Iteration over Composite

Output:

```
>> External:
Data.1
Data.2
Data.A1
Data.A2
>> Method:
Top
   Data.1
   Data.2
   Node.A
      Data.A1
      Data.A2
>> Internal:
Data.1
Data.2
Data.A1
Data.A2
Tree: 4
>> Internal + accumulator:
>> Tree::
:: Data.1
:: Data.2
:: Data.A1
:: Data.A2

>> Tree.stream.count = 4
>> tStream -print ::
Data.1
Data.2
Data.A1
Data.A2
>> tStream -print ::
Data.1
Data.2
Data.A1
Data.A2
>> tStream -print short names::
Data.1
Data.2
>> tStream -print "A" names::
Data.A1
Data.A2
```

CompositeIter.java

```
/*
 *
 * External Iterator for Composite
 *
 *   - (and internal iterator)
 *   - Revised composite iteration method;
 *     - Uses iteration over lists of iterators;
 *       which basically creates recursive ~chains of iterators,
 *       instead of a central stack control
 *     - For each item, or composite,
 *       iteration over it asks for its iterator,
 *       which then would as necessary use subsidiary iterators.
 *     - Notice the benefit of the composite pattern here,
 *       both parts & composites iterate identically!
 *     - [Of course, this is a bit inefficient, in that we use
 *         an iterator to get each individual part.]
 *
 *     - Added Stream methods (J8)
 */
package composite.iter2a;

// ******************************************************
//  Type Specific: Composite with String values in Leaf's
// ******************************************************

import java.util.*;
import java.util.stream.Stream;
import java.util.stream.StreamSupport;

public class CompositeIter2a {
        public static void main(String argv[]) {

                // Build a sample system ...
                Composite tree =
                        new Node("Top").
                        add( new Leaf("Data.1") ).
                        add( new Leaf("Data.2") ).
                        add( new Node("Node.A").                    // should Nodes have data?
                                        add( new Leaf("Data.A1") ).
                                        add( new Leaf("Data.A2") ) );

                // ---------- Now print via external iterator..
                System.out.println(">> External:");
                for (Composite c : tree)              // Iteration...
                        System.out.println(c);;
```

```
                // ---------- recursive method
                System.out.println(">> Method:");
                tree.print();

                // ---------- Now do via internal iterator
                System.out.println(">> Internal:");
                Printer printer = new Printer();   // create printing functor
                tree.doIterator(printer);
                System.out.println("Tree: "      + printer.value() );

                // ---------- Again, using accumulator...
                System.out.println(">> Internal + accumulator:");
                Functor<String,Composite> getTree = new Functor<String,Composite>() {
                        String tree="";
                        public void compute(Composite c) {
                                tree += ":: "+c+"\n";
                        }
                        public String value() { return tree; }
                };
                tree.doIterator(getTree);
                System.out.println(">> Tree:: \n"      + getTree.value() );

                // ---------- Now use J8 Streams
                Long c = tree.stream().count();
                System.out.println(">> Tree.stream.count = "      + c );

                Stream<Composite> tStream = tree.stream();
                System.out.println( ">> tStream -print ::");
                tStream.forEach(e -> e.print());
                // Or;
                System.out.println( ">> tStream -print ::");
                tree.stream().forEach(Composite::print);

                // Another Example;
                System.out.println( ">> tStream -print short names::");
                tree.stream().filter( e -> e.toString().length() <= 6)
                                .forEach(Composite::print);
                // Or;  All "A*" Nodes (only)
                System.out.println( ">> tStream -print \"A\" names::");
                tree.stream().filter(s -> (s.toString()).matches(".*A.*"))
                                .forEach(Composite::print);
        }
}

// -----------------------------------------------------------------
// Base of Composite class
abstract class Composite implements Iterable<Composite> {
        String name;
```

```
        void print() { System.out.println(this); }
        abstract void print(String indent);

        // default iterator (for leaf nodes), is single element; this!
        //  thisi s the base-case for the object iteration over a composite structure
        public Iterator<Composite> iterator() {
                List<Composite> lc =  new ArrayList<Composite>();
                lc.add(this);                   // Also include the composite (Leaf) itself as a component!
                return lc.iterator();
        }

        // internal iterator;
        //   broadcast & delegate
        void doAll(Functor<?,Composite> f) {
                // This assumes an "external" iterator that does all the work
                for (Composite c : this)              // Iteration...
                        f.compute(c);
        }

        // Stream methods (J8)
        public Stream<Composite> stream() {
           return StreamSupport.stream(this.spliterator(), false);
        }

        public Stream<Composite> parallelStream() {
           return StreamSupport.stream(this.spliterator(), true);
        }
}

//-----------------------------------------------------------------
//Leaf nodes; concrete components...
class Leaf extends Composite {
        String data;

        Leaf(String text) {
                data = text;
        }
        public void print() { print(""); }
        public void print(String indent) {
                System.out.println( indent + data);
        }
        public String toString() {
                return data;
        }
}

//-----------------------------------------------------------------
```

```
//The composition...
class Node extends Composite{
	String data;
	List<Composite> parts = new ArrayList<Composite>();

	// Some convenience constructors...
	Node(String name) {
		data = name;	// perhaps encapsulate this in a node, and add it?
	}
	Node( Composite  c ) { parts.add(c);}
	Node add(Composite c) {
		parts.add(c);
		return (this);	// chaining...
	}

	// user methods
	public void print() { print(""); }
	public void print(String indent) {
		System.out.println( indent + data);
		for ( Composite c : parts )
			c.print( indent+"   ");  // Increase indent at elach level
	}
	public String toString() {
		return data;
	}

	//
	public Iterator<Composite> iterator() {
		// Issue: Does not include the nesting component (this)...
		return new myIterator(parts);
	}

	/* This is the interesting part;
	 *  We will iterate through each component.
	 *  - Simple components are taken care of by the default
	 *    (base class) iterator, which just gives the one item.
	 *  - Composites are handled here;
	 *    We will go throgh our list, and ask each item for its iterator,
	 *    and thus recursively (DFS) traverse the whole structure.
	 */
	class myIterator implements Iterator<Composite> {
		Iterator<Composite> main, sub;
		Composite Cp;

		myIterator(List<Composite> Cs ) {
			main = Cs.iterator();
			sub = main.next().iterator();	// Assume non-empty(?)
		}
```

```
		public boolean hasNext() {	// check for next non-empty iterator
			if ( sub.hasNext() )
				return true;
			else
				if ( main.hasNext() ) {
					sub = main.next().iterator();
					return hasNext();
				} else
					return false;
		}

		public Composite next() {
			return sub.next();			// Here's the recursion!
		}
		public void remove() { main.remove(); }
	}
}

//-------------------------------------------------------------
// a Functor for internal iterations
interface Functor<R,T> {
	void compute(T c);
	R value();
}

//Hmm, not much interesting to do with our data!
// If we had a Functor with an additional indent level argument,
//    we could print it nicer...
//  But that is unusual…
class Printer implements Functor<Integer, Composite> {
	protected int count;
	// protected String indent;

	public Integer value() {return count;}

	public void compute(Composite c){
		System.out.println(c);
		count++;
	}
}
```