Exceptions

- Definition: Unexpected condition that cannot easily be handled locally
- Examples:
  - Array index out of bounds
  - Null pointer dereference
  - Problem during I/O
- Alternative mechanisms:
  - Funny return value
  - Global variable ("errno")
  - Pass closure

Java Example

class ExceptionExample {
    static void m(boolean b) {
        if (!b)
            throw new RuntimeException("!B");
        System.out.println("M " + b);
    }
    public static void main(String[] args) {
        try {
            m("true".equals(args[0]));
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("A");
        } catch (RuntimeException e) {
            System.out.println("R " + e.getMessage());
        } finally {
            System.out.println("F");
        }
    }
}

Rules for Exceptions

- Abrupt control transfer to first matching catch clause ("handler")
- While no handler in current method: unwind stack, look in caller
- On the way, execute "finally" blocks
  - Also execute finally blocks when no exception (fall through or return)

Exceptions in catch or finally

Legend:
N = normal
A = abrupt
V, R, S = reasons

Output for Example

dyn9002089027:/tmp> javac ExceptionExample.java
dyn9002089027:/tmp> java ExceptionExample true
M true
T

A
R

A
S

A
R

A
S

N

finally[N]

try

A
V

V

finally[R]

A
R

N

N

finally[N]

exit[N]

finally[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N]

exit[N].
Exception Chaining

class R extends Exception { R(Throwable x) { super(x); } }

catch (final V v) {
  System.out.println("V " + v.getMessage());
  throw new R(v);
}

Exception in thread "main" R: V: (... original message ...)
Caused by: V: (... original message ...)

Checked Exceptions

Must be caught locally or declared in throws clause

Checked Exceptions

• Must be caught locally or declared in throws clause

Concurrency

• Also known as multi-threading
  • Thread = language level active entity
    – Own program counter (PC) and stack
    – Can run at the same time as (concurrently with) other threads
    – May share memory (globals, heap) with other threads
  • Ada "task" is roughly same as Java "thread"
  – Scott book uses "task" for different concept

Example

http://www.bowdoin.edu/~allen/pl/parallel/BoundedBuffer.java

Sequence Diagram

main:MainThread
producer:Producer
consumer:Consumer
Thread States

- **new**
- **created**
- **start**
- **runnable** (on ready queue)
- **running**
- **blocked**
- **yield**
- **return from run**
- **schedule**
- **enter sync./wait/sleep**
- **leave sync.**
- **notify/notifyAll**
- **terminated**

---

### Buffer, version 1 (buggy!)

1. ```java
class Buffer {
    private Date[] buf;
    private int in = -1, out = -1, count = 0;
    public Buffer(int s) {
        buf = new Date[s];
    }
    public synchronized void put(Date d) {
        while (count >= buf.size) {
            // do nothing
        }
        count++;
        buf[++in % buf.size] = d;
    }
    public synchronized Date get() {
        while (count == 0) {
            // do nothing
        }
        count--;
        Date d = buf[++out % buf.size];
        //
        return d;
    }
    }
```  

**Race Condition**

- Multiple concurrent non-atomic accesses to shared data, at least one is a write
  - Threads are “racing” against each other
- Difficult to debug:
  - Silent (may crash much later, or just produce wrong answer)
  - Non-deterministic (hard to reproduce, hard to fix with confidence)
- Critical section = code that temporarily violates invariant on shared data

---

### Buffer, version 2 (still buggy!)

1. ```java
class Buffer {
    private Date[] buf;
    private int in = -1, out = -1, count = 0;
    public Buffer(int s) {
        buf = new Date[s];
    }
    public synchronized void put(Date d) {
        while (count >= buf.size) {
            // do nothing
        }
        count++;
        buf[++in % buf.size] = d;
    }
    public synchronized Date get() {
        while (count == 0) {
            // do nothing
        }
        count--;
        Date d = buf[++out % buf.size];
        //
        return d;
    }
    }
```  

