the code with a wide spectrum of invalid or unexpected inputs, such as can be created via software fault injection and mutation testing (that is also sometimes referred to as fuzz testing). One of the most difficult types of software for which to write exception handling routines is protocol software, since a robust protocol implementation must be prepared to receive input that does not comply with the relevant specification(s).

In order to ensure that meaningful regression analysis can be conducted throughout a software development lifecycle process, any exception handling testing should be highly automated, and the test cases must be generated in a scientific, repeatable fashion. Several commercially available systems exist that perform such testing.

In runtime engine environments such as Java or .NET, there exist tools that attach to the runtime engine and every time that an exception of interest occurs, they record debugging information that existed in memory at the time the exception was thrown (call stack and heap values). These tools are called automated exception handling or error interception tools and provide 'root-cause' information for exceptions.

Exception synchronicity

Somewhat related with the concept of checked exceptions is *exception synchronicity*. Synchronous exceptions happen at a specific program statement whereas **asynchronous exceptions** can raise practically anywhere.[37][38] It follows that asynchronous exception handling can't be required by the compiler. They are also difficult to program with. Examples of naturally asynchronous events include pressing Ctrl-C to interrupt a program, and receiving a signal such as "stop" or "suspend" from another thread of execution.

Programming languages typically deal with this by limiting asynchronicity, for example Java has deprecated the use of its ThreadDeath exception that was used to allow one thread to stop another one.[39] Instead, there can be semi-asynchronous exceptions that only raise in suitable locations of the program or synchronously.

Condition systems

Common Lisp, Dylan and Smalltalk have a condition system[40] (see Common Lisp Condition System) that encompasses the aforementioned exception handling systems. In those languages or environments the advent of a condition (a "generalisation of an error" according to Kent Pitman) implies a function call, and only late in the exception handler the decision to unwind the stack may be taken.

Conditions are a generalization of exceptions. When a condition arises, an appropriate condition handler is searched for and selected, in stack order, to handle the condition. Conditions that do not represent errors may safely go unhandled entirely; their only purpose may be to propagate hints or warnings toward the user.[41]

Continuable exceptions

This is related to the so-called *resumption model* of exception handling, in which some exceptions are said to be *continuable*: it is permitted to return to the expression that signaled an exception, after having taken corrective action in the handler. The condition system is generalized thus: within the handler of a non-serious condition (a.k.a. *continuable exception*), it is possible to jump to predefined restart points (a.k.a. *restarts*) that lie between the signaling expression and the condition handler. Restarts are functions closed over some lexical environment, allowing the programmer to repair this environment before exiting the condition handler completely or unwinding the stack even partially.

Restarts separate mechanism from policy

Condition handling moreover provides a separation of mechanism from policy. Restarts provide various possible mechanisms for recovering from error, but do not select which mechanism is appropriate in a given situation. That is the province of the condition handler, which (since it is located in higher-level code) has access to a broader view.

An example: Suppose there is a library function whose purpose is to parse a single syslog file entry. What should this function do if the entry is malformed? There is no one right answer, because the same library could be deployed in programs for many different purposes. In an interactive log-file browser, the right thing to do might be to return the entry unparsed, so the user can see it—but in an automated log-summarizing program, the right thing to do might be to supply null values for the unreadable fields, but abort with an error, if too many entries have been malformed.

That is to say, the question can only be answered in terms of the broader goals of the program, which are not known to the general-purpose library function. Nonetheless, exiting with an error message is only rarely the right answer. So instead of simply exiting with an error, the function may *establish restarts* offering various ways to continue—for instance, to skip the log entry, to supply default or null values for the unreadable fields, to ask the user for the missing values, *or* to unwind the stack and abort processing with an error message. The restarts offered constitute the *mechanisms* available for recovering from error; the selection of restart by the condition handler supplies the *policy*.

