CSPC31: Principles of Programming Languages

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015

Ph: 999 470 4853
E-Mail: balakrishnan@nitt.edu

Books

- **Text Books**
  - ✓ **Robert W. Sebesta, *"Concepts of Programming Languages"*, Tenth Edition, Addison Wesley, 2012.**
  - ✓ Michael L. Scott, *"Programming Language Pragmatics"*, Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
  - ✓ Allen B Tucker, and Robert E Noonan, *"Programming Languages – Principles and Paradigms"*, Second Edition, Tata McGraw Hill, 2007.
  - ✓ R. Kent Dybvig, *"The Scheme Programming Language"*, Fourth Edition, MIT Press, 2009.
  - ✓ Jeffrey D. Ullman, *"Elements of ML Programming"*, Second Edition, Prentice Hall, 1998.
  - ✓ Richard A. O'Keefe, *"The Craft of Prolog"*, MIT Press, 2009.
  - ✓ W. F. Clocksin, C. S. Mellish, *"Programming in Prolog: Using the ISO Standard"*, Fifth Edition, Springer, 2003.

Chapters

Chapter No.	Title
1.	Preliminaries
~~2.~~	~~Evolution of the Major Programming Languages~~
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
~~5.~~	~~Names, Binding, Type Checking and Scopes~~
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
~~11.~~	~~Abstract Data Types and Encapsulation Constructs~~
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages

Chapter 14 – Exception Handling and Event Handling

Objectives

- Describes the fundamental concepts of exception handling
- Introduction to the design issues for exception handling
- Description and evaluation of exception-handling facilities of C++
- Event handling

Miscellaneous

```
int main()
{
    int x = -1;
    cout << "Before try \n";
    try
    {
      cout << "Inside try \n";
      if (x < 0)
      {
        throw x;
        cout << "After throw (Never executed) \n";
      }
    }
    catch (int x)
    {
      cout << "Exception Caught \n";
    }
    cout << "After catch (Will be executed) \n";
    return 0;
}
```

O/P:

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

```
int main()
{
    try
    {
      throw 10;
    }
    catch (char *excp)
    {
        cout << "Caught " << excp;
    }
    catch (...)
    {
        cout << "Default Exception\n";
    }
    return 0;
}
```

O/P:

```
Default Exception
```

Miscellaneous

```
int main()
{
   try
   {
     throw 'a';
   }
   catch (int x)
   {
      cout << "Caught " << x;
   }
→ catch (...)
   {
      cout << "Default Exception\n";
   }
   return 0;
}
```

O/P: Default Exception

```
int main()
{
   try
   {
     throw 'a';
   }
   catch (int x)
   {
      cout << "Caught ";
   }
   return 0;
}
```

If an exception is thrown and not caught anywhere, the program terminates abnormally

O/P: terminate called after throwing an instance of 'char'
This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information.

Miscellaneous

// This function signature is fine by the compiler, but not recommended. Ideally, **the function should specify all uncaught exceptions and function signature should be "*void fun(int *ptr, int x) throw (int *, int)*"**

```
void fun(int *ptr, int x)
{
   if (ptr == NULL)
      throw ptr;
   if (x == 0)
      throw x;
}
int main()
{
   try
   {
     fun(NULL, 0);
   }
   catch(...)
   {
      cout << "Caught exception from fun()";
   }
   return 0;
}
```

Unlike Java, in C++, all exceptions are unchecked. Compiler doesn't check whether an exception is caught or not. **Eg:** In C++, it is not necessary to specify all uncaught exceptions in a function declaration. Although it's a recommended practice to do so. **Eg:** The following program compiles fine, but ideally signature of fun() should list unchecked exceptions

O/P: Caught exception from fun()

// Here we specify the exceptions that this function throws.

```
void fun(int *ptr, int x) throw (int *, int)
{
   if (ptr == NULL)
      throw ptr;
   if (x == 0)
      throw x;
}
int main()
{
   try
   {
     fun(NULL, 0);
   }
   catch(...)
   {
      cout << "Caught exception from fun()";
   }
   return 0;
}
```

O/P: Caught exception from fun()

Miscellaneous

```
import java.io.*;
class Main
{
   public static void main(String[] args)
   {
     FileReader file = new
     FileReader("C:\\test\\a.txt");
     BufferedReader fileInput = new
     BufferedReader(file);
      for (int counter = 0; counter < 3;
     counter++)
        System.out.println(fileInput.readLin
     e());
      fileInput.close();
   }
}
```

Checked: Are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword

O/P: Exception in thread "main" java.lang.RuntimeException: Uncompilable source code - unreported exception java.io.**FileNotFoundException**; must be caught or declared to be thrown at Main.main(Main.java:5)

```
import java.io.*;
class Main
{
   public static void main(String[] args)
throws IOException
{
      FileReader file = new
FileReader("C:\\test\\a.txt");
      BufferedReader fileInput = new
BufferedReader(file);
      // Print first 3 lines of file "C:\test\a.txt"
      for (int counter = 0; counter < 3;
counter++)
         System.out.println(fileInput.readLine
());
      fileInput.close();
   }
}
```

Note: To fix the problem, we either need to specify list of exceptions using throws, or we need to use try-catch block. We have used throws in the above program. Since ***FileNotFoundException*** is a subclass of ***IOException***, we can just specify *IOException* in the throws list and make the above program compiler-error-free

