Concepts of Programming Languages

Lecture 19 - Exception Handling

Patrick Donnelly

Montana State University

Spring 2014

Administrivia

Assignments:

Programming #4 : due 04.28

Reading:

Chapter 14

Ishmael: Surely all this is not without meaning.

Moby Dick by Herman Melville

Purpose

To simplify programming and make applications more robust.

What does robust mean?

Purpose

To simplify programming and make applications more robust.

What does robust mean?

In a language without exception handling

- When an exception occurs, control goes to the operating system, where a message is displayed and the program is terminated

In a language with exception handling

- When an exception occurs, control goes to the operating system, where a message is displayed and the program is terminated

Pascal Fragment

```
(* Pascal - what can go wrong here?*)

reset(file, name);
sum := 0.0;
count := 0;

while (not eof(file)) do begin
    read(file, number);
    sum := sum + number;
    count := count + 1;
end;

ave := sum / count;
```

Basic Concepts

Many languages allow programs to trap input/output errors.

Definition

An ***exception*** is any unusual event, either erroneous or not, detectable by either hardware or software, that may require special processing

Definition

The special processing that may be required after detection of an exception is called ***exception handling***.

Definition

The exception handling code unit is called an ***exception handler***.

Basic Concepts

Definition

An exception is ***raised*** when its associated event occurs.

A language that does not have exception handling capabilities can still define, detect, raise, and handle exceptions (user defined, software detected)

Alternatives:

- Send an auxiliary parameter or use the return value to indicate the return status of a subprogram
- Pass a label parameter to all subprograms (error return is to the passed label)
- Pass an exception handling subprogram to all subprograms

Advantages of Built-in Exception Handling

Error detection code is tedious to write and it clutters the program

Exception handling encourages programmers to consider many different possible errors

Exception propagation allows a high level of reuse of exception handling code

Design Issues

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 the exception be passed to the handler?

Where does execution continue, if at all, after an exception handler completes its execution? (continuation vs. resumption)

Is some form of finalization provided?

How are user-defined exceptions specified?

Should there be default exception handlers for programs that do not provide their own?

