Exception handling

Exception handling is a programming language construct or computer hardware mechanism designed to handle the occurrence of **exceptions**, special conditions that change the normal flow of program execution.

Programming languages differ considerably in their support for exception handling (as distinct from error checking, which is normal program flow that codes for responses to contingencies such as unsuccessful termination of invoked operations). In some programming languages there are functions which cannot be safely called on invalid input data or functions which return values which cannot be distinguished from exceptions. For example, in C the *atoi* (ASCII to integer conversion) function may return 0 (zero) for any input that cannot be parsed into a valid value. In such languages the programmer must either perform error checking (possibly through some auxiliary global variable such as C's errno) or input validation (perhaps using regular expressions).

The degree to which such explicit validation and error checking is necessary is in contrast to exception handling support provided by any given programming environment. Hardware exception handling differs somewhat from the support provided by software tools, but similar concepts and terminology are prevalent.

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*. Depending on the situation, the handler may later resume the execution at the original location using the saved information. For example, a page fault will usually allow the program to be resumed, while a division by zero might not be resolvable transparently.

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

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. a zero denominator in division) or when a resource it relies on is unavailable (like a missing file, or a hard disk error). 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.

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.

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.[1]

Exception safety

A piece of code is said to be **exception-safe**, if run-time failures within the code will not produce ill effects, such as memory leaks, garbled stored data, or invalid output. Exception-safe code must satisfy invariants placed on the code even if exceptions occur. There are several levels of exception safety:

1. **Failure transparency**, also known as the **no throw guarantee**: Operations are guaranteed to succeed and satisfy all requirements even in presence of exceptional situations. If an exception occurs, it will not throw the exception further up. (Best level of exception safety.)
2. **Commit or rollback semantics**, also known as **strong exception safety** or **no-change guarantee**: Operations can fail, but failed operations are guaranteed to have no side effects so all data retain original values.[2]

3. **Basic exception safety**: Partial execution of failed operations can cause side effects, but invariants on the state are preserved. Any stored data will contain valid values even if data has different values now from before the exception.
4. **Minimal exception safety** also known as **no-leak guarantee**: Partial execution of failed operations may store invalid data but will not cause a crash, and no resources get leaked.
5. **No exception safety**: No guarantees are made. (Worst level of exception safety)

For instance, consider a smart vector type, such as C++'s std::vector or Java's ArrayList. When an item *x* is added to a vector *v*, the vector must actually add *x* to the internal list of objects and also update a count field that says how many objects are in *v*. It may also need to allocate new memory if the existing capacity isn't large enough. This memory allocation may fail and throw an exception. Because of this, a vector that provides failure transparency would be very difficult or impossible to write. However, the vector may be able to offer the strong exception guarantee fairly easily; in this case, either the insertion of *x* into *v* will succeed, or *v* will remain unchanged. If the vector provides only the basic exception safety guarantee, if the insertion fails, *v* may or may not contain *x*, but at least it will be in a consistent state. However, if the vector makes only the minimal guarantee, it's possible that the vector may be invalid. For instance, perhaps the size field of *v* was incremented but *x* wasn't actually inserted, making the state inconsistent. Of course, with no guarantee, the program may crash; perhaps the vector needed to expand but couldn't allocate the memory and blindly ploughs ahead as if the allocation succeeded, touching memory at an invalid address.

Usually at least basic exception safety is required. Failure transparency is difficult to implement, and is usually not possible in libraries where complete knowledge of the application is not available.

Verification of exception handling

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 (which 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 verification 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.

Exception support in programming languages

Many computer languages, such as Actionscript, Ada, BlitzMax, C++, C#, D, ECMAScript, Eiffel, Java, ML, Object Pascal (e.g. Delphi, Free Pascal, and the like), Objective-C, Ocaml, PHP (as of version 5), PL/1, Prolog, Python, REALbasic, Ruby, Visual Prolog and most .NET languages have built-in support for exceptions and exception handling. In those languages, the event of an exception (more precisely, an exception handled by the language) searches back through the stack of function calls until an exception handler is found, with some languages calling for unwinding the stack as the search progresses. That is, if function *f* contains 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 for which this is not true is Common Lisp with its Condition System. Common Lisp calls the exception handler and does not unwind the stack. This allows to continue the computation at exactly the same place where the error occurred (for example when a previously missing file is now available). Mythryl's stackless implementation 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 need and does not provide this construct, and the Resource-Acquisition-Is-Initialization technique should be used to free such resources instead.[3] 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.