Miscellaneous

```
class Base {};
class Derived: public Base {};
int main()
{
  Derived d;
  try
  {
    // Some monitored code
     throw d;
  }
  catch(Base b)
  {
     cout<<"Caught Base Exception";
  }
  catch(Derived d)
  {  //This catch block is NEVER executed
     cout<<"Caught Derived Exception";
  }
  getchar();
  return 0;
}
```

A derived class exception should be caught before a base class exception

O/P: Caught **Base** Exception

```
class Base {};
class Derived: public Base {};
int main()
{
  Derived d;
  try
  {
     // Some monitored code
     throw d;
  }
  catch(Derived d)
  {
     cout<<"Caught Derived Exception";
  }
  catch(Base b)
  {
     cout<<"Caught Base Exception";
  }
  getchar();
  return 0;
}
```

O/P: Caught **Derived** Exception

Miscellaneous

In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using "throw;

```
int main()
{
   try
   {
      try
      {
         throw 20;
      }
      catch (int n)
     {
         cout << "Handle Partially ";
         throw; // Re-throwing an exception
      }
   }
   catch (int n)
   {
      cout << "Handle remaining ";
   }
   return 0;
}
```

O/P: Handle Partially Handle remaining

```
class Test {
public:
   Test()
   {
     cout << "Constructor of Test " << endl;
   }
   ~Test()
   {
     cout << "Destructor of Test " << endl;
   }
};
int main()
{
   try
   {
      Test t1;
      throw 10;
   }
   catch (int i)
   {
      cout << "Caught " << i << endl;
   }
}
```

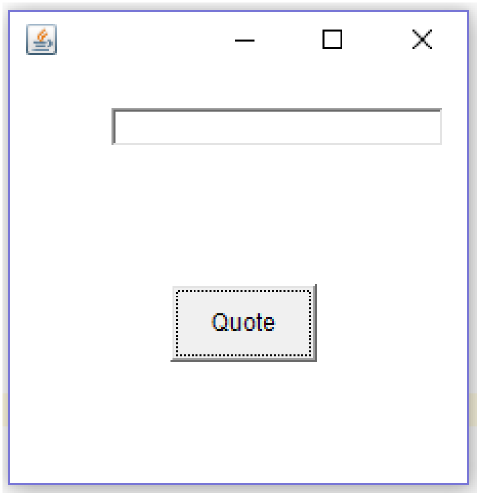
When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block

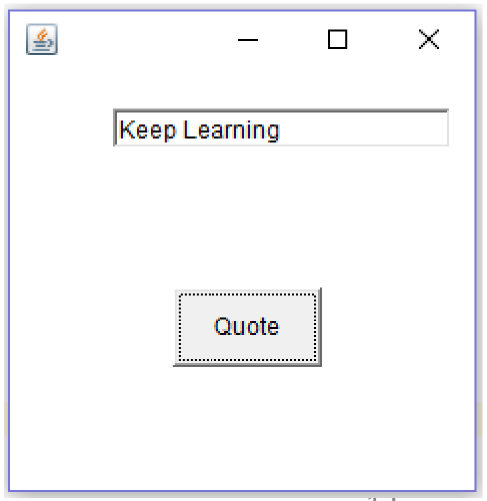
O/P: Constructor of Test
Destructor of Test
Caught 10

Miscellaneous

Event Handling

```
import java.awt.*;
import java.awt.event.*;
class  EventHandle extends Frame
     implements ActionListener
{
   TextField textField;
   EventHandle()
   {
      textField = new TextField();
      textField.setBounds(60,50,170,20);
      Button button = new Button("Quote");
      button.setBounds(90,140,75,40);
       //1
       button.addActionListener(this);
       add(button);
       add(textField);
       setSize(250,250);
       setLayout(null);
       setVisible(true);
     }
//2
public void actionPerformed(ActionEvent e)
{
     textField.setText("Keep Learning");
}
 public static void main(String args[])
{
   new EventHandle();
}
}
```




Keep Learning

Introduction

- Most computer hardware systems are capable of detecting certain run-time error conditions, such as floating-point overflow
- Early programming languages were designed and implemented in such a way that the user program could neither detect nor attempt to deal with such errors
- In these languages, the occurrence of such an error simply causes the program to be terminated and control to be transferred to the operating system
- The typical operating system reaction to a run-time error is to display a diagnostic message, which may be meaningful and therefore useful, or highly cryptic
- After displaying the message, the program is terminated
- In the case of input and output operations, however, the situation is somewhat different
- Eg: A Fortran Read statement can intercept input errors and end-of-file conditions, both of which are detected by the input device hardware
- In both cases, the Read statement can specify the label of some statement in the user program that deals with the condition
- In the case of the end-of-file, it is clear that the condition is not always considered an error
- In most cases, it is nothing more than a signal that one kind of processing is completed and another kind must begin
- In spite of the obvious difference between end-of-file and events that are always errors, such as a failed input process, Fortran handles both situations with the same mechanism

```
Read(Unit=5, Fmt=1000, Err=100, End=999) Weight
```

