Classes ... making the manufacturer. It’s now time to turn our attention to the manufacturer of all these reference objects. As an example ...

public class BankAcc {
    private float balance; // data field to hold values for each BankAcc manufactured

    public float getBalance() { // accessor method
        return balance;
    }

    public void setBalance(float bal) { // mutator method
        balance = bal;
    }

    public float spend(float amt) { // methods to do stuff
        balance -= amt;
    }

    public float deposit(float amt) {
        balance += amt;
    }

    public BankAcc() { // default constructor
        balance = 0.0;
    }

    public BankAcc(float amt) { // another constructor
        balance = amt;
    }
}

So then, how does this get used?

In general it’s a very good idea to default to making as many of the data fields private as possible, and use accessor/mutator (setter/getter) methods to control access to them.

public class GRQ {
    public static void main(String[] args) {
        BankAcc owen = new BankAcc();
        owen.deposit(5000.75);
        System.out.println("Owen has "+ owen.getBalance());
        BankAcc feit = new BankAcc(2000.96);
        feit.spend(3000.50);
        System.out.println("Feit has "+ feit.getBalance());
    }
}

owen can’t get feit’s money, and v.v.
We can enhance the previous class ...

```java
public class BankAcc {
    // static fields 'belong' to the manufacturer, not the made object, so only one copy of it is created, but each object can share it
    private static int numAccs = 0; // data field to hold CLASS values, initialised at compile time when the class is loaded
    private static int[] accNos = new int[100000]; // data field to hold CLASS array for account numbers
    private static float totalAssets = 0.0; // to hold total assets of accounts, if it weren't private then could get by BankAcc.totalAssets

    private float balance; // data field to hold values for each instantiation

    static { // this is (surprisingly?) a method run at compile time (hence no name and no input parentheses!)
        for (int i = 0; i < accNos.length; i++) {
            accNos[i] = 100001 + i; } // end for loop initialising account numbers
    } // end static compile-time method

    public static int getBankAssets() { // this method can be called in main (for example) by BankAcc.getBankAssets();
        return totalAssets; }

    public float getBalance() { // accessor method - perhaps should ask for authorisation?
        return balance; }

    public void setBalance(float balance) { // mutator method - who should authorise this?
        totalAssets += balance; this.balance = balance; }

    public float spend(float amt) { // methods to do stuff - should authenticate user?
        boolean bounce = (balance < amt); // bounce only exists within this method
        balance -= !bounce ? amt : 0.0;
        if (bounce) System.out.println("Sorry, not enough money there at the moment (: ");
        else totalAssets -= amt; }

    public float deposit(float amt) { // perhaps should offer a receipt?
        totalAssets += balance; balance += amt; }

    public BankAcc(float balance) { // another constructor
        totalAssets += balance; this.balance = balance;
        numAccs++; } // note that this gets incremented each time a new account is created!

    public BankAcc() { // default constructor
        this(0.0); } // note the use of this to refer to one of the constructor -- has to be first statement in the constructor
}
```
• So **constructors** are really just nifty methods for initialising the object just after it’s been brought into existence but before it’s been named and hence accessed.

• The word **this** is a **reference** to the current object.

• One aspect of writing programs using **classes** is that it effectively allows us to create our own types - not being restricted only to those already provided in Java.

• We can **inherit** properties of classes in a natural way ...

```java
public class Savings extends BankAcc {
    private float rate;

    public Savings(float balance, float rate) {
        super(balance);
        this.rate = rate;
    }

    // end class Savings
}
```

