Programming Languages

Exceptions and Continuations

CSCI-GA.2110-001
Spring 2023

Exceptions

General mechanism for handling abnormal conditions

One way to improve robustness of programs is to handle errors. How can we do this?

We can check the result of each operation that can go wrong (e.g., popping from a stack, writing to a file, allocating memory).

Unfortunately, this has a couple of serious disadvantages:

1. it is easy to forget to check
2. writing all the checks clutters up the code and obfuscates the common case (the one where no errors occur)

Exceptions let us write clearer code and make it easier to catch errors.

Predefined exceptions in Ada

- Defined in `Standard`:
  - `Constraint_Error` : value out of range
  - `Program_Error` : illegality not detectable at compile-time: unelaborated package, exception during finalization, etc.
  - `Storage_Error` : allocation cannot be satisfied (heap or stack)
  - `Tasking_Error` : communication failure
- Defined in `Ada.IO_Exceptions`:
  - `Data_Error`, `End_Error`, `Name_Error`, `Use_Error`, `Mode_Error`, `Status_Error`, `Device_Error`

Handling exceptions

Any begin-end block can have an exception handler:

```
procedure Test is
  X: Integer := 25;
  Y: Integer := 0;
begin
  X := X / Y;
exception
  when Constraint_Error =>
      Put_Line("did you divide by 0?");
  when others           =>
      Put_Line("out of the blue!");
end;
```

A common idiom

```
function Get_Data return Integer is
  X: Integer;
begin
  loop
    begin
      Get(X);
      return X;    -- if got here, input is valid,
                   --  so leave loop
    exception
      when others =>
        Put_Line("input must be integer, try again");
         -- will restart loop to wait for a good input
    end;
  end loop;
end;
```

User-defined Exceptions

```
package Stacks is
  Stack_Empty: exception;
  ...
end Stacks;
```

```
package body Stacks is
  procedure Pop (X: out Integer;
                 From: in out Stack) is
  begin
    if Empty(From)
      then raise Stack_Empty;
      else ...
  end Pop;
  ...
end Stacks;
```

The scope of exceptions

- an exception has the same visibility as other declared entities: to handle an exception it must be visible in the handler (e.g., caller must be able to see `Stack_Empty`).
- an `others` clause can handle unnamed exceptions

```
when others =>
  Put_Line("disaster somewhere");
  raise;    -- propagate exception,
            --  program will terminate
```

Exception run-time model

How to propagate an exception:

1. When an exception is raised, the current sequence of statements is abandoned (e.g., current `Get` and `return` in example)
2. Starting at the current frame, if we have an exception handler, it is executed, and the current frame is completed.
3. Otherwise, the frame is discarded, and the enclosing *dynamic* scopes are examined to find a frame that contains a handler for the current exception (want dynamic as opposed to static scopes because those are values that caused the problem).
4. If no handler is found, the program terminates.

Note: The current frame is never resumed.

Exception information

- an Ada exception is a label, not a value: we cannot declare exception variables and then assign to them
- but an exception *occurrence* is a value that can be stored and examined
- an exception occurrence may include additional information: source location of occurrence, contents of stack, etc.
- predefined package `Ada.Exceptions` contains needed machinery

Ada.Exceptions (std libraries)

```
package Ada.Exceptions is
  type Exception_Id is private;
  type Exception_Occurrence is limited private;

  function Exception_Identity (X: Exception_Occurrence)
    return Exception_Id;
  function Exception_Name (X: Exception_Occurrence)
    return String;

  procedure Save_Occurrence
    (Target: out Exception_Occurrence;
     Source: Exception_Occurrence);
  procedure Raise_Exception (E: Exception_Id;
                             Message: in String := "")
  ...
end Ada.Exceptions;
```

Using exception information

```
begin
  ...
exception
  when Expected: Constraint_Error =>
     -- Expected has details
    Save_Occurrence(Event_Log, Expected);

  when Trouble: others =>
    Put_Line("unexpected " &
             Exception_Name(Trouble) &
             " raised");
    Put_Line("shutting down");
    raise;
end;
```

Exceptions in C++

- similar *runtime* model,...
- but exceptions are bona-fide values,
- handlers appear in `try/catch` blocks

```
try {
  some_complex_calculation();
} catch (const RangeError& e) {
  // RangeError might be raised
  //  in some_complex_calculation
  cerr << "oops\n";
} catch (const ZeroDivide& e) {
  // same for ZeroDivide
  cerr << "why is denominator zero?\n";
}
```

