16

Creating Custom Streams

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

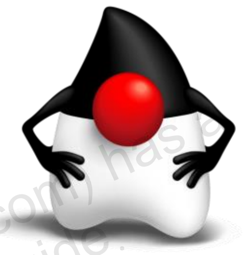
- Create a custom stream based on a custom data type
- Create a custom `Spliterator`

Topics

- Performance and Spliterators
- Custom Spliterators

Performance: Intuition and Measurement

- For small data sets, sequential is usually faster.
- Watch out for boxing.
- Simpler pipelines are easier to intuitively assess.
- Complex pipelines are more difficult to assess.
  - A stream source derived from an Iterator
  - Pipeline containing a limit operation
  - Complex reduction using classification (groupingBy)
- Measure!
  - Use a benchmark tool



Parallel Versus Sequential Example

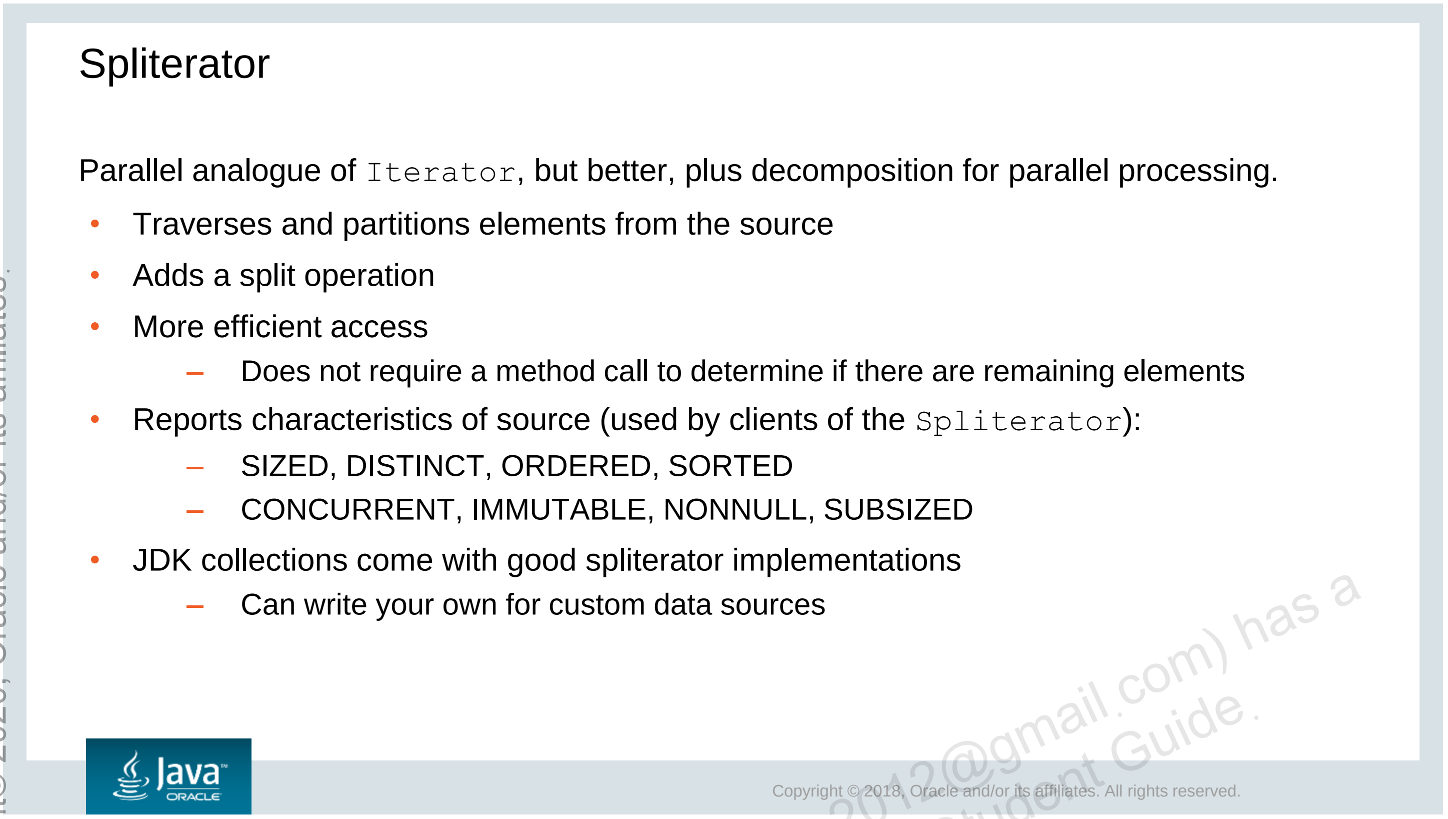
```
long begin = 100_000_000;
long window = 100_000;
getNumPrimes(begin, begin + window);

static void getNumPrimes(long begin, long end ) {
   LongStream newStream = LongStream.range(begin, end)
   .filter(i -> isPrime(i));
   System.out.println("Num of primes = " + newStream.count());
}

static boolean isPrime(long n) {
   return LongStream.rangeClosed(2, (long) sqrt(n))
   .noneMatch(divisor -> n % divisor == 0);
}
```

Use `.parallel()` ?