*super( ) refers to the constructor of the parent class, and is analogous to this, so likewise has to be the first statement in a constructor if it’s to be used at all*
• Suppose we had a `Textbook` class ...

```java
public class Textbook {
    String author, title, publisher;
    int n, isbn, crewyear;
    Preface pre = new Preface ();
    Acknow ack = new Acknow ();
    Contents cont = new Contents ();
    Chapters [] chaps = new Chapters [ n ];
    Index indy = new Index ();
    Exercises trouble = new Exercises ();
    ---------------------
} // end class TextBook
```

then every time it gets invoked via `TextBook FredBloggs = new TextBook () ;`
the particular object (reference) just created is imbued with all the
characteristics of a `TextBook` by this one line! This amounts to a
tremendous saving of effort on our part together with a significant
lessening of potential error. Assuming the relevant classes exist ...

where the `------` would have the constructors plus any useful methods (like
`writeChapter ( int k ) { ...... }` or `makeIndex ()` etc.).

• Of course, not every book is a `TextBook`, we should also have classes for
  `Novels`, `Atlases`, `Cookbooks`, `Dictionaries`, etc..
• Thankfully, Java provides the mechanism of inheritance to save us from too much repetition in dealing with this situation. The essential idea is that a child inherits everything a parent has, but can have some things of its own. This leads to the power of attorney rule: if in some situation you’re expecting a parent but only have a child, then that’s ok since a child can do everything a parent can; if however you’re expecting to see a child but only have a parent, then that is not ok since that child might have had properties the parent doesn’t have! Notice that in this model, no child can have more than one parent.

• The idea then is to put as much commonality as high up in the family tree as possible, so that a Book would have an author, title, publisher, ISBN, cryear. A Sectional would have an array of Chapters called chaps, etc.
• One point needs to be clarified: a child inherits the methods and fields of the parent, it does not inherit the values of any of the parent’s fields! If a parent has a bank account, the child inherits the ability to have a bank account, it doesn’t inherit the money in the parent’s account!!

• One other messy detail ... we can only reach up one level in the family hierarchy via super. So if we have three classes with C extending B which extends A (so that A is the grandparent), and if x is a data field of A (thus inherited by B and C), then within the class C ...

\[
\begin{align*}
\text{x} & \quad \text{variable x in class C} \\
\text{this.x} & \quad \text{ditto} \\
( (A) \text{ this }) . x & \quad \text{variable x in class A} \\
\text{super.x} & \quad \text{variable x in class B} \\
( (B) \text{ this }) . x & \quad \text{ditto} \\
\text{super.super.x} & \quad \text{illegal statement, sorry!!!}
\end{align*}
\]

• Actually, every class is in a hierarchy since even if you don’t specify a parent via extends, Java provides a generic parent class Object! Java does other things by default. If your first statement in a derived ‘child’ class constructor isn’t super, then Java calls super( ) with no arguments automatically. So if the superclass doesn’t have any constructors having no arguments, then the compiler will complain. This is also what happens is a non-explicitly-child class is formed; Java calls the default super( ) from the class Object, so providing a default constructor.
Exceptions. Bad errors cause programs (and sometimes machines!) to crash. It’s better to design our programs to catch exceptional conditions before they become fatal errors.

```java
import java.io.*;

public class PrintInt {
    public static void main ( String [] args ) {
        InputStreamReader isr = new InputStreamReader ( System.in ) ;
        BufferedReader br = new BufferedReader ( isr ) ;
        PrintWriter pw = new PrintWriter ( System.out , true ) ;
        int x ;
        String s ;
        pw.println ( "Enter an integer." ) ;
        try {
            s = br.readLine ( ) ;
            x = Integer.parseInt ( s ) ;
            pw.println ( "The integer was " + x ) ;
        } // end try block
        catch ( Exception e ) {
            pw.println ( e ) ;
        } // end catch block for Exception
    } // end main method
} // end class PrintInt
```

As indicated in the example, the try block is run. If there are no problems then the catch block is ignored. If a problem occurs then the try block terminates immediately and any exception that’s thrown by the problem line gets caught by whichever catch line matches (or includes) the type of exception generated.
If our example had been reading and writing from/to files instead of the keyboard/screen, then if any exceptions had been generated in the try block, the program would have eventually stopped whilst leaving those files open! This is a bad thing. To deal with these sorts of situations, Java provides a finally block to be used after the last catch block. This finally block will be executed whether or not any exceptions are thrown or caught, and could contain lines to close each of the files that had been opened. Essentially the control paths are ...

