Data structures I –
Arrays and Linked lists

Programming

- What makes a program **good** or what is evidence of **good** programming ?
  - it works (as specified!)
  - it is easy to understand and modify
  - it is reasonably efficient
- We address these requirements by using appropriate structures and constructs that manipulate data

Primitives

- The simplest data structures in Java are the primitive types.

Keyword	Description	Size/Format
	(integers)	
`byte`	Byte-length integer	8-bit two's complement
`short`	Short integer	16-bit two's complement
`int`	Integer	32-bit two's complement
`long`	Long integer	64-bit two's complement
	(real numbers)	
`float`	Single-precision floating point	32-bit IEEE 754
`double`	Double-precision floating point	64-bit IEEE 754
	(other types)	
`char`	A single character	16-bit Unicode character
`boolean`	A boolean value (`true` or `false`)	true or false

ARRAYS

Beyond Primitives

- The expression "data structure", however, is usually used to refer to more complex ways of storing and manipulating data, such as arrays, stacks etc.
- We begin by discussing the simplest, but one of the most useful data structures, namely the ***array***.

Declaring arrays:

```
int[] ages;
String[] band;
Double[] vector;
```

Instantiating arrays:

```
ages = new int[6];
band = new String[4];
```

Both together:

```
int[] ages = new int[6];
```

Array Bounds and Length

- Array indexing in Java is zero-based (like c/c++) but unlike Matlab) so

  ```
  int[] ages = new int[6];
  ```

  gives us

  ```
  ages[0], ages[1], ... ages[5]
  ```

- When an array is created its size is held in a public constant which can be accessed as
  ```
  ages.length = 6;
  ```
- Acessing an array element that is out of bounds gives a run time error `ArrayIndexOutOfBoundsException`

Initialisation of Arrays

- When arrays are created using *new* operator the elements are automatically initialised, **but don't bank on it**

`int[] ages = new int[6]; -` what is in array `ages`?

```
for (int i=0; i<6; i++)
  {
  ages[i] = 0;
  }
```

- Can also be initialised another way as

```
int[] ages = {0, 0, 0, 0, 0, 0};
```

Summing Elements of an Array

```
// Program to find the sum of all the elements of an array
class summation
{
    public static void main(String[] args)
    {
    long[] a = new long[101];
    long sum;
    int i,numbers;
    numbers=100;
    // initialise the array a using the loop counter
    for (i=1; i<=numbers; i++)
      {
      a[i]=(long)i;
      }
    sum=0;
    for (i=1; i<=numbers; i++)
      {
      // do summation
      sum = sum + a[i];
      }
    System.out.println("sum of numbers between 1 and "+numbers+" is "+sum);
    }
}
```

Basic Statistics calculations: Mean

```
// Program to find the mean of a set of numbers stored in an array
class statsmean
{
    public static void main(String[] args)
    {
    double[] a = new double[101];
    int i,numbers;
    double sum,mean;
    numbers=100;
    // initialise the array a using the loop counter
    for (i=1; i<=numbers; i++)
      {
      a[i]=(double)i;
      }
    sum=0;
    for (i=1; i<=numbers; i++)
      {
      // do summation
      sum = sum + a[i];
      }
    // calculate mean
    mean = sum/((double)numbers);
    System.out.println("mean of numbers between 1 and "+numbers+" is "+mean);
    }
}
```

Finding the Maximum Element

```
// Program to find the maximum element of an array
class maximumofarray
{
    public static void main(String[] args)
    {
    double maximum;
    int i;
    double [] a={1.5,2.3,4.2,-9.4,2.0,12.9,-5.0,12.9,-0.1,-15.0,0.0};
    maximum=a[0];
    for (i = 1; i <= 10; i++)
      {
      if (a[i] > maximum)
        {
        maximum = a[i];
        }
      }
    System.out.println(“The maximum element is “+maximum);
    }
}
```

Assignment of an Array

- Arrays are treated as Objects in Java and, as such, obey the same rules for equality and assignment.
- An array variable is a *reference* variable. It stores information about where to locate the array elements (like a pointer in c/c++).
- All array variables require the same amount of storage, irrespective of the size of the arrays or the nature of their elements.

```
int[] a = new int[20];
int[] b;
b = a;
```

  - `b` now holds the same address as `a`.
  - only **one array** but there are now **two ways of addressing** it.
  - Any change to `b[i]` will change `a[i]`