Yes

No

This is a good example of where testing really helps as it is quite difficult to foresee. The isPrime method tends to split into many tasks. Also the short-circuiting operation, noneMatch, while supported in parallel operations, works less optimally than for a single task. Alternatively, the getNumPrimes method has a simple task of calling isPrime. Parallelizing here will have the effect of running a number of isPrime methods in parallel, a clean separation.



Spliterator

Parallel analogue of `Iterator`, but better, plus decomposition for parallel processing.

- Traverses and partitions elements from the source
- Adds a split operation
- More efficient access
  - Does not require a method call to determine if there are remaining elements
- Reports characteristics of source (used by clients of the `Spliterator`):
  - SIZED, DISTINCT, ORDERED, SORTED
  - CONCURRENT, IMMUTABLE, NONNULL, SUBSIZED
- JDK collections come with good spliterator implementations
  - Can write your own for custom data sources

See Spliterator documentation for a summary of the characteristics.

Spliterator

```
public interface Spliterator<T> {

   boolean tryAdvance(Consumer<? super T> action);

   void forEachRemaining(Consumer<? super T> action);

   Spliterator<T> trySplit();

   long estimateSize();

   int characteristics();

   Comparator getComparator();
   ...
}
```

If a remaining element exists, performs the given action on it, returning true; else returns false.

Return a new `Spliterator` and adjust this `Spliterator` so they will not cover same elements. Return `null` if no further splits should be made.

If this `Spliterator` has `SORTED` as a characteristic, return the Comparator.

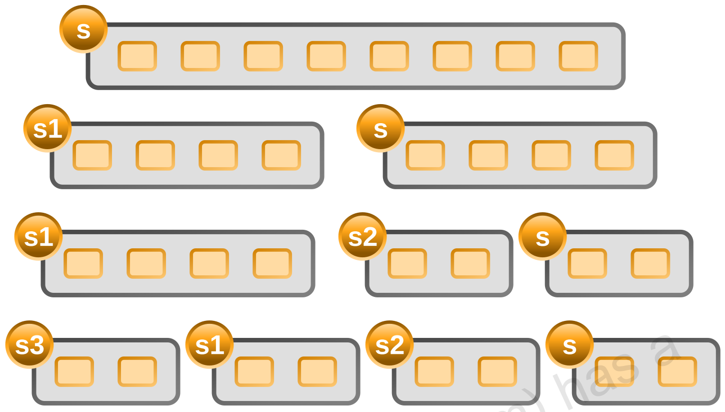
Size is an estimate unless characteristics say otherwise!

Note tryAdvance(). The elements are pushed into the Consumer.

This Consumer instance will be doing the filtering, mapping, and summing (for example). That's how the code gets delivered to where the iteration occurs, and that's how performance is enhanced and parallelism achieved.

Decomposition with `trySplit()`

```
ArrayList l = ...;

Spliterator s = l.spliterator();


Spliterator s1 = s.trySplit();


Spliterator s2 = s.trySplit();


Spliterator s3 = s1.trySplit();
```



In the example, the new Spliterator covers the left-hand side, and the current Spliterator resets itself to cover the remaining right-hand side (of what was the current Spliterator).

It can be implemented in many ways; the new Spliterator does not have to be the left-hand side (or beginning) of the data, nor does it need to split exactly at the halfway point. Later examples will illustrate this.

Integration with Streams

- A collection has a Spliterator.
  - `Collection.spliterator()`
  - JDK collections classes provide optimal implementations.
- A collection-based stream is constructed from the collection's `Spliterator`.

```
public interface Collection<E> extends Iterable<E> {

   default Stream<E> stream() {
      return StreamSupport.stream(spliterator(), false );
   }
   ...
}
```

Resultant `Stream` will not be parallel.

If true is used instead of false, the resultant stream may be parallel, but it is not guaranteed. For some collection types, it is very difficult to split the collection and would require so much work that it would not provide any performance improvement.

A LinkedList is a good example of this, as it must be traversed sequentially in order to get to a possible split point.

Modifying LongStream Spliterator

```
static void getNumPrimes(long begin, long end ) {
   LongStream newStream =
      StreamSupport.longStream(
      new TestSpliterator(range(begin, end).spliterator()),
      true)
      .filter(i -> isPrime(i));
   System.out.println("Num of primes = " + newStream.count());
}

static boolean isPrime(long n) {
   return LongStream.range(2, (long) sqrt(n))
   .noneMatch(divisor -> n % divisor == 0);
}
```

Create a Spliterator by modifying the existing one for a `LongStream`. Note `true` to make parallel.

Spliterator `TestSpliterator`

```
public class TestSpliterator implements Spliterator.OfLong{
    private Spliterator.OfLong split;
    public static AtomicInteger splitNum = new AtomicInteger(1);
    public MyTestSpliterator(Spliterator.OfLong split) {
        this.split = split;
    }
    public boolean tryAdvance(LongConsumer action){
        return split.tryAdvance(action);
    }
    public Spliterator.OfLong trySplit() {
       splitNum.incrementAndGet();
       return split.trySplit();
    }
//... Remaining methods not shown ...
}
```

To find out how many times trySplit called.

