

Preconditions

- Functions often have requirements on their inputs

```
// Return maximum element in A[i..j]
int findMax(int[] A, int i, int j) { ... }
```

- A is nonempty
 - A isn't null
 - i and j must be nonnegative
 - i and j must be less than A.length
 - i < j (maybe)
- These are called *preconditions*

CMSC 330: Organization of Programming Languages

Exceptions

Signaling Errors

- Style 1: Return invalid value

```
// Returns value key maps to, or null if no
// such key in map
Object get(Object key);
```

- Disadvantages?

Dealing with Errors

- What do you do if a precondition isn't met?
- What do you do if something unexpected happens?
 - Try to open a file that doesn't exist
 - Try to write to a full disk

Problems with These Approaches

- What if all possible return values are valid?
 - E.g., `findMax` from earlier slide
 - What about errors in a constructor?
- What if client forgets to check for error?
 - No compiler support
- What if client can't handle error?
 - Needs to be dealt with at a higher level
- Poor modularity- exception handling code becomes scattered throughout program
- 1996 Ariane 5 failure classic example of this ...

Signaling Errors (cont'd)

- Style 2: Return an invalid value and status

```
static int lock_rdev(mdk_rdev_t *rdev) {
    ...
    if (bdev == NULL)
        return -ENOMEM;
    ...
}

// Returns NULL if error and sets global
// variable errno
FILE *fopen(const char *path, const char *mode);
```

Why Ariane 5 failed

- SRI tried to convert a floating point number out of range to integer. Therefore it issued an error message (as a 16 bit number). This 16 bit number was interpreted as an integer by the guidance system and caused the nozzle to move accordingly.
 - The backup SRI performed according to specifications and failed for the same reason.
- Ada range checking was disabled since the SRI was supposedly processing at 80% load and the extra time needed for range checking was deemed unnecessary since the Ariane 4 software worked well.
- The ultimate cause of the problem was that the Ariane 5 has a more pronounced angle of attack and can move horizontally sooner after launch. The “bad value” was actually the appropriate horizontal speed of the vehicle.

Ariane 5 failure

Design issues: In order to save funds and ensure reliability, and since the French Ariane 4 was a successful rocket, the Inertial Reference System (SRI) from Ariane 4 was reused for the Ariane 5.

What happened?: On June 4, 1996 the Ariane 5 launch vehicle failed 39 seconds after liftoff causing the destruction of over \$100 million in satellites.

Cause of failure: The SRI, which controls the attitude (direction) of the vehicle by sending aiming commands to the rocket nozzle, sent a bad command to the rocket causing the nozzle to move the rocket toward the horizontal.

The vehicle tried to switch to the backup SRI, but that failed for the same reason 72 millisecond earlier.

The vehicle had to then be destroyed.

Throwing an Exception

- Create a new object of the class `Exception`, and **throw** it

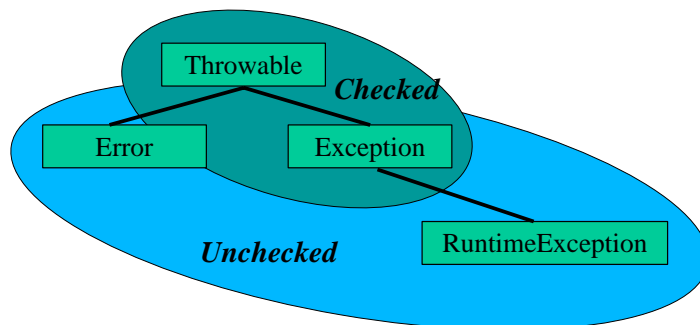
```
if ( i >= 0 && i < a.length )
    return a[i];
throw new ArrayIndexOutOfBoundsException();
```

- Exceptions thrown are part of the return type in Java
 - When overriding method in superclass, cannot throw any more exceptions than parent's version

Better approaches: Exceptions in Java

- On an error condition, we *throw* an exception
- At some point up the call chain, the exception is *caught* and the error is handled
- Separates normal from error-handling code
- A form of non-local control-flow
 - Like goto, but structured

Exception Hierarchy



Method throws declarations

- A method declares the exceptions it might throw
 - `public void openNext() throws`
`UnknownHostException, EmptyStackException`
`{ ... }`
- Must declare any exception the method might throw
 - Unless it is caught in (masked by) the method
 - Includes exceptions thrown by called methods
 - Certain kinds of exceptions excluded

Exception Handling

- All exceptions eventually get caught
- First **catch** with supertype of the exception catches it
- **finally** is always executed

```
try { if (i == 0) return; myMethod(a[i]); }
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("a[] out of bounds"); }
catch (MyOwnException e) {
    System.out.println("Caught my error"); }
catch (Exception e) {
    System.out.println("Caught" + e.toString()); throw e; }
finally { /* stuff to do regardless of whether an exception */
        /* was thrown or a return taken */ }
```

Unchecked Exceptions

- Subclasses of **RuntimeException** and **Error** are unchecked
 - Need not be listed in method specifications
- Currently used for things like
 - **NullPointerException**
 - **IndexOutOfBoundsException**
 - **VirtualMachineError**
- Is this a good design?