Introduction

Read(Unit=5, Fmt=1000, Err=100, End=999) Weight

- Fortran uses simple branches for both input errors and end-of-file
- There is a category of serious errors that are not detectable by hardware but can be detected by code generated by the compiler
- Eg: Array subscript range errors are almost never detected by hardware, but they lead to serious errors that often are not noticed until later in the program execution
- Detection of subscript range errors is sometimes required by the language design
- Eg: Java compilers usually generate code to check the correctness of every subscript expression (they do not generate such code when it can be determined at compile time that a subscript expression cannot have an out-of-range value, for example, if the subscript is a literal)
- In C, subscript ranges are not checked because the cost of such checking was (and still is) not believed to be worth the benefit of detecting such errors
- In some compilers for some languages, subscript range checking can be selected (if not turned on by default) or turned off (if it is on by default) as desired in the program or in the command that executes the compiler

Introduction

- Designers of most contemporary languages have included mechanisms that allow programs to react in a standard way to certain run-time errors, as well as other program-detected unusual events
- Programs may also be notified when certain events are detected by hardware or system software, so that they also can react to these events
- These mechanisms are collectively called exception handling
- Perhaps the most plausible reason some languages do not include exception handling is the complexity it adds to the language

Basic Concepts

- We consider both the errors detected by hardware, such as disk read errors, and unusual conditions, such as end-of-file (which is also detected by hardware), to be exceptions
- We further extend the concept of an exception to include errors or unusual conditions that are software-detectable (by either a software interpreter or the user code itself)
- Accordingly, we define exception to be any unusual event, erroneous or not, that is detectable by either hardware or software and that may require special processing

Introduction

- The special processing that may be required when an exception is detected -> Exception Handling
- This processing is done by a code unit or segment -> Exception Handler
- An exception is raised when its associated event occurs
- In some C-based languages, exceptions are said to be thrown, rather than raised
- Different kinds of exceptions require different exception handlers
- Detection of end-of-file nearly always requires some specific program action
- But, clearly, that action would not also be appropriate for an array index range error exception
- In some cases, the only action is the generation of an error message and an orderly termination of the program
- In some situations, it may be desirable to ignore certain hardware-detectable exceptions—for example, division by zero—for a time
- This action would be done by disabling the exception
- A disabled exception could be enabled again at a later time

Introduction

- The absence of separate or specific exception-handling facilities in a language does not preclude the handling of user-defined, software-detected exceptions
- Such an exception detected within a program unit is often handled by the unit's caller, or invoker
- **One possible design** is to send an auxiliary parameter, which is used as a status variable
- The status variable is assigned a value in the called subprogram according to the correctness and/or normalness of its computation
- Immediately upon return from the called unit, the caller tests the status variable
- If the value indicates that an exception has occurred, the handler, which may reside in the calling unit, can be enacted
- Many of the **C standard library functions** use a variant of this approach: The return values are used as error indicators. **Eg:** int main()

Introduction

- **Another possibility** is to pass a label parameter to the subprogram
- Of course, this approach is possible only in languages that allow labels to be used as parameters
- Passing a label allows the called unit to return to a different point in the caller if an exception has occurred
- As in the first alternative, the handler is often a segment of the calling unit's code
- This is a common use of label parameters in **Fortran**
- **A third possibility** is to have the handler defined as a separate subprogram whose name is passed as a parameter to the called unit
- In this case, the handler subprogram is provided by the caller, but the called unit calls the handler when an exception is raised
- **One problem** with this approach is that one is required to send a handler subprogram with every call to every subprogram that takes a handler subprogram as a parameter, whether it is needed or not
- Furthermore, to deal with several different kinds of exceptions, several different handler routines would need to be passed, complicating the code

Introduction

- If it is desirable to handle an exception in the unit in which it is detected, the handler is included as a segment of code in that unit
- There are some definite advantages to having exception handling built into a language
- First, without exception handling, the code required to detect error conditions can considerably clutter a program
- Eg: Suppose a subprogram includes expressions that contain 10 references to elements of a matrix named mat, and any one of them could have an index out-of-range error. Further suppose that the language does not require index range checking
- Without built-in index range checking, every one of these operations may need to be preceded by code to detect a possible index range error
- Eg: Consider the following reference to an element of mat, which has 10 rows and 20 columns

```
if (row >= 0 && row < 10 && col >= 0 && col < 20)
  sum += mat[row][col];
else
  System.out.println("Index range error on mat, row = " +
                     row + " col = " + col);
```

Introduction

- The presence of exception handling in the language would permit the compiler to insert machine code for such checks before every array element access, greatly shortening and simplifying the source program
- Another advantage of language support for exception handling results from exception propagation
- Exception propagation allows an exception raised in one program unit to be handled in some other unit in its dynamic or static ancestry
- This allows a single exception handler to be used for any number of different program units
- This reuse can result in significant savings in development cost, program size, and program complexity
- A language that supports exception handling encourages its users to consider all of the events that could occur during program execution and how they can be handled
- This approach is far better than not considering such possibilities and simply hoping nothing will go wrong

Introduction

Design Issues

- System might allow both predefined and user-defined exceptions and exception handlers
- Note that predefined exceptions are implicitly raised, whereas user-defined exceptions must be explicitly raised by user code
- Consider the following skeletal subprogram that includes an exception-handling mechanism for an implicitly raised exception:

```
void example() {
  . . .
  average = sum / total;
  . . .
  return;
/* Exception handlers */
  when zero_divide {
    average = 0;
    printf("Error-divisor (total) is zero\n");
  }
  . . .
}
```

