

CS558 Programming Languages

Winter 2008

Lecture 9

EXCEPTIONS

Programs often need to handle **exceptional** conditions, i.e., deviations from “normal” control flow.

Exceptions may arise from

- failure of built-in or library operations (e.g., division by zero, end of file)
- user-defined events (e.g., key not found in dictionary)

Awkward or impossible to deal with these conditions explicitly without distorting normal code.

Most recent languages (Ada, C++, Java, etc.) provide a means to **define**, **raise**, and **handle** exceptions.

EXAMPLE: EXCEPTIONS IN JAVA

```
class Help extends Exception {};  
try {  
  ... if (gone wrong) throw new Help();  
  ... x = a / b; ...  
} catch (Help e) {  
  ...report problem...  
} catch (ArithmeticException e) {  
  x = -99;  
}
```

WHAT TO DO IN AN EXCEPTION?

If there is a statically enclosing handler, the thrown exception behaves much like a goto. In previous example:

```
...  
if (gone wrong) goto help_label;  
..  
help_label: ...report problem...
```

But what if there is no handler explicitly wrapped around the exception-throwing point?

- In most languages, uncaught exceptions **propagate** to next **dynamically** enclosing handler. E.g, caller can handle uncaught exceptions raised in callee.
- Many languages permit a value to be returned along with the exception itself.
- A few languages support **resumption** of the program at the point where the exception was raised.

EXCEPTION HANDLING EXAMPLE

```
class BadThing extends Exception {}

int foo () {
  ... throw new BadThing(); ...
}

bar () {
  int x;
  try {
    x = foo ();
  } catch (BadThing e) {
    x = 0;
  }
}
```

EXCEPTIONS VS. ERROR VALUES

An alternative to user-raised exceptions is to return status values, which must be checked on return:

```
fun find (k0:string)
  (env: (string * int) list)
  : int option =
  case env of
  nil => NONE
  | (k,v)::t =>
    if k = k0 then
      SOME v
    else find k0 t

... case find "abc" e0 of
  SOME v => ... v ...
  | NONE => ...error code...
```

EXCEPTION VS. ERROR VALUES (2)

With exceptions, we can defer checking for (rare) error conditions to a more convenient point.

```
exception NotFound
fun find (k0:string)
  (env: (string * int) list)
  : int =
  case env of
  nil => raise NotFound
  | (k,v)::t =>
    if k = k0 then
      v
    else find k0 t

... let val v = find "abc" e0
  in ... v ...
  end
  handle NotFound => ...error code...
```

IMPLEMENTING EXCEPTIONS (1)

One approach to implementing exceptions is for the runtime system to maintain a **handler stack** with an entry for each handler context currently active.

- Each entry contains a handler code address and a call stack pointer.
- When the scope of a handler is entered (e.g. by evaluating a `try...catch`), the handler's address is paired with the current call stack pointer and pushed onto the handler stack.
- When an exception occurs, the top of the handler stack is popped, resetting the call stack pointer and passing control to the handler's code. If this handler itself raises an exception, control passes to the next handler on the stack, etc.
- Selective handlers work by simply re-raise any exception they don't want to handle (causing control to pass to the next handler on the stack).

EXCEPTIONS ON PURPOSE

- In this execution model, raising an exception provides a way to return quickly from a deep recursion, with no need to pop stack frames one at a time.

Example:

```
fun product l =
  let exception Zero
      fun prod l =
        case l of
          nil => 1
        | (h::t) =>
            if h = 0 then raise Zero
            else h * (prod t)
        in (prod l) handle Zero => 0
    end
```

IMPLEMENTING EXCEPTIONS (2)

The handler-stack implementation makes handling very cheap, but incurs cost each time we enter a new handler scope. If exceptions are very rare, this is a bad tradeoff.

- As an alternative, some runtime systems use a **static table** that maps each code address to the address of the statically enclosing handler (if any).
- If an exception occurs, the table is inspected to find the appropriate handler.
- If there is no handler defined in the current routine, the runtime system looks for a handler that covers the return address (in the caller), and so on up the call-stack.
- The deliberate use of exceptions in the previous example would be unwise if this implementation approach is used.