Exceptions

For now we'll treat this topic somewhat cursorily. Bad errors cause programs (& sometimes machines!) to crash. It's better to design our programs to catch exceptional conditions before they become fatal errors.

```
import java.io.*;

public class PrintInt {
  public static void main (String [] args) {
    InputStreamReader ca = new InputStreamReader(System.in);
    BufferedReader va = new BufferedReader(ca);
    PrintWriter bou = new PrintWriter(System.out, true);

    int x;
    String s;

    bou.println("Enter an integer.");

    try
    {
      s = va.readLine();
      x = Integer.parseInt(s);
      bou.println("The integer was " + x);
    }
    catch (Exception e)
    { bou.println(e); }
  }
}
```

runs this section

catch is immediate

goes down here only if exceptions occur

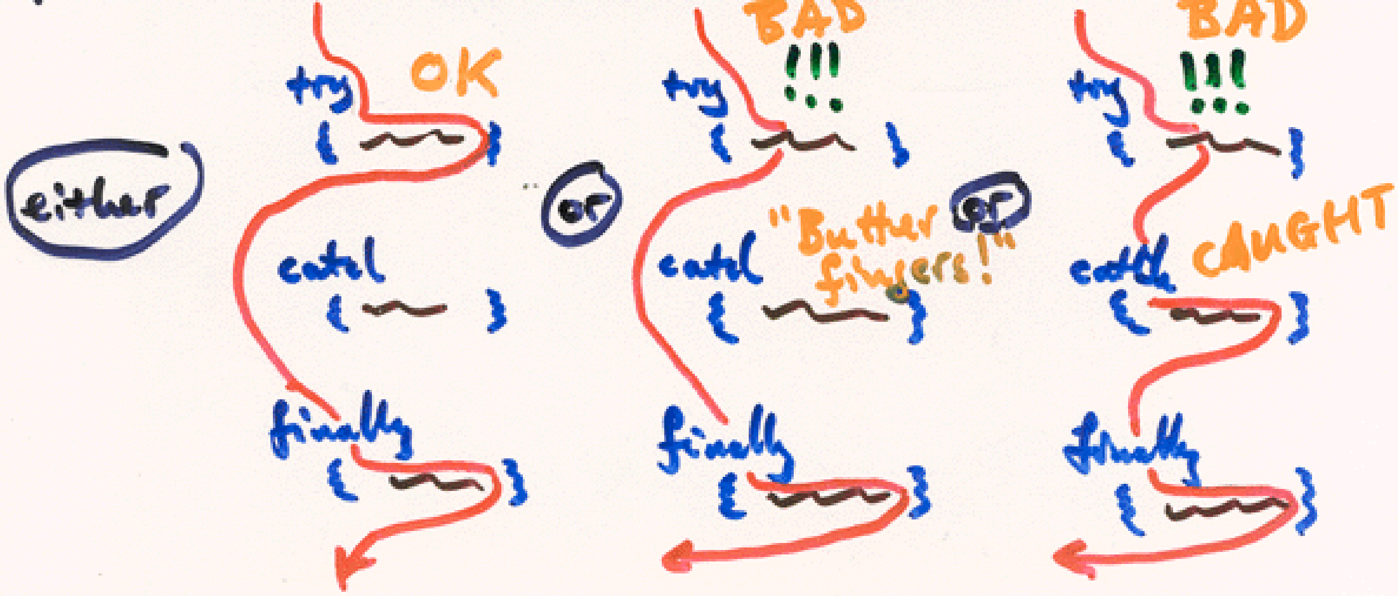
could generate an IOException

could generate a NumberFormatException

We could be more specific, but this will nab the interesting ones – nice & general.

As indicated in the example, the try block is run. If there are no problems then catch 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 (if any) 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 finally to be used like catch, but after the last catch block. This finally block is then executed, 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	overflow, or int ÷ 0
NumberFormatException	bad String → Number
IndexOutOfBoundsException	looking outside array or String
NegativeArraySizeException	Obvious!
NullPointerException	bad use of null reference
SecurityException	Obvious!