- The exception of division by zero, which is implicitly raised, causes control to transfer to the appropriate handler, which is then executed

Introduction

- First design issue for exception handling is how an exception occurrence is bound to an exception handler
- This issue occurs on two different levels
- On the unit level, there is the question of how the same exception being raised at different points in a unit can be bound to different handlers within the unit
- Eg: In the example subprogram, there is a handler for a division-by-zero exception that appears to be written to deal with an occurrence of division by zero in a particular statement (the one shown)
- But suppose the function includes several other expressions with division operators
- For those operators, this handler would probably not be appropriate
- So, 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

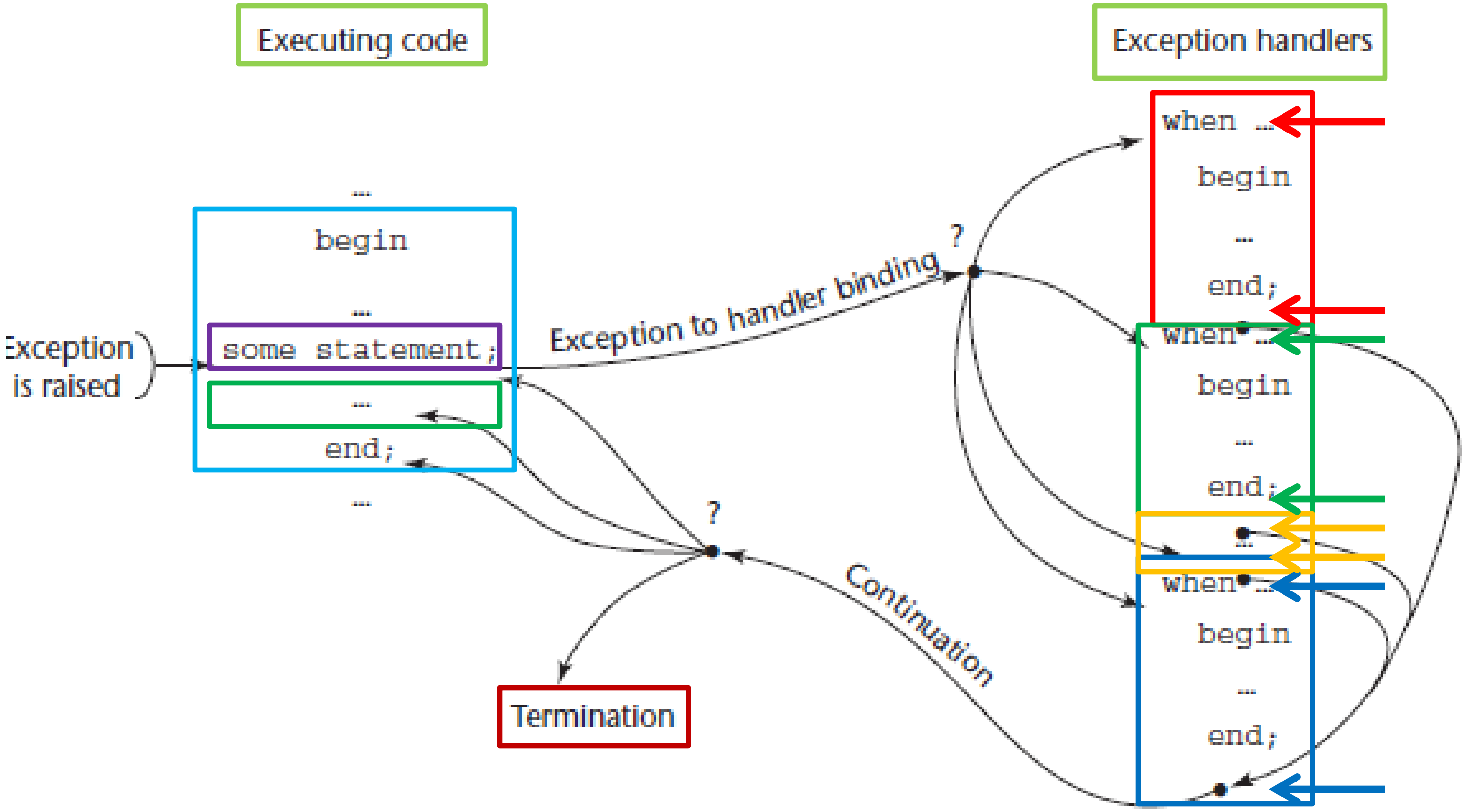
Introduction

- At a higher level, the binding question arises when there is no exception handler local to the unit in which the exception is raised
- In this case, the language designer must decide whether to propagate the exception to some other unit and, if so, where
- How this propagation takes place and how far it goes have an important impact on the writability of exception handlers
- Eg: If handlers must be local, then many handlers must be written, which complicates both the writing and reading of the program
- On the other hand, if exceptions are propagated, a single handler might handle the same exception raised in several program units, which may require the handler to be more general than one would prefer
- An issue that is related to the binding of an exception to an exception handler is whether information about the exception is made available to the handler