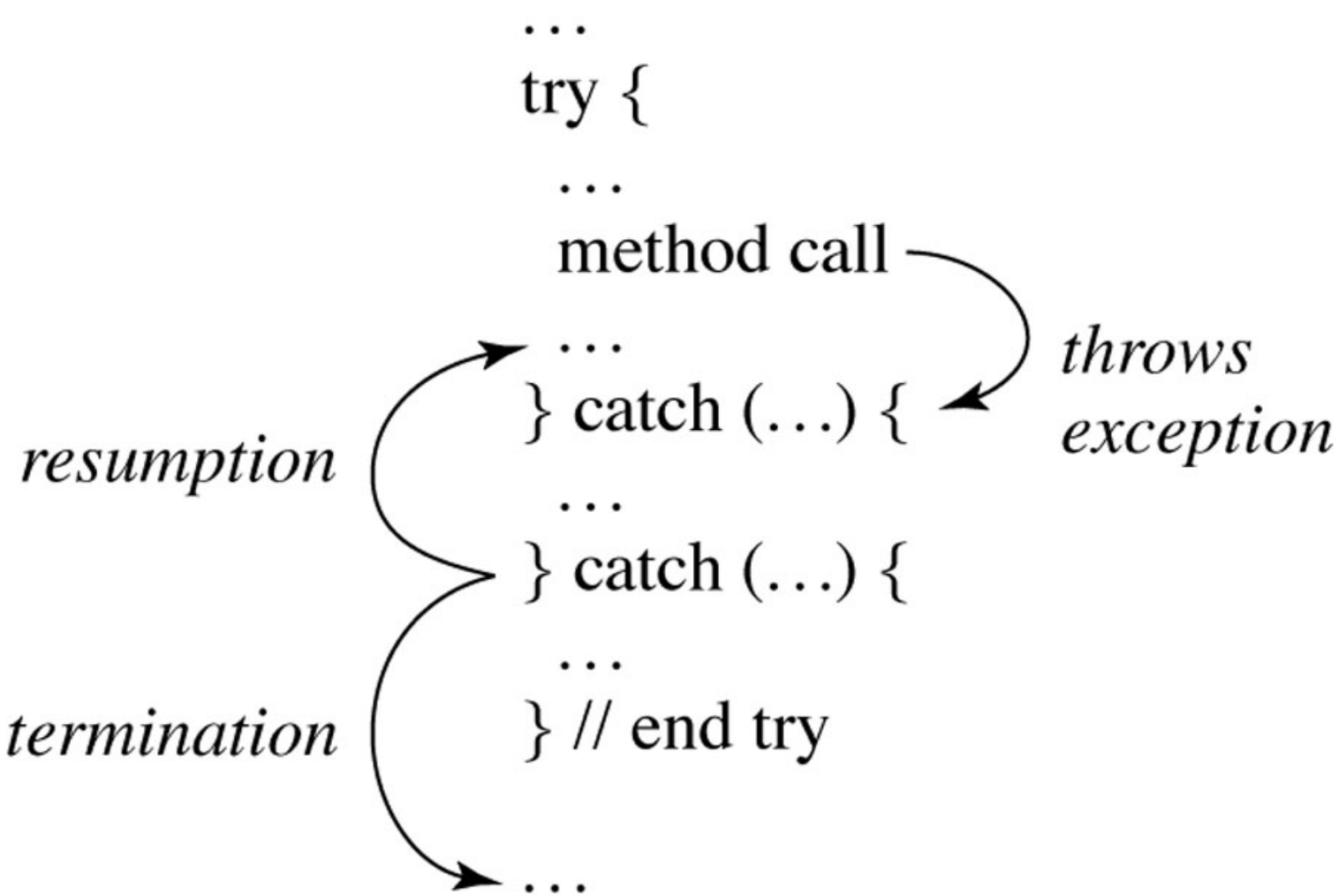
Can predefined exceptions be explicitly raised?

