CS 181E: Fundamentals of Parallel Programming

Instructor: Vivek Sarkar

Co-Instructor: Ran Libeskind-Hadas

http://www.cs.hmc.edu/courses/2012/fall/cs181e/

Recap of Lectures 7 and 8

<u>Lecture 7</u>:

- Critical Sections and the Isolated Statement
- Atomic Variables

<u>Lecture 8</u>:

- Observationally Cooperative Multithreading
  - Code between two successive yield operations is isolated by default

Worksheet #8 solution: Insertion of isolated for correctness

The goal of IsolatedPRNG is to implement a single Pseudo Random Number Generator object that can be shared by multiple tasks. Show the isolated statement(s) that you can insert in method nextSeed() to avoid data races and guarantee proper semantics.

```
class IsolatedPRNG {
  private int seed;
  public int nextSeed() {
    int retVal;
    isolated {
     retVal = seed;

     seed = nextInt(retVal);
    }
    return retVal;
  } // nextSeed()
  . . .
} // IsolatedPRNG
```

```
main() { // Pseudocode
  // Initial seed = 1
  IsolatedPRNG r = new IsolatedPRNG(1);
  async { print r.nextSeed(); ... }
  async { print r.nextSeed(); ... }
} // main()
```

Acknowledgments for Today's Lecture

- Inside the Java Virtual Machine, Chapter 20: Thread Synchronization
  http://www.artima.com/insidejvm/ed2/threadsynch.html
- Concurrency Tutorial: Guarded Blocks
  http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html
- "Actor-based Programming for Scalable Parallel and Distributed Systems", Gul Agha

  http://dl.dropbox.com/u/27020702/actors/Actors.pptx
- Actor Model - Wikipedia
- "Actor Concurrency" by Alex Miller

  http://www.slideshare.net/alexmiller/actor-concurrency
- "The Actor Model - Towards Better Concurrency" by Dror Bereznitsky
  http://www.slideshare.net/drorbr/the-actor-model-towards-better-concurrency
- "Actors in Scala" by Philipp Haller and Frank Sommers

Outline

- **Monitors**
- **Actors**

Monitors --- an object-oriented approach to isolation

- A monitor is an object containing
  - some local variables (private data)
  - some methods that operate on local data (monitor regions)
- Only one task can be active in a monitor at a time, executing some monitor region
  - **Analogous to a critical section**
- Monitors can also be used for
  - Mutual exclusion
  - Cooperation

Monitors – a Diagrammatic summary

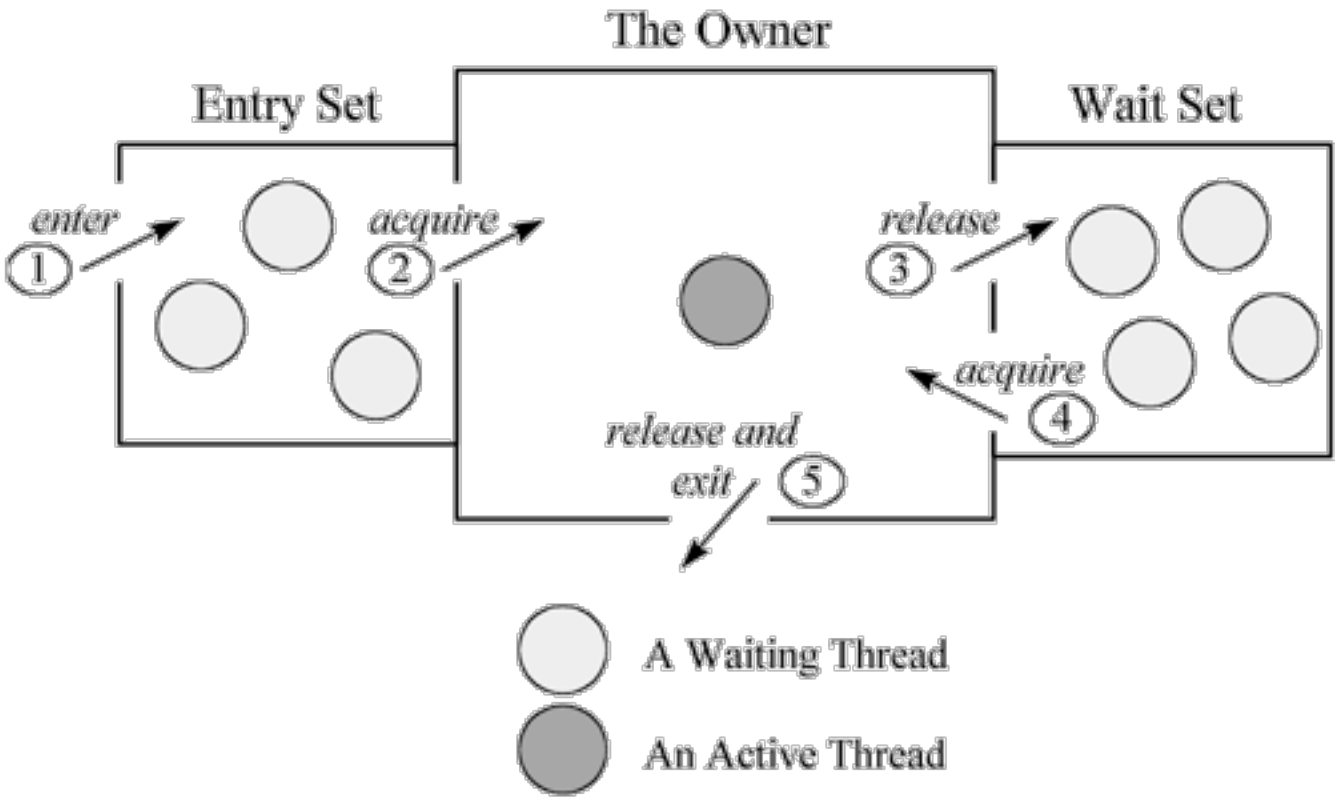


Figure 20-1. A Java monitor.

