• Interfaces. *(cf “The Java Programming Language” book, ch 4.)* These provide a very natural way to design a specification for code to solve a problem, think of them as being contracts for classes. Effectively, cracking a problem can be broken down into designing the solution and then implementing it; the design stage being stipulated by listing ‘skeletal’ classes having methods (but no definitions), and the implementation stage being split between implementing the contracts (classes) and writing code which uses them to actually solve the problem. Hence a programming team could be divided into these three groups, allowing them to work largely in parallel once the design has been set.

• Imagine building a spec sheet for a queue ... folk join at the back and are served (and leave) from the front. Being aggressive, we could insist that after joining a queue, the only way to leave is from the front. This might lead to ...

```java
public interface Queue {
    // note that there aren’t even hints of method definitions below, save for the comments!!!
    void joinQ ( Person p ) ; // the person p gets added to the back of the queue
    Person leaveQ ( ) ; // to return the person at the front, and then dump them in favour of the next person in line
    int getLength ( ) ; // to return the number of people in the queue
    boolean isFull ( ) ; // to return true if there’s no room left in the queue
    boolean isEmpty ( ) ; // to return true if there’s no-one in the queue
} // end interface Queue
```
• Note that promising to implement an interface forces (via the compiler) an implementation of every method listed in that interface (but doesn’t insist on the method definitions being sensible!) ...  

```java
public class Q1 implements Queue {
    void joinQ (Person p) { return ;}
    Person leaveQ () { return new Person ("anonymous") ;}
    int getLength () { return Integer.MAX_VALUE ;}
    boolean isFull () { return true ;}
    boolean isEmpty () { return true ;}
    public Q1 (int n) {
        System.out.println ("This queue is so efficient it takes up almost no space!!!") ;
    } // constructor
    public Q1 ( ) { this (-47) ; } // equally silly default constructor
} // end interface Queue

Sadly, the compiler will be perfectly happy with this! Perhaps rather better might be ...

```java
public class Q2 implements Queue {
    private Person [ ] storage ;
    private int length , front , back , size ;
    private final int DEFAULT_SIZE = 100 ;
    void joinQ (Person p) {
        if ( ! isFull() ) { storage [ back++ ] = p ; length++ ;}
        else System.out.println ("Sorry, full up") ;
    } // end joinQ method
    Person leaveQ () {
        Person temp ;
        if ( ! isEmpty() ) { temp = storage [ front++ ] ; length-- ; return temp ;}
        else System.out.println ("Sorry, queue empty") ; return null ;
    } // end leaveQ method
    int getLength () { return this.length ;}
    boolean isFull () { return ( this.back == size ) ;}
    boolean isEmpty () { return ( length == 0 ) ;}
    public Q2 (int n) {
        size = n ; storage = new Person [ n ] ; length = 0 ; front = 0 ; back = 0 ;
    } // hopefully n > 0 !
    public Q2 ( ) {
        this ( DEFAULT_SIZE ) ; } // or some such default value
} // end class Q2
```
• Interfaces make no presumption about implementations - indeed, there can be multiple strikingly different implementations of any given interface. As further examples (cf the Weiss textbook, ch 3) ...

```java
public class Q3 implements Queue {
    private Person [] storage ;
    private int length , front , back , size ;
    private final int DEFAULT_SIZE = 100 ;

    void joinQ ( Person p ) { if ( ! isFull( ) ) { storage [ ( back++ ) % size ] = p ; length++ ;} else System.out.println ( "Sorry, full up" ) ; } // end joinQ method

    Person leaveQ ( ) { if ( ! isEmpty( ) ) { Person temp = storage [ ( front++ ) % size ] ; length-- ; return temp ;} else System.out.println ( "Sorry, queue empty" ) ; return null; } // end leaveQ method

    int getLength ( ) { return this.length ;}
    boolean isFull ( ) { return ( this.length == size ) ;}
    boolean isEmpty ( ) { return ( length == 0 ) ;}