Introduction

- After an exception handler executes, either control can transfer to somewhere in the program outside of the handler code or program execution can simply terminate
- We term this the question of control continuation after handler execution, or simply continuation
- Termination is obviously the simplest choice, and in many error exception conditions, the best
- However, in other situations, particularly those associated with unusual but not erroneous events, the choice of continuing execution is best
- This design is called resumption
- In these cases, some conventions must be chosen to determine where execution should continue
- It might be the statement that raised the exception, the statement after the statement that raised the exception, or possibly some other unit
- The choice to return to the statement that raised the exception may seem like a good one, but in the case of an error exception, it is useful only if the handler somehow is able to modify the values or operations that caused the exception to be raised
- Otherwise, the exception will simply be reraised
- The required modification for an error exception is often very difficult to predict
- Even when possible, however, it may not be a sound practice
- It allows the program to remove the symptom of a problem without removing the cause

Introduction



Introduction

- When exception handling is included, a subprogram’s execution can terminate in two ways
    - When its execution is complete or when it encounters an exception
    - In some situations, it is necessary to complete some computation regardless of how subprogram execution terminates
    - The ability to specify such a computation is called *finalization*
    - The choice of whether to support finalization is obviously a design issue for exception handling
- Another design issue is the following
    - If users are allowed to define exceptions, how are these exceptions specified? The usual answer is to require that they be declared in the specification parts of the program units in which they can be raised. The scope of a declared exception is usually the scope of the program unit that contains the declaration

Introduction

- In the case where a language provides predefined exceptions, several other design issues follow. For example, should the language run-time system provide default handlers for the built-in exceptions, or should the user be required to write handlers for all exceptions? Another question is whether predefined exceptions can be raised explicitly by the user program. This usage can be convenient if there are software-detectable situations in which the user would like to use a predefined handler
- Another issue is whether hardware-detectable errors can be handled by user programs. If not, all exceptions obviously are software detectable. A related question is whether there should be any predefined exceptions. Predefined exceptions are implicitly raised by either hardware or system software
- Finally, there is the question of whether exceptions, either predefined or user defined, can be temporarily or permanently disabled. This question is somewhat philosophical, particularly in the case of predefined error conditions
- For example, suppose a language has a predefined exception that is raised when a subscript range error occurs. Many believe that subscript range errors should always be detected, and therefore it should not be possible for the program to disable detection of these errors. Others argue that subscript range checking is too costly for production software, where, presumably, the code is sufficiently error free that range errors should not occur

Introduction

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

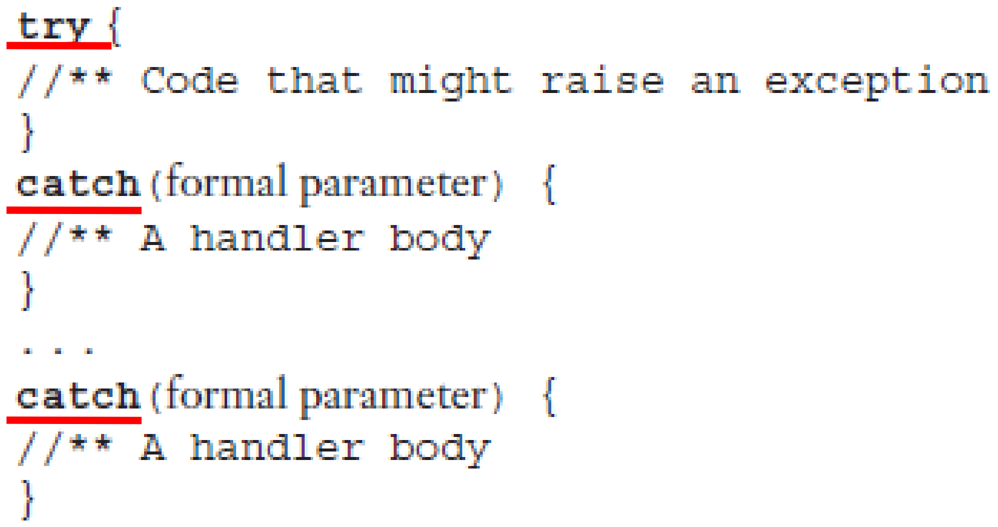
- How and where are exception handlers specified, and what is their scope?
- How is an exception occurrence bound to an exception handler?
- Can information about an exception be passed to the handler?
- Where does execution continue, if at all, after an exception handler completes its execution? (This 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?

Exception Handling in C++

- One major difference between the exception handling of C++ and that of Ada is the absence of predefined exceptions in C++ (other than in its standard libraries)
- Thus, in C++, exceptions are user or library defined and explicitly raised

Exception Handlers

- C++ uses a special construct that is introduced with the reserved word try for this purpose
- A try construct includes a compound statement called the try clause and a list of exception handlers
- The compound statement defines the scope of the following handlers



```
try {
//** Code that might raise an exception
}
catch (formal parameter)  {
//** A handler body
}
. . .
catch (formal parameter)  {
//** A handler body
}
```

Exception Handling in C++

- Each catch function is an exception handler
- A catch function can have only a single formal parameter, which is similar to a formal parameter in a function definition in C++, including the possibility of it being an ellipsis (. . .)
- A handler with an ellipsis formal parameter is the catch-all handler; it is enacted for any raised exception if no appropriate handler was found
- The formal parameter also can be a naked type specifier, such as float, as in a function prototype
  - In such a case, the only purpose of the formal parameter is to make the handler uniquely identifiable
- When information about the exception is to be passed to the handler, the formal parameter includes a variable name that is used for that purpose
- Because the class of the parameter can be any user-defined class, the parameter can include as many data members as are necessary
- In C++, exception handlers can include any C++ code