Figure source: http://www.artima.com/insidejvm/ed2/images/fig20-1.gif

Converting Standard Java Libraries to Monitors

Different approaches:

1. Restrict access to a single task ➔ no modification needed
2. Ensure that each call to a public method is isolated ➔ excessive serialization
3. Use specialized implementations that minimize serialization across public methods ➔ Java Concurrent Collections

- We will focus on three java.util.concurrent classes that can be used freely in HJ programs, analogous to Java Atomic Variables
  - ConcurrentHashMap, ConcurrentLinkedQueue, CopyOnWriteArraySet
- Other j.u.c. classes can be used in standard Java, but not in HJ because they may perform blocking operations
  - ArrayBlockingQueue, CountDownLatch, CyclicBarrier, DelayQueue, Exchanger, FutureTask, LinkedBlockingQueue, Phaser PriorityBlockingQueue, Semaphore, SynchronousQueue

The Java Map Interface

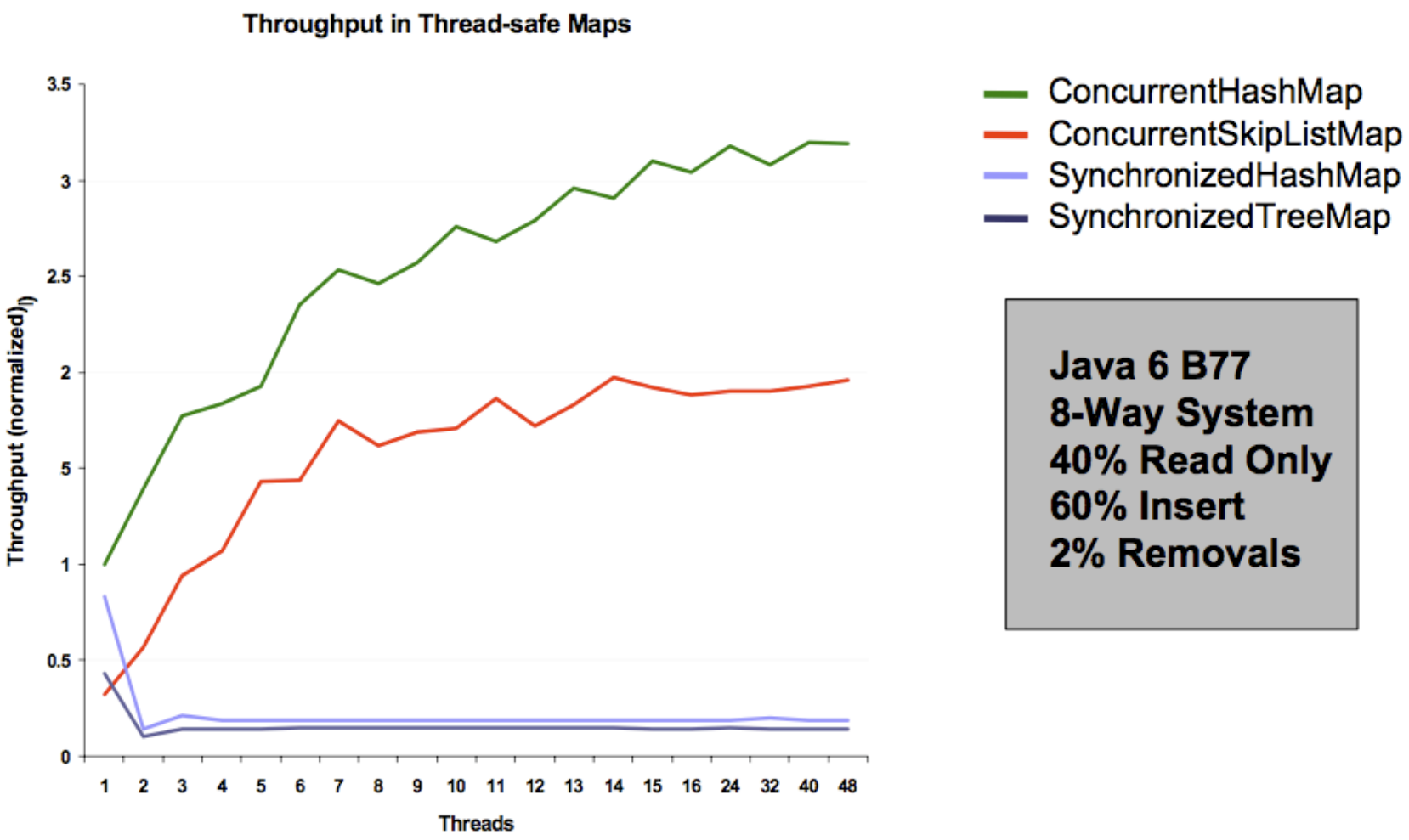
- Map describes a type that stores a collection of key-value pairs
- A Map associates a key with a value
- The keys must be unique
  - the values need not be unique
- Useful for implementing software caches (where a program stores key-value maps obtained from an external source such as a database), dictionaries, sparse arrays, …

- A Map is often implemented with a hash table (HashMap)
- Hash tables attempt to provide constant-time access to objects based on a key (String or Integer)
  - key could be your Student ID, your telephone number, social security number, account number, …
- The direct access is made possible by converting the key to an array index using a hash function that returns values in the range 0 … ARRAY_SIZE-1, typically by using a (mod ARRAY_SIZE) operation

java.util.concurrent.concurrentHashMap

- Implements ConcurrentMap sub-interface of Map
- Allows read (traversal) and write (update) operations to overlap with each other
- Some operations are atomic with respect to each other e.g.,
  - get(), put(), putIfAbsent(), remove()
- Aggregate operations may not be viewed atomically by other operations e.g.,
  - putAll(), clear()
- Expected degree of parallelism can be specified in ConcurrentHashMap constructor
  - ConcurrentHashMap(initialCapacity, loadFactor, concurrencyLevel)
  - A larger value of concurrencyLevel results in less serialization, but a larger space overhead for storing the ConcurrentHashMap