Assignment of an Array

- To create a copy of an array requires more effort

```
b = new int[a.length];
for (int i = 0; i < a.length; i++)
  {
  b[i] = a[i];
  }
```

- There are now two distinct arrays, of the same length and with the same contents.
- Change to either will leave the other unaffected, so the assignment

```
b[5] = 10;
```

does not change `a[5]`

Equality of two Arrays

- To compare two arrays i.e. `a==b` is true if and only if they have the same array reference
- Alternatively, comparison is done that checks
  1. array `a` and `b` have the same array length
  2. on an element by element basis that `a[i]==b[i]` must be true for every comparison

```
same=true;
if (a.length != b.length)
  same=false;
if (same)
  {
  for (int i = 0; i < a.length; i++)
    {
    if (a[i] != b[i])
      same=false;
    }
  }
```

Arrays as Parameters

- Array variables can be used as parameters to methods.
- When an array is passed to a method, a copy is made of the actual parameter.
- Since this is only a *reference to the array*, **only the reference is copied**, which makes for efficient passing of arrays to methods.
- The method now has access to the elements of the array, via the reference, which means that they can be changed as the method chooses.
- The reference itself, however, will not be changed outside the method.

Multi-dimensional Arrays

- The elements of an array can themselves be arrays
- The array is said to be a **multi-dimensional** array
- To declare and create a rectangular integer array with 2 rows and 3 columns:

```
int a[][] = new int[2][3];
```

- It can be initialised by using nested `for` loops such as

```
for (int i = 0; i < a.length; i++)
  {
  for (int j = 0; j < a[i].length; j++)
    {
    a[i][j] = 0;
    }
  }
```

Multi-dimensional Arrays

```
int [][] table = { {1, 2, 3}, {10, 20, 30}};
```

contents :

1	2	3
10	20	30

reference (or location):

0,0	0,1	0,2
1,0	1,1	1,2

- We refer to `table[0][1]` which is 2, `table[1][2]` which is 30, and so on.
- This structure can be used to define a ***matrix***
- Note: **all array indices are integer numbers**

$$\begin{pmatrix} 1 & 2 & 3 \\ 10 & 20 & 30 \end{pmatrix}$$

Matrix Addition

This program uses a nest of two loops to add two 3x3 matrices a and b together, and print the result matrix c.

```
// Program to add two 3x3 matrices
class matrixaddition
{
    public static void main(String[] args)
    {
    int[][] a = {{1,2,3}, {3,4,5}, {5,6,7}}
    int[][] b = {{7,6,5}, {5,4,3}, {3,2,1}}
    int[][] c = new int[3][3]
    int i,j;
    for (i = 0, i < 3; i++)
        {
        for (j = 0; j < 3; j++)
            {
            c[i][j] = a[i][j] + b[i][j];
            }
        System.out.println("a+b = "+c[i][0]+" "+c[i][1]+" "+c[i][2]);
        }
    }
}
```

Matrix Multiplication

- Multiplying two matrices can be done using triple nested loop

```
// Program to multiply two 3x3 matrices
class matrixmultiply
{
    public static void main(String[] args)
    {
    int[][] a = {{1,2,3}, {3,4,5}, {5,6,7}}
    int[][] b = {{7,6,5}, {5,4,3}, {3,2,1}}
    int[][] c = new int[3][3]
    int i,j,k;
    // process rows
    for (i = 0, i < 3; i++)
      // process columns
      for (j = 0; j < 3; j++)
        {
        c[i][j] = 0.0;
        // process row-column interactions and sum them into array c
        for (k = 0; k < 3; k++)
          {
          c[i][j] = c[i][j] + a[i][k] * b[k][j];
          }
        }
      System.out.println(“a.b = “+c[i][0]+” ”+c[i][1]+” ”+c[i][2]);
    }
}
```

```
int[][] a = {{1,2,3}, {3,4,5}, {5,6,7}}
    int[][] b = {{7,6,5}, {5,4,3}, {3,2,1}}
    int[][] c = new int[3][3]
    int i,j,k;
```