**Race Condition**

- Multiple concurrent non-atomic accesses to shared data, at least one is a write
  - Threads are “racing” against each other
- Difficult to debug:
  - Silent (may crash much later, or just produce wrong answer)
  - Non-deterministic (hard to reproduce, hard to fix with confidence)
- Critical section = code that temporarily violates invariant on shared data

---

### Synchronized in Java

- Lock associated with object
  - In case of synchronized method: “this”, i.e., the shared buffer
- When thread attempts to enter:
  - If another thread active: block
- When thread leaves:
  - If another thread waiting, wake it up
- Achieves mutual exclusion

---

### Buffer, version 1 (buggy!)

- Class `Buffer` is declared with a private `Date[]` array.
- Constructor initializes the array.
- `put` method is synchronized to prevent data races.
- `get` method is synchronized to prevent data races.

### Buffer, version 2 (still buggy!)

- Synchronized version of `Buffer` with potential race conditions.
- Setting `in` and `out` indices using `% buf.size` to handle wrap-around.
- Potential data race when multiple threads access `buf` simultaneously.

---

**Race Condition**

- Code that temporarily violates invariant on shared data
- Download buffer content using `get` function
- Blocks or releases lock
- Method returns a `Date` object

---

**Buffer, version 2 (still buggy!)**

- Synchronized version of `Buffer` with potential race conditions.
- Setting `in` and `out` indices using `% buf.size` to handle wrap-around.
- Potential data race when multiple threads access `buf` simultaneously.

---

**Syncrhonized in Java**

- Lock associated with object
  - In case of synchronized method: “this”, i.e., the shared buffer
- When thread attempts to enter:
  - If another thread active: block
- When thread leaves:
  - If another thread waiting, wake it up
- Achieves mutual exclusion

---

**Race Condition**

- Multiple concurrent non-atomic accesses to shared data, at least one is a write
  - Threads are “racing” against each other
- Difficult to debug:
  - Silent (may crash much later, or just produce wrong answer)
  - Non-deterministic (hard to reproduce, hard to fix with confidence)
- Critical section = code that temporarily violates invariant on shared data
Deadlock

- Producer loops until there is a free slot; but since producer holds lock, consumer cannot create free slot
- Coffman conditions:
  - Mutual exclusion
  - Hold-and-wait
  - No preemption
  - Circular wait
- To fix example: wait, but don't hold
  - Condition notification

Buffer, version 3 (correct)

```java
1. class Buffer {
2. private Date[] buf;
3. private int in = -1, out = -1, count = 0;
4. public Buffer(int s) { buf = new Date[s]; }
5. public synchronized void put(Date d) {
6.   while (count >= buf.size) wait();
7.   count++;
8.   buf[++in % buf.size] = d;
9.   notify();
10. }
11. public synchronized Date get() {
12.   while (count == 0) wait();
13.   count--;
14.   Date d = buf[++out % buf.size];
15.   notify();
16.   return d;
17. }
18. }
```

How to write correct concurrent code

- Don't write concurrent code
- If you have to write concurrent code: don't use shared memory
- If you have to use shared memory:
  - Share immutable data
  - Reuse libraries of concurrent containers
- If you can't reuse libraries:
  - Read a book
  - Use annotations

Shared Memory Concurrency

<table>
<thead>
<tr>
<th>Java</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>subclass of Thread</td>
<td>task type</td>
</tr>
<tr>
<td>instance of a subclass of Thread</td>
<td>task, or instance of task type</td>
</tr>
<tr>
<td>Class where all methods are synchronized</td>
<td>protected type</td>
</tr>
<tr>
<td>while (tc) wait();</td>
<td>wait(c);</td>
</tr>
<tr>
<td>notify();</td>
<td>signal(c);</td>
</tr>
</tbody>
</table>

Message Passing

- Concurrent programming without shared memory (e.g., distributed)
- Prevents data races, can still have deadlocks
- In C or Fortran: MPI
- In Java:
  - Libraries (sockets)
  - Frameworks (J2EE)
- In Ada: part of language
  - task has 'entry' subroutines
  - Call blocked until task does "accept"

Bounded Buffer with Message Passing

(Scott Figure 12.21)