    public Q3 ( int n ) { size = n ; storage = new Person [ size ] ; length = 0 ; front = 0 ; back = 0 ;} // hopefully n > 0 !
    public Q3 ( ) { this ( DEFAULT_SIZE ) ; }
} // end class Q3
```

The difference between the Q2 and Q3 implementations is that for Q2 we stored the data in a linear array (wasting space as it emptied, and losing queue re-usability), whereas for Q3 the storage was in a ‘quasi-circular’ array (allowing continual use). It’s worth noting that Q2 might be well-suited in a situation having limited resource (eg queueing for food or for concert tickets) whereas Q3 would be a better match for a renewable resource (eg printing or processor access, or customer service).

• Thinking more structurally about a queue, we could build a more elegant flavour ...

CS 2110 Data Structures - queues
The nifty thing about this approach is that the queue never gets full (until the computer does!), and is always being sized dynamically according to need (so no wasted space). Of course, nothing is ever ideal; so in this case there is some computational overhead in creating each node, so in a situation where the queue size will remain roughly the same it might be better to use an array flavour, but where the size changes frequently and significantly, this pointer-based approach is better. Effectively, each time we add a Person, a Node is created to house them, and it’s that Node which is tacked onto the end of the queue. Each time a Person leaves, the pointer (front) advances, so that then nothing is referring to the previous front of the queue, so that it can be garbage-collected, thus allowing reclamation of space.
A very similar structure, but which allows joining and leaving from anywhere within, is called a ‘linked list’. This could be implemented with arrays, but we’ll focus on a similar pointer-based approach, using our existing Node class, and thinking of leaving being the deletion of a Person from the ‘current’ location, and joining as interposing a person right after the current spot ...

```java
public class L1 implements List {
    private Node header, current; private int length;
    public void join ( Person p ) throws BadInsertionPoint {
        if ( current == null ) throw new BadInsertionPoint();
        Node temp = new Node ( p , current.getNext() ) ;
        current.setNext ( temp ) ; length++ ;
    } // end join method
    public void leave ( ) {
        if ( ! isEmpty( ) && inList ( ) ) {
            getPrevious ( ).setNext ( current.getPrev() ) ;
            length-- ;
        } else System.out.println ( "Sorry, list is empty" ) ;
    } // end leave method
    private Node getPrevious ( ) {
        for ( Node n = header ; n != null && n.getNext ( ) != null ; n = n.getNext ( ) )
            if ( n.getNext ( ) == current ) return n ;
        return null ;  } // this return is only reached if no ‘previous’ was found
    public void setCurrent ( Person p ) { // sets ‘current’ to point to where p is
        for ( current = header ; current != null ; current = current.getNext( ) )
            if ( current.getData ( ).equals ( p ) ) return ;
    } public boolean inList ( ) { return ( current != null && current != header ) ;}
    public int getLength ( ) { return this.length ;}
    public boolean isFull ( ) { return false ;}
    public boolean isEmpty ( ) { return ( length == 0 ) ;}
    public L1 ( int n ) { header = new Node ( ) ; current = header; length = 0 ;}
    public L1 ( ) { this ( 0 ) ;}
} // end class L1
```

public interface List {
    void join ( Person p ) ; // p inserted after ‘current location’
    void leave ( ) ; // the current locn’s content is deleted
    int getLength ( ) ; // the number of people in the list
    boolean isFull ( ) ;
    boolean isEmpty ( ) ;
    void setCurrent ( Person p ) ; // set ‘cur loc’ to where p is
} // end interface List

joining a linked list

leaving a linked list
• Actually this is a poor approach; having just one ‘current location’ will make it hard to sort a list (we’d need at least two ‘currents’ for that), and there’s some serious awkwardness in the way we manipulated current. Far better would be to divvy up the work between the list and the arrows pointing into the list (iterators) according to where things rightly belong, but we’ll delay doing this until we’ve introduced the other two main data structures.

• **A stack** is another kind of emaciated list, this time only allowing entering and leaving from one end (the ‘top’)

```java
public class S1 implements Stack {
    private Node top;
    private int length;
    public void join(Person p) {
        top = new Node(p, isEmpty() ? null; top);
        length++;
    }
    public Person leave() {
        if (!isEmpty()) {
            Person temp = top.getData();
            top = top.getNext();
            length--;
            return temp;
        } else System.out.println("Sorry, stack is empty");
    }
    public int getLength() { return this.length; }
    public boolean isFull() { return false; }
    public boolean isEmpty() { return (length == 0); }
    public S1(int n) { length = 0; } // n is irrelevant here
    public S1() { this(0); }
} // end class S1
```

