The goal of IsolatedPRNG is to implement a single Pseudo Random Number Generator object that can be shared by multiple tasks. Show the isolated construct(s) that you can insert in method nextSeed() to avoid data races in the sample main program (pseudocode is fine).

```
1. class IsolatedPRNG {
2.   private int seed;
3.   public int nextSeed() {
4.     int retVal;
5.     isolated {
6.       retVal = seed;
7.       seed = nextInt(retVal);
8.     }
9.     return retVal;
10.   }
11. } // nextSeed()
12. . . .
13. } // IsolatedPRNG
```

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

What might happen if line 5 and line 6 were enclosed in separate isolated statements?
What is a “Eureka Style” Computation?

• Many optimization and search problems attempts to find a result with a certain property or cost
• Announce when a result has been found
  • An "aha!" moment – **Eureka** event
  • Can make rest of the computation unnecessary

==> Opportunities for “speculative parallelism”, e.g., Parallel Search, Branch and Bound Optimization, Soft Real-Time Deadlines, Convergence Iterations, . . .

Image source: http://www.netstate.com/states/mottoes/images/ca_eureka.jpg
Simple Example: Search in a 2-D Matrix

```java
class AsyncFinishSearch {
    def atomicRefFactory() {
        val initValue = [-1, -1]
        return new AtomicRef(initValue)
    }
    def doComputation(matrix, goal) {
        val token = atomicRefFactory()
        finish
            for rowIndices in matrix.chunks() {
                async
                    for (r in rowIndices) {
                        processRow(matrix(r), r, goal, token)
                    // return either [-1, -1] or a valid index [i, j]
                    return token.get()
            }
    }
    def processRow(rowData, r, goal, token) {
        for (c in rowData.indices()) {
            if goal.matches(rowData(c)) // eureka!!
                token.set([r, c])
            return
        }
    }
}
```
Challenges in Parallelizing a Eureka-Style Computation

- Detecting eureka events
  — need to pass token around as extra argument

- Terminating executing tasks after eureka
  — manual termination via cancellation tokens can be a burden
  — throwing an exception does not impact parallel tasks
  — “killing” a parallel task can lead to unpredictable results
Example of Manual termination via Cancellation Tokens

- Manual periodic checks with `returns`
- User controls responsiveness

```python
async
for r in rowIndices:
    if token.eureka():
        return
    processRow(matrix(r), r, goal, token)

def processRow(rowData, r, goal, token):
    for c in rowData.indices():
        if token.eureka():
            return
        if goal.matches(rowData(c)):
            token.set([r, c])
```

- Cumbersome to write
- Impossible to support inaccessible functions
HJlib solution: the Eureka construct

1. eureka = eurekaFactory()
2. finish (eureka) S1
   - Multiple finish’es can register on same Eureka
   - Wait for all tasks to finish as before
     - Except that some tasks may terminate early when eureka is resolved
3. async
   - Inherits eureka registrations from immediately-enclosing finish
4. offer()
   - Triggers eureka event on registered eureka
5. check()
   - Causes task to terminate if eureka resolved
class AsyncFinishEurekaSearch {
    def eurekaFactory() {
        ...
    }
    def doComputation(matrix, goal) {
        val eu = eurekaFactory()
        finish (eu) // eureka registration
        for rowIndices in matrix.chunks()
            async
                for r in rowIndices
                    processRow(matrix(r), r, goal)
        return eu.get()
    }
    def processRow(rowData, r, goal) {
        for c in rowData.indices()
            check([[r, c]]) // cooperative termination check
            if goal.matches(rowData(c))
                offer([[r, c]]) // trigger eureka event
    }
}
def eurekaFactory() {
    val initialValue = [-1, -1]
    return new SearchEureka(initialValue)
}

def eurekaFactory() {
    val K = 4
    return new CountEureka(K)
}

def eurekaFactory() {
    // comparator to compare indices
    val comparator = (a, b) -> {
        ((a.x - b.x) == 0) ? (a.y - b.y) : (a.x - b.x)
    }
    val initialValue = [INFINITY, INFINITY]
    return new MinimaEureka(initialValue, comparator)
}

def eurekaFactory() {
    val time = 4.seconds
    return new TimerEureka(time)
}

def eurekaFactory() {
    val units = 400
    return new EngineEureka(units)
}
Binary Tree Search Example

