Capsule-oriented Programming
Concurrency is a Modularity problem.
Concurrency features may affect every module’s invariants thus breaking modular reasoning.
We must **restore** modular reasoning.
Actors can send messages to other Actors
Actors can create a finite number of new Actors
Actors can send new Actors as messages
Actors can run code in response to messages
Actors have local heap and stacks
Actors can send asynchronous messages
Actors can send synchronous messages
Messages are buffered
Message delivery is guaranteed
Message ordering is guaranteed
Message fairness is guaranteed

YMMV.
_ can send messages to other _
_ can create a finite number of new _
_ can send _ as messages
_ can run code in response to messages
_ have local heap and stacks
_ can send asynchronous messages
_ can send synchronous messages
messages are buffered
message delivery is guaranteed
message ordering is guaranteed
message fairness is guaranteed

Confinement is key.

_ = Actors

✓
✓
✓
✓
✓
✓
✓
✓
✓

Confinement is key.
Problems Concretely

_ = Actors

Impedance mismatch for sequential programmers

- Have local heap and stacks
- Can send asynchronous messages
- Can send synchronous messages
- Messages are buffered
- Message delivery is guaranteed
- Message ordering is guaranteed
- Message fairness is guaranteed

C++ looks like C. Java/C# look like C++.
Actors can send messages to other actors.
Actors can create a finite number of new actors.
Actors can send actions as messages.

Message fairness is guaranteed.

Message delivery is guaranteed.
Message ordering is guaranteed.

Harder to build precise, efficient analyses for safety, compilation, etc...
How to solve it?
<table>
<thead>
<tr>
<th>Feature</th>
<th>Actors</th>
<th>Capsules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can send messages to other</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Can create a finite number of new</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Can send _ as messages</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Can run code in response to messages</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Have local heap and stacks</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Can send asynchronous messages</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Can send synchronous messages</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Messages are buffered</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Message delivery is guaranteed</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Message ordering is guaranteed</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Message fairness is guaranteed</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
A Concrete Example: Panini

Capsule-oriented programming on the JVM.
Capsule-oriented programming entails

- dividing program logic into capsules
- composing capsules into a system
capsule Buffer {
  ...
  
} capsule declaration
capsule Buffer {
  ...
}

capsule declaration

reuse by composition
capsule Buffer {

    ArrayList list = new ArrayList();

    ...

}
capsule Buffer {

  ArrayList list = \textbf{new} ArrayList();

  ...

} \textbf{state declaration}

\textbf{use existing OO code as is}
capsule Buffer {
  ArrayList list = new ArrayList();
  ...
}

states confined by design

use existing OO code as is

state declaration
capsule Buffer {

    ArrayList list = new ArrayList();

    Integer count() {
        return list.size();
    }

    Boolean add(Object o) {
        ...}

}
capsule Buffer {

ArrayList list = new ArrayList();

Integer count() {
  return list.size();
}

Boolean add(Object o) {
  ...
}

procedure declarations

args & return values confined by design
capsule Buffer {

    ArrayList list = new ArrayList();

    Integer count() {
        return list.size();
    }

    Boolean add(Object o) { ... }

}
capsule Buffer {

ArrayList list = new ArrayList();

Integer count() {
    return list.size();
}

Boolean add(Object o) { ... }
}
system Exchange {
    ...
}

system declaration
capsule Buffer { ... }  
capsule Producer (Buffer b) { ... }  
capsule Consumer (Buffer b) { ... }  

system Exchange {

Buffer b;  
Producer p; Consumer c;  

...  

}
system Exchange {
    Buffer b;
    Producer p; Consumer c;

    p(b);

    c(b);
}

capsule instance wiring

p b c
Are they solved?
We have refactored 134,000+ LOC of OO programs to capsule-oriented programs. JG, NAS, StreamIt,…

Freshmen wrote some of these capsule-oriented programs.
How is the performance?
Where are we going?
0.9.0胶囊，系统

0.9.1更好的顺序一致性

0.9.2更好的编译策略

0.9.3更好的隔离分析（错误）

0.9.4托管人用于共享

Open Source MPL 1.1
Concurrency is a Modularity problem, reconciling them could be useful for sequential programmers.
Java program with **threads** and **synchronization** to compute \(\pi\)

```java
class Worker implements Runnable {
    private final CountDownLatch doneSignal;
    Worker(long num, CountDownLatch doneSignal) {
        this.num = num;
        this.doneSignal = doneSignal;
    }
    Random prng = new Random();
    Number _circleCount = null; //Emulates return value of worker
    public void run() {
        _circleCount = new Number(0);
        for (long j = 0; j < num; j++) {
            double x = prng.nextDouble();
            double y = prng.nextDouble();
            if (((x + x + y * y) < 1) _circleCount.incr();
        }
        doneSignal.countDown();
    }
    class Master {
        void assign(long totalCount, int numWorkers) {
            CountDownLatch I = new CountDownLatch(numWorkers);
            Worker[] workers = new Worker[numWorkers];
            for (int i = 0; i < numWorkers; i++)
                workers[i] = new Worker(totalCount / numWorkers, I);
            new Thread(workers[]).start();
        }
        try {
        I.await();
    } catch (InterruptedException e) { // Error recovery */
        Number[] results = new Number[numWorkers];
        for (int i = 0; i < numWorkers; i++)
            results[i] = workers[i].getCircleCount();
        long total = 0;
        for (Number result: results) total += result.value();
        double pi = 4.0 * total / totalCount;
    }
    public class PI {
        public static void main(String[] args) {
            Master master = new Master();
            master.assign(60000000, 10);
        }
    }
}
```

Panini program to compute \(\pi\)

```java
capsule Worker (int num) {
    Random prng = new Random();
    Number compute() {
        Number _circleCount = new Number(0);
        for (int i = 0; i < num; i++) {
            double x = prng.nextDouble();
            double y = prng.nextDouble();
            if (((x + x + y * y) < 1) _circleCount.incr();
        }
        return _circleCount;
    }
}
capsule Master (int total, Worker[] workers) {
    Number[] results = new Number(workers.length);
    for (int i = 0; i < workers.length; i++)
        results[i] = workers[i].compute(); // Inter-capsule procedure calls, return a duck immediately
    long total = 0;
    for (Number result: results) total += result.value(); // Ducks are transparent, no refactoring
    double pi = 4.0 * total / totalCount;
}
system PI {
    Master master; Worker workers[10];
    master(50000000, workers);
    for (Worker w: workers) w(50000000);
}
```

**Cleaner code**, **Compiler-checked** Familiar semantics

Complex code

No correctness guarantees

Performance results

**Similar Performance Benefits**