HJ isolated construct (Recap)

```haskell
isolated (\ -> <body> );
```

- Isolated construct identifies a critical section
- Two tasks executing isolated constructs are guaranteed to perform them in mutual exclusion
  - Isolation guarantee applies to (isolated, isolated) pairs of constructs, not to (isolated, non-isolated) pairs of constructs
- Isolated constructs may be nested
  - An inner isolated construct is redundant
- Blocking parallel constructs are forbidden inside isolated constructs
  - Isolated constructs must not contain any parallel construct that performs a blocking operation e.g., `finish`, `future get`, `next`
  - Non-blocking async operations are permitted, but isolation guarantee only applies to creation of async, not to its execution
- Isolated constructs can never cause a deadlock
  - Other techniques used to enforce mutual exclusion (e.g., locks) can lead to a deadlock, if used incorrectly
Object-based isolation (Recap)

`isolated(obj1, obj2, ..., () -> <body>)`

- In this case, programmer specifies list of objects for which isolation is required
- Mutual exclusion is only guaranteed for instances of isolated constructs that have a common object in their object lists
  - Standard isolated is equivalent to “isolated(*)” by default i.e., isolation across all objects
- Example:
  - `isolated(a,b,()->{..})` and `isolated(c,d,()->{..})` can execute in parallel
  - `isolated(a,b,()->{..})` and `isolated(b,c,()->{..})` cannot execute in parallel

DoublyLinkedListNode Example revisited with Object-Based Isolation

```java
1. class DoublyLinkedListNode {
2.    DoublyLinkedListNode prev, next;
3.    ...
4.    void delete() {
5.        isolated(this.prev, this, this.next, () -> { // object-based isolation
6.            this.prev.next = this.next;
7.            this.next.prev = this.prev;
8.        });
9.    ...
10. }
11. } // DoublyLinkedListNode
12. ...
13. static void deleteTwoNodes(final DoublyLinkedListNode L) {
14.    finish(() -> {
15.        DoublyLinkedListNode second = L.next;
16.        DoublyLinkedListNode third = second.next;
17.        async(() -> { second.delete(); });
18.        async(() -> { third.delete(); });
19.    });
20. }
```
Read-Write Object-based isolation in HJ

```java
isolated(readMode(obj1), writeMode(obj2), …, () -> <body> );
```

- Programmer specifies list of objects as well as their read-write modes for which isolation is required
- Not specifying a mode is the same as specifying a write mode (default mode = read + write)
- Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists such that one of the accesses is in writeMode