Some of the standard run-time exceptions are ...

- ArithmeticException
- NumberFormatException
- IndexOutOfBoundsException
- SecurityException
- NullPointerException
- NegativeArraySizeException

Some other standard checked exceptions ...

- EOFException
- FileNotFoundException
- IOException

These checked exceptions must be dealt with either by try/catch blocks within the method, or by having a try/catch arrangement higher up in one of the calling programs to catch the exception coupled with an appropriate throws statement in the method declaration(s) to throw the exception “upstairs”.

CS 2110 Java structure- exceptions
We can even create our own exceptions by extending (inheriting from) the Throwable class or one of its subclasses. For example ...

```java
import java.io.*;
public class SnazzyProgram {
    public static void main ( String [] args ) {
        try {
            FileReader fr = new FileReader ( "crawled.txt" ) ;
            BufferedReader br = new BufferedReader ( fr ) ;
            String [] emails = new String [ 10000 ] ;
            String temp = br.readLine ( ) ;
            for ( int i = 0 ; i < emails.length && temp != null ; i++ ) {
                emails [ i ] = checkEmail ( temp ) ;   temp = br.readLine ( ) ;
            } // end for loop reading file
        } // end try block
        catch ( BadEmailException bee ) {
            System.out.println ( bee.getMessage( ) ) ;}
        catch ( FileNotFoundException fnf ) {
            System.out.println ( fnf.getMessage( ) ) ;}
        catch ( IOException io ) {
            System.out.println ( io.getMessage( ) ) ;}
        finally { if ( fr != null ) fr.close ( ) ;}
    } // end main method
}

public static String checkEmail ( String s ) throws BadEmailException {
    try {
        if ( s == null ) throw new BadEmailException ( ) ;
        for ( int i = 0 ; i < s.length( ) ; i++ )
            if ( s.indexOf ( '@' ) == -1 ) throw new BadEmailException(s);
        return s ;
    } // end try block
} // end static checkEmail ( ) method

class BadEmailException extends Exception {
    public final String fake ;
    public BadEmailException ( String faked ) {
        super ( "This email address is missing an @ symbol. It was " + faked ) ;
        this.fake = faked ;
    }
    public BadEmailException ( ) {
        super ( "Something bad happened!" ) ;
        this.fake = null ;
    }
} // end fresh exception class
```
Threads. So far, we have been running a single thread of control, but it’s often convenient to be able to run either several threads independently and concurrently, or have threads branch off and yet still communicate with one another. We’ll look at a few elementary approaches to multi-threading (for a more in-depth discussion, follow the link to the Java Specification on the course homepage and read ch 17) ...

There is a `Thread` class, so we could do ...

```java
Thread sausage = new Thread();
```

to create a new thread `sausage` which can be configured and run. However `sausage.run();` won’t do anything ... the computer doesn’t know anything special about running sausages!! Better would be to extend the `Thread` class and then redefine `run()` in the derived class ...

```java
public class PingPong extends Thread { // from the java.sun.com thread tutorial
    private String word; // the word to print
    private int delay; // the delay in millisecs to pause

    public PingPong(String parole, int pendant) {
        this.word = parole; this.delay = pendant;
    } // end constructor

    public void run() { // overriding Thread's run() method
        try {
            // since sleep can throw an InterruptedException
            for (; ; ) { // never stop (unless interrupted) !!!!
                System.out.print(word + " ");
                sleep(delay);
            }
        } catch (InterruptedException ie) { return; }
    } // end overriding of the run() method
} // end class PingPong
```