See also

- Exception handling syntax
- Automated exception handling
- Exception safety
- Continuation
- Defensive programming
- setjmp/longjmp
- Triple fault
- Vectored Exception Handling (VEH)

References

1. "Hardware Exceptions Detection" (http://processors.wiki.ti.com/index.php/Hardware_Exceptions_Detection). TEXAS INSTRUMENTS. 2011-11-24. Archived from the original (http://processors.wiki.ti.com/) on 2011-11-24. Retrieved 2012-10-05.
2. Xiaoye Li and James Demmel (1994). "Faster Numerical Algorithms via Exception Handling, IEEE Transactions on Computers, 43(8)". pp. 983–992.
3. W.Kahan (July 5, 2005). "A Demonstration of Presubstitution for ∞/∞" (http://www.cs.berkeley.edu/~wkahan/Grail.pdf) (PDF).
4. John Hauser (1996). "Handling Floating-Point Exceptions in Numeric Programs, ACM Transactions on Programming Languages and Systems 18(2)". pp. 139–174.
5. http://www.irisa.fr/pampa/EPEE/Ariane5.html
6. https://www.ima.umn.edu/~arnold/disasters/ariane5rep.html
7. Kiniry, J. R. (2006). "Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application". *Advanced Topics in Exception Handling Techniques*. Lecture Notes in Computer Science **4119**. p. 288. doi:10.1007/11818502_16 (https://dx.doi.org/10.1007%2F11818502_16). ISBN 978-3-540-37443-5.
8. All Exceptions Are Handled, Jim Wilcox, http://granitestatehacker.kataire.com/2008/02/all-exceptions-are-handled.html
9. Weimer, W and Necula, G.C. (2008). "Exceptional Situations and Program Reliability" (http://www.cs.virginia.edu/~weimer/p/weimer-toplas2008.pdf) (PDF). *ACM Transactions on Programming Languages and Systems, vol 30 (2)*.
10. Gabriel & Steele 2008, p. 3.
11. White 1979, p. 194.
12. Stroustrup 1994, p. 392.
13. Stroustrup 1994, 16.6 Exception Handling: Resumption vs. Termination, pp. 390–393.
14. C.A.R. Hoare. "The Emperor's Old Clothes". 1980 Turing Award Lecture
15. Scott Meyers, The Most Important C++ Software...Ever (http://www.artima.com/cppsource/top_cpp_software.html), 2006
16. D. Cameron, P. Faust, D. Lenkov, M. Mehta, "A portable implementation of C++ exception handling", *Proceedings of the C++ Conference* (August 1992) USENIX.
17. Graham Hutton, Joel Wright, "Compiling Exceptions Correctly (http://www.cs.nott.ac.uk/~gmh/exceptions.pdf)". *Proceedings of the 7th International Conference on Mathematics of Program Construction*, 2004.
18. Lajoie, Josée (March–April 1994). "Exception handling – Supporting the runtime mechanism". *C++ Report* **6** (3).
19. Schilling, Jonathan L. (August 1998). "Optimizing away C++ exception handling". *SIGPLAN Notices* **33** (8): 40–47. doi:10.1145/286385.286390 (https://dx.doi.org/10.1145%2F286385.286390).
20. "[1] (http://software.intel.com/en-us/articles/how-to-implement-software-exception-handling/)", Intel Corporation.