Exception Handling in C++

Binding Exceptions to Handlers

- C++ exceptions are raised only by the explicit statement throw, whose general form in EBNF is

  throw [expression];
- The brackets here are metasymbols used to specify that the expression is optional
- A throw without an operand can appear only in a handler
- When it appears there, it reraises the exception, which is then handled elsewhere
- The type of the throw expression selects the particular handler, which of course must have a “matching” type formal parameter
- An exception raised in a try clause causes an immediate end to the execution of the code in that try clause
- The search for a matching handler begins with the handlers that immediately follow the try clause
- The matching process is done sequentially on the handlers until a match is found
- This means that if any other match precedes an exactly matching handler, the exactly matching handler will not be used
- Therefore, handlers for specific exceptions are placed at the top of the list, followed by more generic handlers
- The last handler is often one with an ellipsis (. . .) formal parameter, which matches any exception
- This would guarantee that all exceptions were caught

Exception Handling in C++

- If an exception is raised in a try clause and there is no matching handler associated with that try clause, the exception is propagated
- If the try clause is nested inside another try clause, the exception is propagated to the handlers associated with the outer try clause
- If none of the enclosing try clauses yields a matching handler, the exception is propagated to the caller of the function in which it was raised
- If the call to the function was not in a try clause, the exception is propagated to that function's caller
- If no matching handler is found in the program through this propagation process, the default handler is called

Continuation

- After a handler has completed its execution, control flows to the first statement following the try construct (the statement immediately after the last handler in the sequence of handlers of which it is an element)
- A handler may reraise an exception, using a throw without an expression, in which case that exception is propagated

Miscellaneous

```
int main()
{
    try
    {
        try
        {
            throw 20;
        }
        catch (int n)
        {
            cout << "Handle Partially ";
            throw; // Re-throwing an exception
        }
    }
    catch (int n)
    {
        cout << "Handle remaining ";
    }
    return 0;
}
```

In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using "throw;

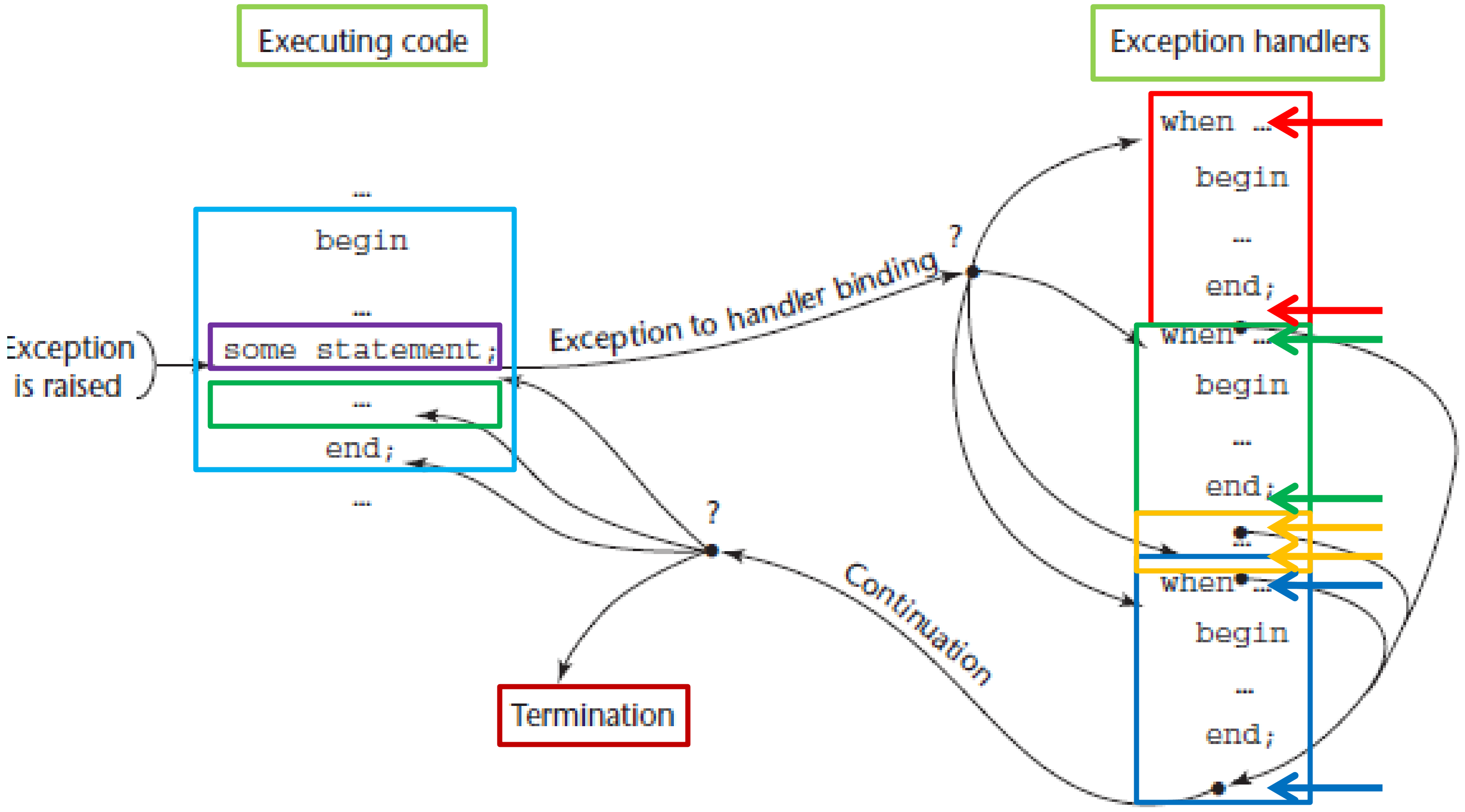
O/P: Handle Partially Handle remaining

```
class Test {
public:
    Test()
    {
       cout << "Constructor of Test " << endl;
    }
    ~Test()
    {
       cout << "Destructor of Test " << endl;
    }
};
int main()
{
    try
    {
        Test t1;
        throw 10;
    }
    catch (int i)
    {
        cout << "Caught " << i << endl;
    }
}
```