**Sorted List example**

1. public boolean contains(Object object) {
2.     return isolatedWithReturn( readMode(this), () -> {
3.         Entry pred, curr;
4.         ...
5.         return (key == curr.key);
6.     });
7. }
8.
9. public int add(Object object) {
10.    return isolatedWithReturn( writeMode(this), () -> {
11.        Entry pred, curr;
12.        ...
13.        if (...) return 1; else return 0;
14.    });
15. }
```

java.util.concurrent library

- Atomic variables
  - Efficient implementations of special-case patterns of isolated statements
- Concurrent Collections:
  - Queues, blocking queues, concurrent hash map, ...
  - Data structures designed for concurrent environments
- Executors, Thread pools and Futures
  - Execution frameworks for asynchronous tasking
- Locks and Conditions
  - More flexible synchronization control
  - Read/write locks
- Synchronizers: Semaphore, Latch, Barrier, Exchanger, Phaser
  - Tools for thread coordination
- **WARNING:** only a small subset of the full java.util.concurrent library can safely be used with HJlib
  - Atomic variables are part of the safe subset
  - We will study the full library later this semester as part of Java Concurrency
java.util.concurrent.atomic.AtomicInteger

- Constructors
  - `new AtomicInteger()`  
    - Creates a new AtomicInteger with initial value 0
  - `new AtomicInteger(int initialValue)`  
    - Creates a new AtomicInteger with the given initial value

- Selected methods
  - `int addAndGet(int delta)`  
    - Atomically adds delta to the current value of the atomic variable, and returns the new value
  - `int getAndAdd(int delta)`  
    - Atomically returns the current value of the atomic variable, and adds delta to the current value

- Similar interfaces available for LongInteger

Work-Sharing Pattern using AtomicInteger

1. import java.util.concurrent.atomic.AtomicInteger;
2. . .
3. String[] X = ...; int numTasks = ...;
4. int[] taskId = new int[X.length];
5. AtomicInteger a = new AtomicInteger();
6. . .
7. finish(() -> {
8.   for (int i=0; i<numTasks; i++)
9.     async(() -> {
10.       . . .
11.       int j = a.getAndAdd(1);
12.       // can also use a.getAndIncrement()
13.       if (j >= X.length) break;
14.       taskId[j] = i; // Task i processes string X[j]
15.     . . .
16.   } while (true);
17. });  
18. }); // finish-for-async
java.util.concurrent.AtomicInteger methods and their 
equivalent isolated constructs (pseudocode)

<table>
<thead>
<tr>
<th>j.u.c.atomic Class and Constructors</th>
<th>j.u.c.atomic Methods</th>
<th>Equivalent HJ isolated statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomicInteger</td>
<td>int j = v.get();</td>
<td>int j; isolated (v) j = v.val;</td>
</tr>
<tr>
<td></td>
<td>v.set(newVal);</td>
<td>isolated (v) v.val = newVal;</td>
</tr>
<tr>
<td>AtomicInteger()</td>
<td>int j = v.getAndSet(newVal);</td>
<td>int j; isolated (v) { j = v.val; v.val = newVal; }</td>
</tr>
<tr>
<td>// init = 0</td>
<td>int j = v.addAndGet(delta);</td>
<td>isolated (v) { v.val += delta; j = v.val; }</td>
</tr>
<tr>
<td>AtomicInteger(init)</td>
<td>int j = v.getAndAdd(delta);</td>
<td>isolated (v) { j = v.val; v.val += delta; }</td>
</tr>
<tr>
<td></td>
<td>boolean b =</td>
<td>boolean b;</td>
</tr>
<tr>
<td></td>
<td>v.compareAndSet</td>
<td>isolated (v)</td>
</tr>
<tr>
<td></td>
<td>(expect,update);</td>
<td>if (v.val==expect) {v.val=update; b=true;}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else b = false;</td>
</tr>
</tbody>
</table>

Methods in java.util.concurrent.AtomicInteger class and their equivalent HJ isolated statements. Variable v refers to an AtomicInteger object in column 2 and to a standard non-atomic Java object in column 3. val refers to a field of type int.

java.util.concurrent.atomic.AtomicReference

- **Constructors**
  - `new AtomicReference()`
    - Creates a newAtomicReference with initial value 0
  - `new AtomicReference(Object init)`
    - creates a new AtomicReference with the given initial value

- **Selected methods**
  - `int getAndSet(Object newRef)`
    - Atomically get current value of the atomic variable, and set value to newRef
  - `int compareAndSet(Object expect, Object update)`
    - Atomically check if current value = expect. If so, replace the value of the atomic variable by update and return true. Otherwise, return false.
Methods in `java.util.concurrent.AtomicReference` class and their equivalent HJ isolated statements. Variable `v` refers to an AtomicReference object in column 2 and to a standard non-atomic Java object in column 3. `ref` refers to a field of type `Object`.

AtomicReference<T> can be used to specify a type parameter.

---

Parallel Spanning Tree Algorithm using AtomicReference

1. class V {
2.     V [] neighbors; // adjacency list for input graph
3.     AtomicReference<V> parent; // output value of parent in spanning tree
4.     boolean tryLabeling(final V n) {
5.         return parent.compareAndSet(null, n);
6.     } // tryLabeling
7.     void compute() {
8.         for (int i=0; i<neighbors.length; i++) {
9.             final V child = neighbors[i];
10.            if (child.tryLabeling(this))
11.                async(() -> { child.compute(); }); // escaping async
12.         }
13.     } // compute
14. } // class V
15. . . .
16. . . .
17. root.parent = root; // Use self-cycle to identify root
18. finish(() -> { root.compute(); });
19. . . .


**COMP 322 Worksheet 21 solution:** Eureka-style Speculative Parallelism.

The code snippet below performs a eureka-style search on a 2-D array with a fixed number of tasks. Each task uses the `next()` operation to ensure the computation progresses in a lock step manner where all tasks execute one call to `check()` before performing the equality (==) comparison. What does the program print when it completes execution? Also, print the number of == comparisons performed by the program. Remember that once the search eureka has been resolved (via a call to `offer()`), subsequent calls to `check()` will cause the task to terminate.

```java
final int numRows = 10;
final int numCols = 100;
final int[][] dataArray = new int[numRows][numCols];
for (int i = 0; i < numRows; i++) {
    for (int j = 0; j < numCols; j++) {
        dataArray[i][j] = 100 * i + j;
    }
}
final int searchElement = 625;
final HjSearchEureka<int[][]> eureka = newSearchEureka();
finish(eureka, () -> {
    forasyncPhased(0, numRows - 1, (i) -> {
        for (int j = 0; j < numCols; j++) {
            final int[] elemIndex = {i, j};
            eureka.check(elemIndex);
            next(); // barrier
            if (dataArray[i][j] == searchElement) {
                eureka.offer(elemIndex);
            }
        }
    }); // forasyncPhased
});// finish
final int[] index = eureka.get();
System.out.println("Result = " + Arrays.toString(index));

Answer:

Result: [6, 25] (found by task i=6 in iteration j=25)
Number of == comparisons: 260

Due to the presence of the barriers, each of the 10 tasks performs a comparison operation before the next comparison operation is performed by any of the other tasks. Since the call to `offer()` occurs at the 26th iteration (j=25), the total number of comparisons is 10x26 = 260.