```java
public interface Stack {
    void join(Person p); // p inserted at the top
    Person leave(); // the top is deleted
    int getLength(); // # of people in the stack
    boolean isFull();
    boolean isEmpty();
} // end interface Stack
```
• Stacks are surprisingly powerful animals - much of what goes on in your computer is done via stacks (of commands, of values, etc.). However, as quick examples ...

(i) Two ways of writing arithmetic expressions are *infix*, as in \( a + ((b-c)\times((d+e)-(f/g))) \) and *postfix* (or *reverse Polish*), as in \( a \ b \ c - \ d \ e + f \ g / - * + \). To evaluate the latter, we can put the values on a stack, and on seeing an operator, apply it by pulling the top off the stack (twice) and then putting the newly computed value on the stack ...

(ii) As you might guess from the above, we can also use a stack to convert an infix to a postfix expression, essentially using the stack as a ‘storage-until-ready’ holding location ...

This will produce the postfix version above from the infix flavour term-by-term under the algorithm:
(i) output non-operands immediately
(ii) rank operands by strength, so + and - are at the bottom, * and / are next, and “(” is the highest
(iii) on reading an operand, pop from the stack (into the output) until seeing a stronger one, then push that operand
(iv) on reading a “)”, pop everything from the stack until the “(”, which is discarded (popped but not output)
(v) if there’s no more input, then pop from the stack until empty
• Trees are essentially lists on steroids, with nodes having multiple ‘nexts’, a binary tree having two ‘nexts’ (a left and a right) from each node.

So a binary tree could be modeled using nodes as in the code on the right. A common application is for binary search trees, which assign choices on insertion to place left or right of a ‘current’ node depending on whether the new value to be inserted is smaller or larger than the ‘current’ one. This can allow for a log(n) search speed … certainly faster than the linear search on a regular array!! For example, the sequence of numbers, 47, 72, 53, 26, 41, 97, 83, 14, would yield the following tree:

Of course, whilst insertion is easy, deletion of a value requires rather more care if the binary nature of the tree is to be preserved! Removing 83 from our example is trivial, or losing 97 instead is easy, but deleting 72 would need some finesse, especially if the parts of the tree below 53 and 97 were to be far more greatly populated.
Before we continue with trees, we should really address the weakness in our list class. As hinted on page 20 of these slides, a far cleaner approach would be to separate according to purpose or function, so that Lists would purely hold data, a fresh class being built of Iterators whose function is to point into a List (rather like having integers to iterate through an array of data).

public interface List {
    void join (Person p, Itr t) throws BadInsertionPoint; // to insert p after t's location
    void remove (Person p) throws ItemNotFoundException; // to delete the (first) appearance of p
    Itr find (Person p) throws ItemNotFoundException; // to return a pointer to the (first) appearance of p
    int getLength (); // the number of people in the list
    boolean isFull ();
    boolean isEmpty ();
    Itr first (); // returns pointer to the first node in list
} // end interface List

So here we use an Iterator to point into the list (to allow insertion *after* a given location). Notice that we've restricted the List interface to have methods which feel list-like, and we devolve to the Itr interface those methods which are more 'pointy' (gathering data or moving around).

public interface Itr {
    boolean isInList (); // is this pointing into the list
    boolean atEnd (); // is this pointing at the last node
    Node getNode (); // returns current node being pointed to
    Person getData () throws BadPointer; // returns data in current
    Itr getCopy (); // returns a copy of this iterator
    Itr pointNext (); // returns iterator pointing to next
    void plusPlus (); // move pointer to 'next' node
    void minusMinus (); // move pointer to 'previous' node
} // end interface Itr
There are many choices we could make in writing the actual implementations of these interfaces, and as we’ll see, there are questions of actual ‘taste’ which arise, and these can be frequently hard to resolve entirely satisfactorily ...

```java
public class ItrP implements Itr {
    private L2 list; // allows us to specify which list we're navigating
    private Node current;

    public boolean isInList() {
        return current != null && current.getData() != null;
    }

    public boolean atEnd() { return current.getNext() == null; }

    public Node getNode() { return current; }

    public Person getData() throws BadPointer {
        if (!isInList()) throw new BadPointer();
        return current.getData();
    }

    public Itr getCopy() { return new ItrP(this.list, current); }

    public Itr pointNext() { return new ItrP(this.list, current.getNext()); }

    public void plusPlus() {
        if (current != null) current = current.getNext();
    }

    public void minusMinus() {
        if (current != null && current.getData() != null) {
            // so not the header
            for (Itr i = new ItrP(this.list); !i.atEnd(); i.plusPlus())
                if (i.pointNext().getNode() == current) current = i.getNode();
        } // end if current != header, if otherwise we simply let current stay there
    }

    public ItrP(L2 list, Node n) { this.list = list; this.current = n; }
    public ItrP(L2 list) { this(list, ((L2) list).getHeader()); }
    public ItrP() { this(null, null); }
}
```

