Programming Languages

Exceptions and Concurrency

G22.2110

Summer 2010

Exceptions

General mechanism for handling abnormal conditions

Category	Examples	How raised
predefined	constraint violations, I/O errors, communication errors, other illegalities	by the runtime system
user-defined	pop from empty stack	explicitly by user code

- exception handlers specify remedial actions or proper shutdown
- exceptions can be stored and re-raised later

Error handling

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 unnamable exceptions partially

```
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 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 (part of 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:

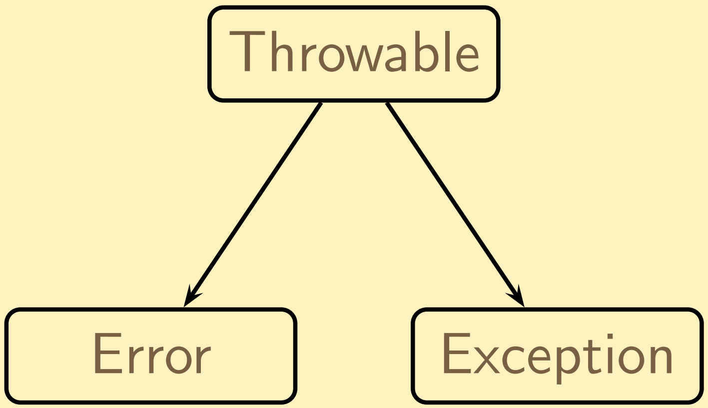
```
class Matherr { }; // a bare object, no info
class Overflow : public Matherr {...};
class Underflow : public Matherr {...};
class ZeroDivide : public Matherr {...};

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)
}
```

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

Exception class hierarchy



- any class extending `Exception` is a *checked* exception
- system errors are extensions of `Error`; these are *unchecked* exceptions

Checked exceptions must be either handled or declared in the method that throws them; this is checked by the compiler.

If a method might throw an exception, callers should know about it

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

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
}
```

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)

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**:

```
(begin
  ...
  (call/cc (lambda (k) (set! here k)) ;; set label
  ...
  (here ())          ;; ``goto'' here
  ...
)
```

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
```

Tasking

- concurrent programming
- declaration, creation, activation, termination
- synchronization and communication
- time and delays
- conditional communication
- non-determinism

Concurrent programming

- synchronous and asynchronous models of communication
- description of concurrent, independent activities
- a *task* is an independent thread of control, with own stack, program counter and local environment.
- Ada tasks communicate through
  - rendezvous (think "meeting someone for a date")
  - protected objects
  - shared variables
- Java threads communicate through shared objects (preferably synchronized)
- C++ has no core language support for concurrency

Task Declarations

A task type is a limited type

```
task type Worker;         -- declaration;
                          --  public interface

type Worker_Id is access Worker;

task body Worker is       -- actions performed in lifetime
begin
  loop                    -- Runs forever;
    compute;              --  will be shutdown
  end loop;               --  from the outside.
end Worker;
```

More Task Declarations

- a task type can be a component of a composite
- number of tasks in a program is not fixed at compile-time.

```
W1, W2: Worker;   -- two individual tasks

type Crew is array (Integer range <>) of Worker;

First_Shift: Crew (1 .. 10); -- group of tasks

type Monitored is record
  Counter: Integer;
  Agent: Worker;
end record;
```

Task Activation

When does a task start running?

- if statically allocated $\Longrightarrow$ at the next `begin`
- if dynamically allocated $\Longrightarrow$ at the point of allocation

```
declare
   W1, W2: Worker;
   Joe: Worker_Id := new Worker; -- Starts working now
   Third_Shift: Crew(1..N);      -- N tasks
begin     -- activate W1, W2, and the Third_Shift
   ...
end;      -- wait for them to complete
          -- Joe will keep running
```

Task Services

- a task can perform some actions on request from another task
- the interface (declaration) of the task specifies the available actions (entries)
- a task can also execute some actions on its own behalf, without external requests or communication