Then if we have ...

```java
public static void main(String[] args) {
    new PingPong("ping", 333).start();
    new PingPong("PONG", 1000).start();
} // end main method
```

we’ll get `ping` appearing on the screen every 1/3 second and `PONG` every second (they’ll have fractionally different start times) ...

```
ping PONG ping ping ping PONG ping ping ping PONG ping
```

Hence two separate and independently running threads.
• There’s another more or less equivalent way of doing this, especially useful if you want to inherit from some class and don’t want to use up your one inheritance opportunity by having to extend the Thread class ...

```java
public class RunPingPong implements Runnable { // from the “java programming language” book
    private String word;
    private int delay;

    public RunPingPong ( String parole , int pendant ) {
        this.word = parole ;    this.delay = pendant ;
    } // end constructor

    public void run ( ) {
        try {
            for (   ;   ;   ) {
                System.out.print ( word + " ");
                Thread.sleep ( delay ) ;
            } // only exit from for loop is via an interrupt
            catch ( InterruptedException ie ) { return ; }
        } // end implementing the run ( ) method
    } // end class RunPingPong
}
```

Then if we have ...

```java
public static void main ( String [ ] args ) {
    Runnable little , bigger ;
    little = new RunPingPong ( “ping” , 333 ) ;
    bigger = new RunPingPong ( “PONG” , 1000 ) ;
    new Thread ( little ).start ( ) ;
    new Thread ( bigger ).start ( ) ;
} // end main method
```

we’ll get the same behaviour as before.

• We’ll say more about interfaces like Runnable in the next section. For now you can think of them as mold-like superclasses which only have methods - declared, but never defined. They come with an implied contract to define every method they have if you want to implement (néé extend) them. For the case of Runnable, it only declares one method, namely run( ).
• Suppose we want some rudimentary control on when particular data fields can be accessed. Consider for example a bank account in a multi-threaded environment. So what if the same account data (e.g. balance) could be accessed simultaneously by independent threads?

![Timeline diagram showing two threads accessing balance]

It could be argued that each thread ran ‘correctly’, but because thread B read the value of balance after thread A’s read but before thread A had completed its calculation, thread B incremented ‘the wrong’ value, hence leaving the balance as if that $2000 have never been deposited. Such a situation is called a race condition.

• The same situation could occur even with a simple expression like oops++ ; which technically comprises three operations: read oops, add one to oops, write oops back into memory. Concurrent writes or read/writes on a value are dangerous!!! (Concurrent reads are safe.) To safeguard this situation ...
Synchronizing per se makes no guarantee of the order of access, but does ensure that only one synchronized method at a time can have access to any data fields addressed within that method.

(Synched methods of any given instantiated object block each other, and synched static methods block each other at the class level, but there is no mutual blocking of static vs non-static methods. Note that because a child class could potentially override a parent’s synched method, it’s the case that in the child that method is actually synched only if it’s explicitly declared as synched in the child class.)

DANGER! Synchronizing is not a universal panacea, indeed it can be quite devastating if used without care .......... Imagine a bunch of quick processes waiting while a synched laborious process rambles on. Worse still, suppose you have two instances x and y of a class G having synched methods hug( ) and hugback( ) which act on G’s, and suppose hug invokes hugback( ). Then

thread A .......... x.hug ( y ) ......................... y.hugback ( x ) waiting
so thread A has a lock on y so now A wants a lock on x but is blocked by B

thread B ....................... y.hug ( x ) ............................... x.hugback ( y ) waiting
so thread B has a lock on x so now B wants a lock on y but is blocked by A

OOPS!!!

