Delegation

There are times when it's sensible to delegate certain tasks within a program. Javaspeak for this is **methods** (alias functions or subroutines). For example...

```
public class Stats
{
   public static void main (String[] args)
   {
      int a = 7, b = 12 , c = 24;
      double dl;

      dl = avg (a, b);

      System.out.println (" The average of "+a
         + " and " +b+ " is " +dl+ ".");

      System.out.println (" Including " +c
         + " changes this to " + avg(a,b,c)+".");
   }

   public static double avg (int x, int y)
   {
      return (x+y)/2.0;
   }
   public static double avg (int x, int y, int z)
   { return (x + y + z)/3.0; }
}
```

Values copied

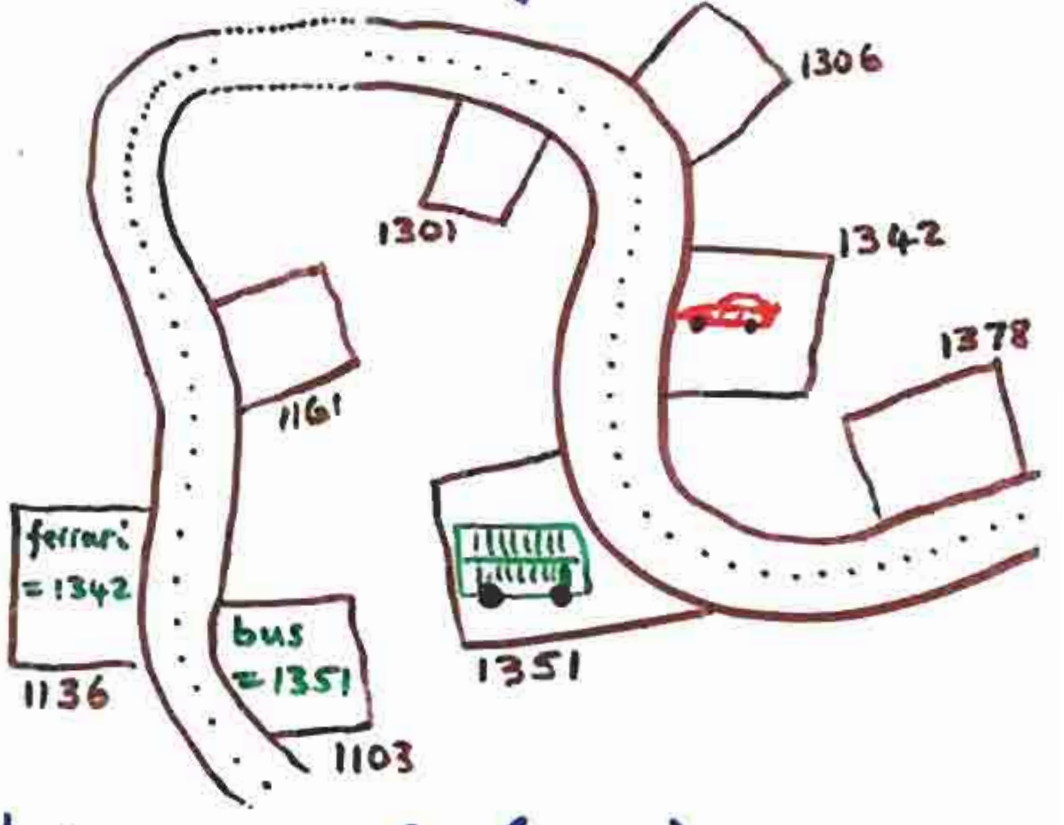
unambiguous

overloading

For the primitive types, actual values are copied and passed. For any other 'types', since Java doesn't really know how big they might be, instead of passing/copying whatever constitutes value, Java passes/copies the address/reference + the object, leaving the object wherever it happens to sit in memory. Indeed...

```
Car ferrari = new Car(red);
```

