Principles of Programming Languages(CS F301)

BITS Pilani
Hyderabad Campus

Prof.R.Gururaj
CS&IS Dept.

Exception Handling (Ch.13 & 14 of T1)

BITS Pilani
Hyderabad Campus

Prof R Gururaj

Introduction to Exception Handling(Ch.14)

innovate achieve lead

Most computer HW systems are capable of detecting certain runtime error conditions.

Early programming languages were designed in such a way that the user program could neither detect nor attempt to deal with such errors.

In these programs the occurrence of errors will simply cause the program to be terminated and transfer the control back to the OS.

The OS's reaction to this will be to display a diagnostic message which may be meaningful and useful, or very cryptic.

There is a category of serious errors that are not detected by HW but can be detected by code generated by the compiler.

Ex: Array subscript range error.

This detection sometimes required by the language design.

Ex. Java compilers usually generate code to check the correctness of every subscript expression, and can be detected at compile time if they are literals. Else not possible at compile time.

In C we don't have such check.

In some languages subscript range check is an optional facility and can be turned off if as desired.

The designers of most contemporary languages have included mechanisms that allow programs to react in a standard way to certain runtime errors, as well as other program detected unusual events.

Programs may also be notified when certain events are detected by HW or OS so that they also can react to these events.

These mechanisms are collectively called as *Exception Handling*.

Because of the complexity this Exception Handling mechanism adds, some languages do not support this feature.

We consider both the errors detected by HW such as disk read error, and unusual conditions such as end-of-file to be *Exceptions*.

We define *Exception* to be any unusual event , erroneous or not that is detected by either HW or SW and that may be required special processing.

The special processing that may be required when an exception is detected is called *Exception Handling*.

This processing is done by a code unit or segment called *Exception Handler*.

Exceptions are said to be *thrown* or *raised*.

The absence of separate or specific exception-handling facility in a language does not hinder the handling errors of user-defined, software-detected exceptions .

Such an exception detected within a program unit is often handled by it's caller or invoker.

Advantages of support for exception Handling at Language level.

1. Compiler can insert machine code for such checks greatly shortening and simplifying the source program.
2. Exceptions can be propagated from program unit to other, thus allowing single exception handler to be used for any number of program units. Saving development time, cost, and reduced complexity.
3. Encourages its users to consider all of the events that could occur during the program execution and how they can be handled.

Predefined exceptions are implicitly *raised*.

Where as the user-defined exceptions must be explicitly raised by the user code.

It should be possible to bind the exceptions that can be raised by particular statements to particular handlers, even though the same exception can be raised by many different statements.

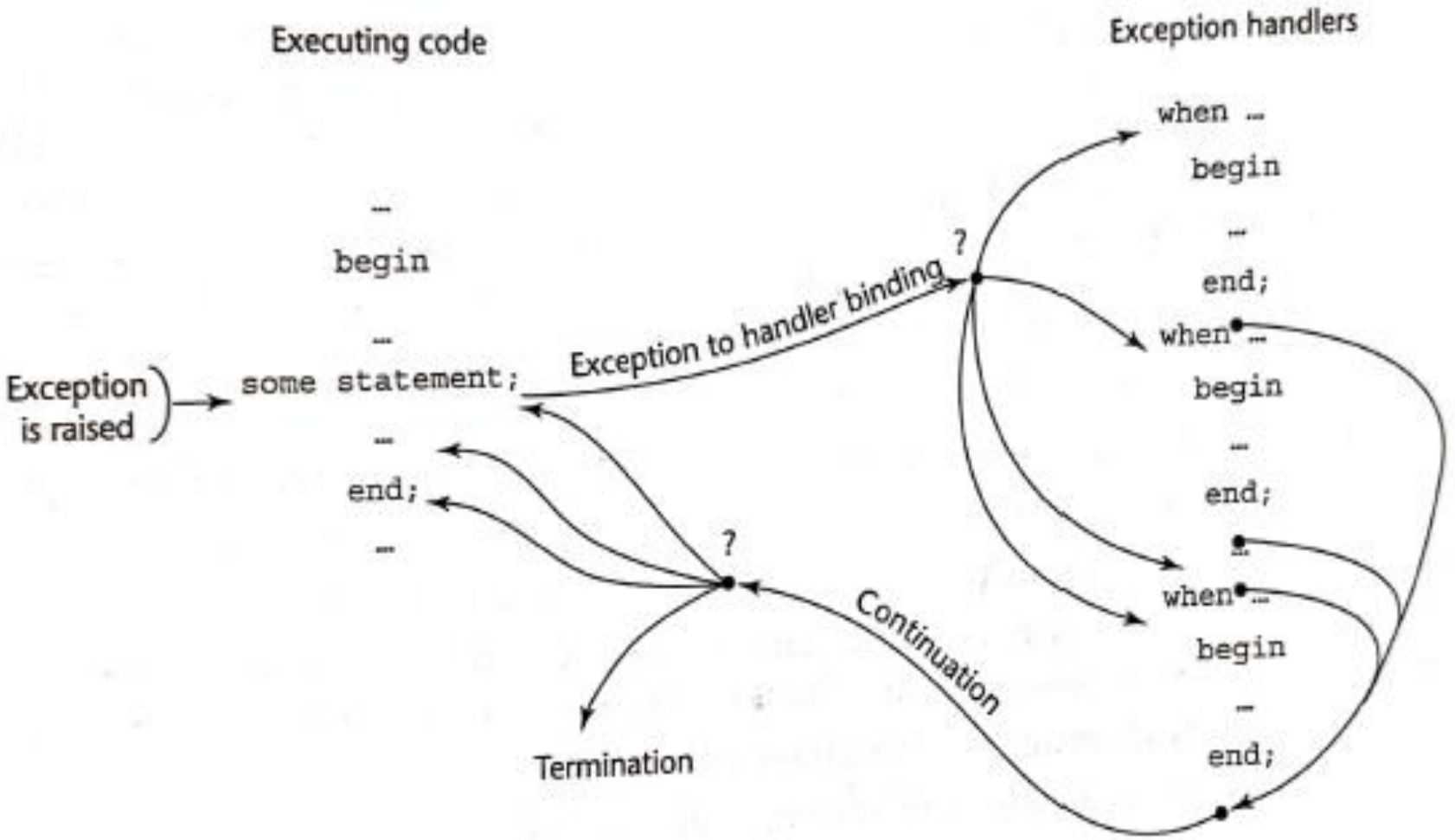
Issue related to *binding* is- whether the information about the exception is made available to the handler?