1	2	3
3	4	5
5	6	7

7	6	5
5	4	3
3	2	1

1*7 + 2*5 + 3*3	1*6 + 2*4 + 3*2	1*5 + 2*3 + 3*1

```
// process rows
    for (i = 0, i < 3; i++)
      // process columns
      for (j = 0; j < 3; j++)
        {
        c[i][j] = 0.0;
        // process row-column interactions and sum them into
array c
        for (k = 0; k < 3; k++)
          {
          c[i][j] = c[i][j] + a[i][k] * b[k][j];
          }
        }
      System.out.println("a.b = "+c[i][0]+" "+c[i][1]+"
"+c[i][2]);
    }
}
```

Array Access and Memory Allocation

- One of the principal reasons arrays are used so widely is that their elements can be accessed in ***constant time***.
- So the time taken to access `a[1]` is the same as `a[100]`
- The address of a[i] can be determined arithmetically by adding a suitable offset to the address of the head of the array AND the array is stored ***contiguously*** in memory .
- This is a +ve and a –ve since once the space is allocated, it cannot be extended (here arrays are static data structures).
- On the other hand, arrays can be allocated dynamically in Java (not covered in this course).

OTHER ABSTRACT DATA TYPES

Abstract Data Types

- Array structures are useful but have limitations and can be inefficient
- Structures with a number of different components or attributes are sometimes called ***composites*** or ***abstract data types*** (ADTs).
- ADTs separate the *specification* (what kind of thing we're working with, and what operations can be performed on it) and *implementation* (how the thing and its operations are actually implemented).

Benefits of ADTs

- The benefits of using ADTs include:
  - Code is easier to understand (e.g. it is easier to see "high-level" steps being performed, not obscured by low-level code or “clutter”).
  - Implementations of ADTs can be changed (e.g. for efficiency) without requiring changes to the program that uses the ADTs.
  - ADTs can be reused in future programs

ADTs defined using Classes

- *Class* features of object-oriented programming languages like Java make it easy for programmers to use ADTs
- Each ADT corresponds to a *class*
- The operations on the ADT are the class's *public methods*
- The user, or client of the ADT only needs to know about the method *interfaces* (the *names* of the methods, the *types* of the parameters, the purpose of the methods, and what if any values they return), **not the actual implementation**.

Public and Private views of ADTs

- There are two parts to each ADT:
  - The *public* or *external* part, which consists of:
    - The conceptual picture (the user's view of what the object looks like, how the structure is organized). e.g. a 2D matrix
    - The conceptual operations (what the user can do to the ADT). e.g. addition of two matrices
  - The *private* or *internal* part, which consists of:
    - The representation (how the structure is actually stored). e.g. a linked list of elements
    - The implementation of the operations (the actual code). e.g. creating a new linked list

Operations on ADTs

- There are many possible operations that could be defined for each ADT.

- Commonly used include
  - Initialize
  - Add/insert data
  - Access data
  - Edit/modify data
  - Remove/delete data
  - ...