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

The C++ derivative Embedded C++ excludes exception handling support as it can substantially increase the size of the object code.

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 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.) Two schemes are most common. The first, *dynamic registration*, generates code that continually updates structures about the program state in terms of exception handling.[4] 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.[5]

The second scheme, and the one implemented in many production-quality C++ compilers, is a *table-driven* approach. This creates static tables at compile and link time that relate ranges of the program counter to the program state with respect to exception handling.[6] 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, albeit at the cost of some space, although said space can be allocated into read-only, special-purpose data sections that are not loaded or relocated until and unless an exception is thrown.[7] This second approach is also superior in terms of achieving thread safety.

Other definitional and implementation schemes have been proposed as well.[8] For languages that support metaprogramming, approaches that involve no overhead at all have been advanced.[9]

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 fails, triggering an exception in its caller (so that the abnormal event is not ignored!), after fixing the object's state by re-establishing the invariant (the "organized" part).
- Retry: try the algorithm again, usually after changing some values so that the next attempt will have a better chance to succeed.

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 a "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.

Checked exceptions

The designers of Java devised[10] [11] checked exceptions,[12] 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.

This is related to exception checkers that exist at least for OCaml.[13] The external tool for OCaml is both invisible (i.e. it does not require any syntactic annotations) and facultative (i.e. it is possible to compile and run a program without having checked the exceptions, although this is not suggested 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.[14] These features don't include the compile time checking which is central in the concept of checked exceptions, and hasn't (as of 2006) been incorporated into major programming languages other than Java.[15]

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.[16] 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.[16] Explicit use of empty exception specifications can, however, allow C++ compilers to perform significant code and stack layout optimizations that normally have to be suppressed when exception handling may take place in a function.[7] Some analysts view the proper use of exception specifications in C++ as difficult to achieve.[17] In the upcoming C++ language standard (C++0x), the use of exception specifications as specified in the current version of the standard (C++03), is deprecated.

Views on usage

Checked exceptions can, at compile time, reduce the incidence of unhandled exceptions surfacing at runtime in a given application; the unchecked exceptions (RuntimeExceptions and Errors) 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 by either 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,[18] and mapping lower level exceptions to these types, preferably wrapped using the 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 minimal throws Exception declaration or catch (Exception e) is sufficient for satisfying the checking in Java. While this may have some use, it essentially circumvents the checked exception mechanism, and forces all calling code to catch (Exception e) or add throws Exception to its method signature.

Unchecked exception types should not be handled except, with consideration, at the outermost levels of scope. These often represent scenarios that do not allow for recovery: RuntimeExceptions frequently reflect programming defects,[19] and Errors generally represent unrecoverable JVM failures. The view is that, even in a language that supports checked exceptions, there are cases where the use of checked exceptions is not appropriate.

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.[20] [21] 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 lost thread stopping and resuming.[22] 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 [23] (see Common Lisp Condition System) which 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 which do not represent errors may safely go unhandled entirely; their only purpose may be to propagate hints or warnings toward the user.[24]

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*.

References

[1] All Exceptions Are Handled, Jim Wilcox, http://poliTechnosis.kataire.com/2008/02/all-exceptions-are-handled.html

[2] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1997/N1077.asc

[3] Bjarne Stroustrup's FAQ (http://www2.research.att.com/~bs/bs_faq2.html#finally)

[4] D. Cameron, P. Faust, D. Lenkov, M. Mehta, "A portable implementation of C++ exception handling", *Proceedings of the C++ Conference* (August 1992) USENIX.

[5] 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.