Wrapping the standard spliterator allows you to experiment a little by choosing to use the standard method implementations for most methods, but add some extra code to a critical method. Generally the API spliterators are very good, and it's unlikely they can be improved, but you may learn about how the spliterator is functioning, which will help you review your approach.

It's worth remembering that *within* a spliterator, code is executed sequentially, even if the spliterator is producing a parallel stream. The notion is "thread confinement". Each spliterator is executed by a single thread; the parallelism comes from multiple threads operating on multiple spliterators. Thus, methods can modify instance state of this spliterator without worrying about locking. However, as `splitNum` operates on static state of `TestSpliterator`, which is shared across multiple threads, it uses an `AtomicInteger` instead of a plain `int`.

Using TestSpliterator

```
long begin = 100_000_000;
long window = 100_000;
getNumPrimes(begin, begin + window);
System.out.println("Number splits: " +
   TestSpliterator.splitNum);
```

```
Number splits: 5
```

While the number of splits chosen by the JDK is not determinable, you can write your own custom Spliterator and in your code determine, say, the maximum number of splits that will be allowed.

Topics

- Performance and Spliterators
- Custom Spliterators

Creating a Custom Spliterator

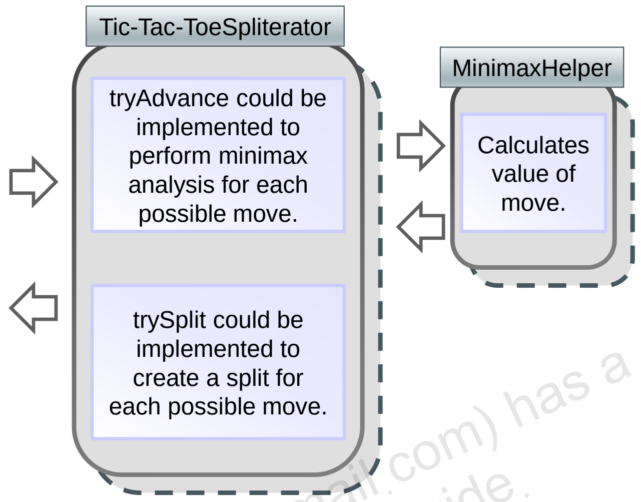
The standard Spliterators for Collections in the JDK are very good.

- It's not necessary to write a custom Spliterator for collections in the JDK.
- It may be useful to write a custom Spliterator for a Collection you have created.
  - Even then:
    - Determine if it is really needed
    - Determine if the collection is really needed
    - Determine if parallel support is needed

Is a Custom Spliterator Needed for a Game Engine?

Writing a game engine like chess or even tic-tac-toe may seem to require a custom spliterator..

```
Spliterator s0 =
  new TicTacToeSpliterator(myTic);
Stream<eStatus> ticStream =
  StreamSupport.stream(s0, true);
Collector moveCollector1 =
  Collectors.toList();

// Now select move with best
// value (eStatus) from List.
```


Tic-Tac-ToeSpliterator
tryAdvance could be implemented to perform minimax analysis for each possible move.
trySplit could be implemented to create a split for each possible move.
MinimaxHelper
Calculates value of move.

In a partially completed tic-tac-toe game like the following (assume the game engine is playing 'X'):

```
O X - -
- - X O
- - - -
O X - O
```

`myTic` is of type `TicTacToe` and contains an array that represents this move. It also contains methods to determine what is a legal move and whether a move wins by creating a line.

In the example position, the positions marked with a '-' represent a possible move. Therefore the Spliterator must:

1. Try all possible responses for each move, all possible responses to that, and so on until the end of the game.
2. Determine the score (win, loss, draw) of each of these possible moves by calling a MinimaxHelper class.
3. Return a stream based on the values determined by MinimaxHelper.

   For example, in the position shown the results for each possible move (reading left to right, top to bottom) are: loss, tie, tie, tie, tie, tie, tie, tie, loss.

   (Code for this implementation is found in the example folder for this lesson.)

A Tic-Tac-Toe Engine Using the `map` Method of `Stream`

But the tic-tac-toe engine can be implemented just as easily by using the `map` method. Remember N * Q performance model—here N is small (< 15) while Q is very large.

```
List<TicTacToe> candidateMoves = new ArrayList<>();
// Code to populate the List with TicTacToe objects that represent
// all possible moves

Stream<TicTacToe> ticStream = candidateMoves.parallelStream();
List<eStatus> possMoves = ticStream
      .map(n -> MinimaxHelper.getMoveValue(n, computerPlayer))
      .collect(Collectors.toList());

// Choose move with best value (eStatus)
```

In this example, there is no custom Spliterator. As the stream encounters each element, the map function calls MinimaxHelper to determine the value of that move.

(Code for this implementation is found in the example folder for this lesson.)

The N * Q performance model is covered in the lesson titled "Parallel Streams." Here it is again.

N = Size of the source data set

Q = Cost per element through the pipeline

N * Q ~= Cost of the pipeline

As stated in the lesson titled "Parallel Streams," generally a data set should contain more than 10,000 items before showing a difference in performance. In the tic-tac-toe example given, the data set is never more than the number of free moves. In a 4 X 4 tic-tac-toe game, that's less than 16. But because Q is so very great, given that it recursively investigates so many board states, there is still a significant advantage to running in parallel.