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

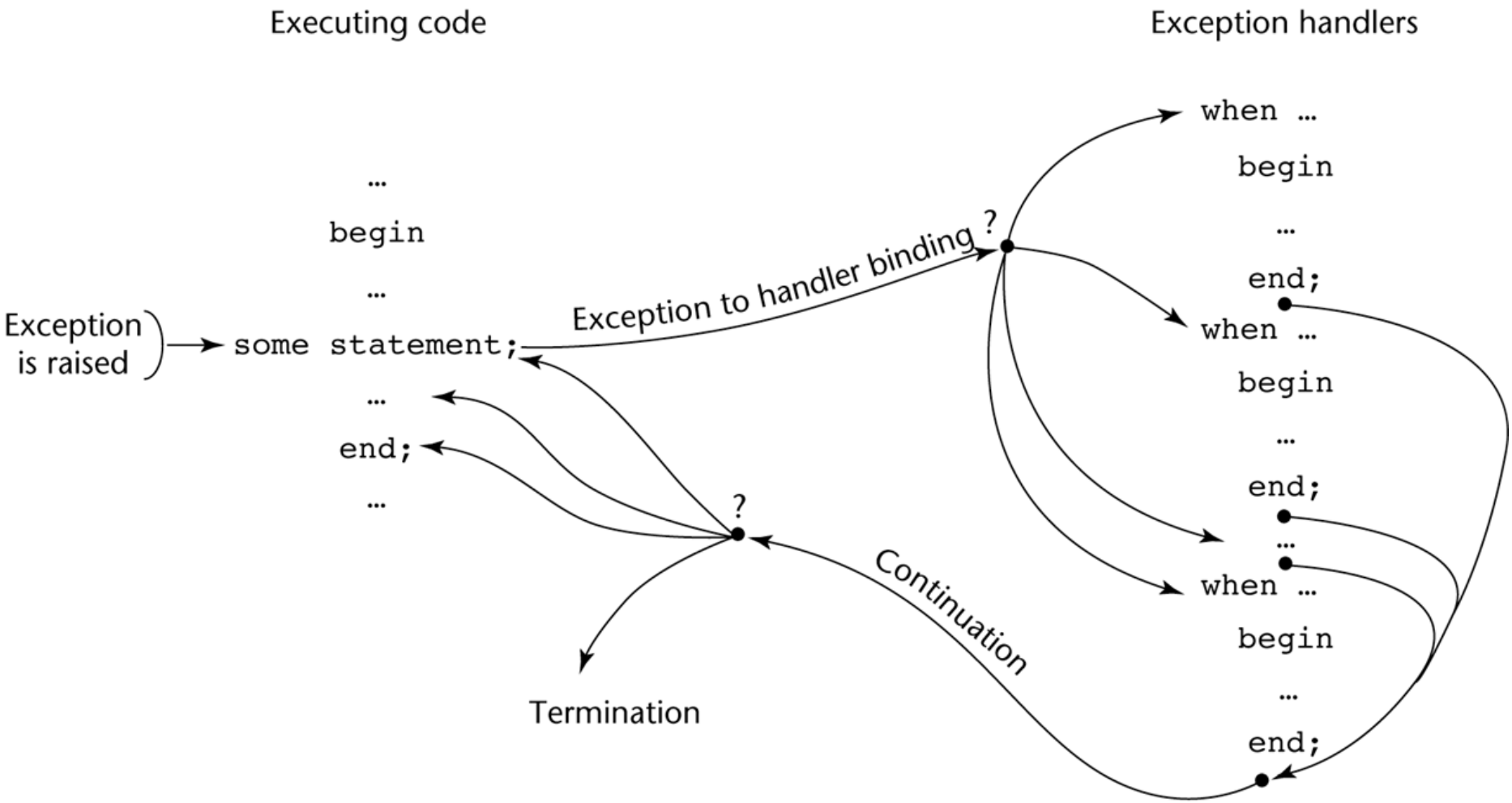
Are there any predefined exceptions?

How can exceptions be disabled, if at all?

Exception Handling



Exception Control Flow



Exception Handling in C++

Added to C++ in 1990

Design is based on that of CLU, Ada, and ML

Exception Handling in C++

Added to C++ in 1990

Design is based on that of CLU, Ada, and ML

Exception Handlers Form:

```
try {
      //  code that is expected
      //  to raise an exception
}catch (formal parameter) {
      // handler code
}
...
catch (formal parameter) {
      //  handler code
}
```

The `catch` Function

`catch` is the name of all handlers–it is an overloaded name, so the formal parameter of each must be unique

The formal parameter need not have a variable

It can be simply a type name to distinguish the handler it is in from others

The formal parameter can be used to transfer information to the handler

The formal parameter can be an ellipsis, in which case it handles all exceptions not yet handled

Throwing Exceptions

Exceptions are all raised explicitly by the statement:

```
throw [expression];
```

The brackets are metasymbols

A `throw` without an operand can only appear in a handler; when it appears, it simply re-raises the exception, which is then handled elsewhere

The type of the expression disambiguates the intended handler

Unhandled Exceptions

An unhandled exception is propagated to the caller of the function in which it is raised

This propagation continues to the main function

If no handler is found, the default handler is called

Continuation

After a handler completes its execution, control flows to the first statement after the last handler in the sequence of handlers of which it is an element

Other design choices:

- All exceptions are user-defined
- Exceptions are neither specified nor declared
- The default handler, `unexpected`, simply terminates the program; `unexpected` can be redefined by the user
- Functions can list the exceptions they may raise
- Without a specification, a function can raise any exception (the `throw` clause)

Evaluation

It is odd that exceptions are not named and that hardware- and system software-detectable exceptions cannot be handled

Binding exceptions to handlers through the type of the parameter certainly does not promote readability