[6] Lajoie, Josée (March–April 1994). "Exception handling – Supporting the runtime mechanism". *C++ Report* **6** (3).

[7] Schilling, Jonathan L. (August 1998). "Optimizing away C++ exception handling". *SIGPLAN Notices* **33** (8): 40–47. doi:10.1145/286385.286390.

[8] " How to Implement Software Exception Handling (http://intel.com/cd/ids/developer/asmo-na/eng/81438.htm)", Intel Corporation.

[9] 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.

[10] LISTSERV 15.0 - RMI-USERS Archives (http://archives.java.sun.com/cgi-bin/wa?A2=ind9901&L=rmi-users&F=P&P=36083)

[11] Google Answers: The origin of checked exceptions (http://answers.google.com/answers/threadview?id=26101)

[12] Java Language Specification, chapter 11.2. http://java.sun.com/docs/books/jls/third_edition/html/exceptions.html#11.2

[13] OcamlExc -- An uncaught exceptions analyzer for Objective Caml (http://caml.inria.fr/pub/old_caml_site/ocamlexc/ocamlexc.htm)

[14] Modula-3 - Procedure Types (http://www1.cs.columbia.edu/graphics/modula3/tutorial/www/m3_23.html#SEC23)

[15] Bruce Eckel's MindView, Inc: Does Java need Checked Exceptions? (http://www.mindview.net/Etc/Discussions/CheckedExceptions)

[16] Bjarne Stroustrup, *The C++ Programming Language* Third Edition, Addison Wesley, 1997. ISBN 0-201-88954-4. pp. 375-380.

[17] Reeves, J.W. (July 1996). "Ten Guidelines for Exception Specifications". *C++ Report* **8** (7).

[18] Bloch 2001:178 Bloch, Joshua (2001). *Effective Java Programming Language Guide*. Addison-Wesley Professional. ISBN 0-201-31005-8.

[19] Bloch 2001:172

[20] Asynchronous Exceptions in Haskell - Marlow, Jones, Moran (ResearchIndex) (http://citeseer.ist.psu.edu/415348.html)

[21] Safe asynchronous exceptions for Python. http://www.cs.williams.edu/~freund/papers/02-lwl2.ps

[22] Java Thread Primitive Deprecation (http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html)
[23] What Conditions (Exceptions) are Really About (http://danweinreb.org/blog/what-conditions-exceptions-are-really-about)
[24] Condition System Concepts (http://www.franz.com/support/documentation/6.2/ansicl/section/conditio.htm)

External links

- Article " PHP exception handling (http://www.chrisjhill.co.uk/Articles/PHP_exception_handling)" by Christopher Hill
- Article " Unchecked Exceptions - The Controversy (http://java.sun.com/docs/books/tutorial/essential/exceptions/runtime.html)"
- Article " Practical C++ Error Handling in Hybrid Environments (http://ddj.com/dept/debug/197003350)" by Gigi Sayfan
- Article " C++ Exception Handling (https://db.usenix.org/events/wiess2000/full_papers/dinechin/dinechin.pdf)" by Christophe de Dinechin
- Article " Exceptional practices (http://java.sun.com/developer/technicalArticles/Programming/exceptions2/index.html)" by Brian Goetz
- Article " Programming with Exceptions in C++ (http://oreillynet.com/pub/a/network/2003/05/05/cpluspocketref.html)" by Kyle Loudon
- Article " Object Oriented Exception Handling in Perl (http://perl.com/pub/a/2002/11/14/exception.html)" by Arun Udaya Shankar
- Article " Exception Handling in C without C++ (http://www.on-time.com/ddj0011.htm)" by Tom Schotland and Peter Petersen
- Article " Structured Exception Handling Basics (http://www.gamedev.net/reference/programming/features/sehbasics/)" by Vadim Kokielov
- Article " All Exceptions Are Handled (http://politechnosis.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
- 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
- Descriptions from Portland Pattern Repository (http://c2.com/cgi/wiki?CategoryException)
- 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)
- The Trouble with Checked Exceptions (http://artima.com/intv/handcuffsP.html) - a conversation with Anders Hejlsberg
- Does Java Need Checked Exceptions? (http://www.mindview.net/Etc/Discussions/CheckedExceptions)
- Problems and Benefits of Exception Handling (http://neil.fraser.name/writing/exception/)
- Understanding and Using Exceptions in .NET (http://codebetter.com/blogs/karlseguin/archive/2006/04/05/142355.aspx)
- Java Exception Handling (http://tutorials.jenkov.com/java-exception-handling/index.html) - Jakob Jenkov
- Visual Prolog Exception Handling (http://wiki.visual-prolog.com/index.php?title=Exception_Handling) (wiki article)
- Type of Java Exceptions (http://javapapers.com/core-java/java-exception/explain-type-of-exceptions-or-checked-vs-unchecked-exceptions-in-java/)
- Java : How to rethrow exceptions without wrapping them. (http://robaustin.wikidot.com/rethrow-exceptions) - Rob Austin
- How to handle class constructors that fail (http://www.amcgowan.ca/blog/computer-science/how-to-handle-class-constructors-that-fail/)