It's also interesting that even then improvements to the minimax algorithm can have a huge effect. Sometimes jumping on parallelizing might not be the best way to increase performance even when it is a possible approach.

A Custom Spliterator Example for a Custom Collection

If a custom binary or n-ary tree is required, stream support may also be necessary. The next example illustrates:

- Two approaches for a Spliterator for an n-ary Tree
- Support for parallelizing the Spliterator
- Traversal only for preorder traversal

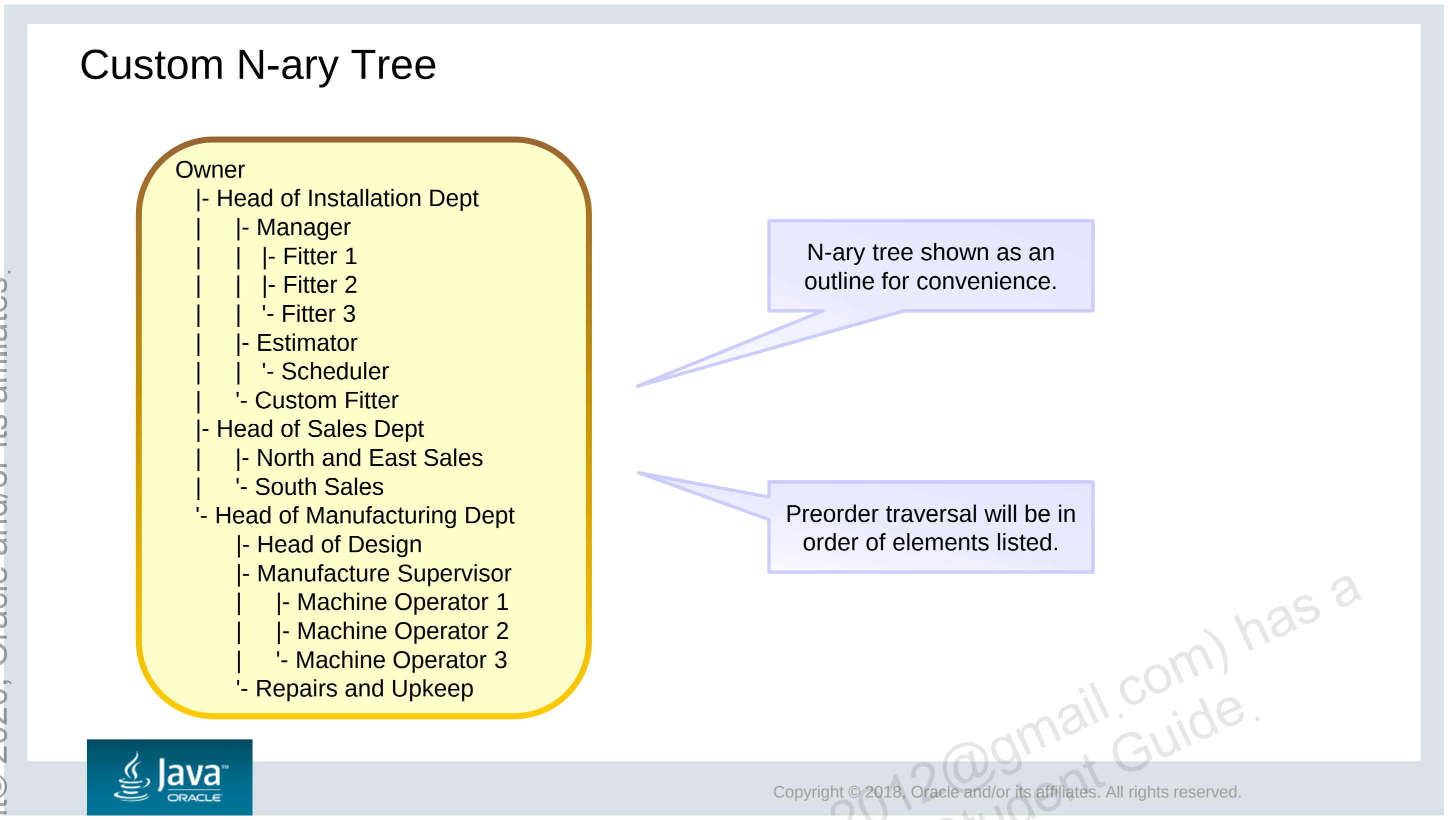
The Java API includes tree data types as:

- TreeMap and TreeSet, which use binary trees internally
- Nashorn trees
- Swing trees

It does not include any simple tree data types that can be traversed in the standard inorder, preorder, or postorder order. If you need this functionality, you may need to create a custom tree type.

Some external libraries provide tree types; for example, TreeTraverser is provided by the Guava library.

The Tree used in the following slides and in the practices illustrates the creation of a suitable Spliterator to traverse the tree. Note that if the Spliterator supports parallel streams, the order may not be retained.