Concurrent Collection Performance



Java 6 B77
8-Way System
40% Read Only
60% Insert
2% Removals

Example usage of ConcurrentHashMap in org.mirrorfinder.model.BaseDirectory

```
1  public abstract class BaseDirectory extends BaseItem implements Directory {
2    Map files = new ConcurrentHashMap();
3    . . .
4    public Map getFiles() {
5      return files;
6    }
7    public boolean has(File item) {
8      return getFiles().containsValue(item);
9    }
10   public Directory add(File file) {
11     String key = file.getName();
12     if (key == null) throw new Error(. . .);
13     getFiles().put(key, file);
14     . . .
15     return this;
16   }
17   public Directory remove(File item) throws NotFoundException {
18     if (has(item)) {
19       getFiles().remove(item.getName());
20       . . .
21     } else throw new NotFoundException("can't remove unrelated item");
22   }
23 }
```

Listing 1: Example usage of ConcurrentHashMap in org.mirrorfinder.model.BaseDirectory [1]

java.util.concurrent.ConcurrentLinkedQueue

- `Queue` interface added to `java.util`
  - interface Queue extends Collection and includes

```
boolean offer(E x); // same as add() in Collection
E poll(); // remove head of queue if non-empty
E remove(o) throws NoSuchElementException;
E peek(); // examine head of queue without removing it
```

- Non-blocking operations
  - Return `false` when full
  - Return `null` when empty
- Fast thread-safe non-blocking implementation of Queue interface: `ConcurrentLinkedQueue`

Example usage of ConcurrentLinkedQueue in org.apache.catalina.tribes.io.BufferPool15Impl

```
1  class BufferPool15Impl implements BufferPool.BufferPoolAPI {
2    protected int maxSize;
3    protected AtomicInteger size = new AtomicInteger(0);
4    protected ConcurrentLinkedQueue queue = new ConcurrentLinkedQueue();
5    . . .
6    public XByteBuffer getBuffer(int minSize, boolean discard) {
7      XByteBuffer buffer = (XByteBuffer) queue.poll();
8      if ( buffer != null ) size.addAndGet(-buffer.getCapacity());
9      if ( buffer == null ) buffer = new XByteBuffer(minSize,discard);
10     else if ( buffer.getCapacity() <= minSize ) buffer.expand(minSize);
11     . . .
12     return buffer;
13   }
14   public void returnBuffer(XByteBuffer buffer) {
15     if ( (size.get() + buffer.getCapacity()) <= maxSize ) {
16       size.addAndGet(buffer.getCapacity());
17       queue.offer(buffer);
18     }
19   }
20 }
```

Listing 2: Example usage of ConcurrentLinkedQueue in org.apache.catalina.tribes.io.BufferPool15Impl

java.util.concurrent.CopyOnWriteArraySet

- Set implementation optimized for case when sets are not large, and read operations dominate update operations in frequency
- This is because update operations such as add() and remove() involve making copies of the array
  - Functional approach to mutation
- Iterators can traverse array "snapshots" efficiently without worrying about changes during the traversal.

Example usage of CopyOnWriteArraySet in org.norther.tammi.spray.freemarker.DefaultTemplateLoader

```
1  public class DefaultTemplateLoader implements TemplateLoader, Serializable
2  {
3    private Set resolvers = new CopyOnWriteArraySet();
4    public void addResolver(ResourceResolver res)
5    {
6      resolvers.add(res);
7    }
8    public boolean templateExists(String name)
9    {
10       for (Iterator i = resolvers.iterator(); i.hasNext();) {
11         if (((ResourceResolver) i.next()).resourceExists(name)) return true;
12       }
13       return false;
14   }
15   public Object findTemplateSource(String name) throws IOException
16   {
17     for (Iterator i = resolvers.iterator(); i.hasNext();) {
18       CachedResource res = ((ResourceResolver) i.next()).getResource(name);
19       if (res != null) return res;
20     }
21     return null;
22   }
23  }
```

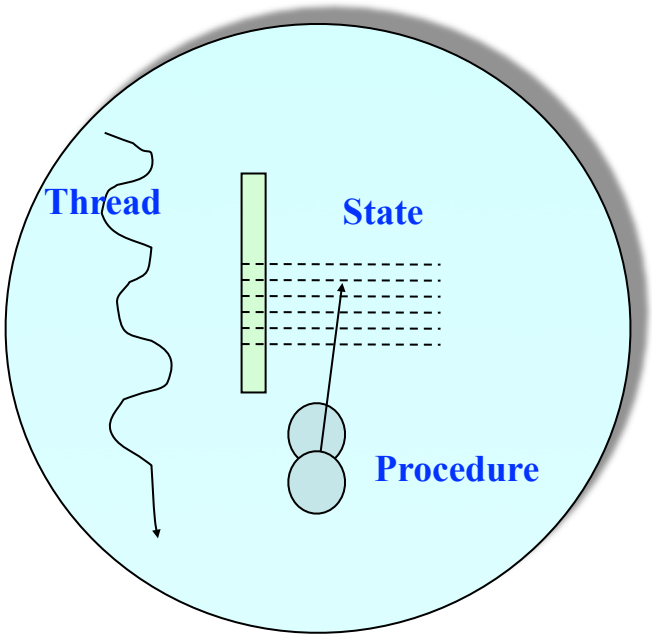
Listing 3: Example usage of CopyOnWriteArraySet in org.norther.tammi.spray.freemarker.DefaultTemplateLoader

