Exceptions

Based on the slides of Maurizio Gabbrielli

Example

- The function **`Average`** Calculates the average of a vector; if the vector is empty, it raises the exception

```
class EmptyExcp extends Throwable {int x=0;};

int average(int[] V) throws EmptyExcp(){
   if (length(V)==0) throw new EmptyExcp();
   else {int s=0; for (int i=0, i<length(V), i++) s=s+V[i];}
   return s/length(V);
};
...
try{...
     average(W);
     ...
}
catch (EmptyExcp e) {write('Array␣empty);␣}
```

Exceptions: Structured Exit

- Terminate part of a computation
  - Exiting from a construct
  - Passing data through the jump
  - Returning control to the newest management point
  - Activation records no longer needed are deallocated
    - Other resources, including heap space, can be freed
- Two constructs
  - Declaration of the exception manager
  - Command/expression to raise the exception

Propagate the Exception

- If the exception is not handled in the current routine:
  - The procedure ends, the exception is re-raised to the call point
  - If the exception is not handled by the caller, the exception is propagated along the chain
  - Until you meet an handler or you reach the top-level, which provides a default handler
- The respective frames are removed from the stack:
  - For each frame that is removed, the status of the registers are restored
- Each routine has a "hidden" handler that restores the state and propagates the exceptions along the stack

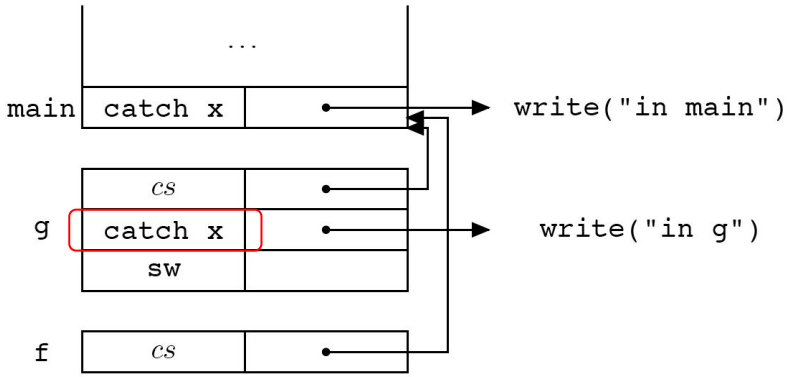
Exception propagates along the dynamic chain

```
{
void f() throws X {
    throw new X();
    }

void g (int sw) throws X {
    if (sw == 0) {f();}
    try {f();} catch (X e) {write("in g");}
    }
...
try {g(1);}
catch (X e)  {write("in main");}
}
```

Which of the two handlers runs?

Exception propagates along the dynamic chain

```
{
void f() throws X {
   throw new X();
   }

void g (int sw) throws X {
   if (sw == 0) {f();}
   try {f();} catch (X e) {write("in g");}
   }
...
try {g(1);}
catch (X e)  {write("in main");}
}
```



Implement exceptions

- Simple:
  - At the beginning of a protected block
    - Put on the stack the handler
  - When an exception is raised:
    - Remove the first handler on the stack and see if it's the right one
    - If not, raise again the exception and repeat
- Inefficient in the (most frequent) case where the exception does not occur:
  - Every try block we have to put and take stuff off the stack

Implement exceptions 2

- Better solution:
  - The compiler generates a table where, for each protected block and for each handler there is a pair <block_addr,handler_addr >
  - The table is sorted on the first element (statically)
  - When an exception is raised:
    - Binary search in the table on the first element
    - Transferring the control to the corresponding handler address
  - If the handler re raise the exception repeat

Finally blocks (Java, C#, Python …)

```
try {
        statements
}     catch (ex-type1 identifier1) {
              statements
} catch (ex-type2 identifier2) {
              statements
}   finally {
              statements
}
```

- Finally of a try block practically always gets executed
- Avoid having cleanup code accidentally bypassed by a return, continue, or break
- + readability and maintainability of code

Finally Example

```
try {
 throw null; // throws NullPointerException!
} catch (Exception e)  {
  int oops = 1/0; // throws ArithmeticException!
} finally {
  System.out.println("Finally!");
  // still gets executed!
}
System.out.println("What about me???");
  // doesn't get executed!
```

Finally and Return in Java

```
int i = 0;
try {
  i = 2;
  return i;
} finally {
  i = 12;
}
```

And what if instead of int you have an object?

What if you “return 12” in the finally?

Finally and Return in Java

- From Specification:

  A return statement with an Expression attempts to transfer control to the invoker of the method that contains it; the value of the Expression becomes the value of the method invocation.

  The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any try statements within the method or constructor whose try blocks contain the return statement, then any finally clauses of those try statements will be executed, in order, innermost to outermost, before control is transferred to the invoker of the method or constructor. Abrupt completion of a finally clause can disrupt the transfer of control initiated by a return statement.

Finally and Return in Java

- With throw in both try and finally block
  - The try block throws SomeException
  - The finally block kicks in and throws SomeOtherException
  - The second throw “wins” and the first exception is discarded

- A return in a finally block will cause an exception thrown from a try block to be discarded
- Throw in a finally block will cause a normal return value from a try block to be discarded

Suggested Exercises

- Chapter 7 exercises 6-9 (higher order + exceptions)