Defining and throwing exceptions

The program throws an object. There is nothing needed in the declaration of the type to indicate it will be used as an exception.

```
struct ZeroDivide {
  int lineno;
  ZeroDivide (...) { ... }  // constructor
  ...
};

...
if (x == 0)
  throw ZeroDivide(...);    // call constructor
                             //  and go
```

Exceptions and inheritance

A handler names a class, and can handle an object of a derived class as well:

```
class Matherr { }; // a bare object, no info
class Overflow : public Matherr {...};
class Underflow : public Matherr {...};
class ZeroDivide : public Matherr {...};

try {
  weatherPredictionModel(...);
} catch (const Overflow& e) {
  // e.g., change parameters in caller
} catch (const Matherr& e) {
  // Underflow, ZeroDivide handled here
} catch (...) {
  // handle anything else (ellipsis)
}
```

Rethrowing exceptions in C++

When an exception can be only partially handled it should be rethrown.

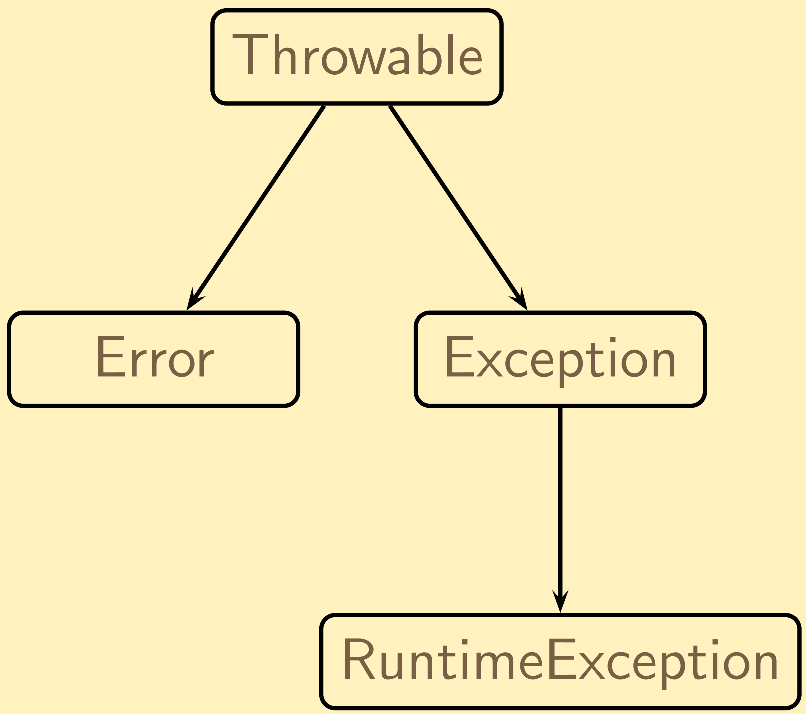
- Rethrow the *same* object, not a new object. This preserves important information such as the stack trace.
- Accomplished with parameterless `throw`.

```
void test(string en) {
  try {
    if (!en.compare("First")) {
      throw MyFirstException();
    } else {
      throw MySecondException();
    }
  } catch (MyException& e) {
    // partially handle here
    throw;
  }
}
```

Exceptions in Java

- Model and terminology similar to C++:
  - exceptions are objects that are thrown and caught
  - `try` blocks have handlers, which are examined in succession
  - a handler for an exception can handle any object of a derived class
- Differences:
  - all exceptions are extensions of predefined class `Throwable`
  - checked exceptions are part of method declaration
  - the `finally` clause specifies clean-up actions
    - in C++, cleanup actions are idiomatically done in destructors
  - `try-with-resources` automatically releases resources upon exit from the `try` block.

Exception class hierarchy



- System errors are extensions of `Error` and `RuntimeException`.
- System errors are *unchecked* exceptions. Examples: `ClassCastException`, `NullPointerException`, `OutOfMemoryError`.
- All other exception classes are *checked*. These exceptions must be either handled or declared in the method that throws them; this is checked by the compiler.

