• 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;
        }
    }
    // end class BankAcc

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.

So then, how does this get used?

    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());
        }
    }
    // end main method
    // end class GRQ
• 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++ ;  }

    public BankAcc ( ) {
        // default constructor
        this ( 0.0 ) ;  // note that this gets incremented each time a new account is created!
    }
}
```

CS 2110 Java structure - classes 2

Thursday, 1 July 2010
• 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
  ```
Suppose we had a `Textbook` class ...

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

ten 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 ...

```java
public class TextBook {
    //end class TextBook
}
```

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 ...

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

• 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 the 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 BadEmail
        catch ( FileNotFoundException fnf ) {
            System.out.println ( fnf.getMessage( ) ) ;
        } // catch FileNotFound
        catch ( IOException io ) {
            System.out.println ( io.getMessage( ) ) ;
        } // catch IO hiccups
        finally { if ( fr != null )   fr.close ( ) ;}
    } // end main method
} // end class SnazzyProgram

public static String checkEmail ( String s ) throws BadEmailException {
    if ( s == null )   throw new BadEmailException ( ) ;
    if ( s.indexOf ( '@' ) == -1 ) throw new BadEmailException(s) ;
    return s ;
} // end static checkEmail ( ) method

public 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 ;
    } // end constructor
    public BadEmailException ( ) {
        super ( "Something bad happened!" ) ;
        this.fake = null ;// or simply super( ) above
    } // end constructor
} // end fresh exception class
```

The diagram shows the inheritance hierarchy of exceptions in Java, starting from `Object` and going through `Exception` and `Throwable` classes.