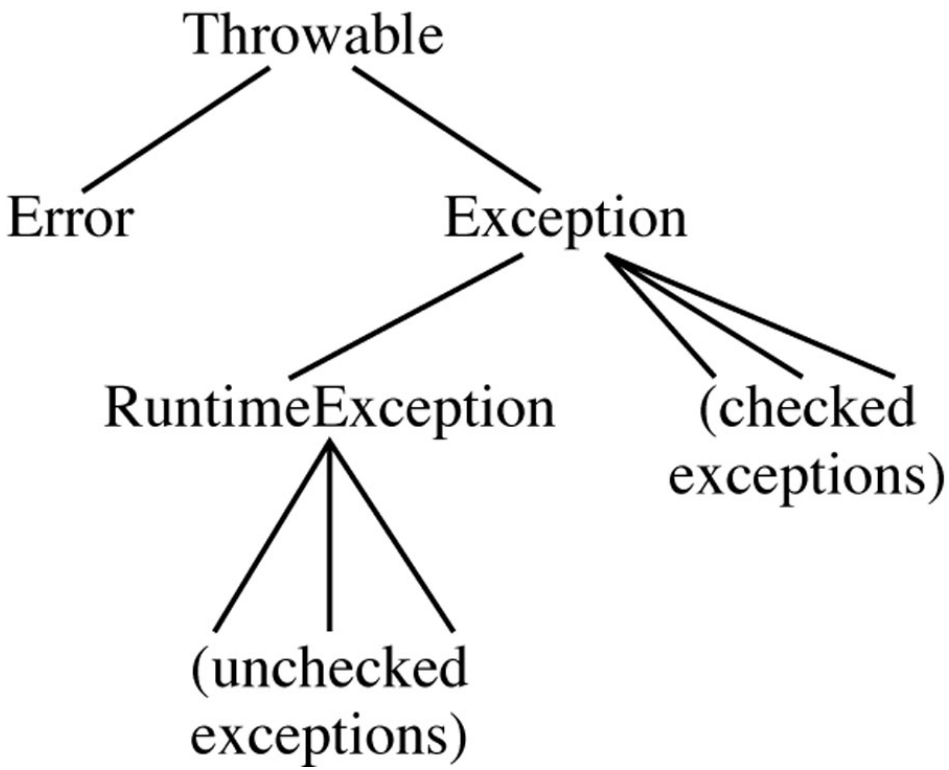
C++ Example

```
#include <iostream.h>
int main () {
  char A[10];
  cin >> n;
  try {
    for (int i=0; i<n; i++){
      if (i>9) throw "array index error";
      A[i]=getchar();
    }
  }
  catch (char* s)
  { cout << "Exception: " << s << endl; }
  return 0;
}
```

Exception Handling in Java

Based on that of C++, but more in line with OOP philosophy

All exceptions are objects of classes that are descendants of the `Throwable` class



Classes of Exceptions

The Java library includes two subclasses of `Throwable`:

`Error`:

- Thrown by the Java interpreter for events such as heap overflow
- Never handled by user programs

`Exception`

- User-defined exceptions are usually subclasses of this
- Has two predefined subclasses, `IOException` and `RuntimeException`
- e.g.,`ArrayIndexOutOfBoundsException` and `NullPointerException`

Java Exception Handlers

Like those of C++, except every catch requires a named parameter and all parameters must be descendants of `Throwable`

Syntax of try clause is exactly that of C++

Exceptions are thrown with throw, as in C++, but often the throw includes the new operator to create the object, as in:

```
throw new MyException();
```

Creating a New Exception Class

```
class StackUnderflowException extends Exception {
   public StackUnderflowException() { super(); }
   public StackUnderflowException(String s){  super(s);}
}
```

Missing Argument Exception

```
public static void main(String[] arg) {
    try {
    if (arg.length < 1)  {
    System.err.println("Missing argument.");
    displayUsage( );
    System.exit(1);
    }
    process(new BufferedReader(new FileReader(arg[0])));
  } catch (FileNotFoundException e) {
    System.err.println("Cannot open file: " + arg[0]);
    System.exit(1);
  }
}
```