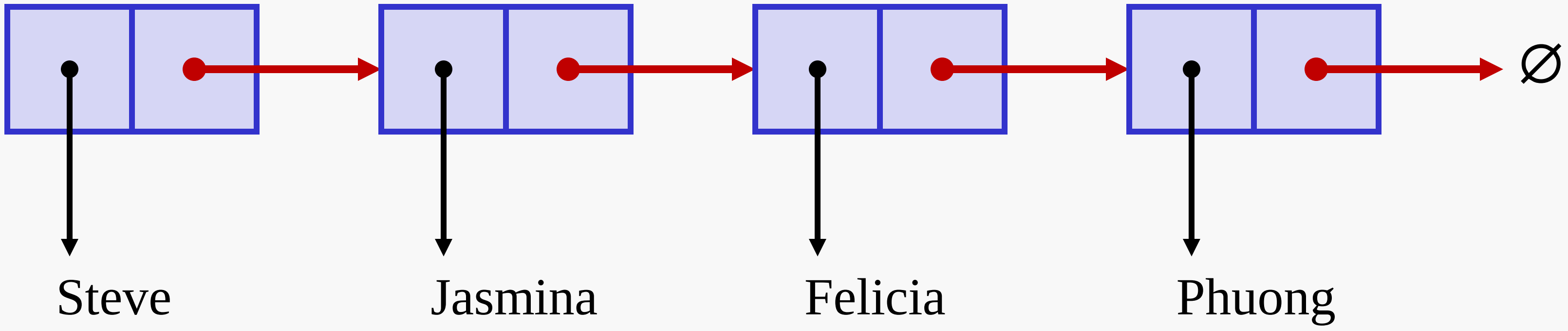
LINKED LISTS

Lists

- There are different kinds of ways to *link* lists of information together

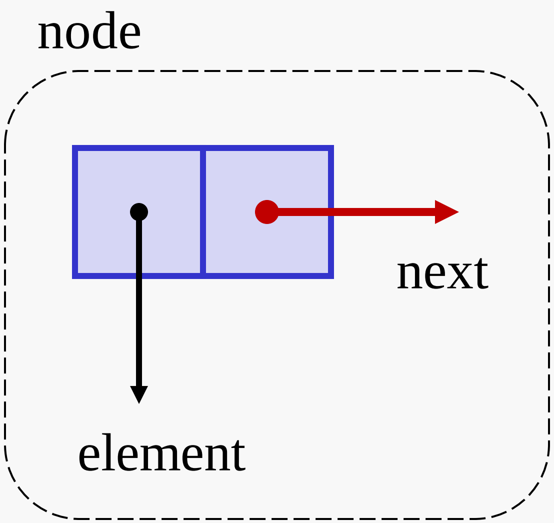
- A *linked list* is based on the concept of a *self-referential object* - an object that refers to an object of the same class.

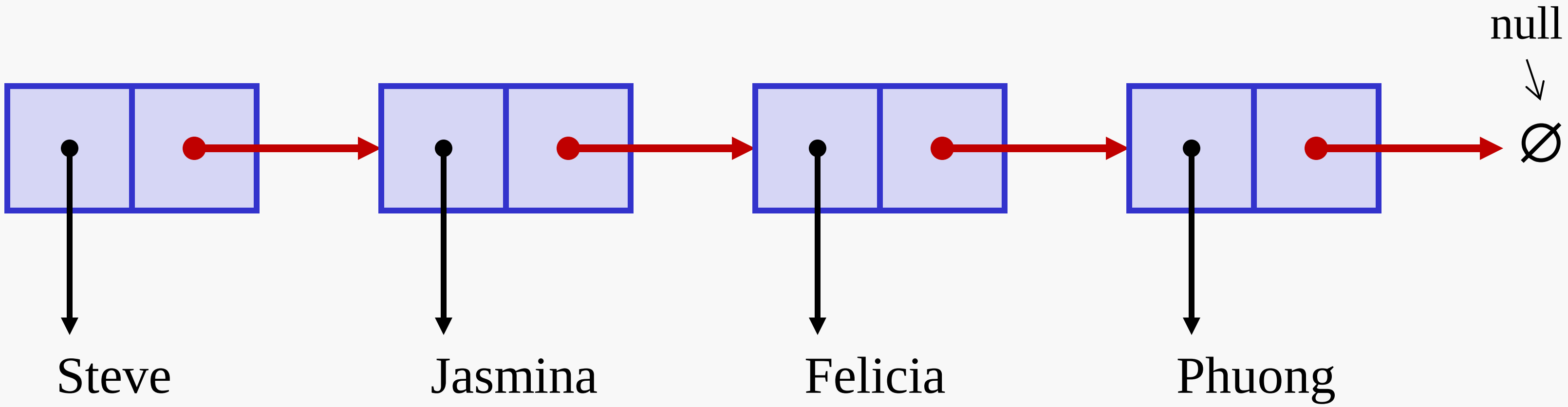
- Example of a list of names that are linked together



Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





Before we create a Linked List

- We need to define the ADT that reflects the structure of
  - element
  - link to next node
- A list will be represented internally by a single list of nodes (`SLinkedList)` that is made up of the `head` of the list
- Initially we set `head` to `null` to indicate an empty list
- ***We will illustrate using a list of Strings…***

The StringNode Class for a node containing a *String*

```
public class StringNode {
  // Instance variables:
  private String element;
  private StringNode next;
  /** Creates a node with null references to its element and
   next node. */
  public StringNode() {
    this("", null);
  }
  /** Creates a node with the given element and next node. */
  public StringNode(String e, StringNode n) {
    element = e;
    next = n;
  }
```

The StringNode Class for a node containing a *String (continued)*

```
// Accessor methods:
  public String getElement() {
    return element;
  }
  public StringNode getNext() {
    return next;
  }
  // Modifier methods:
  public void setElement(String newElem) {
    element = newElem;
  }
  public void setNext(StringNode newNext) {
    next = newNext;
  }
}
```

