Exceptions: Structured Exit

Terminate part of computation

* Jump out of construct
* Pass data as part of jump
* Return to most recent site set up to handle exception
* Unnecessary activation records may be deallocated
  + May need to free heap space, other resources

Two main language constructs

* Declaration to establish exception handler
* Statement or expression to raise or throw exception

Often used for unusual or exceptional condition, but not necessarily

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, application, table

Description automatically generated

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.

On the rationale for such a design for the C++ mechanism, Stroustrup notes: At the Palo Alto 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. 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.

When sth went wrong (Whatever I do after this will be out of the semantic that I define) you have 2 options:

* Terminate
* Try to resume (programmer decide what to do about this – exception handling)

Opposing Views… (Idea against resumption)

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."

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".

All of the exceptions come at runtime bc if I know that they are gonna come at compile time, I will program them for those so I can get my logic right. With exception handling I am opening another path which is different than my logic for my program.

Exception Handling

We distinguish between two such classes of events:

* Those that are detected by hardware: e.g., disk read errors, end-of-file
* Those that are software-detectable: e.g., subscript range errors

**Definition**: An exception is an unusual event that is detectable by either hardware or software and that may require special processing.

**Terminology**: The special processing that may be required when an exception is detected is called exception handling. The processing is done by a code unit or segment called an exception handler. An exception is raised when its associated event occurs.

User-Defined Exception Handling

When a language does not include specific exception handling facilities, the user often handles software detections by him/herself.

This is typically done in one of three ways:

* Use of a status variable (or flag) which is assigned a value in a subprogram according to the correctness of its computation. [Used in standard C library functions]
* Use of a label parameter in the subprogram to make it return to different locations in the caller according to the value of the label. [Used in Fortran].
* Define the handler as a separate subprogram and pass its name as a parameter to the called unit. But this means that a handler subprogram must be sent with every call to every subprogram.

Advantages to Built-in Exception Handling

* Without built-in Exception Handling, the code required to detect error conditions can considerably clutter a program.
* Built-in Exception Handling often allows exception propagation. i.e., an exception raised in one program unit can be handled in some other unit in its dynamic or static ancestry. A single handler can thus be used in different locations.
* Built-in Exception Handling forces the programmer to consider all the events that could occur and their handling. This is better than not thinking about them.
* Built-in Exception Handling can simplify the code of programs that deal with unusual situations. (Such code would normally be very convoluted without it).

Illustration of an Exception Handling Mechanism

Graphical user interface, application

Description automatically generated

Design Issues: Exception Binding

Binding an exception occurrence to an exception handler:

* At the unit level: how can the same exception raised at different points in the unit be bound to different handlers within the unit? Farklı yerlerde aynı exception olunca aynı handleri çok fazla kez mi yazacağız?
* At a higher level: if there is no exception handler local to the unit, should the exception be propagated to other units? If so, how far? [Note: if handlers must be local, then many need to be written. If propagation is permitted, then the handler may need to be too general to really be useful.] Tracking will be difficult.

Design Issues: Continuation

After an exception handler executes, either control can transfer to somewhere in the program outside of the handler code, or program execution can terminate.

* Termination is the simplest solution and is often appropriate.
* Resumption is useful when the condition encountered is unusual, but not erroneous. In this case, some convention should be chosen as to where to return:
  + At the statement that raised the exception?
  + At the statement following the statement that raised the exception?
  + At some other unit?

Design Issues: Others

Is finalization—the ability to complete some computations at the end of execution regardless of whether the program terminated normally or because of an exception—supported?

How are user-defined exceptions specified?

Are there pre-defined exceptions?

Should it be possible to disable predefined exceptions?

If there are pre-defined exceptions, should there be default exception handlers for programs that do not provide their own?

Can pre-defined exceptions be explicitly raised?

Are hardware-detectable errors treated as exceptions that may be handled?

Why would you disable predefined exception in your language like zero division exception?

* flexibility
  + You don’t always want to deal with the exception.

But if I put it in my language, it is important.

Exception Handling in Java: Class Hierarchy

Diagram

Description automatically generated

Exception handling is typed heavily to types.

Exception Handling in Java: Exception Handlers

A try construct includes a compound statement called the try clause and a list of exception handlers:

Text

Description automatically generated

Binding happens in unit level.

Exception Handling in Java: Binding Exceptions to Handlers

An exception is thrown using the throw statement.

E.g.: throw new MyException (“a message to specify the location of the error”)

Binding: If an exception is thrown in the compound statement of a try construct, it is bound to the first handler (catch function) immediately following the try clause whose parameter is the same class as the thrown object, or an ancestor of it. If a matching handler is found, the throw is bound to it and it is executed.

Exception Handling in Java: The finally clause

Sometimes, a process must be executed regardless of whether an exception is raised or not and handled or not.

This occurs, for example, in the case where a file must be closed or an external resource released, regardless of the outcome of the program.

This is done by adding a *finally* clause at the end of the list of handlers, just after a complete try construct.

The finally clause is executed in all cases whether or not try throws an exception, and whether or not it is caught by a catch clause.

Event Handling I

Event handling is similar to exception handling.

The difference is that while exceptions can be created whether explicitly by user code or implicitly by hardware or a software interpreter, events are created by external actions (like keyboard hit), such as user interactions though a graphical user interface (GUI)

In event-driven programming, parts of the program are executed at completely unpredictable times, often triggered by user interactions with the executing program.

Event Handling II

An event is a notification that something specific has occurred, such as a mouse click on a graphical button.

An event handler is a segment of code that is executed in response to the appearance of an event.

Event handling is useful in Web applications such as:

* commercial applications where a user clicks on buttons to select merchandise or
* Web form completion, where event handling is used to verify that no error or omission has occurred in the completion of a form.

Java supports two different approaches to presenting interactive displays to users: either from application programs or from applets.

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

Diagram

Description automatically generatedDiagram

Description automatically generated

Table

Description automatically generated

Typing of Exceptions

Typing of raise <exn>

* Definition of typing: expression e has type t if normal termination of e produces value of type t
* Raising an exception is not normal termination
  + Example: 1 + raise X

Typing of handle <exception> => <value>

* Converts exception to normal termination
* Need type agreement
* Examples
  + 1 + ((raise X) handle X => e) Type of emust be int (why?)
  + 1 + (e1 handle X => e2) Type of e1,e2 must be int (why?)

Exceptions and Resource Allocation

Graphical user interface, text, application

Description automatically generated