Inputs:
- binary tree, T
- id for each node in T, in breadth-first order e.g., root.id = 0, root.left.id = 1, root.right.id = 2, ...
- value for each node in T that is the search target

Outputs:
- calls to offer() resolve eureka
- calls to check() can lead to early termination
- final value of eureka contains id of a node with value == elemToSearch

```java
final HjSearchEureka<Integer> eureka = newSearchEureka(null);
finish(eureka, () -> {
    async(0) -> {
        searchBody(eureka, rootNode, elemToSearch);
    }
});
```

```java
private static void searchBody(
    final HjSearchEureka<Integer> eureka, final Node rootNode,
    final int elemToSearch) throws SuspendableException {
    eureka.check(rootNode.id);
    if (rootNode.value == elemToSearch) {
        eureka.offer(rootNode.id);
    }
    if (rootNode.left != null) {
        async(0) -> {
            searchBody(eureka, rootNode.left, elemToSearch);
        }
    }
    if (rootNode.right != null) {
        async(0) -> {
            searchBody(eureka, rootNode.right, elemToSearch);
        }
    }
}
```
Tree Min Index Search Example

Inputs:
- binary tree, T
- id for each node in T, in breadth-first order e.g., root.id = 0, root.left.id = 1, root.right.id = 2, ...
- value for each node in T that is the search target

Outputs:
- calls to offer() update eureka with minimum id found so far (among those that match)
- calls to check() can lead to early termination if the argument is >= than current minimum in eureka
- final value of eureka contains minimum id of node with value == elemToSearch

```java
final HjExtremaEureka<Integer> eureka = newExtremaEureka(
    Integer.MAX_VALUE, (Integer i, Integer j) -> j.compareTo(i));
finish(eureka, () -> {
    async(() -> {
        minIndexSearchBody(eureka, rootNode, elemToSearch);
    });
});

private static void minIndexSearchBody(
    final HjExtremaEureka<Integer> eureka, final Node rootNode, final int elemToSearch) throws SuspendableException {
    eureka.check(rootNode.id);
    if (rootNode.value == elemToSearch) {
        eureka.offer(rootNode.id);
    }
    if (rootNode.left != null) {
        async(() -> {
            minIndexSearchBody(eureka, rootNode.left, elemToSearch);
        });
    }
    if (rootNode.right != null) {
        async(() -> {
            minIndexSearchBody(eureka, rootNode.right, elemToSearch);
        });
    }
}
```

Inputs:
- binary tree, T
- id for each node in T, in breadth-first order e.g., root.id = 0, root.left.id = 1, root.right.id = 2, ...
- value for each node in T that is the search target

Outputs:
- calls to offer() update eureka with minimum id found so far (among those that match)
- calls to check() can lead to early termination if the argument is >= than current minimum in eureka
- final value of eureka contains minimum id of node with value == elemToSearch
AND-composition of Eurekas

```python
class AsyncFinishEurekaDoubleSearch {
    def doComputation(matrix, goal1, goal2) {
        val eu1 = eurekaFactory()
        val eu2 = eurekaFactory()
        val eu = eurekaComposition(AND, eu1, eu2)
        finish (eu) // eureka registration
        for rowIndices in matrix.chunks()
            async
            for r in rowIndices
                processRow(matrix(r), r, goal1, goal2)
                // eu1.get() or eu2.get() to determine which goal was found
        return eu.get()
    }

    def processRow(rowData, r, goal1, goal2) {
        for c in rowData.indices()
            val checkArg = [[r, c], [r, c]] // pair for eu1 and eu2
            check(checkArg) // cooperative termination check
            val foundIdx1 = goal1.matches(rowData(c)) ? [r, c] : [-1, -1]
            val foundIdx2 = goal2.matches(rowData(c)) ? [r, c] : [-1, -1]
            val foundIdx = [foundIdx1, foundIdx2] // pair for eu1 and eu2
            offer(foundIdx) // possible eureka event
    }
}
```