- exceptions4c: An exception handling framework for C (http://code.google.com/p/exceptions4c/)

Article Sources and Contributors

Exception handling *Source*: http://en.wikipedia.org/w/index.php?oldid=422045034 *Contributors*: A Quest For Knowledge, Aaron Rotenberg, Abarnea 2000, Abdull, Ahy1, Andreas Kaufmann, Andrew Hampe, Aomarks, Ascánder, Astatine, Auntof6, Aurelien, B-rat, BMF81, BenFrantzDale, Betterusername, Blaxthos, Bmaisonnier, Brockert, Btx40, BurntSky, C17GMaster, CSProfBill, Cadr, Caiyu, Cander0000, Chealer, Chesterloke, Chu Jetcheng, Danrah, Dasch, David.Monniaux, David.kaplan, Decltype, Demitsu, Derek farn, Descubes, Doug Bell, DropDeadGorgias, Edcolins, Elektrik Shoos, Enric Naval, Eric B. and Rakim, Erkan Yilmaz, Esap, Etaoin, EvanED, EvanGrim, Everyking, Ewlloyd, FabianNiemann, Feb30th1712, Firsfron, Fubar Obfusco, Furrykef, Fuzzie, Gaal, Gary King, Green caterpillar, Greenrd, HaeB, Hairy Dude, Hans Bauer, Harriv, Hcheney, Homerjay, Hosterweis, Hu12, I take exception, J Casanova, JLaTondre, Jeanfoucault, JimD, Jkl, John Vandenberg, Jonhanson, Joswig, Jóna Þórunn, Kavadi carrier, Kku, Kseg, Kulandai, LilHelpa, Linas, Lowellian, Lucio, Luís Felipe Braga, MONGO, Magnus Bakken, Mahadevan R, Marudubshinki, Massysett, Melah Hashamaim, Mike Fikes, Mikething, Mintleaf, Mipadi, Munificent, Musiphil, Naveen84nv, Nbarth, NeilFraser, Neilc, NickBush24, Nigelj, OranL, Orderud, Patrick, PdDemeter, Peterdjones, Phresnel, Pianohacker, Piet Delport, PrologFan, Prosfilaes, Ptrb, QuantumEngineer, Radagast83, Random832, RedWolf, Rjgodoy, Rling, Rufous, Runtime, SAE1962, ScottBlomquist, SeanProctor, Smeezekitty, Smyth, Sycomonkey, TheTito, Thecheesykid, Thecodemachine, Thomas Linder Puls, Tigrisek, TimBentley, Tmaufer, Tobias Bergemann, Tompsci, TotoBaggins, Triddle, TuukkaH, UtherSRG, Vald, Venugopal.ch, Veryfaststuff, Vicarious, Vocaro, Widefox, Xezbeth, Yerpo, Yoric, Ysangkok, Zark77, Zippanova, 215 anonymous edits

License