Outline

- Monitors
- <u>Actors</u>

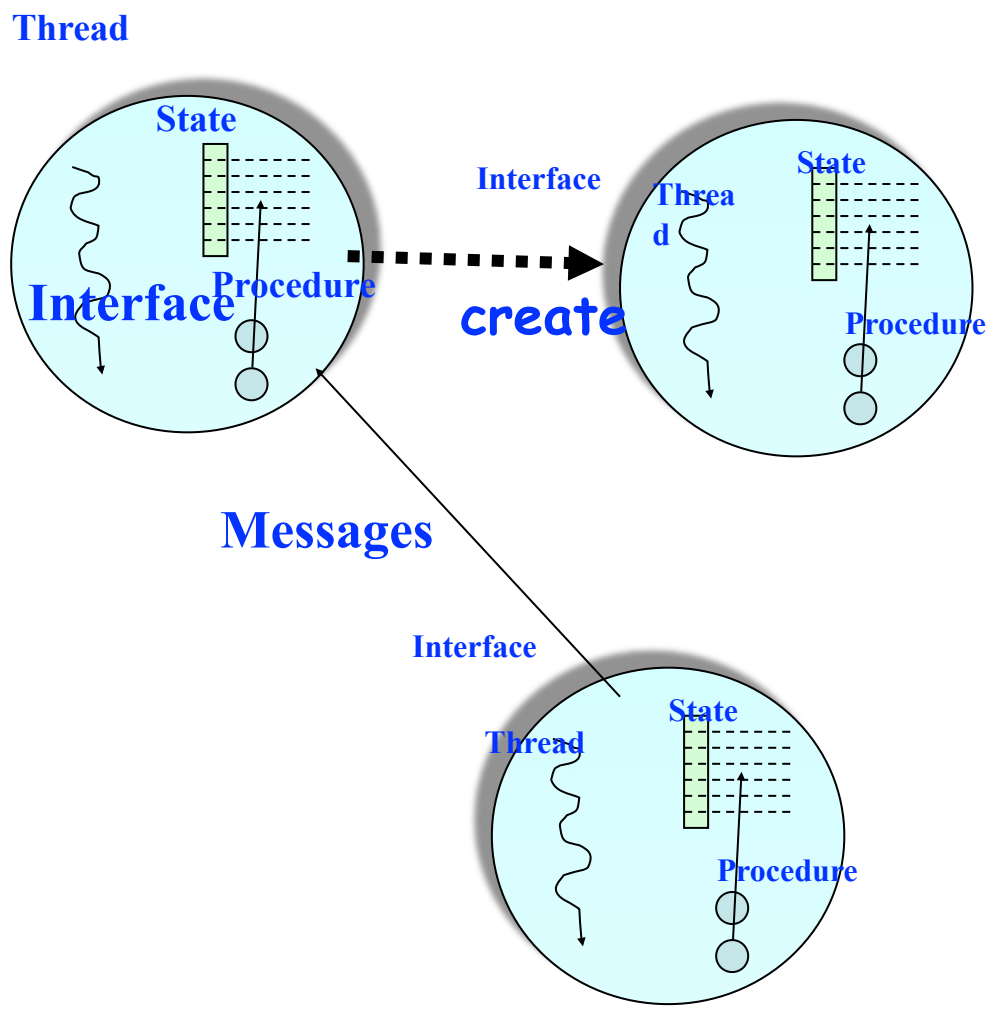
Actors as concurrent objects

- An actor is an autonomous, interacting component of a parallel system.
- An actor has:
  - an immutable identity (name, virtual address)
  - a mutable local state (encapsulated)
  - procedures to manipulate local state (provide an interface)
  - a thread of control

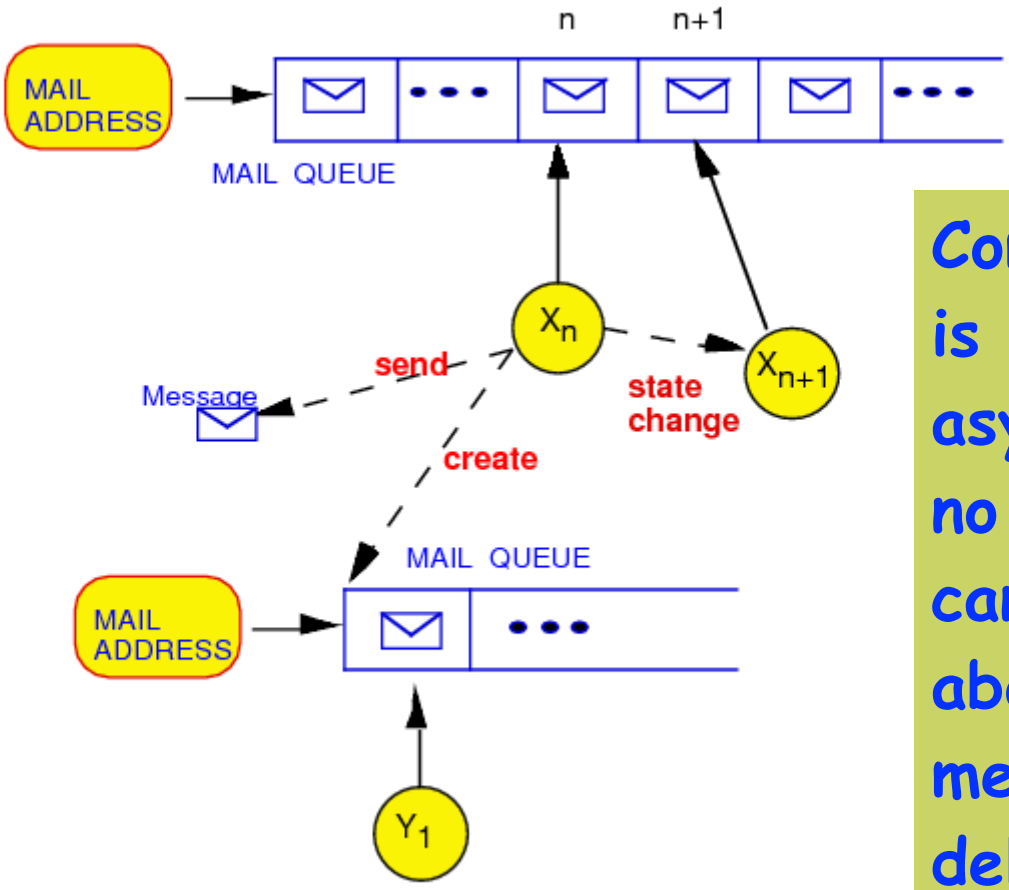


The Actor Model: Fundamentals

- An actor may:
  - process messages
  - send messages
  - change local state
  - create new actors