21. M. Hof, H. Mössenböck, P. Pirkelbauer, "Zero-Overhead Exception Handling Using Metaprogramming (http://www.ssw.uni-linz.ac.at/Research/Papers/Hof97b.html)", *Proceedings SOFSEM'97*, November 1997, *Lecture Notes in Computer Science 1338*, pp. 423-431.
22. Re: Toward a more "automatic" RMI = compatible with basic RMI philosophy (http://archives.java.sun.com/cgi-bin/wa?A2=ind9901&L=rmi-users&F=P&P=36083), Ann Wollrath (JavaSoft East). A post on the RMI-USERS mailing list, 22 January 1999.
23. "Google Answers: The origin of checked exceptions" (http://answers.google.com/answers/threadview?id=26101). Answers.google.com. Retrieved 2011-12-15.
24. Java Language Specification, chapter 11.2. http://java.sun.com/docs/books/jls/third_edition/html/exceptions.html#11.2
25. "OcamlExc - An uncaught exceptions analyzer for Objective Caml" (http://caml.inria.fr/pub/old_caml_site/ocamlexc/ocamlexc.htm). Caml.inria.fr. Retrieved 2011-12-15.
26. "Modula-3 - Procedure Types" (http://www1.cs.columbia.edu/graphics/modula3/tutorial/www/m3_23.html#SEC23). .cs.columbia.edu. 1995-03-08. Retrieved 2011-12-15.
27. "Bruce Eckel's MindView, Inc: Does Java need Checked Exceptions?" (http://www.mindview.net/Etc/Discussions/CheckedExceptions). Mindview.net. Retrieved 2011-12-15.
28. Bjarne Stroustrup, *The C++ Programming Language* Third Edition, Addison Wesley, 1997. ISBN 0-201-88954-4. pp. 375-380.
29. Reeves, J.W. (July 1996). "Ten Guidelines for Exception Specifications". *C++ Report* **8** (7).
30. Sutter, Herb (3 March 2010). "Trip Report: March 2010 ISO C++ Standards Meeting" (http://herbsutter.com/2010/03/13/trip-report-march-2010-iso-c-standards-meeting/). Retrieved 24 March 2010.
31. Mössenböck, Hanspeter (2002-03-25). "Advanced C#: Variable Number of Parameters" (http://ssw.jku.at/Teaching/Lectures/CSharp/Tutorial/Part2.pdf) (PDF). http://ssw.jku.at/Teaching/Lectures/CSharp/Tutorial/: Institut für Systemsoftware, Johannes Kepler Universität Linz, Fachbereich Informatik. p. 32. Retrieved 2011-08-05.
32. Bill Venners; Bruce Eckel (August 18, 2003). "The Trouble with Checked Exceptions: A Conversation with Anders Hejlsberg, Part II" (http://www.artima.com/intv/handcuffs.html). p. 2 (http://www.artima.com/intv/handcuffs2.html).
33. Bloch 2001:178 Bloch, Joshua (2001). *Effective Java Programming Language Guide*. Addison-Wesley Professional. ISBN 0-201-31005-8.
34. "Advantages of Exceptions (The Java™ Tutorials > Essential Classes > Exceptions)" (http://download.oracle.com/javase/tutorial/essential/exceptions/advantages.html). Download.oracle.com. Retrieved 2011-12-15.
35. Bloch 2001:172
36. "Unchecked Exceptions The Controversy (The Java™ Tutorials > Essential Classes > Exceptions)" (http://download.oracle.com/javase/tutorial/essential/exceptions/runtime.html). Download.oracle.com. Retrieved 2011-12-15.
37. "Asynchronous Exceptions in Haskell - Marlow, Jones, Moran (ResearchIndex)" (http://citeseer.ist.psu.edu/415348.html). Citeseer.ist.psu.edu. Retrieved 2011-12-15.
38. Safe asynchronous exceptions for Python. http://www.cs.williams.edu/~freund/papers/02-lwl2.ps
39. "Java Thread Primitive Deprecation" (http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html). Java.sun.com. Retrieved 2011-12-15.
40. What Conditions (Exceptions) are Really About (2008-03-24). "What Conditions (Exceptions) are Really About" (http://web.archive.org/web/20130201124021/http://danweinreb.org/blog/what-conditions-exceptions-are-really-about). Danweinreb.org. Retrieved 2014-09-18.
41. "Condition System Concepts" (http://www.franz.com/support/documentation/6.2/ansicl/section/conditio.htm). Franz.com. 2009-07-21. Retrieved 2011-12-15.

- Gabriel, Richard P.; Steele, Guy L. (2008). *A Pattern of Language Evolution* (http://www.dreamsongs.com/Files/PatternOfLanguageEvolution.pdf) (PDF). LISP50: Celebrating the 50th Anniversary of Lisp. pp. 1–10. doi:10.1145/1529966.1529967 (https://dx.doi.org/10.1145%2F1529966.1529967). ISBN 978-1-60558-383-9.
- Goodenough, John B. (1975a). *Structured exception handling*. Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '75. pp. 204–224.

doi:10.1145/512976.512997 (https://dx.doi.org/10.1145%2F512976.512997).