When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block

O/P:
```
Constructor of Test
Destructor of Test
Caught 10
```

Introduction



Exception Handling in C++

Other Design Choices

- Exception handling of C++ is simple
- There are only user-defined exceptions, and they are not specified (though they might be declared as new classes)
- There is a default exception handler, unexpected, whose only action is to terminate the program
- This handler catches all exceptions not caught by the program
- It can be replaced by a user-defined handler
    - The replacement handler must be a function that returns void and takes no parameters
    - The replacement function is set by assigning its name to set_terminate
- Exceptions cannot be disabled
- A C++ function can list the types of the exceptions (the types of the throw expressions) that it could raise
- This is done by attaching the reserved word throw, followed by a parenthesized list of these types, to the function header

```
int fun() throw (int, char *) { . . . }
```

- If the function does throw some unlisted exception, the program will be terminated

Miscellaneous

```
// Here we specify the exceptions that this
function throws.

void fun(int *ptr, int x) throw (int *, char)
{
    if (ptr == NULL)
      throw 5;
    if (x == 0)
       throw 6;
}
int main()
{
    try
    {
      fun(NULL, 0);
    }
    catch(...)
    {
       cout << "Caught exception from fun()";
    }
    return 0;
}
```

Program will be terminated

Exception Handling in C++

- If the types in the throw clause are classes, then the function can raise any exception that is derived from the listed classes
- If a function header has a throw clause and raises an exception that is not listed in the throw clause and is not derived from a class listed there, the default handler is called
- Note that this error cannot be detected at compile time
- The list of types in the list may be empty, meaning that the function will not raise any exceptions
- If there is no throw specification on the header, the function can raise any exception
- The list is not part of the function's type
- If a function overrides a function that has a throw clause, the overriding function cannot have a throw clause with more exceptions than the overridden function
- Although C++ has no predefined exceptions, the standard libraries define and throw exceptions, such as out_of_range, which can be thrown by library container classes, and overflow_error, which can be thrown by math library functions

Miscellaneous

```
class FileException
{
   Exception 1: FileNotFound
   Exception 2: End-of-File
};
void fun(int *ptr, int x) throw (int, FileException)
{
    if (ptr == NULL)
       throw Read-Only;
    if (x == 0)
        throw Write-Only;
}
```

```
int main()
{
   Derived d;
   try
  {
       fun(NULL, 0);
   }
  catch(FileException b) ←
  {
        cout<<"Caught Base Exception";
   }
   getchar();
   return 0;
}
```

If a function header has a throw clause and raises an exception that is not listed in the throw clause and is not derived from a class listed there, then the program won't be terminated. Instead, the default handler is called

Miscellaneous

```
// Here we specify the exceptions that this
function throws.

void fun(int *ptr, int x) throw (int *, int)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
}
int main()
{
    try
    {
      fun(NULL, 0);
    }
    catch(...)
    {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

O/P: Caught exception from fun()

```
// Here we specify the exceptions that this
function throws.

void fun(int *ptr, int x) throw ()
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
}
int main()
{
    try
    {
      fun(NULL, 0);
    }
    catch(...)
    {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Compilation Error

Miscellaneous

```
class demo {

};

int main()

{

   try

  {

     throw demo();

   }

   catch (demo d)

   {

     cout << "Caught exception of demo
    class \n";

   }

}
```

O/P: Caught exception of demo class

```
class demo1 {
};

class demo2 {
};

int main()
{
   for (int i = 1; i <= 2; i++) {
      try
      {
         if (i == 1)
            throw demo1();

         else if (i == 2)
            throw demo2();
      }
      catch (demo1 d1)
      {
         cout << "Caught exception of demo1 class
\n";
      }

     catch (demo2 d2)
     {
         cout << "Caught exception of demo2 class
\n";
      }
   }
}
```

O/P: Caught exception of demo1 class
Caught exception of demo2 class

Miscellaneous

```
class demo1 {
};

class demo2 : public demo1 {
};

int main()
{
   for (int i = 1; i <= 2; i++) {
      try
     {
         if (i == 1)
            throw demo1();

         else if (i == 2)
            throw demo2();
      }
      catch (demo1 d1)
     {
         cout << "Caught exception of demo1 class \n";
      }
      catch (demo2 d2)
      {
         cout << "Caught exception of demo2 class \n";
      }
   }
}
```