Arrival Order Nondeterminism

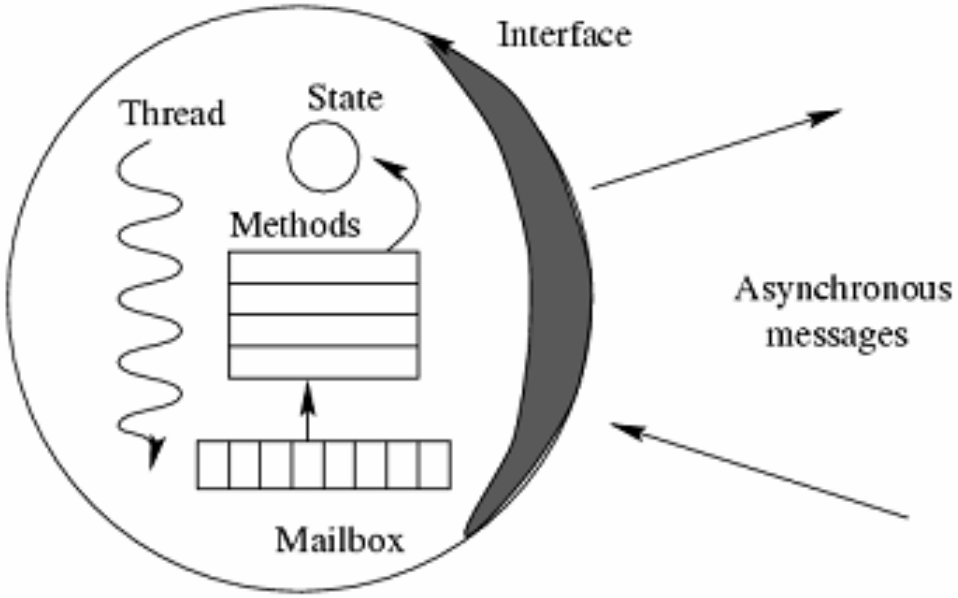


Communication is asynchronous: no assumption can be made about order of message delivery

Actor anatomy

Actors = encapsulated state + behavior (methods) +

thread of control + mailbox



Actor Model

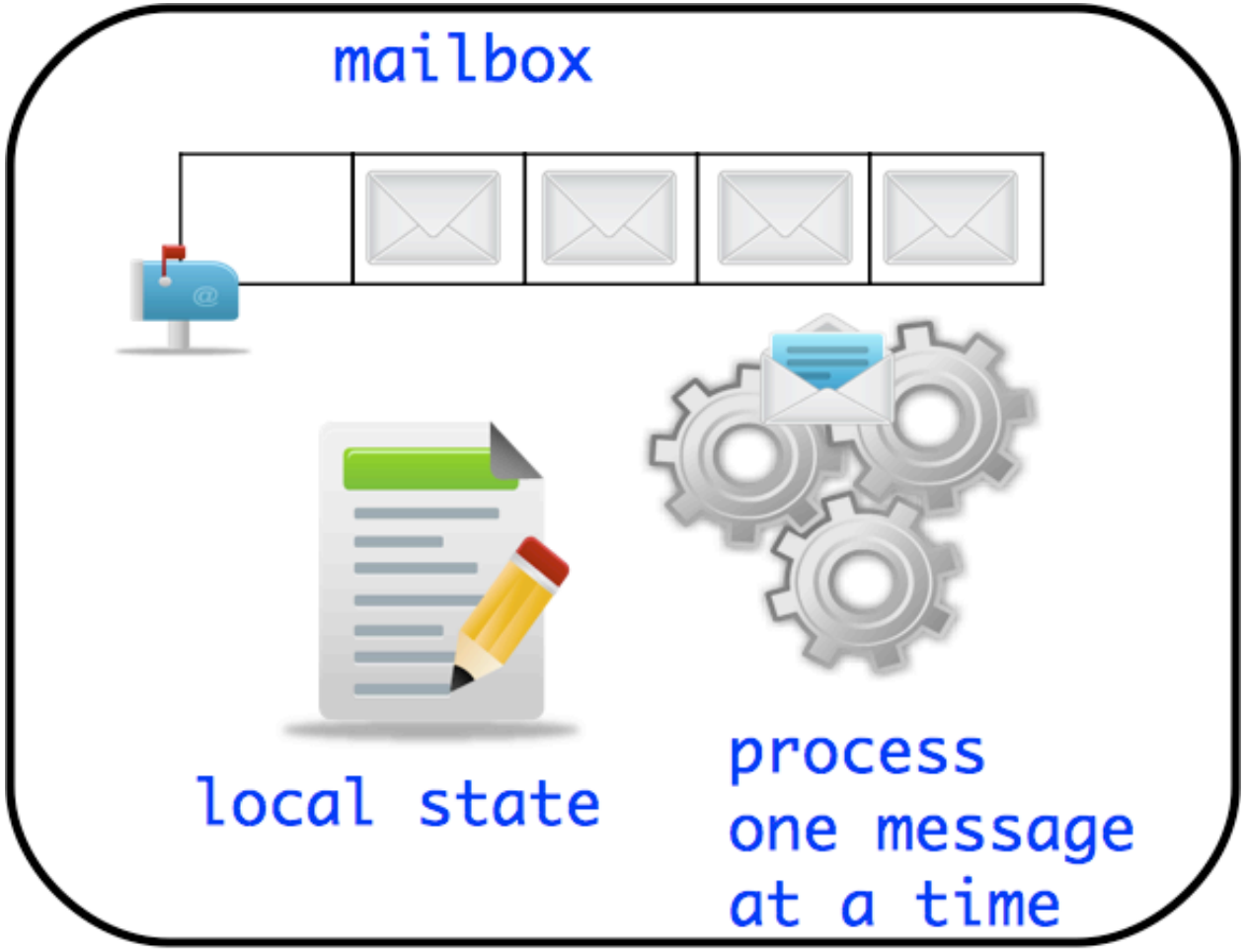
- A message-based concurrency model to manage mutable shared state
- First defined in 1973 by Carl Hewitt