Constructor for LinkedList

- Calling the `SLinkedList` Constructor creates a list with one node called the `head` – the element value is “” and the next pointer is to `null`.

```
public class SLinkedList {

    protected StringNode head;
    public  SLinkedList() {
        head = new StringNode();
    }
```

The LinkedList Class for Singly Linked Lists

```
// For a Single Linked List of Strings
public class SLinkedList
  {
  ...
  // Checks if list is empty
  public boolean isEmpty();
  // add node to head of list
  public void addFirst(String element);
  // remove node from head of list
  public void removeFirst();
  // add node to tail of list
  public void addLast(String element);
  // remove node from tail of list
  public void removeLast();
  // add node containing element after 1st occurrence of entryafter
  public void addMid(String element, String entryafter);
  // remove first node containing element
  public void removeMid(String element);
  // print out the single linked list defined by thelist
  public static void printList(SLinkedList thelist)
}
```

Inserting at the Head

1. Allocate a new node

   `new StringNode();`

2. Insert new element

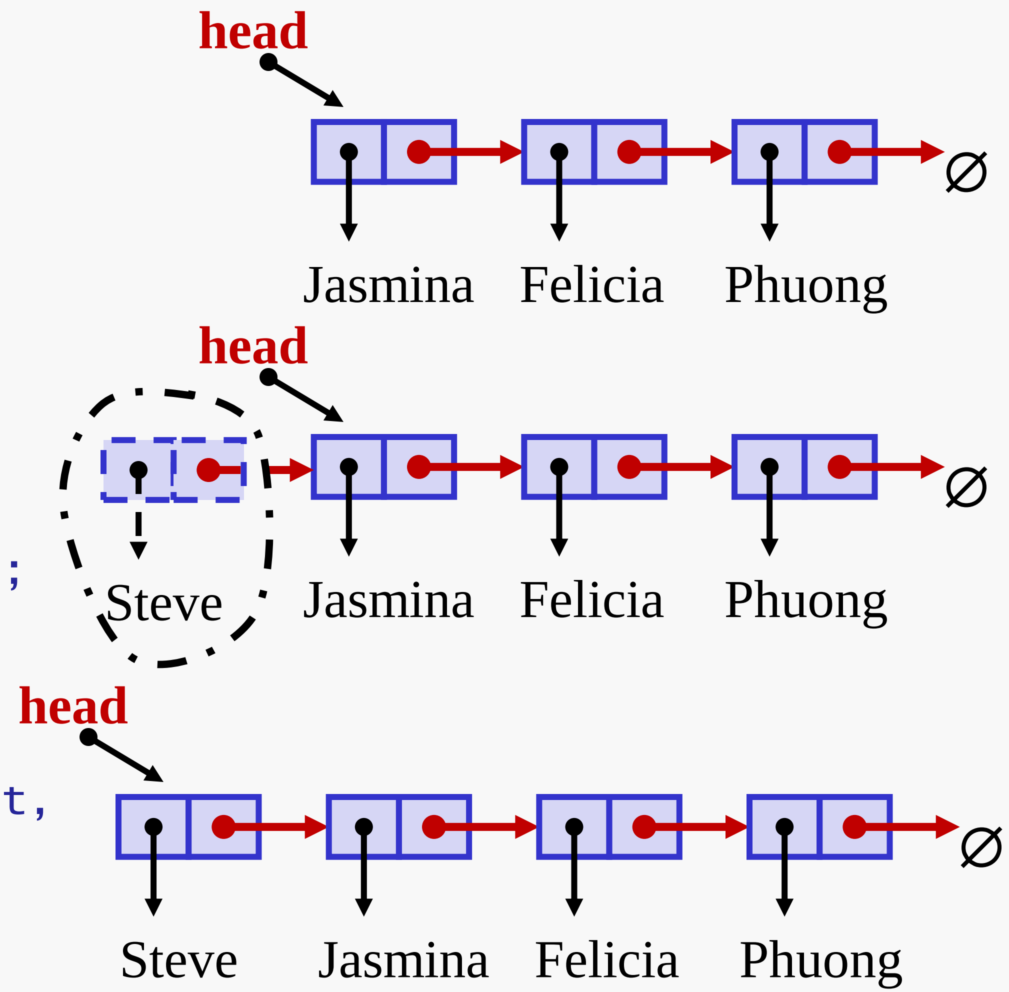
   `new  StringNode(element,);`

3. Have new node point to old head

   `new StringNode(element, head);`