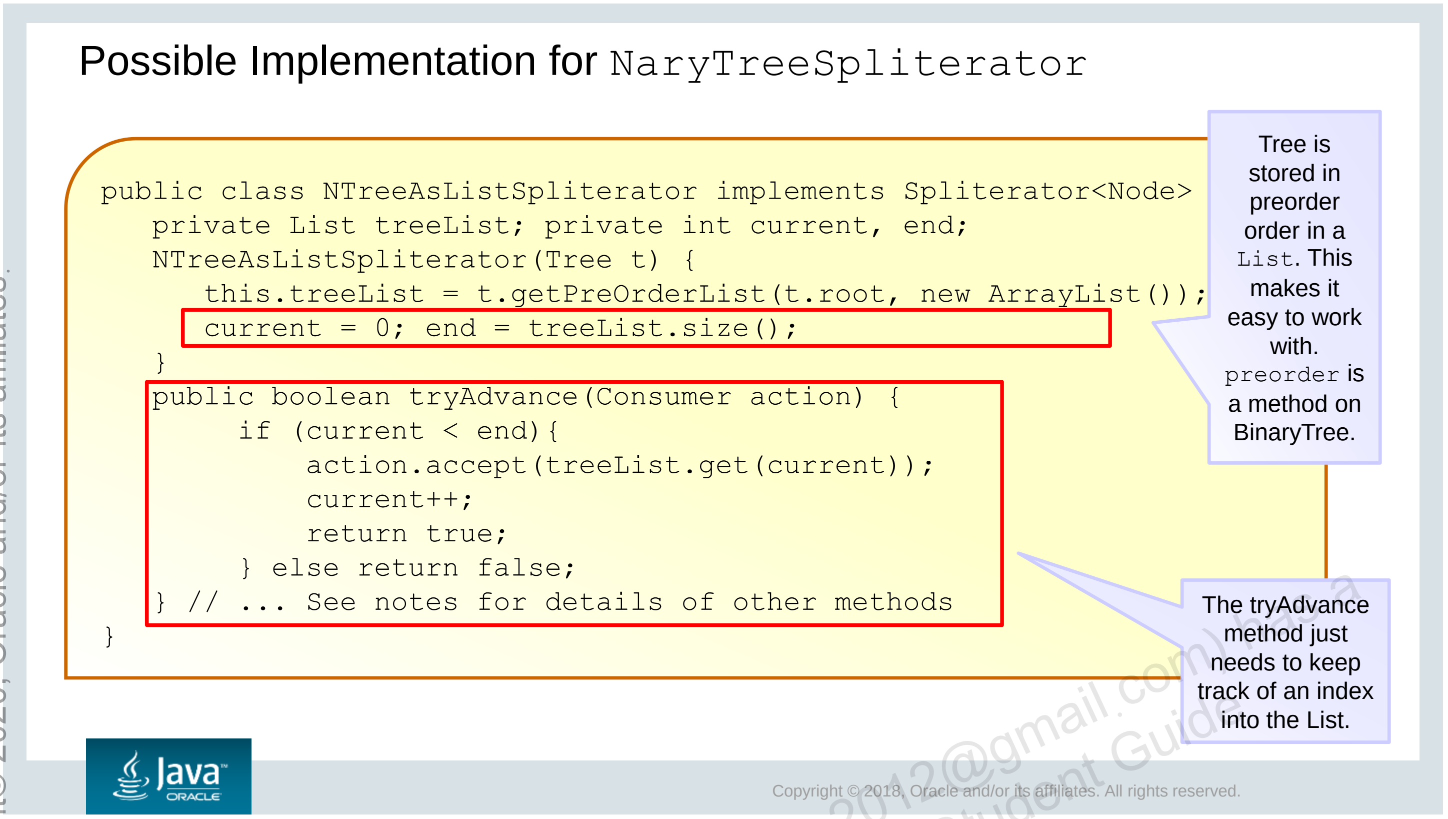
Custom N-ary Tree

```
Owner
   |- Head of Installation Dept
   |     |- Manager
   |     |   |- Fitter 1
   |     |   |- Fitter 2
   |     |   '- Fitter 3
   |     |- Estimator
   |     |   '- Scheduler
   |     '- Custom Fitter
   |- Head of Sales Dept
   |     |- North and East Sales
   |     '- South Sales
   '- Head of Manufacturing Dept
         |- Head of Design
         |- Manufacture Supervisor
         |     |- Machine Operator 1
         |     |- Machine Operator 2
         |     '- Machine Operator 3
         '- Repairs and Upkeep
```

N-ary tree shown as an outline for convenience.

Preorder traversal will be in order of elements listed.

N-ary trees have a set of nodes, and each node can have any number of children. A bill of materials (BOM) can be represented by an N-ary tree as can a company organization, as they are both hierarchical data. The example here is for a very simple company organization.