Invalid Input Exception

```
while (true) {
  try {
    System.out.print ( "Enter number : ");
    number = Integer.parseInt(in.readLine ( ));
    break;
  } catch (NumberFormatException e) {
    System.out.println ( "Invalid number, please reenter.");
  } catch (IOException e) {
    System.out.println("Input error occurred, please reenter.");
  } // try
} // while
```

Binding Exceptions to Handlers

Binding an exception to a handler is simpler in Java than it is in C++

- An exception is bound to the first handler with a parameter is the same class as the thrown object or an ancestor of it

An exception can be handled and rethrown by including a `throw` in the handler (a handler could also throw a different exception)

Throwing an Exception

```
class Stack {
  int stack[];
  int top = 0;
  ...

  public int pop() throws StackUnderflowException {
    if (top <= 0)
        throw new StackUnderflowException("pop on empty stack");
    return stack[--top];
  }
  ...
}
```

Continuation

If no handler is found in the `try` construct, the search is continued in the nearest enclosing `try` construct, etc.

If no handler is found in the method, the exception is propagated to the method's caller

If no handler is found (all the way to main), the program is terminated

To insure that all exceptions are caught, a handler can be included in any `try` construct that catches all exceptions

Simply use an Exception class parameter

Of course, it must be the last in the `try` construct

Checked and Unchecked Exceptions

The Java throws clause is quite different from the throw clause of C++

Exceptions of class `Error` and `RunTimeException` and all of their descendants are called unchecked exceptions; all other exceptions are called checked exceptions

Checked exceptions that may be thrown by a method must be either:

- Listed in the `throws` clause, or
- Handled in the method

Other Design Choices

A method cannot declare more exceptions in its `throws` clause than the method it overrides

A method that calls a method that lists a particular checked exception in its `throws` clause has three alternatives for dealing with that exception:

- Catch and handle the exception
- Catch the exception and throw an exception that is listed in its own `throws` clause
- Declare it in its `throws` clause and do not handle it

The `finally` Clause

Can appear at the end of a `try` construct

Form:

```
finally {
...
}
```

Purpose: To specify code that is to be executed, regardless of what happens in the `try` construct

Example

A try construct with a finally clause can be used outside exception handling

```
try {
    for (index = 0; index < 100; index++) {
        ...
        if (...) {
            return;
        } //** end of if
    } //** end of try clause
    finally {
        ...
    } //** end of try construct
```

Assertions

Statements in the program declaring a boolean expression regarding the current state of the computation

When evaluated to true nothing happens

When evaluated to false an `AssertionError` exception is thrown

Can be disabled during runtime without program modification or recompilation

Two forms:

- `assert condition;`
- `assert condition: expression;`

AssertException Class

```
class AssertException extends RuntimeException {
    public AssertException( ) { super( ); }

    public AssertException(String s) { super(s); }
}
```

Assert Class

```
class Assert {
   static public final boolean ON = true;
   static public void isTrue(boolean b) {
      if (!b) { throw new AssertException("Assertion failed"); }
   }

   static public void shouldNeverReachHere() {
      throw new AssertException("Should never reach here");
   }
}
```

Using Asserts

```
class Stack {
  int stack[];
  int top = 0;
  ...

  public boolean empty() { return top <= 0; }

  public int pop() {
    Assert.isTrue(!empty());
    return stack[--top];
  }
  ...
}
```

Evaluation

The types of exceptions makes more sense than in the case of C++

The `throws` clause is better than that of C++ (The `throw` clause in C++ says little to the programmer)

The `finally` clause is often useful

The Java interpreter throws a variety of exceptions that can be handled by user programs

Summary

Ada provides extensive exception-handling facilities with a comprehensive set of built-in exceptions.

C++ includes no predefined exceptions

Exceptions are bound to handlers by connecting the type of expression in the `throw` statement to that of the formal parameter of the `catch` function

Java exceptions are similar to C++ exceptions except that a Java exception must be a descendant of the `Throwable` class. Additionally Java includes a `finally` clause