Some other standard 'checked' exceptions ...

java.io.EOFException	hit end of file too soon!
java.io.FileNotFoundException	Obvious!
java.io.IOException	most I/O problems

These checked exceptions must be dealt with. Any method that could give rise to any of these must either use try/catch to catch that exception within the method, or have a try/catch arrangement higher up in one of the calling programs (at worst ultimately in main) to catch the exception as well as an appropriate throws statement in the method declaration (or in each method declaration if there's a string of nested methods) to throw the exception "upstairs", e.g. `public static int Goal (int x) throws IOException`

Java has lots of nifty library routines. One of these useful repositories is obtained by ...

```
import java.util.*;
```

For example, if you have a String of stuff (perhaps read in, or returned by some method), it could be that what you are really interested in is separated by spaces. Then...

```
StringTokenizer t;
String cuckoo = "Once upon a time 7 168";

t = new StringTokenizer(cuckoo);
```

builds t as (the address of) the original String cuckoo broken up into chunks/tokens ...

each of which is still a String, but which can be accessed as if it were a stream like BufferedReader. Then

```
t.countTokens(); ⟶ 6
```

and

```
t.nextToken(); ⟶ Once
t.nextToken(); ⟶ upon
```

↓ successively

etc..

Classes

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;

    public float getBalance()
         { return balance; }
    public void setBalance( float bal)
         { balance = bal; }
    public float spend ( float amt )
         { balance -= amt;
           System.out.println ("You spent "+ amt);
         }
    public float deposit ( float amt )
         { balance += amt;
           System.out.println ("You deposited "+ amt);
         }

    public BankAcc()
         { balance = 0.0 ; }
    public BankAcc ( float amt )
         { balance = amt ; }
  }
```

can't access except via methods in the 'manufacturer', i.e. actually in class

field/data

accessor method

mutator method

method/function

can access via any object of 'type' BankAcc but only for that incarnation of the object!!

no return type

same name as the class name

constructors

So then, how does this get used?

```
public class GRQ
  {
    public static void main (String [] args)
      {
        BankAcc owen = new BankAcc();
        float x;
        owen.deposit(5000.75);
      x=owen.getBalance();
        System.out.println (x);
        BankAcc feit = new BankAcc(5000.75);
        feit.spend(3000.50);
      x=feit.getBalance(); System.out.println(x);
      }
  }
```

owen can't get feit's money, and v.v..

The only new thing here is the idea of constructors, which are really just nifty ways of initializing the fields of an object, or doing other useful things just after the object has been brought into existence but before it's been named & hence accessed.

Sometimes there can be confusion with names of 'variables'; for this problem the word this is a reference to the current object...

```
public BankAcc(float balance)
    { this.balance = balance; }
```

local name limited to the scope of the method defn.

the 'balance' of the current object

We could also have written...

```
public BankAcc()
    { this(0.0); }
```

On this occasion, this calls whichever constructor matches its use. For this to work, the call to this has to be the first statement in the constructor.

Before we extend this, a couple of side observations are worth making...

If we were to add the following line to our BankAcc class

```
private static int numAccs = 0;
```

and then change our constructor(s) to

```
public BankAcc (float balance)
  {
    this.balance = balance;
    numAccs ++;
  }
```

then we'd have in numAccs the total number of times a BankAcc has been created. This is because static fields 'belong' to the manufacturer, not the manufactured object, so only one copy of it is ever created, but each object can share it. To change its value however, we would have to write...

```
BankAcc.numAccs = 10216;
```

As you can guess from numAccs being set to 0, any initialization performed on a static field in a class occurs only once — when the class is loaded. We can do the same thing with static methods, so...

```
private static int accNos [] = new int [10000];

static
  {
    for (int i=0 ; i< accNos.length ; i++)
      accNos[i] = 100001 + i ;
  }
```

included in our BankAcc class definition would give us an array of account numbers ready for use by each BankAcc object. Notice that there's no point giving this method a return type or name since it can never be accessed & run after loading!

One aspect of writing programs using classes is that it allows us to create our own types — not being restricted only to those already provided in Java. For example, if we had a Textbook class...

```
public class Textbook
  {
    . . . . . .
    . . . . . . .