Java "throws" declaration

Checked exceptions must be declared with `throws` if the exception could be emitted by the method:

```
public void replace (String name,
                     Object newValue) throws NoSuch
{
   Attribute attr = find(name);
   if (attr == null) throw new NoSuch(name);
   newValue.update(attr);
}
```

To avoid declaring `NoSuch`, the programmer must catch `NoSuch`.

This exception declaration requirement alerts programmers of the (checked) exceptions that might be thrown when a given method is called.

Mandatory cleanup actions

Some cleanups must be performed whether the method terminates normally or throws an exception.

```
public void parse (String file) throws IOException {
  BufferedReader input =
    new BufferedReader(new FileReader(file));
  try {
    while (true) {
      String s = input.readLine();
      if (s == null) break;
      parseLine(s);  // may fail somewhere
    }
  } finally {
    if (input != null) input.close();
  }  //regardless of how we exit
}
```

Java try-with-resources

- An alternative to `finally`, as of Java 7
- Resources declared at the top of the `try` block are automatically closed regardless of outcome
- Resources only visible within `try` (not `catch`)
- Resources must implement the `AutoCloseable` interface
- Similar to the C# `using` clause and the `IDisposable` interface

```
try (FileInputStream inputStream =
                  new FileInputStream(f);) {
  // use the inputStream to read a file

  } catch (FileNotFoundException e) {
    log.error(e);
  } catch (IOException e) {
    log.error(e);
  }
```

Exceptions in ML

- runtime model similar to Ada/C++/Java
- `exception` is a single type (like a `datatype` but dynamically extensible)
- declaring new sorts of exceptions:

```
exception StackUnderflow
exception ParseError of { line: int, col: int }
```

- raising an exception:

```
raise StackUnderflow
raise (ParseError { line = 5, col = 12 })
```

- handling an exception:

  $expr_1$ `handle` *pattern* `=>` $expr_2$

  If an exception is raised during evaluation of $expr_1$, and *pattern* matches that exception, $expr_2$ is evaluated instead

A closer look

```
exception DivideByZero

fun f i j =
  if j <> 0
    then i div j
    else raise DivideByZero
```

```
(f 6 2
 handle DivideByZero => 42)    (* evaluates to 3 *)
```

```
(f 4 0
 handle DivideByZero => 42)    (* evaluates to 42 *)
```

Typing issues:

- the type of the body and the handler must be the same
- the type of a `raise` expression can be *any type* (whatever type is appropriate is chosen)

Exception Best Practices

- Throw the most specific type possible (e.g. not `Exception`)
- Catch the most specific exception first
- Populate exception objects with useful information
- Do not ignore exceptions when catching them
- Do not use exception handling as an ordinary control structure
- Document the exceptions thrown by any given method (if not Java)
- Log exceptions for later debugging (unless rethrowing)

Call-with-current-continuation

Available in Scheme and SML/NJ; usually abbreviated to `call/cc`.
In Scheme, it is called `call-with-current-continuation`.

A *continuation* represents the computation of "rest of the program".

`call/cc` takes a function as an argument. It calls that function with the current continuation (which is packaged up as a function) as an argument. If this continuation is called with some value as an argument, the effect is as if `call/cc` had itself returned with that argument as its result.

The current continuation is the "rest of the program", starting from the point when `call/cc` returns.

```
(call/cc (lambda (c) (c 5)))         ;; returns 5
(call/cc (lambda (c) 5))             ;; so does this
(call/cc (lambda (c) (+ 1 (c 5)))) ;; ditto
```

The power of continuations

We can implement many control structures with `call/cc`:

- **return**:

```
(lambda (x)
  (call/cc (lambda (ret)
    ...              ;; body of function
    (ret 76)         ;; call continuation with result
    ...
  ))
)
```

- **goto**:

```
(define cont #f)
 (define (object)
  (let ((i 0))

   (call/cc (lambda (k) (set! cont k)))

   ; The next time cont is called, we start here.
   (set! i (+ i 1))
   i))
```

Exceptions via call/cc

Exceptions can also be implemented by `call/cc`:

- Need global stack: `handlers`
- For each `try`/`catch`:

```
(call/cc (lambda (k)
           (begin
             (push handlers (lambda ()
                              (begin
                                (pop handlers)
                                (catch-block)
                                (k ()))))
             (try-block)
             (pop handlers))))
```

- For each `raise`:

```
((top handlers))    ; call the top function on
                    ;  the handlers stack
```