- Goodenough, John B. (1975). "Exception handling: Issues and a proposed notation" (http://www.cs.colorado.edu/~bec/courses/csci5535-s09/reading/goodenough-exceptions.pdf) (PDF). *Communications of the ACM* **18** (12): 683–696. doi:10.1145/361227.361230 (https://dx.doi.org/10.1145%2F361227.361230).
- White, Jon L (May 1979). *NIL - A Perspective* (http://www.softwarepreservation.org/projects/LISP/MIT/White-NIL_A_Perspective-1979.pdf) (PDF). Proceedings of the 1979 Macsyma User's Conference.

External links

- Article "When Should you catch RuntimeExceptions? (http://10kloc.wordpress.com/2013/03/09/runtimeexceptions-try-catch-or-not-to-catch/)"
- A Crash Course on the Depths of Win32 Structured Exception Handling (http://www.microsoft.com/msj/0197/exception/exception.aspx) by Matt Pietrek - Microsoft Systems Journal (1997)
- Article "All Exceptions Are Handled (http://granitestatehacker.kataire.com/2008/02/all-exceptions-are-handled.html)" by James "Jim" Wilcox
- Article "An Exceptional Philosophy (http://www.dlugosz.com/Magazine/WTJ/May96/)" by John M. Dlugosz
- Article "C++ Exception Handling (https://db.usenix.org/events/wiess2000/full_papers/dinechin/dinechin.pdf)" by Christophe de Dinechin
- Article "Exception Handling in C without C++ (http://www.on-time.com/ddj0011.htm)" by Tom Schotland and Peter Petersen
- Article "Exceptional practices (http://java.sun.com/developer/technicalArticles/Programming/exceptions2/index.html)" by Brian Goetz
- Article "Object Oriented Exception Handling in Perl (http://perl.com/pub/a/2002/11/14/exception.html)" by Arun Udaya Shankar
- Article "PHP exception handling (http://www.chrisjhill.co.uk/Articles/PHP_exception_handling)" by Christopher Hill
- Article "Practical C++ Error Handling in Hybrid Environments (http://ddj.com/dept/debug/197003350)" by Gigi Sayfan
- Article "Programming with Exceptions in C++ (http://oreillynet.com/pub/a/network/2003/05/05/cpluspocketref.html)" by Kyle Loudon
- Article "Structured Exception Handling Basics (http://www.gamedev.net/reference/programming/features/sehbasics/)" by Vadim Kokielov
- Article "Unchecked Exceptions - The Controversy (http://java.sun.com/docs/books/tutorial/essential/exceptions/runtime.html)"
- Conference slides Floating-Point Exception-Handling policies (pdf p. 46) (http://www.eecs.berkeley.edu/~wkahan/Boulder.pdf) by William Kahan
- Descriptions from Portland Pattern Repository (http://c2.com/cgi/wiki?CategoryException)
- Does Java Need Checked Exceptions? (http://www.mindview.net/Etc/Discussions/CheckedExceptions)
- exceptions4c: An exception handling framework for C (http://code.google.com/p/exceptions4c/)
- Another exception handling framework for ANSI/ISO C (https://github.com/takanuva/ANSI-ISO-C-Exception-Handling)
- How to handle class constructors that fail (http://www.amcgowan.ca/blog/computer-science/how-to-handle-class-constructors-that-fail/)
- Java Exception Handling (http://tutorials.jenkov.com/java-exception-handling/index.html) - Jakob Jenkov
- Java: How to rethrow exceptions without wrapping them. (http://robaustin.wikidot.com/rethrow-exceptions) - Rob Austin

- Paper "Exception Handling in Petri-Net-based Workflow Management (http://www.informatik.uni-hamburg.de/TGI/pnbib/f/faustmann_g1.html)" by Gert Faustmann and Dietmar Wikarski
- Problems and Benefits of Exception Handling (http://neil.fraser.name/writing/exception/)
- The Trouble with Checked Exceptions (http://artima.com/intv/handcuffsP.html) - a conversation with Anders Hejlsberg
- Type of Java Exceptions (http://javapapers.com/core-java/java-exception/explain-type-of-exceptions-or-checked-vs-unchecked-exceptions-in-java/)
- Understanding and Using Exceptions in .NET (http://codebetter.com/blogs/karlseguin/archive/2006/04/05/142355.aspx)
- Visual Prolog Exception Handling (http://wiki.visual-prolog.com/index.php?title=Exception_Handling) (wiki article)

Categories: Control flow | Software anomalies