  }
```

then every time it gets invoked...

```
    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.

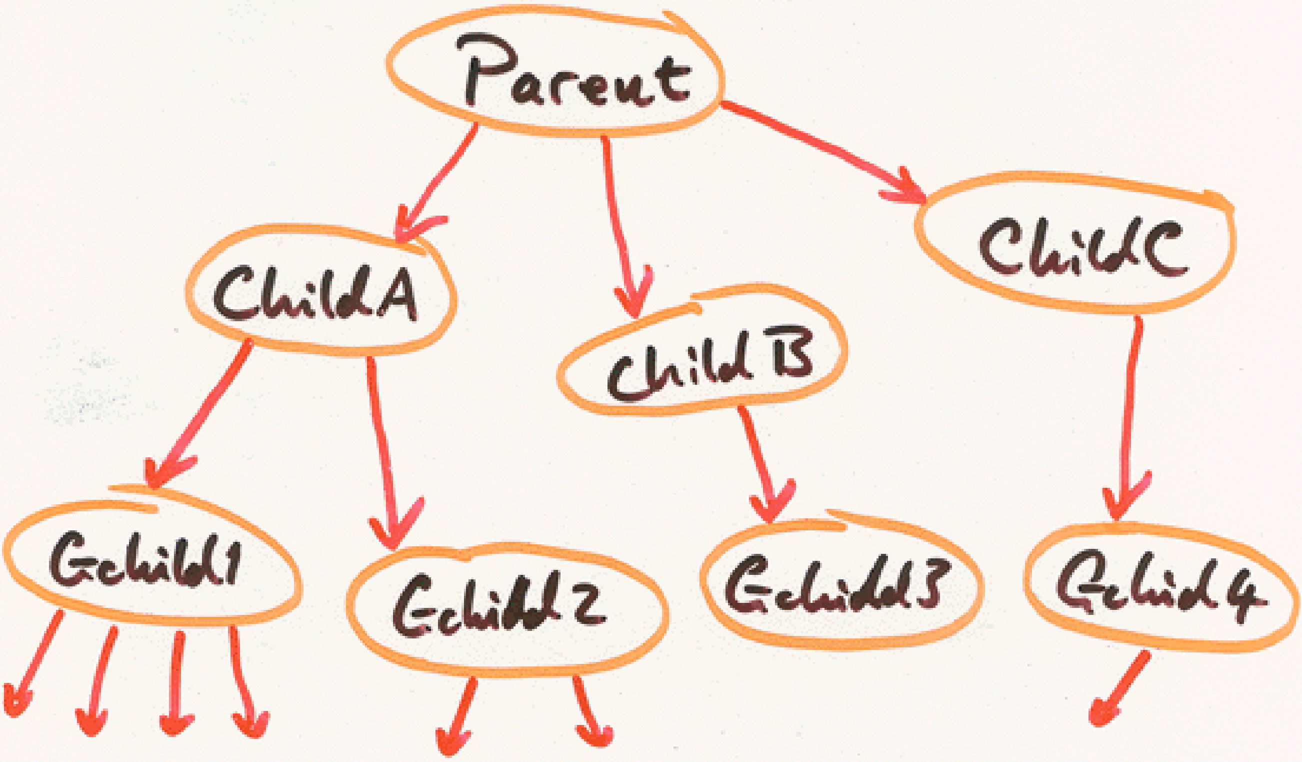
Assuming the relevant classes exist ...

```
public class Textbook
  {
     String author, title, publisher;
     int n, isbn, cryear;
     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();
     .......
  }
```

where the .... would have the constructors, plus any useful methods (like writeChapter(int k) or makeIndex() etc.).

