# << Lecture 16 >>

## Summary

- In this lecture, we continued and [momentarily] closed the topic of pointers and linked data structures
- We also covered *Exceptions* and the *try-catch* statement
- The midterm will cover everything up to Exceptions

## < Pointers and linked data structures >

- In Java, variables whose type subtypes Object are *pointers*
- *Pointer*: an indication of a memory address, to an object or value that resides in memory
- Why do they matter? Objects can link objects

## Example: node in a list

```
class Node{
   public Node next;
   public int size;

   public Node() {
      next=null;
      size=1;
   }
   public Node( Node nxt )  {
      next = nxt;
      location = nxt.location + 1;
   }
}
```

- Now, let us use Node:

```
Node a = new Node();
Node b = new Node(a);
Node c = new Node(c);
Node d = new Node(d);
for( Node j=d; j!=null; n=n.next )
   System.out.println( j.size + ',' + j);
```

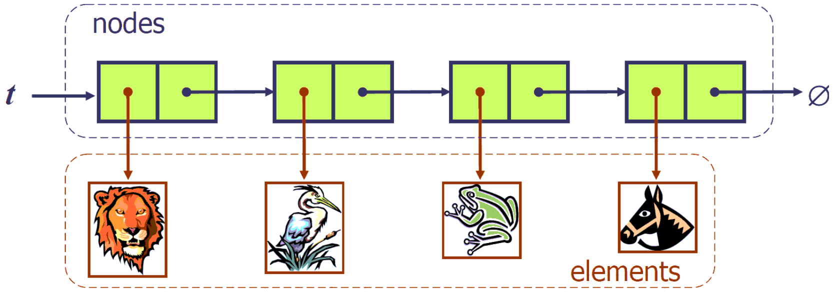
- *Lesson*: we can create “big” stuff
- *Observation*: changing the fields of Node, we can have something like an Array or a Map (i.e., like Python's **dict**)
- We created a simplified version of a **linked list**

## Thinking about the linked list

- Linked lists store items much like an array
    - But with flexible length, unlike array
- Problem: where to put a new node?
    - At the head? At the end? In the middle?
    - Do we need objects sorted???
- Problem: how to retrieve data?
    - *i*-th possition? Or retrieve position by object?

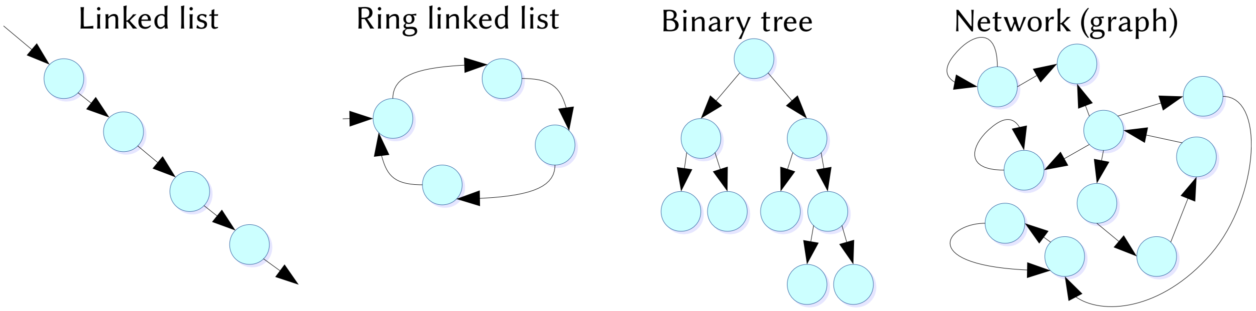
## Drawing a linked list

A linked list consists of a succession of nodes where every node points to the *next* node, until hitting the *null* pointer. But also, every node contains some additional information. The picture below (from the book's slides) illustrates this.



## Questions about linked data structures

- Using pointers allows to develop complex structures
    - No need to limit ourselves to *arity*=1



- Different needs demand different data structures
    - Basic operations (semantics of the data model)
    - Algorithmic efficiency (mathematical analysis)

## Encapsulation for linked lists

- It is better to have a *front* class for a linked list, and then have the nodes be accessed by that class only
    - Why would you access the nodes from within other classes anyway?
- Typical decisions:
    - Where to add new items?
    - How to remove items?
    - How to retrieve items? (In some order? By index?)
    - How to combine lists