Possible Implementation for `NaryTreeSpliterator`

```
public class NTreeAsListSpliterator implements Spliterator<Node>
   private List treeList; private int current, end;
   NTreeAsListSpliterator(Tree t) {
      this.treeList = t.getPreOrderList(t.root, new ArrayList());
      current = 0; end = treeList.size();
   }
   public boolean tryAdvance(Consumer action) {
        if (current < end){
            action.accept(treeList.get(current));
            current++;
            return true;
        } else return false;
   } // ... See notes for details of other methods
}
```

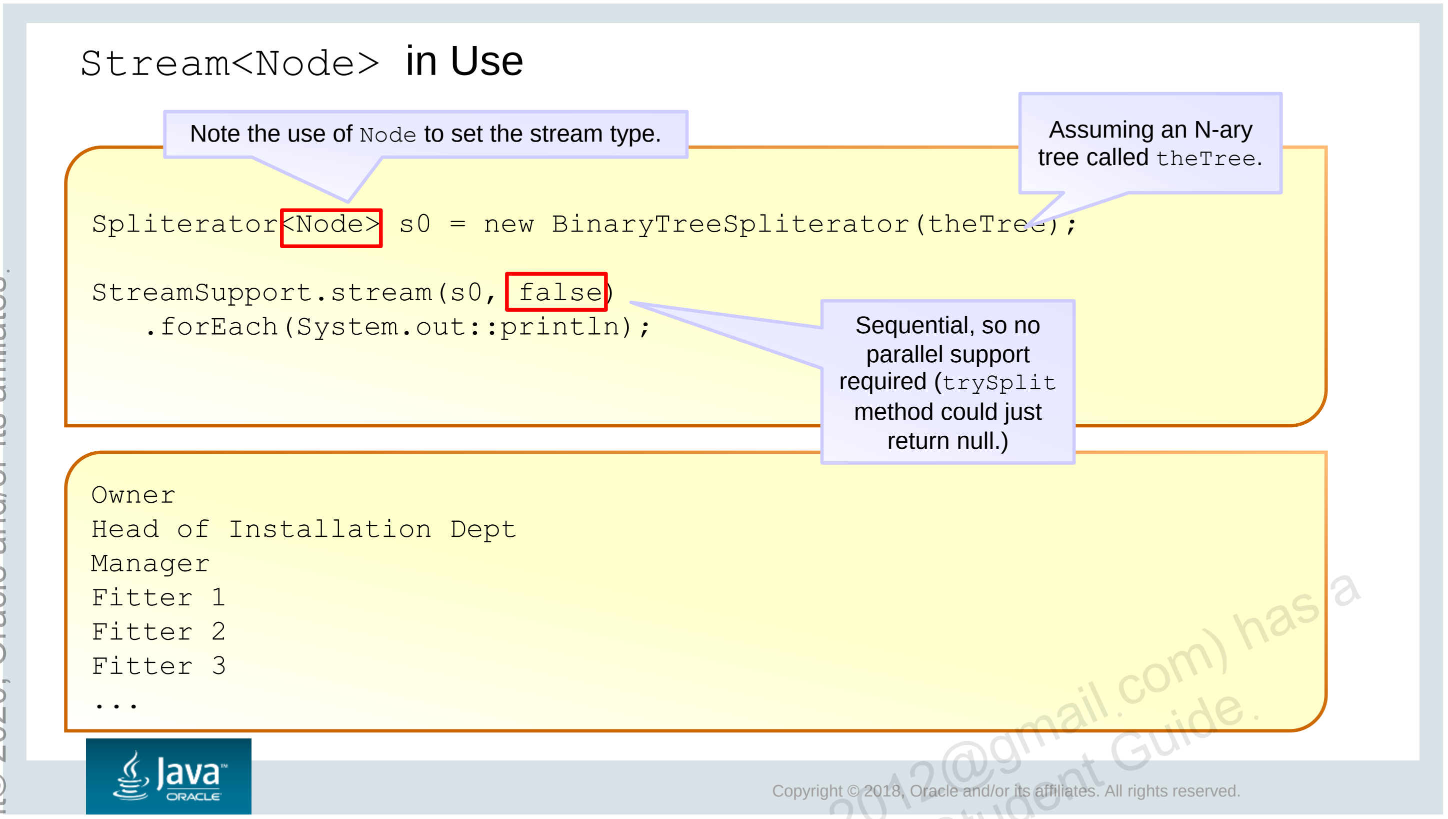
Tree is stored in preorder order in a `List`. This makes it easy to work with. `preorder` is a method on BinaryTree.

The tryAdvance method just needs to keep track of an index into the List.

This is a very simple approach to traversing the tree. If the tree is used to populate a List in preorder order, then the Spliterator can simply work with the List making traversal and splitting quite straightforward. Part of the constructor for the Spliterator could specify preorder—indeed the traversal could be specified with a lambda expression!

The remaining three methods can be implemented very simply as here:

```
 @Override
   public Spliterator<Node> trySplit() {
        return null;
   }

   @Override
   public long estimateSize() {
       return end - current;
   }

   @Override
   public int characteristics() {
       return 0;
}
```