  Further theoretical development by Henry Baker and Gul Agha
- Key Ideas:
  - Everything is an Actor!
  - Analogous to "everything is an object" in OOP
  - Encapsulate shared state in Actors
  - Mutable state is not shared
- Other important features (we will get to these later)
  - Asynchronous message passing
  - Non-deterministic ordering of messages

Actors' Behavior

- **Actors are passive and lazy**
  - Only respond if messages are sent to them
    - Messages may come from other actors or from main program (environment)
  - Only process one message at a time
    - Pending messages are stored in a "mailbox"
    - Parallelism comes from multiple actors processing messages in parallel
  - Mutate local state **only** while processing a message
  - Mutating local state can result in actor responding differently to subsequent messages

Actor



Actor Analogy - Email

- Email accounts are a good simple analogy to Actors
- To notify some information to (i.e. change some state of) A1 another account A2 sends an email (i.e. sends a message) to A1
- A1 has a mailbox to store all incoming messages
- A1 can read (i.e. process) one email at a time
  - At least that is what normal people do :)
- Reading an email can change how you respond to a subsequent email
  - e.g. receiving pleasant news while reading current email can affect the response to a subsequent email
- Actor creation (stretching the analogy)
  - Create a new email account that can send/receive messages

Actor Life Cycle

Actor states

- New: Actor has been created
  - e.g., email account has been created
- Started: Actor can receive and process messages
  - **e.g., email account has been activated**
- Terminated: Actor will no longer processes messages
  - **e.g., termination of email account after graduation**

Using Actors in HJ

- Create your custom class which extends hj.lang.Actor<Object> ,and implement the void process() method

```
class MyActor extends Actor<Object> {
  protected void process(Object message) {
    System.out.println("Processing " + message);
} }
```

- Instantiate and start your actor

```
Actor<Object> anActor = new MyActor(); anActor.start()
```

- Send messages to the actor

```
anActor.send(aMessage); //aMessage can be any object in general
```

- Use a special message to terminate an actor

```
protected void process(Object message) {
  if (message.someCondition()) exit();
}
```

- **Actor execution implemented as async tasks in HJ**
  - **Can use finish to await their completion**

Hello World Example

```
1. public class HelloWorld {
2.   public static void main(String[] args) {
3.     EchoActor actor = new EchoActor();
4.     actor.start(); // don't forget to start the actor
5.     actor.send("Hello"); // asynchronous send (returns immediately)
6.     actor.send("World");
7.     actor.send(EchoActor.STOP_MSG);
8.   }
9. }
10.class EchoActor extends Actor<Object> {
11.  static final Object STOP_MSG = new Object();
12.  private int messageCount = 0;
13.  protected void process(final Object msg) {
14.    if (STOP_MSG.equals(msg)) {
15.       println("Message-" + messageCount + ": terminating.”);
16.       exit(); // never forget to terminate an actor
17.     } else {
          messageCount += 1;
18.       println("Message-" + messageCount + ": " + msg);
19.} } }
```

Sends are asynchronous in actor model, but HJ Actor library preserves order of messages between same sender and receiver

Integer Counter Example

Without Actors:

```
int counter = 0;
public void foo() {
  // do something
  isolated {
    counter++;
  }
  // do something else
}
public void bar() {
  // do something
  isolated {
    counter--;
  } }
```

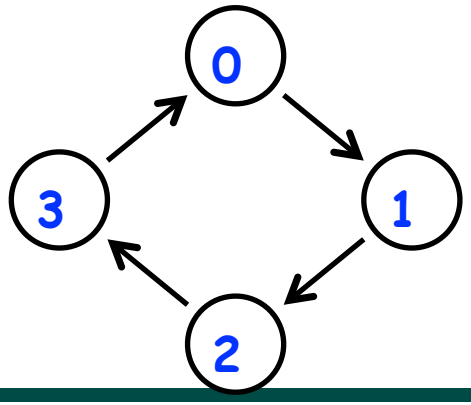
- Can also use atomic variables instead of isolated

With Actors:

```
class Counter extends Actor {
  private int counter = 0; // local state
  public void process(Message msg) {
    if (msg instanceof IncMessage) {
      counter++;
    } else if (msg instanceof DecMessage){
      counter--;
} } }
. . .
 Counter counter = new Counter();
 public void foo() {
// do something
counter.send(new IncrementMessage(1));
// do something else
}
 public void bar() {
// do something
counter.send(new DecrementMessage(1));
}
```

ThreadRing (Coordination) Example

```
1.  finish {
2.    int numThreads = 4;
3.    int numberOfHops = 10;
4.    ThreadRingActor[] ring =
        new ThreadRingActor[numThreads];
5.    for(int i=numThreads-1;i>=0; i--) {
6.      ring[i] = new ThreadRingActor(i);
7.      ring[i].start();
8.      if (i < numThreads - 1) {
9.        ring[i].nextActor(ring[i + 1]);
10.   } }
11.   ring[numThreads-1].nextActor(ring[0]);
12.   ring[0].send(numberOfHops);
13. } // finish
```

