an alternative(?) linked-list in Java – a brief discussion

version 1

These are singularly linked lists. There are 2 types of node: one acting as null links the other (a classic linked-list node) containing data and a link to the next node in the list. A complete list is simply a reference to its first node.

Nodes of both types implement the following...

```
public interface ListNode
{
	public int length();
	public String toString();
}
```

Null nodes are simply defined as...

```
public class NullList implements ListNode
{
	public int    length()   { return 0;  }
	public String toString() { return ""; }
}
```

Nodes containing data|link fields can be specified as generics as...

```
public class DataNode<T> implements ListNode
{
	private T data;
	private ListNode link;

	public DataNode(T data, ListNode link)
	{	this.data = data;
		this.link = link;
	}
	@Override
	public String toString()
	{	return data.toString() + ", " + link.toString();
	}

	@Override
	public int length()
	{	return 1+ link.length();
	}
}
```

Output...

```
// test run...
System.out.println( "example #1 - using DataNode v.1" );
ListNode x = new DataNode<String>("kipper",
                  new DataNode<Integer>(12,
                        new DataNode<String>("carp", new NullList())));
System.out.println( "list = " + x.toString());
System.out.println( "size = " + x.length());

// generates output...
example #1 - using DataNode v.1
list = kipper, 12, carp,
size = 3
```

version 2

You may notice that DataNode does not use typed methods so can be untyped...

```
public class DataNode2 implements ListNode
{
      private Object   data;
      private ListNode link;

      public DataNode2(Object data, ListNode link)
      {     this.data = data;
            this.link = link;
      }
      public String toString()
      {     return data.toString() + ", " + link.toString();
      }
      public int length()
      {     return 1+ link.length();
      }
}
```

Output...

```
// test run...
System.out.println( "\nexample #2 - using DataNode2" );
x = new DataNode2("mango",
            new DataNode2(200, new DataNode2("melon", new NullList())));
System.out.println( "list = " + x.toString());
System.out.println( "size = " + x.length());

// generates output...
example #2 - using DataNode2
list = mango, 200, melon,
size = 3
```

version 3

Simplifies the null node by using an enum? Is this ok or a bit of a cludge?

```
public enum NullList2 implements ListNode
{
      NULL_PTR;

      public int    length()   { return 0;  }
      public String toString() { return ""; }
}
```

Output...

```
// test run...
System.out.println( "\nexample #3 - DataNode2 & NULL_PTR" );
x = new DataNode2("robin",
      new DataNode2(333, new DataNode2("sparrow", NullList2.NULL_PTR)));
System.out.println( "list = " + x.toString());
System.out.println( "size = " + x.length());

// generates output...
example #3 - DataNode2 & NULL_PTR
list = robin, 333, sparrow,
size = 3
```

version 4

Uses a simple Defs class (containing static finals) in place of the enum…

```
/**
 * a constants & singletons wrapper
 */
public abstract class Defs
{
      public static final ListNode NULL_NODE = new NullList();
}
```

Output...

```
// test run...
System.out.println( "\nexample #4 - using Defs" );
x = new DataNode2("apple",
      new DataNode2("pear", new DataNode2(1234, Defs.NULL_NODE)));
System.out.println( "list = " + x.toString());
System.out.println( "size = " + x.length());

// generates output...
example #4 - using Defs
list = apple, pear, 1234,
size = 3
```

questions

1. what are the strengths & weaknesses (or maybe just good & bad points) of the different versions?
2. which version do you prefer & why?
3. to what extent does the approach implement a decorator?