“deadlock”

public class BankAcc {
    private float balance ;

    public synchronized float spend ( float amt ) {
        balance -= amt ; return balance ;
    }

    public synchronized float deposit ( float amt ) {
        balance += amt ; return balance ;
    }

} // end class BankAcc

public class BankAcc {
    private float balance ;

    public synchronized float spend ( float amt ) {
       balance -= amt ; return balance ;
    }

    public synchronized float deposit ( float amt ) {
       balance += amt ; return balance ;
    }

} // end class BankAcc

whichever of these methods is in the first thread to access balance blocks any further access to balance by either of these methods until done. Actually, the blocking is even more aggressive*.
• We can also synch whole chunks of code as a ‘local’ statement. Consider the following method to convert an int array to absolute values ...

    public static void abs ( int [ ] values ) {
        synchronized ( values ) {
            for ( int i = 0 ; i < values.length ; i++ )
                if ( values [ i ] < 0 ) values [ i ] = - values [ i ] ;
        } // end synched block on the values array
    } // end abs method

(Although safe in a single-threaded environment without the ‘synchronized’ epithet, it must be synched if multi-threaded, otherwise some other thread might access value[i] after the abs method reads the boolean test, and then overwrite value[i] with 23, so that when abs multiplies and writes, it will leave the array with value[i] = -23. Being able to synch smaller chunks of code is a valuable option, since in general we don’t want to force other processes to have to wait longer than necessary, plus it can help evade some potential deadlocks. In particular this allows us to have a more fine-grained control on acquiring and releasing locks, for instance, we could use this to allow parallel calls to multiple methods within a class yet still protecting access to disjoint data fields.)

• If you already have code written without any thought of multi-threading, rather than rework the whole code with intricate synchs, you can create an extended class to override the appropriate methods, declare them synchronized, and then forward method calls through the super reference. If only occasional synchronised access is needed, then it’s usually easier just to use a synched statement as above. (More on threads later in the course.)
• 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)* ...

    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 ...
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.

```java
public class Q4 implements Queue {
    private Node front, back; private int length;

    public void joinQ(Person p) {
        if (isEmpty()) { back = new Node(p); front = back; }
        else { back.setNext(new Node(p)); back= back.getNext(); }
        length++;
    } // end joinQ method
    public Person leaveQ() {
        if (!isEmpty()) { Person temp = front.getData();
            front = front.getNext();
            length--; return temp; }
        else { System.out.println("Sorry, I am empty"); return null; }
    } // end leaveQ method
    public int getLength() { return this.length; }
    public boolean isFull() { return false; }
    public boolean isEmpty() { return (length == 0); }

    public Q4(int n) { length = 0; } // n is irrelevant here
    public Q4() { this(0); }
} // end class Q4
```

```java
public class Node {
    private Person data; // holds the data
    private Node next; // points to next in line

    public Node() {
        data = null; next = null;
    }
    public Node(Person p) {
        this(p, null);
    }
    public Node(Person p, Node n) {
        data = p; next = n;
    }
    public Node(Person p) {
        this(p, null);
    }
    public Person getData() { return data; }
    public void setData(Person p) { data = p; }
    public Node getNext() { return next; }
    public void setNext(Node n) { next = n; }
}
```

```java
front
null
```
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.getNext ( ) ) ;
            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 ;  }
    public void setCurrent ( Person p ) {
        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
```

![Diagram of a linked list showing joining and leaving operations](image)

---

CS 2110 Data Structures - lists 19
• 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++;
    }
    // end join method
    public Person leave() {
        if (!isEmpty()) {
            Person temp = top.getData();
            top = top.getNext(); length--;
            return temp;
        } else System.out.println("Sorry, stack is empty");
    }
    // end leave method
    public int getLength() { return this.length; }
    public boolean isFull() { return false; }
    public boolean isEmpty() { return (length == 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 \(a + ((b-c)*((d+e)-(f/g)))\) 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:

```
public class BTNode {
    private Person data ; // holds the data
    private BTNode left , right ; // points to ‘children’

    public Person getData () { return data ;}
    public void setData ( Person p ) { data = p ;}
    public BTNode getLeft () { return left ;}
    public void setLeft ( BTNode L ) { left = L ;}
    public BTNode getRight () { return right ;}
    public void setRight ( BTNode R ) { right = R ;}

    public BTNode ( Person p , BTNode L , BTNode R ) {
        data = p ;
        left = L ;
        right = R ;
    }
    public BTNode ( Person p ) { this ( p , null , null ) ;}
    public BTNode ( ) ( this ( null ) ;}
}
```

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 ItemNotFound ; // to delete the (first) appearance of p
        Itr find ( Person p ) throws ItemNotFound ; // 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 voidplusplus() {
        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((L2) list.getHeader());
    }

    public ItrP() {
        this(null, null);
    }
}

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

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

    public Litr find ( Person p ) throws ItemNotFound {
        Litr 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 ( ) ) {
            Litr 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 isEmpty () { return ( length == 0 ); }
    public Litr first () { return new LitrP ( this , header.getNext ( ) ); }

    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).

```java
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.
Suppose you are given a ‘rule’ such as

\[ a_n = n \ a_{n-1} \]

and also know that

\[ a_1 = 1 \]

Then we can use this to build a sequence of numbers

\[ 1, 2, 6, 24, 120, 720, \ldots \]

which you recognise as factorials. It’s easy to see this by building up from the bottom

\[ 1, 2 \times 1, 3 \times (2 \times 1), 4 \times (3 \times 2 \times 1), \ldots \]

although using this approach to show that this will really only produce factorials would take an infinite amount of time!
We could prove this in an intuitively rigorous inductive way by

1. Remark that \( a_1 = 1 = (1)! \)

2. Notice that if we were to assume that \( a_n = n! \) then
   \[
   a_{n+1} = (n+1) \times a_n \quad \text{by our ‘rule’}
   = (n+1) \times (n!) \quad \text{by our assumption}
   = (n+1)!
   \]

This is rather like saying

1. I can put my foot on the first rung of a ladder.

2. IF I’m on any rung of the ladder THEN I can step onto the next rung.

This way of arguing, called induction, is very nice because

a. We don’t have to do infinitely many steps.

b. It’s “jolly obvious” that we’ve covered every case!
• Instead of working from the bottom up, we could work from the top down (provided our ‘top’ is only ‘finitely high’) …

If you follow the arrows, you can see that this process first finds the BOTTOM, and then assembles the calculation as it returns to the top. Obviously, if there is no bottom then we will be waiting a jolly long time for any results!! Let’s see this process in Java code.

```java
public int fact ( int n ) {
    if ( n == 1 ) return 1 ;
    else            return  n * fact ( n-1 ) ;
}
```

Then our `fact ( n )` behaves just like our $a_n$, and it would be invoked by

```java
ans = fact ( 6 ) ;
```

for example - producing the same bottom-hungry routine we saw for $a_n$. 

• In fact, any sequence defined by a recurrence relation can be converted very easily into recursive code. Without making any comments about efficiency (!), recursive code is typically very short.

• As experiments, first you should run this code (previous page) to compute fact(10) and then fact(100) - but you might like to change from int to BigInteger to avoid the sad consequences of int wrap around! After that, try to find the 10th and the 100th term in the following Fibonacci sequence, and then look at the schematic of the recursive calls on the previous page to understand what’s (not) going on (and then fix it)!

$$a_n = a_{n-1} + a_{n-2}$$

$$a_1 = 1, \text{ and } a_2 = 1$$

• It’s worth noting that very similar code can be used to compute $C_{n,r}$, the binomial coefficients (for the intuition behind this look at Pascal’s triangle)

$$C_{n,r} = C_{n-1,r} + C_{n-1,r-1} \text{ with } C_{n,n} = 1, \text{ and } C_{n,0} = 1$$

and powers of a (an example of a ‘divide and conquer’ approach)

$$a^n = (a^{n/2})^2 \text{ for } n \text{ even, } a^n = a(a^{n/2})^2 \text{ for } n \text{ odd, and } a^0 = 1$$

Note the vastly faster computation for powers when coded this way!
• It’s interesting to see how recursive methods could be implemented within the machine. The key idea is to use a stack to remember parameters and local variables across recursive calls, and for each method invocation to get its own stack frame.