O/P: Caught exception of demo1 class
Caught exception of demo1 class

```
class demo {
   int num;

public:
   demo(int x)
   {
      try {
         if (x == 0)
            // catch block would be called
            throw "Zero not allowed ";
         num = x;
         show();
      }
      catch (const char* exp) {
         cout << "Exception caught \n ";
         cout << exp << endl;
      }
   }
   void show()
   {
      cout << "Num = " << num << endl;
   }
};

int main()
{
   // constructor will be called
   demo(0);
   cout << "Again creating object \n";
   demo(1);
}
```

O/P: Exception caught
Zero not allowed
Again creating object
Num = 1

Exception Handling in C++

Evaluation

- Unhandled exceptions in functions are propagated to the function's caller
- There are no predefined hardware-detectable exceptions that can be handled by the user, and exceptions are not named
- Exceptions are connected to handlers through a parameter type in which the formal parameter may be omitted
- The type of the formal parameter of a handler determines the condition under which it is called but may have nothing whatsoever to do with the nature of the raised exception
- Therefore, the use of predefined types for exceptions certainly does not promote readability
- It is much better to define classes for exceptions with meaningful names in a meaningful hierarchy that can be used for defining exceptions
- The exception parameter provides a way to pass information about an exception to the exception handler

Introduction to Event Handling

- Event handling is similar to exception handling
- In both cases, the handlers are implicitly called by the occurrence of something, either an exception or an event
- While exceptions can be created either explicitly by user code or implicitly by hardware or a software interpreter, events are created by external actions, such as user interactions through a graphical user interface (GUI)
- In conventional (non–event-driven) programming, the program code itself specifies the order in which that code is executed, although the order is usually affected by the program’s input data
- In event-driven programming, parts of the program are executed at completely unpredictable times, often triggered by user interactions with the executing program
- Particular kind of event handling discussed here is related to GUIs
- Therefore, most of the events are caused by user interactions through graphical objects or components, often called widgets
- The most common widgets are buttons
- Implementing reactions to user interactions with GUI components is the most common form of event handling

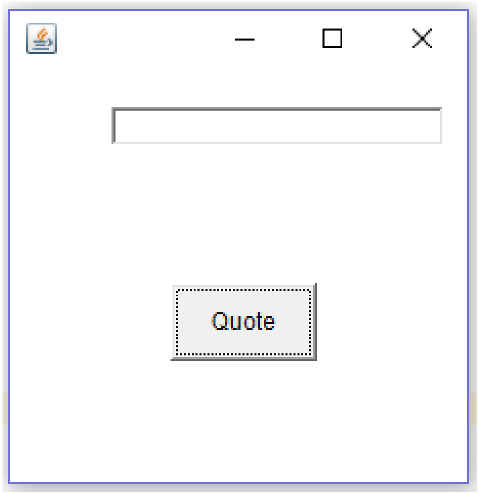
Introduction to Event Handling

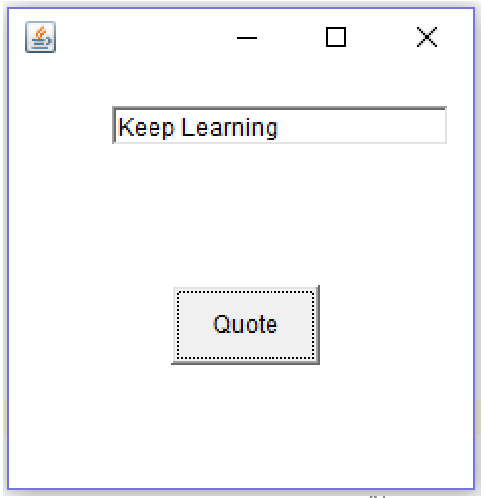
- An event is a notification that something specific has occurred, such as a mouse click on a graphical button
- Strictly speaking, an event is an object that is implicitly created by the run-time system in response to a user action, at least in the context in which event handling is being discussed here
- An event handler is a segment of code that is executed in response to the appearance of an event
- Event handlers enable a program to be responsive to user actions
- Common use of event handlers is to check for simple errors and omissions in the elements of a form, either when they are changed or when the form is submitted to the Web server for processing
- Using event handling on the browser to check the validity of form data saves the time of sending that data to the server, where their correctness then must be checked by a server-resident program or script before they can be processed
- This kind of event-driven programming is often done using a client-side scripting language, such as JavaScript

Miscellaneous

Event Handling

```
import java.awt.*;
import java.awt.event.*;
class  EventHandle extends Frame
      implements ActionListener
{
   TextField textField;
   EventHandle()
   {
     textField = new TextField();
     textField.setBounds(60,50,170,20);
     Button button = new Button("Quote");
     button.setBounds(90,140,75,40);
      //1
      button.addActionListener(this);
      add(button);
      add(textField);
      setSize(250,250);
      setLayout(null);
      setVisible(true);
    }
//2
public void actionPerformed(ActionEvent e)
{
     textField.setText("Keep Learning");
}
 public static void main(String args[])
{
   new EventHandle();
}
}
```




Keep Learning