## Example code from class

```
// First example

// I used this example to illustrate the concept of a linked list.
// The Node objects store the list's size so far and the pointer.

public class NodeExample {

    static class Node {

        public Node next;
        public int size;

        public Node() {
            next = null;
            size = 1;
        }

        public Node(Node nxt) {
            next = nxt;
            size = nxt.size + 1;
        }
    }

    public static void main(String[] xxxxx) {
        Node a = new Node();
        Node b = new Node(a);
        Node c = new Node(b);
        Node d = new Node(c);
        for (Node j = d; j != null; j = j.next) {
            System.out.println(j.size);
        }
    }

}
```

```
// Second example

// The Node objects stores characters and the pointers.
// The idea here is to, in a way, create an array-like sequence
// of characters.

public class NodeExample1 {

    static class Node {

        public Node next;
        public char daChar;

        public Node(char arg) {
            next = null;
            daChar = arg;
        }

        public void append(char arg) {      // RECURSIVE DEFINITION
            if (next==null)
                next = new Node(arg);
            else
                next.append(arg);
        }
    }

    public static void main(String[] xxxxx) {
        Node a = new Node('h');
        a.append('e');
        a.append('l');
        a.append('l');
        a.append('o');
        for (Node j = a; j != null; j = j.next) {
            System.out.print(j.daChar + " ");
        }
    }

}



// Note: I defined class Node as static and inside class NodeExample,1,2.
//       Why? Well, I wanted to duplicate the files (see the other pages)
//       and I did not feel like renaming the Node class all the time.
//       Then, since it is nested in another class, it has to be “static”
//       so that I can access it from the main. If I were not to reuse
//       the same name for the class, then I could have just defined it
//       outside another class, i.e., in the usual way.
//       I won't ask about classes with the static modifier in the midterm.
// Third example

// This is a more developed example, which incorporates encapsulation.
// I defined class CharLL to represent a linked list of characters.
// Encapsulation is insufficient, for field Node head should be private;
// CharLL should deal with the nodes by itself alone.
// There are also error situations not handled properly; this is but a
// very simple example.

public class NodeExample2 {

    static class Node {

        public Node next;
        public char daChar;

        public Node(char arg) {
            next = null;
            daChar = arg;
        }

        public void append(char arg) {
            if (next==null)
                next = new Node(arg);
            else
                next.append(arg);
        }
    }

    static class CharLL {
        Node head;
        public CharLL() {
            head = null;
        }
        public CharLL(char arg) {
            head = new Node(arg);
        }
        public CharLL(String txt) {
            head = new Node(txt.charAt(0));
            for(int i=1;i<txt.length();i++)
                head.append(txt.charAt(i));
        }
        public void append(String txt) {
            for(int i=0;i<txt.length();i++)
                head.append(txt.charAt(i));
        }
    }

    public static void main(String[] xxxxx) {
        CharLL ll = new CharLL("Whatever");
        for (Node j = ll.head; j != null; j = j.next) {
            System.out.print(j.daChar + " ");
        }
    }

}
```

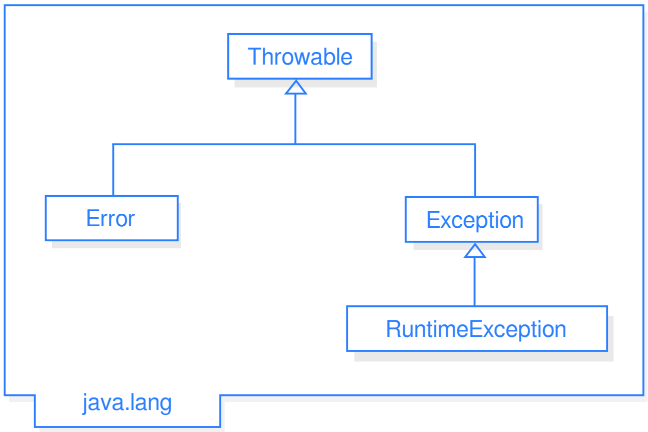
# (Back to Java essentials...)

## Exceptions in Java

- You can expect the possibility that methods throw *exceptions*, and you can catch these
- For example:

```
try {
  int x = 10/0;
} catch( ArithmeticException e) {
  System.out.println( e );
}
```

- The code: **try** initiates the block that you are concerned with; **catch** captures errors, if they occurred
- An **Exception** is an object modeling the error
- **ArithmeticException** subtypes **Exception**



(CC Arunreginald, link https://commons.wikimedia.org/wiki/File:Java_exception_classes.svg)

- You can check the hierarchies in java.lang, which happens to contain Throwable, here: http://docs.oracle.com/javase/7/docs/api/java/lang/package-tree.html
- Different types of subtypes? Example:

```
try{
    …
} catch( IOException e ) {
    …
} catch( NullPointerException e) {
    …
} catch( Exception e ) {
    …
} …
```

- The order of the catch statements matters—you should go from specific to general

- I am lazy, so I prefer to catch **everything**:

```
try {
   …
} catch( Exception e ) {   // yes, I capture the supertype
   …
}
```

- You can also raise exceptions at will:

```
throw new Exception("I AM A MEANIE EXCEPTION");
throw new ArithmeticException("I AM A MEANIE EXCEPTION");
```

- What NetBeans shows as errors are usually exceptions!

**<< Lecture 16 will resume from here >>**