```java
public interface Itr {
    boolean isInList();
    boolean atEnd();
    Node getNode();
    Person getData() throws BadPointer;
    Itr getCopy();
    Itr pointNext();
    void plusPlus();
    void minusMinus();
}
```

Now we can implement our list more purely as a storage device, utilising the list iterator to allow us to move around through the data.

```java
public class L2 implements List {
    private Node header;  private int length;
    public Node getHeader () { return this.header; }

    public void join ( Person p , Itr t )
        throws BadInsertionPoint {
        if ( t == null ) throw new BadInsertionPoint ( );
        t.getNode ().setNext ( new Node ( p , t.pointNext ().getNode ( ) ) );
        length++;  }
    } // end join method

    public Itr find ( Person p )
        throws ItemNotFound {
        Itr t = isEmpty () ? null : first ( );
        try {
            while ( t.isInList () && ! t.getData ().equals ( p ) )
                t.plusPlus ( );
        } catch ( BadPointer bp ) {} // can't happen since t isInList
        if ( t == null ) throw new ItemNotFound ( );
        return t ;  }
    } // end find method

    public void remove ( Person p )
        throws ItemNotFound {
        if ( ! isEmpty ( ) ) {
            Itr t = find ( p );  // throws exception if not found
            t.minusMinus ( );
            t.getNode ().setNext ( t.pointNext ().pointNext ().getNode ( ) );
            length--;  }
        else  System.out.println ( "Sorry, list is empty" ) ;
    } // end remove method

    public int getLength ( ) { return this.length ; }
    public boolean isFull ( ) { return false ; }
    public boolean isEmpty ( ) { return ( length == 0 ) ; }
    public Itr first ( ) { return new ItrP ( this , header.header.get ( ) ) ; }

    public L2 ( int n ) { header = new Node ( ) ;  length = 0 ; }
    public L2 ( ) { this ( 0 ) ;  }
} // end class L2

This implementation has aspects which are unsatisfactory, for example having such ready access to the header node is dangerous. Also, it’s time we addressed the performance hit from restricting our attention to singly linked lists; the extra space taken by having doubly linked nodes is negligible compared to the linear cost (versus constant time) of finding a node’s previous!
So assuming a node containing data (Person) and a next and prev (Node) permits the following more efficient implementation of our iterator interface (the L2 list implementation remains, though with the getHeader( ) method deleted).

```
public class ItrP implements Itr {
    private Node current;

    public boolean isInList () {
        return current != null && current.getData () != null ;
    }
    public boolean atEnd () { return current.getNext () == null ;}
    public Node getNode () { return current ;}
    public Person getData () throws BadPointer {
        if ( ! isInList () ) throw new BadPointer ( ) ;
        return current.getData () ;
    }
    public Itr getCopy () { return new ItrP ( current ) ;}
    public Itr pointNext () { return new ItrP ( current.getNext () ) ;}
    public void plusPlus () {
        if (current != null) current = current.getNext () ;
    }
    public void minusMinus () { // ignore current == null or the header
        if (isInList ( ) ) current = current.getPrev () ;
    }
    public ItrP ( Node n ) { this.current = n ; }
    public ItrP () { this ( null ) ; }
} // end class ItrP
```

You might like to try rewriting these implementations with the getNode( ) method deleted, since that too is a tad unsafe (naturally this means adjusting the Itr interface!!).

Clearly it would be more than a nuisance to have to write fresh versions of all these data structures for every kind of object (Person, Sausage, Rhubarb, ...). Whilst we could handle this by having our structures be based on Objects and then exploiting casting, there is a convenient solution to this via Java’s generic types, which we’ll introduce after a quick detour on recursion and induction.

You should also try writing classes to implement stacks, queues and binary trees with separate classes of iterators to facilitate manipulation of their data. However, in order to write a clean binary tree implementation using tree iterators to insert and delete values, we should first remind ourselves about recursion.