A stack frame contains storage for

- local variables of the method
- parameters of the method
- return info (address and return value)
- any other bookkeeping info

is pushed with each recursive call, and popped when the method returns (leaving any value on top of the stack).

example showing the change of state of the stack of stack frames for computing the value of $2^5$ recursively (as on the previous page)
The processor has to keep track of all this . . . one approach is to have a frame base register (FBR), so that when a method is invoked, the FBR points to its stack frame. Then when the method invocation returns, the FBR is restored to its value before the invocation (courtesy of part of the return info in the stack frame).

Computational activity only takes place in the topmost (most recently pushed) stack frame.

A rather nice application of recursion can be found when parsing ‘sentences’ constructed according to formal grammars. These actually underlie the formal structure of programming languages (as well as many other things!).
A Grammar

Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → bunnies
Verb → like
Verb → see

• Our sample grammar has these rules:
  ▪ A Sentence can be a Noun followed by a Verb followed by a Noun
  ▪ A Noun can be ‘boys’ or ‘girls’ or ‘bunnies’
  ▪ A Verb can be ‘like’ or ‘see’

• Grammar: set of rules for generating sentences in a language
• Examples of Sentence:
  ▪ boys see bunnies
  ▪ bunnies like girls
  ▪ ...
• White space between words does not matter
• The words boys, girls, bunnies, like, see are called tokens or terminals
• The words Sentence, Noun, Verb are called nonterminals
• This is a very boring grammar because the set of Sentences is finite (exactly 18 sentences)
A Recursive Grammar

Sentence → Sentence and Sentence
Sentence → Sentence or Sentence
Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → bunnies
Verb → like
Verb → see

- This grammar is more interesting than the last one because the set of Sentences is infinite

- Examples of Sentences in this language:
  - boys like girls
  - boys like girls and girls like bunnies
  - boys like girls and girls like bunnies and girls like bunnies
  - boys like girls and girls like bunnies and girls like bunnies and girls like bunnies
  - ........

- What makes this set infinite?
  Answer:
    - Recursive definition of Sentence
Sentences with Periods

PunctuatedSentence → Sentence .
Sentence → Sentence and Sentence
Sentence → Sentence or Sentence
Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → bunnies
Verb → like
Verb → see

- Add a new rule that adds a period only at the end of the sentence.
- The tokens here are the 7 words plus the period (.)
- This grammar is ambiguous:
  boys like girls
  and girls like boys
  or girls like bunnies
Grammar for Simple Expressions

E → integer
E → ( E + E )

• Simple expressions:
  ▪ An E can be an integer.
  ▪ An E can be ‘(’ followed by an E followed by ‘+’ followed by an E followed by ‘)’

• Set of expressions defined by this grammar is a recursively-defined set
  ▪ Is language finite or infinite?
  ▪ Do recursive grammars always yield infinite languages?

• Here are some legal expressions:
  ▪ 2
  ▪ (3 + 34)
  ▪ ((4+23) + 89)
  ▪ ((89 + 23) + (23 + (34+12)))

• Here are some illegal expressions:
  ▪ (3
  ▪ 3 + 4

• The tokens in this grammar are (, +, ), and any integer
Parsing

- Grammars can be used in two ways
  - A grammar defines a language (i.e., the set of properly structured sentences)
  - A grammar can be used to parse a sentence (thus, checking if the sentence is in the language)

- To parse a sentence is to build a parse tree
  - This is much like diagramming a sentence

- Example: Show that \(((4+23) + 89)\) is a valid expression E by building a parse tree

```
       E
      / \  / \
     E   E + 89
    / \ /   / \    
   E  + E  4 23
```