4. Update `head` to point to new node

   ```
   head = new  StringNode(element,
   head);
   ```



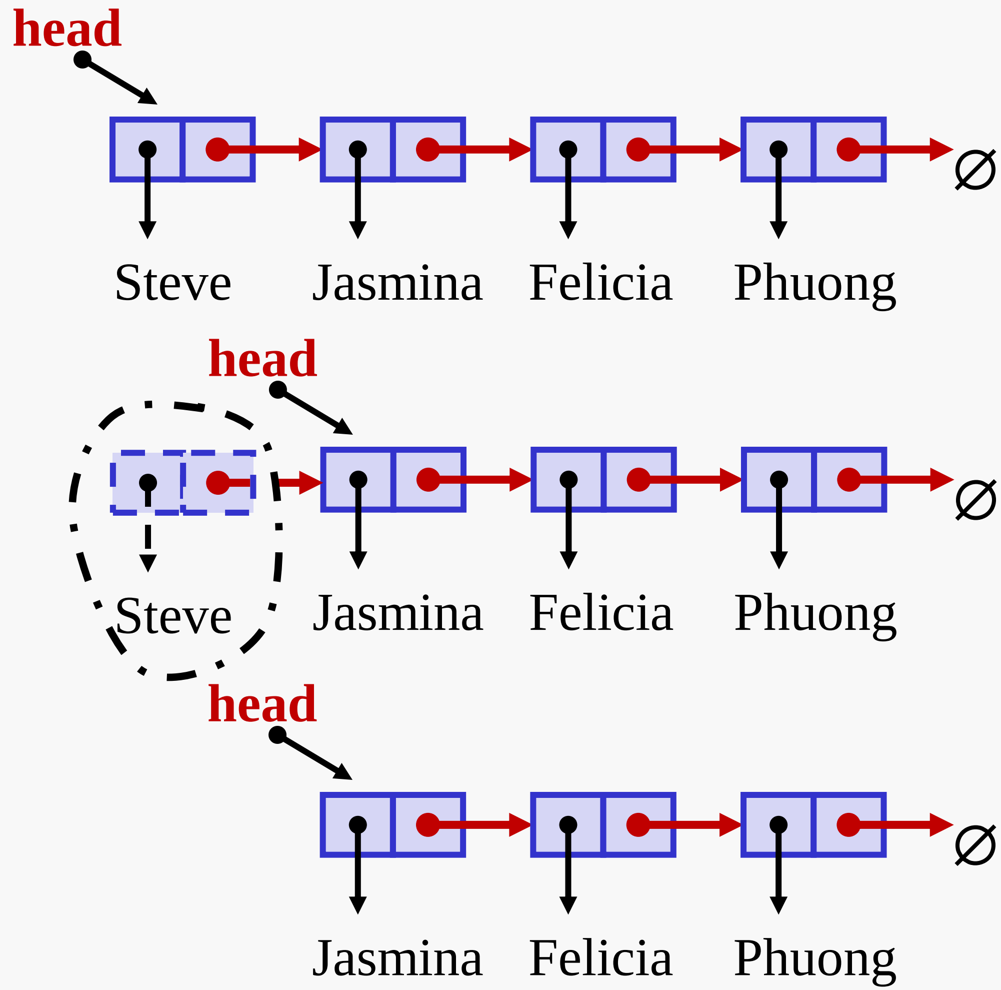
Inserting at the Head

- The following method inserts a new node at the `head` of the list. Whether the list was empty or not.

```
// add node to head of list
public void addFirst(String element)
  {
  head = new StringNode(element, head);
  }
```

- the statement **`new StringNode(element, head)`** does the steps
  1. Allocate a new node
  2. Insert new element
  3. Have new node point to old head
- the assignment **`head =`** does the last step
  4. Update `head` to point to new node

Removing at the Head



1. Update `head` to point to next node in the list

```
head = head.getNext();
```

2. Allow garbage collector to reclaim the former first node

```
oldhead.setNext(null);
```

Removing at the Head

- The following method removes the element at the `head` of the list. The element referred to by the previous head's **`next`** field now becomes the new head of the list.

```
// remove node at head of list
public void removeFirst()
  {
  StringNode oldhead;
  oldhead = head;
  if (head != null) {
     head = head.getNext();
     oldhead.setNext(null);
     }
  else {
     throw new NoSuchElementException();
     }
  }
```

Linked lists

To be continued…