```
task type Device is
  entry Read (X: out Integer);
  entry Write (X: Integer);
end Device;
```

Synchronization: The Rendezvous

- caller makes explicit request: *entry call*
- callee (server) states its availability: *accept statement*
- if server is not available, caller blocks and queues up on the entry for later service
- if both present and ready, parameters are transmitted to server
- server performs action
- `out` parameters are transmitted to caller
- caller and server continue execution independently

Example: semaphore

Simple mechanism to prevent simultaneous access to a *critical section*: code that cannot be executed by more than one task at a time

```
task type semaphore is
  entry P;    -- Dijkstra's terminology
  entry V;    -- from the Dutch
  -- Proberen te verlangen (wait) [P];
  -- verhogen [V] (post when done)
end semaphore;

task body semaphore is
begin
  loop
    accept P;
      -- won't accept another P
      --  until a caller asks for V
    accept V;
  end loop;
end semaphore;
```

Using a semaphore

- A task that needs exclusive access to the critical section executes:

```
Sema : semaphore;
...
Sema.P;
-- critical section code
Sema.V;
```

- If in the meantime another task calls `Sema.P`, it blocks, because the semaphore does not accept a call to `P` until after the next call to `V`: the other task is blocked until the current one releases by making an entry call to `V`.
- programming hazards:
  - someone else may call `V` $\Longrightarrow$ race condition
  - no one calls `V` $\Longrightarrow$ other callers are *livelocked*

Delays and Time

- A `delay` statement can be executed anywhere at any time, to make current task quiescent for a stated interval:

```
delay 0.2;   -- type is Duration, unit is seconds
```

- We can also specify that the task stop until a certain specified time:

```
delay until Noon;   -- Noon defined elsewhere
```

Conditional Communication

- need to protect against excessive delays, deadlock, starvation, caused by missing or malfunctioning tasks
- timed entry call: caller waits for rendezvous a stated amount of time:

```
select
  Disk.Write(Value => 12,
             Track => 123); -- Disk is a task
or
  delay 0.2;
end select;
```

- if `Disk` does not accept within 0.2 seconds, go do something else

Conditional Communication (ii)

- conditional entry call: caller ready for rendezvous only if no one else is queued, and rendezvous can begin at once:

```
select
  Disk.Write(Value => 12, Track => 123);
else
  Put_Line("device busy");
end select;
```

- print message if call cannot be accepted immediately

Conditional communication (iii)

- the server may accept a call only if the internal state of the task is appropriate:

```
select
  when not Full =>
    accept Write (Val: Integer) do ... end;
or
  when not Empty =>
    accept Read (Var: out Integer) do ... end;
or
  delay 0.2;  -- maybe something will happen
end select;
```

- if several guards are open and callers are present, any one of the calls may be accepted – non-determinism

Concurrency in Java

- Two notions
  - `class Thread`
  - `interface Runnable`
- An object of class `Thread` is mapped into an operating system primitive

```
interface Runnable {
  public void run ();
}
```

- Any class can become a thread of control by supplying a `run` method

```
class R implements Runnable { ... }
```

```
Thread t = new Thread(new R(...));
t.start();
```

Threads at work

```
class PingPong extends Thread {
  private String word;
  private int delay;
  PingPong (String whatToSay, int delayTime) {
    word = whatToSay;  delay = delayTime;
  }

  public void run () {
    try {
      for (;;) { // infinite loop
        System.out.print(word + " ");
        sleep(delay); // yield processor
      }
    } catch (InterruptedException e) {
      return; // terminate thread
    }
  }
}
```

Activation and execution

```
public static void main (String[] args) {
  new PingPong("ping", 33).start();  // activate
  new PingPong("pong", 100).start(); // activate
}
```

- call to `start` activates thread, which executes `run` method
- threads can communicate through shared objects
- classes can have synchronized methods to enforce critical sections