Development of Exns (cont'd)

- PL/I (mid-1960s)
 - “ON-units” can be established for a block to handle one of 23 predefined errors
 - Most recently established ON-unit used for condition
 - And exit from a block cancels ON-units for it
 - Like nested try/catch blocks
- Turned out to be difficult to use
 - Needed to use global variables to communicate with exception handling code
 - Fixup actions occurred where ON statement executed
 - Led to some strange behavior

The Development of Exceptions

- Lisp 1.5 (mid-1950s)
 - Certain built-in functions could trigger errors
 - Would result in program termination or debugger entry
 - **errset e**
 - Evaluate **e**, and ignore any errors that occurred
 - Key: wanted to be able to continue execution after an error

Development of Exns (cont'd)

- CLU (mid-1970s)
 - A language designed for data abstraction

```
proc_name = proc (formals)
    signals (exn_1 (params), ...
           exn_n (params))
-- procedure body --
end proc_name
```

Exns can have params

Exceptions part of interface

Source: Hutto, notes on exceptions

Development of Exns (cont'd)

- Throw within ON-unit resulted in recursion

```
FACTORIAL: PROC;
DCL (N INIT(1) ,F INIT(1)) FIXED BIN (31);
ON FINISH BEGIN;
    PUT LIST (N, "FACTORIAL =", F) ;
    PUT SKIP;
    N = N + 1;
    F = N * F
    STOP;
END;
STOP;
END FACTORIAL;
```

Raise FINISH condition
(jump to top of loop,
indefinitely until overflow)

Source: MacLaren, Exception Handling in PL/I

Development of Exns (cont'd)

- Ada (late 1970s)
 - Exceptions independent of procedures
 - But can't have parameters
 - Exceptions propagate through call chain until they are handled

Development of Exns (cont'd)

- Handling exceptions in CLU

```
statement
except
    exn_1 (params) : stmt_1
    ...
    exn_n (params) : stmt_n
end
```

Source: Hutto, notes on exceptions

- Exception *must* be handled in caller
 - If not, built-in exception *failure* is raised

Implementing Exns in Java (cont'd)

```
void foo();
Code:
 0:  aconst_null
 1:  astore_1
 2:  aload_1
 3:  invokevirtual #2;
    //hashCode
 6:  pop
 7:  goto 19
```

```
10:  astore_1
11:  getstatic #4; //System.out
14:  ldc #5; //String caught
16:  invokevirtual #6; //println
19:  return
Exception table:
 from  to  target type
  0     7     10  NullPointerException
```

- Exception table tells JVM what handlers there are for which region of code
 - Notice that putting this “off to the side” keeps it out of the main code path
 - (Though less important for an interpreted language)

CMSC 330

22

Implementing Exceptions in Java

- JVM knows about exceptions, and has built-in mechanism to handle them

```
public class A {
    void foo() {
        try {
            Object f = null;
            f.hashCode();
        }
        catch (NullPointerException e) {
            System.out.println("caught");
        }
    }
}
```

CMSC 330

21

Implementing Exns in C++ (cont'd)

- Some ideals:
 - Type-safe transmission of arbitrary data from throw to handler
 - No added cost to code that doesn't throw exns
 - Handlers can be written to catch multiple exns
 - Exceptions should work in multi-threaded programs

CMSC 330

24

Implementing Exns in C++

- C++ has exception mechanism similar to Java
 - Differences: arbitrary objects can be thrown;
 - C++ code is compiled, so need to generate code for exception handling
- Key assumptions (from Stroustrup, Design and Evolution of C++)
 - Exceptions are primarily for error handling
 - Exception handlers rare compared to function defs
 - Exceptions are infrequent compared to function calls
 - Exceptions should be part of the language

CMSC 330

23

Implementing Exns in C++ (cont'd)

- Design battle: resumption vs. termination
 - Resumption: an exception handler can resume computation at the place where the exception was thrown
 - Termination: throwing an exception terminates execution at the point of the exception
- C++ settled on termination
 - What do you think?