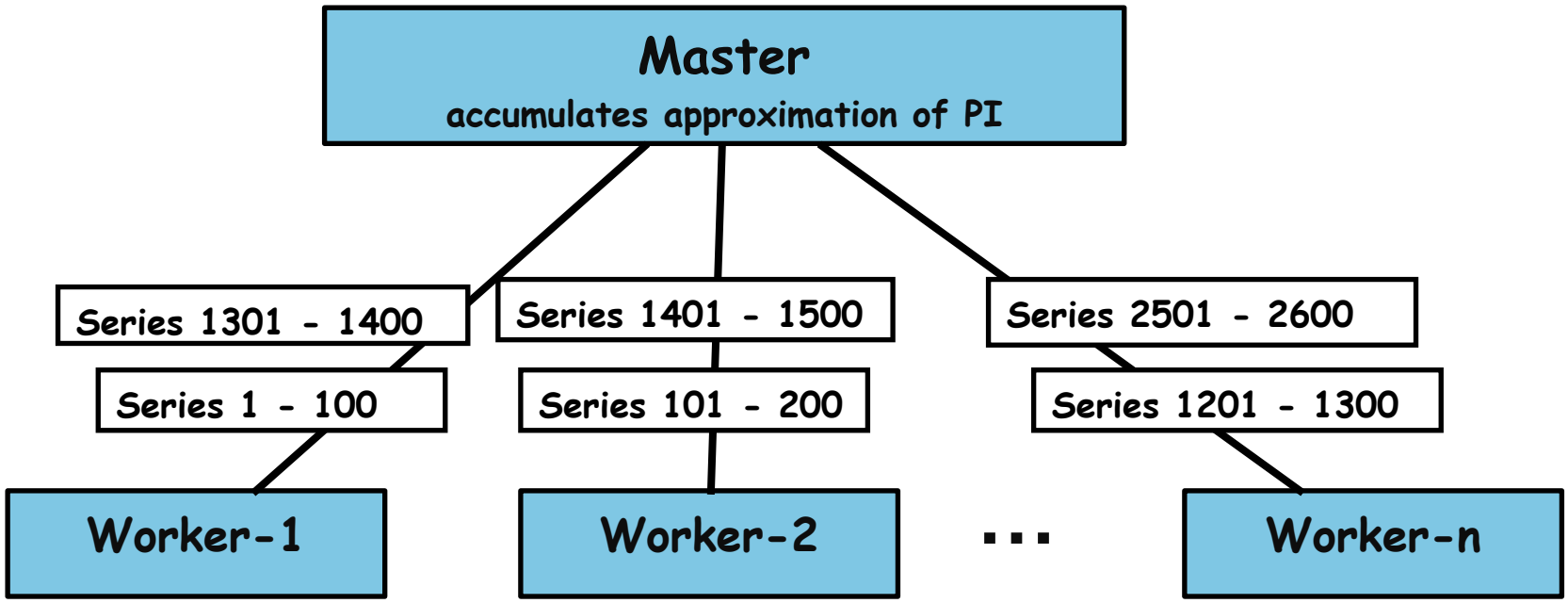



```
14. class ThreadRingActor
15.     extends Actor<Object> {
16.   private Actor<Object> nextActor;
17.   private final int id;
18.   ...
19.   public void nextActor(
        Actor<Object> nextActor) {...}
20.   void process(Object theMsg) {
21.     if (theMsg instanceof Integer) {
22.       Integer n = (Integer) theMsg;
23.       if (n > 0) {
24.         println("Thread-" + id +
25.           " active, remaining = " + n);
26.         nextActor.send(n - 1);
27.       } else {
28.         println("Exiting Thread-"+ id);
29.         nextActor.send(-1);
30.         exit();
31.       }
32.     } else {
33.     /* ERROR - handle appropriately */
34. } } }
```

Pi Computation Example

$$\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots.$$

- Use Master-Worker technique:



Source: http://www.enotes.com/topic/Pi

Pi Calculation --- Master Actor

```
class Master extends Actor<Object> {
  private double result = 0; private int nrMsgsReceived = 0;
  private Worker[] workers;
  Master(nrWrkrs, nrEls, nrMsgs) {...} // constructor
  void start() {
    super.start(); // Starts the master actor
    // Create and start workers
    workers = new Worker[nrWrkrs];
    for (int i = 0; i < nrwrkrs; i++) {
      workers[i] = new Worker();
      workers[i].start();
    }
    // Send messages to workers
    for (int j = 0; j < nrMsgs; j++) {
      someWrkr = ... ; // Select worker for message j
      someWrkr.send(new Work(...));
    }
  } // start()
```

Pi Calculation --- Master Actor (contd)

```
19.   void exit() {
20.     for (int i = 0; i < nrWrkrs; i++) workers[i].send(new Stop());
21.     super.exit(); // Terminates the actor
22.   } // exit()
23.   void process(final Object msg) {
24.     if (msg instanceof Result) {
25.       result += ((Result) msg).result;
26.       nrMsgsReceived += 1;
27.       if (nrMsgsReceived == nrMsgs) exit();
28.     }
29.     // Handle other message cases here
30.   } // process()
31. } // Master
32.  . . .
33.  // Main program
34.  Master master = new Master(w, e, m);
35.  finish master.start();
36.  println("PI = " + master.getResult());
```

Pi Calculation --- Worker Actor

```
class Worker extends Actor<Object> {
  void process(Object msg) {
    if (msg instanceof Stop) exit();
    else if (msg instanceof Work) {
      Work wm = (Work) msg;
      double result = calculatePiFor(wm.start, wm.end)
      master.send(new ResultMessage(result));    }
  } // process()

  private double calculatePiFor(int start, int end) {
    double acc = 0.0;
    for (int i = start; i < end; i++) {
      acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1);
    }
    return acc;
  }
} // Worker
```