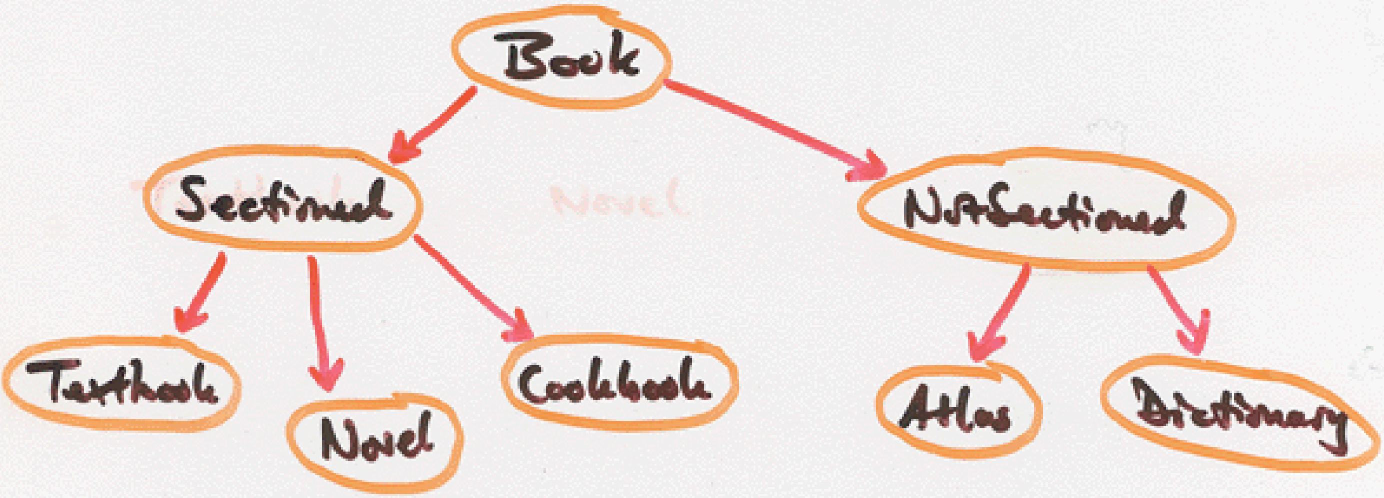
Of course, not every book is a Textbook, so we should also have classes for Novels, Cookbooks, Atlases, Dictionarys (!), etc..

Thankfully, Java provides the mechanism of inheritance to save us from too much repetition in dealing with this situation.



The essential idea of inheritance 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 need a parent but only have a child, then that's OK since a child can do everything a parent can; if however you need a child but only have a parent, then that's 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 the family tree as possible ...



so that a Book would have an author, title, publisher, isbn, cryear. A Sectioned would have an array of chapters called chaps. A NonSectioned might have a single chapter-like entity instead. A Textbook would add pre, ack, cont, indx, and trouble; yet a Novel would skip trouble and indx, but probably add disclaimer; and so on. The syntax for this is ...

```
public class Sectioned extends Book
    {
        . . . . . .   ← new stuff not in Book
    }
```

and

```
public class Textbook extends Sectioned
    {
        . . . . . .   ← new stuff not in
    }                   Sectioned
```

One point needs to be clarified: a child inheri 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 !!

We'll go through an extensive example shortly.

There is an analogous reference super to our earlie reference this which accesses one level higher in the hierarchy ...

```
public class BankAcc
 {
    private float balance;
    public String name;
    public BankAcc(float balance)
        { this.balance = balance; }
    public float getBalance() { return balance; }
 }


public class Savings extends BankAcc
 {
    private float rate;
    public Savings (float balance, float rate)
        { super(balance); this.rate = rate;}
 }
```

also inherits fields for balance and name

also inherits the getBalance method

super + ( ) refers to the constructor of the parent

Here the word super is analogous to this, except that it refers to the constructor of the parent class. If you're going to use it, then it has to be the first statement in the method, even before any variable declarations!

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 isn't super, then Java calls super() with no arguments automatically — so if the superclass has no constructors having no arguments, then the compiler will complain. This is also what happens if a non-explicitly-child class is formed; Java calls the default super() from the class Object, so providing a default constructor.

We can play super/this games so that

this.a

refers to the variable a in the current class, and

super.a

refers to the variable a in the parent class — very useful if a means different things in the two classes!

To emphasize... if C is a child of B which is a child of A

'type casting'

x	variable x in class C
this.x	" " " " "
super.x	" x " " B
((B)this).x	" " " " "
((A)this).x	" x " " A
super.super.x	illegal statement, sorry!!