`Stream<Node>` in Use

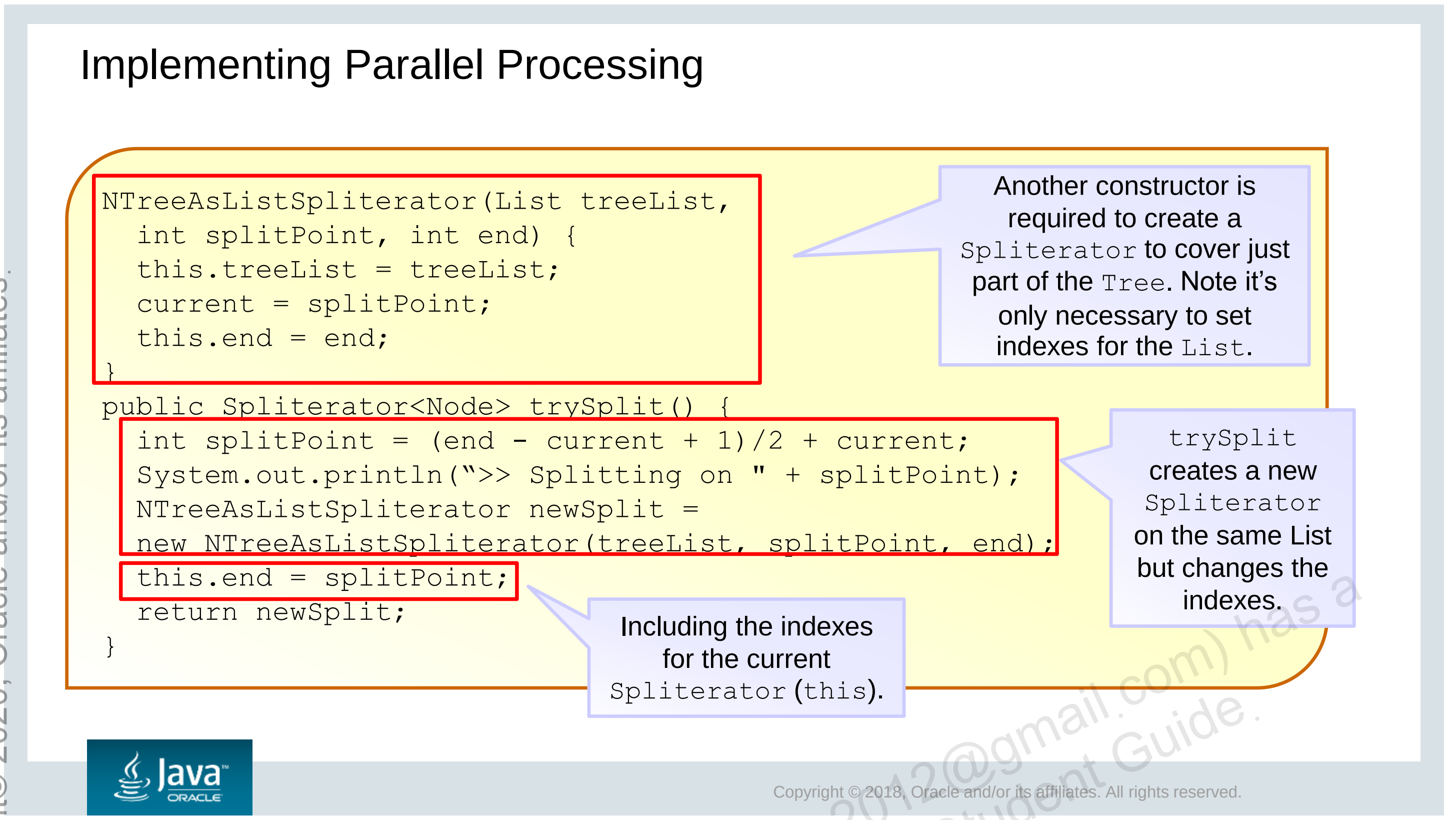
Note the use of `Node` to set the stream type.

Assuming an N-ary tree called `theTree`.

```
Spliterator<Node> s0 = new BinaryTreeSpliterator(theTree);

StreamSupport.stream(s0, false)
    .forEach(System.out::println);
```

Sequential, so no parallel support required (`trySplit` method could just return null.)

```
Owner
Head of Installation Dept
Manager
Fitter 1
Fitter 2
Fitter 3
...
```