Limitations of Actor Model

- Deadlocks possible
  - Deadlock occurs when all started (but non-terminated) actors have empty mailboxes
- Data races possible when messages include shared objects
- Simulating synchronous replies requires some effort
  - e.g., does not support addAndGet()
- Implementing truly concurrent data structures is hard
  - No parallel reads, no reductions/accumulators
- Difficult to achieve global consensus
  - Finish and barriers not supported as first-class primitives

==> These limitations can be overcome by using a hybrid model that combines task parallelism with actors

Actors - Simulating synchronous replies

- Actors are inherently asynchronous
- Synchronous replies require blocking operations e.g., async await

```
class CountMessage {
  ... ddf = new DataDrivenFuture();
  int localCount = 0;

static int getAndIncrement(
       CounterActor counterActor) {

   ... msg = new CountMessage();
   counterActor.send(msg);
   // use ddf to wait for response
   // THREAD-BLOCKING
   finish { async await(msg.ddf) { }}
   // return count from the message
   return msg.localCount;
} }
```

```
class CounterActor extends Actor {
   int counter = 0;
   void process(Object m) {

      if (m instanceof CountMessage){
         CountMessage msg = ...
         counter++;
           msg.localCount = counter;
           msg.ddf.put(true);
       } ...
    } }
```

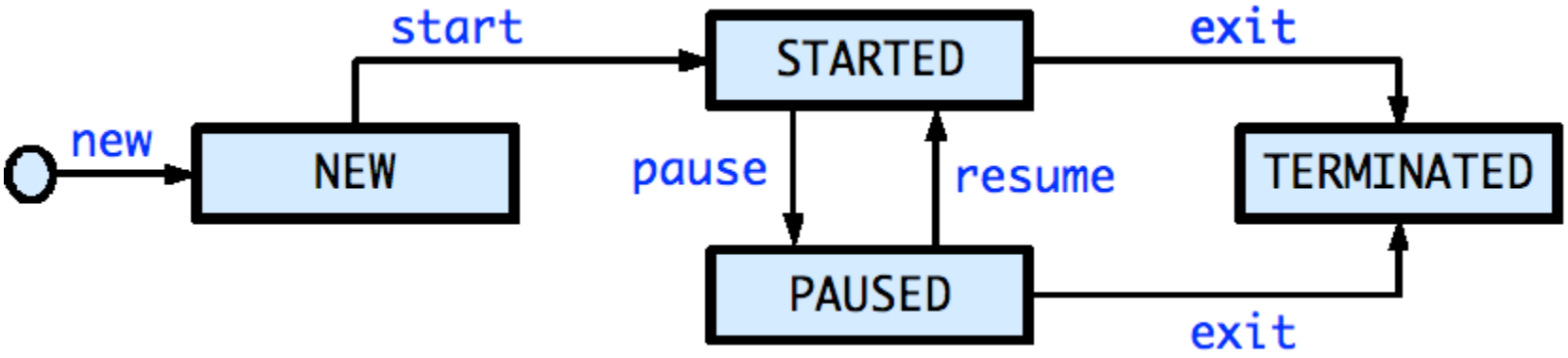
Actors – Global Consensus

- Global consensus is simple with barriers/phasers but can be complex with actors e.g.,
  - First send message from master actor to participant actors signaling intention
  - Wait for all participants to reply they are ready. Participants start ignoring messages sent to them apart from the master
  - Once master confirms all participants are ready, master sends the request to each participant and waits for reply from each
  - Master notifies participants that consensus has been reached, everyone can go back to normal functioning

Parallelizing Actors in HJ

- Two techniques:
  - Use finish construct to wrap asyncs in message processing body
    - Finish ensures all spawned asyncs complete before next message is
  - Allow escaping asyncs inside process() message
    - WAIT! Won't escaping asyncs violate the one-message-at-a-time rule in actors
    - Solution: Use pause and resume

Actors: pause and resume



- Paused state: actor will not process subsequent messages until it is resumed
- Pause an actor before returning from message processing body with escaping asyncs
- Resume actor when it is safe to process the next message
- Akin to Java's wait/notify operations with locks

Synchronous Reply using Async-Await (without pause/resume)

```
class SynchronousReplyActor1 extends Actor {
void process(Message msg) {
    if (msg instanceof Ping) {
        finish {
            DataDrivenFuture<T> ddf = new DataDrivenFuture<T>();
            otherActor.send(ddf);
            async await(ddf) {
                T synchronousReply = ddf.get();
                // do some processing with synchronous reply
            }
        }
    } else if (msg instanceof ...) { ... } } }
```

Synchronous Reply using Pause/Resume

```
class SynchronousReplyActor2 extends Actor {
  void process(Message msg) {
    if (msg instanceof Ping) {
      DataDrivenFuture<T> ddf = new DataDrivenFuture<T>();
      otherActor.send(ddf);
      async await(ddf) { // this async processes synchronous reply
         T synchronousReply = ddf.get();
         // do some processing with synchronous reply
         resume(); // allow actor to process next message
      }
      pause(); // when paused, the actor doesn't process messages
    } else if (msg instanceof ...) { ... } } }
```

Worksheet #9 (to be done in pairs): Interaction between finish and actors

Name 1: _____ Name 2: _____

What would happen if the end-finish operation from slide 29 was moved from line 13 to line 11 as shown below?

```
1.  finish {
2.    int numThreads = 4;
3.    int numberOfHops = 10;
4.    ThreadRingActor[] ring = new ThreadRingActor[numThreads];
5.    for(int i=numThreads-1;i>=0; i--) {
6.      ring[i] = new ThreadRingActor(i);
7.      ring[i].start();
8.      if (i < numThreads - 1) {
9.        ring[i].nextActor(ring[i + 1]);
10.   } }
11.  } // finish
12. ring[numThreads-1].nextActor(ring[0]);
13. ring[0].send(numberOfHops);
```