Continuation- control continuation after handler is executed.

Options- resumption or termination

Finalization.

Exception Handling Control Flow



Design Issues

The exception-handling design issues can be summarized as follows:

- How and where are exception handlers specified, and what is their scope?
- How is an exception occurrence bound to an exception handler?
- Can information about an exception be passed to the handler?
- Where does execution continue, if at all, after an exception handler completes its execution? (This is the question of continuation or resumption.)
- Is some form of finalization provided?
- How are user-defined exceptions specified?
- If there are predefined exceptions, should there be default exception handlers for programs that do not provide their own?
- Can predefined exceptions be explicitly raised?
- Are hardware-detectable errors treated as exceptions that may be handled?
- Are there any predefined exceptions?
- Should it be possible to disable predefined exceptions?

Ada provides extensive exception handling facilities and a small collection of built-in exceptions.

The handlers are attached to the program entities.

But it possible to propagate exceptions implicitly or explicitly to other program units if no local handlers are available.

C++ includes no predefined exceptions.

Excepting those defined in standard library.

C++ exceptions are objects of primitive type, a predefined class or a user-defined class.

Exceptions are bound to handlers by connecting the type of the expression in the throw statement to that formal parameter of the handler.

Handlers have the same name- *catch*.

The C++ *throw* clause of a method lists the types of exceptions the method could throw.

Java exceptions are objects whose ancestry must trace back to a class that descends from the *Throwable* class

There are two categories of exceptions- checked and unchecked.

Checked exceptions are a concern for the user program and the compiler. Unchecked exceptions can occur anywhere and are often ignored by user programs.

Clauses-

throws, finally, try

Java now includes *asserts* statement which facilitate defensive programming.

Using assert in Java

Another relatively new addition to Java is the keyword **assert**. It is used during program development to create an *assertion*, which is a condition that should be true during the execution of the program. For example, you might have a method that should always return a positive integer value. You might test this by asserting that the return value is greater than zero using an **assert** statement. At run time, if the condition actually is true, no other action takes place. However, if the condition is false, then an **AssertionError** is thrown. Assertions are often used during testing to verify that some expected condition is actually met. They are not usually used for released code.

```
class Demo
{
        public static void main(String args[]){
        int x=Integer.parseInt(args[0]);
        System.out.println(x);
        assert x<100;
        System.out.println("Valid integer:");

          }
}
```

Event Handling

An *event* is a notification that something occurred that requires special processing.

Events are often created by user interactions with a program through a graphical user interface.

Java event handlers are called through event listeners.

An event listener must be registered for an event if it is to be notified when the event occurs.

Windows Forms is the original approach to building GUI components and handling events in .NET languages.

A C# application builds a GUI in this approach by subclassing the Form class.

All .NET event handlers are registered by creating an EventHandler object and assigning it to the predefined delegate associated with the GUI object that can raise the vent.

Summary of Ch.14 (Exception Handling.)

innovate achieve lead

1. Introduction
2. Exception and Exception handler
3. Exception handling control flow
4. Design issues
5. Ada, C++, Java Exception handling
6. Java assert.
7. Java and .NET event handling in GUI