Implementing Parallel Processing

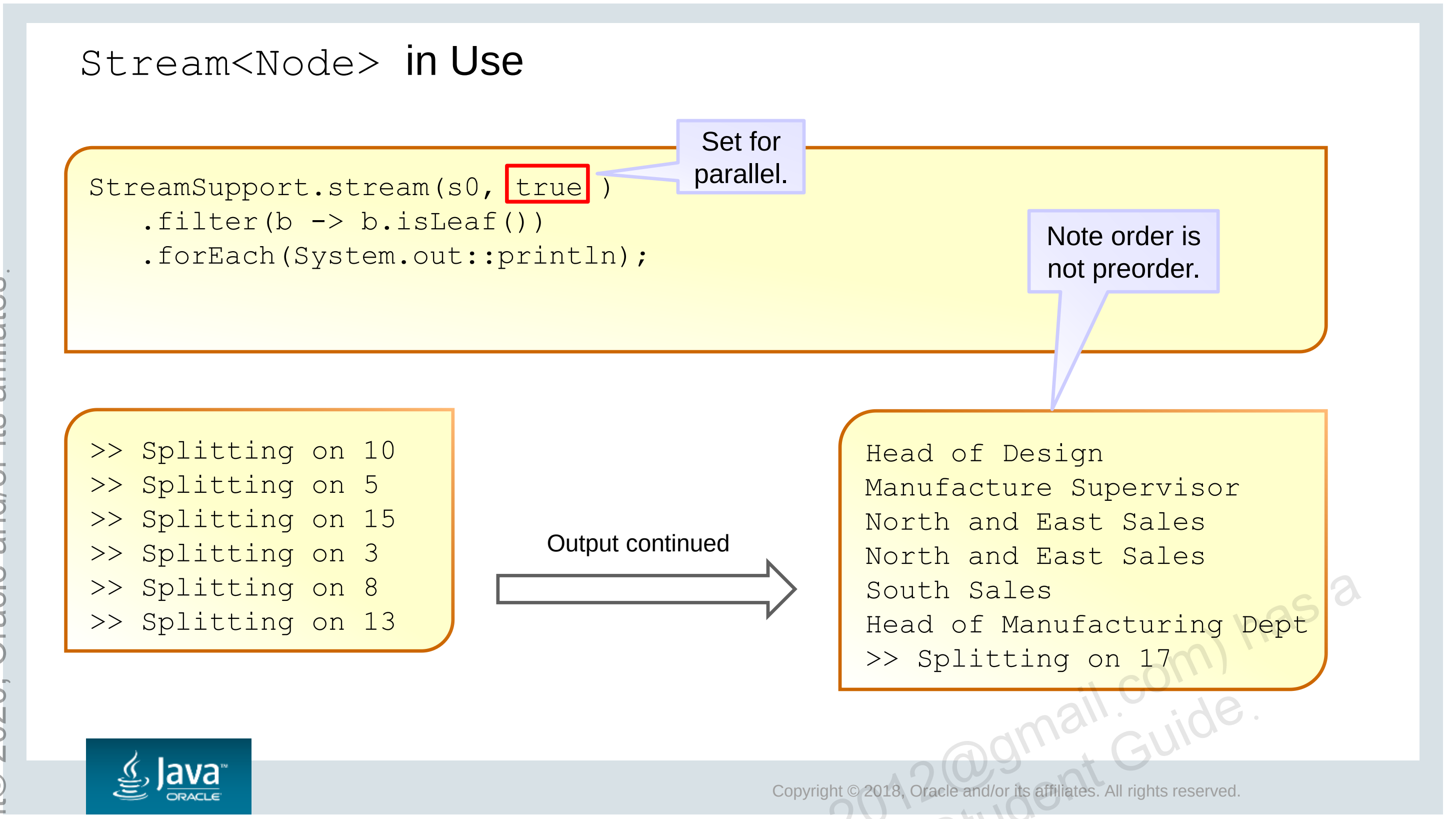
```
NTreeAsListSpliterator(List treeList,
  int splitPoint, int end) {
  this.treeList = treeList;
  current = splitPoint;
  this.end = end;
}
public Spliterator<Node> trySplit() {
  int splitPoint = (end - current + 1)/2 + current;
  System.out.println(">> Splitting on " + splitPoint);
  NTreeAsListSpliterator newSplit =
  new NTreeAsListSpliterator(treeList, splitPoint, end);
  this.end = splitPoint;
  return newSplit;
}
```

Another constructor is required to create a `Spliterator` to cover just part of the `Tree`. Note it's only necessary to set indexes for the `List`.

`trySplit` creates a new `Spliterator` on the same List but changes the indexes.

Including the indexes for the current `Spliterator` (`this`).

This example is just to illustrate that splitting on a List is relatively easy. However, if specifying the order of traversal is important, the performance advantage of parallel processing may be reduced somewhat by the overhead of preserving order. This issue is similar to the performance versus ordering issues with the forEach and forEachOrdered Stream methods.



`Stream<Node>` in Use

```
StreamSupport.stream(s0, true )
    .filter(b -> b.isLeaf())
    .forEach(System.out::println);
```

Set for parallel.
Note order is not preorder.
Output continued

```
>> Splitting on 10
>> Splitting on 5
>> Splitting on 15
>> Splitting on 3
>> Splitting on 8
>> Splitting on 13
```

```
Head of Design
Manufacture Supervisor
North and East Sales
North and East Sales
South Sales
Head of Manufacturing Dept
>> Splitting on 17
```

Summary of `NTreeAsListSpliterator`

Using a preordered List of the tree nodes is easy to implement, but:

- It requires the entire tree to be traversed and rendered as a `List` before any traversal takes place.
  - This needs to be done for each new traversal order required.

Another approach could be:

- Use an N-ary tree to provide the data for the `Spliterator`
- Have `tryAdvance` move one step along the tree in preorder order
- Create splits by:
  - Creating a new root node for a new tree based on the current tree
  - Changing the current tree to cut off the new tree when traversing

An example of an N-ary Tree Spliterator will be examined in the practice for this lesson.

Summary

In this lesson, you should have learned how to:

- Create a custom `Spliterator`
- Create a custom stream based on a custom data type

Practice 16: Overview

This practice covers the following topics:

- Practice 16-1: Examine the PrimeNumbersExample Application
- Practice 16-2: Using JMH (Java Microbench Harness)
- Practice 16-3: Run the Tictactoe Game Engine
- Practice 16-4: Examine a Custom Spliterator