Example: Show that ((4+23) + 89) is a valid expression E by building a parse tree
Recursive Descent Parsing

- Idea: Use the grammar to design a recursive program to check if a sentence is in the language
- To parse an expression $E$, for instance
  - We look for each terminal (i.e., each token)
  - Each nonterminal (e.g., $E$) can handle itself by using a recursive call
- The grammar tells how to write the program!

```java
boolean parseE() {
    if (first token is an integer) return true;
    if (first token is '(') {
        parseE();
        Make sure there is a '+' token;
        parseE();
        Make sure there is ')' token;
        return true;
    }
    return false;
}
```

```java
public static Node parseE(Scanner scanner) {
    if (scanner.hasNextInt()) {
        int data = scanner.nextInt();
        return new Node(data);
    }
    check(scanner, '(');
    left = parseE(scanner);
    check(scanner, '+');
    right = parseE(scanner);
    check(scanner, ')');
    return new Node(left, right);
}
```
Using a Parser to Generate Code

• We can modify the parser so that it generates stack code to evaluate arithmetic expressions:

\[
\begin{align*}
2 & \quad \text{PUSH 2} \\
& \quad \text{STOP} \\
(2 + 3) & \quad \text{PUSH 2} \\
& \quad \text{PUSH 3} \\
& \quad \text{ADD} \\
& \quad \text{STOP}
\end{align*}
\]

• Goal: Method parseE should return a string containing stack code for expression it has parsed

• Method parseE can generate code in a recursive way:
  - For integer \( i \), it returns string “PUSH ” + \( i \) + “\n”
  - For \((E_1 + E_2)\),
    - Recursive calls for \( E_1 \) and \( E_2 \) return code strings \( c_1 \) and \( c_2 \), respectively
    - For \((E_1 + E_2)\), return \( c_1 + c_2 + “ADD\n”\)
  - Top-level method should tack on a STOP command after code received from parseE
Does Recursive Descent Always Work?

• There are some grammars that cannot be used as the basis for recursive descent
  ▪ A trivial example (causes infinite recursion):
    ✷ $S \rightarrow b$
    ✷ $S \rightarrow Sa$

• Can rewrite grammar
  ✷ $S \rightarrow b$
  ✷ $S \rightarrow bA$
  ✷ $A \rightarrow a$
  ✷ $A \rightarrow aA$

• For some constructs, recursive descent is hard to use
  ▪ Can use a more powerful parsing technique (there are several, but not in this course)
Syntactic Ambiguity

- Sometimes a sentence has more than one parse tree
  
  \[
  S \rightarrow A \mid aaxB \\
  A \rightarrow x \mid aAb \\
  B \rightarrow b \mid bB
  \]
  
  - The string aaxbb can be parsed in two ways

- This kind of ambiguity sometimes shows up in programming languages

  if E1 then if E2 then S1 else S2

  \textit{Which then \textit{does the else go with}?}

- This ambiguity actually affects the program’s meaning

- How do we resolve this?
  - Provide an extra non-grammar rule (e.g., the \textit{else} goes with the closest \textit{if})
  - Modify the language (e.g., an if-statement must end with a \textit{fi})
  - Operator precedence (e.g., \(1 + 2 \times 3\) should always be parsed as \(1 + (2 \times 3)\), not \((1 + 2) \times 3\)
  - Other methods (e.g., Python uses amount of indentation)
• On page 26 we deferred coding tree navigation until after a reminder on recursion, however, instead of covering this in depth in lectures, we’ll leave coding these for the current homework set -- probably more helpful in enabling you to ‘get into’ the mindset of building ‘nano-code-bots’ to be released at the root of a tree and which spawn recursive bots as needed (active discussion in lecture describing the construction of in-order traversals).

• Also on page 26 we referred to generics (introduced in Java 1.5) as a way to avoid having to build data structures for each particular type of object, so it’s time to see what they are and how to use them.