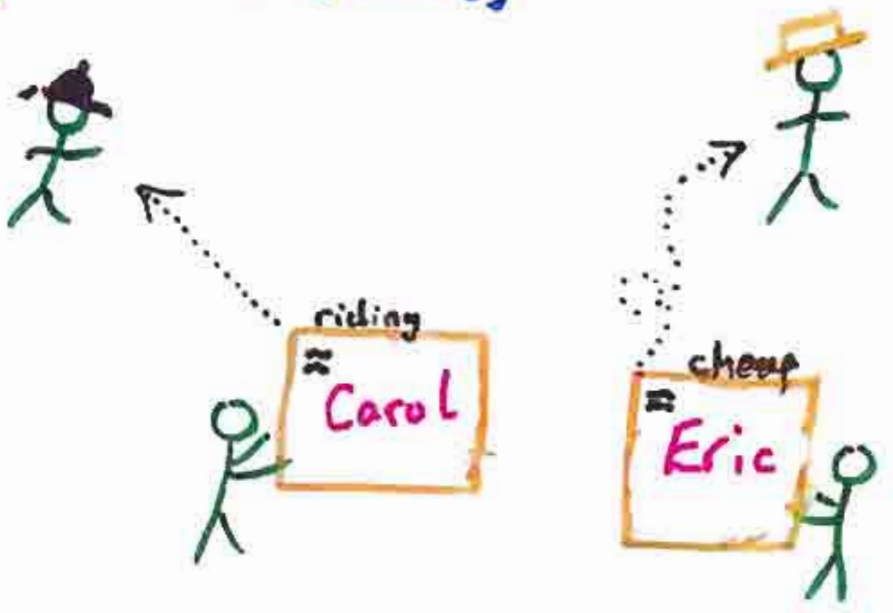
does not make ferrari the actual car, it makes instead ferrari the address of the actual car.




```
Car bus = new Car(green);
```

This leads to various consequences...

```
Hat cheap = new Hat(white);
Hat riding = new Hat(black);
```



Then

```
Hat tasteless = new Hat(white);
```

gives us 2 white Hats, but

```
cheap != tasteless
```

because the addresses are different.

However,

```
cheap = riding;
```

makes the changes in red which mean

```
cheap == riding
```

both being the same address, the address of the riding hat ——— the address of the cheap hat has been lost in the process!

This raises the obvious question of what address a reference holds if it's not currently referencing any particular object. The answer is:

null

If this situation is not picked up by the compiler, but only noticed at run-time, then we get the common error message:

NullPointerException

if a reference was declared but not assigned the address of an actual existant object.

It's also worth noting that once an object is no longer being referenced, as in cheap after the assignment cheap = riding; Java allows the memory allocated to the object cheap to be written over (whenever this may happen) — this is called automatic garbage collection.

A reference cannot reference a primitive variable.

As a final comment in this vein, suppose

yummy(rhubarb);

is a call to a method yummy with declaration

public int yummy(eeeee custard) {----}

then if eeeee is a primitive type the variable custard copies the value of rhubarb, but if eeeee is a reference then custard points to the same object rhubarb does.

We should observe that in

```
Car ferrari = new Car(red);
```

the word new creates an anonymous object according to the manufacturer class Car which has been told explicitly to make it red; the phrase Car ferrari makes ferrari an allowable reference to a Car-like object; finally, the = assigns the address of the anonymous car to the name ferrari !!!

Strings

We've seen already that there is a non-primitive class String for which the operator + is defined by 'concatenation'. Some other things about the String class are worth listing...

```
String vide = "";
```

no space !!

creates vide as an empty string. Beware that 'a' is a single character whereas "a" is a String with length one, and char is primitive but String is reference, so they cannot be the same.

Beware also that == and != apply to Strings as they do to all other references — they refer to addresses not values. To test value, use either

binary `trois.equals(two)` → true/false

or lexicographic `trois.compareTo(two)` → neg. no./0/pos. no.

Arrays

Arrays in Java are references, so

```
int [] boxes;   or   int boxes [];
```

declares boxes to be the reference for an integer array which doesn't yet exist (so the value of boxes is null). To create we of course use new

```
boxes = new int [2010];
```

This array might be initialized by ...

```
for (int i=0; i < boxes.length ; i++)
    boxes [i] = i * i % 12;
```

To avoid the dreaded NullPointerException, if our array was instead

```
Rhubarb [] holes = new Rhubarb [2010];
```

then the initialization would be ...

```
for (int i=0; i < holes.length ; i++)
    holes [i] = new Rhubarb (i);
```

assuming that the class Rhubarb knows what to do with an integer i!

Again, since arrays are references, be careful with =, ==, and !=. Of course, arrays are passed by reference into methods just like any other reference —— another standard source of errors.

Other String tricks are...

```
myriad.length();
```

which if the String myriad is "Did I blink" returns the integer value 11, and...

```
myriad.charAt(7);
```

which returns the character value 'l' since the count starts with D being position 0 — NOT 1!!
Also...

```
myriad.substring(3,8);
```

returns the string "␣I␣bl". Any primitive type can be converted to a String by...

```
toString(97.46);
```

which returns the string (really the reference to the string!) "97.46", and if PI were to be defined in Math with the natural meaning then

```
toString(Math.PI);
```

would return the address of a horribly long String!

Finally, the following have equivalent effect:

```
int n = Integer.parseInt("96");
int n = Integer.valueOf("96").intValue();
int n = new Integer("96").intValue();
```

However, for primitives other than integer (ie other than byte, short, int, long) only the analogues of the last two forms work.

Two (and higher) dimensional arrays are handled similarly ...

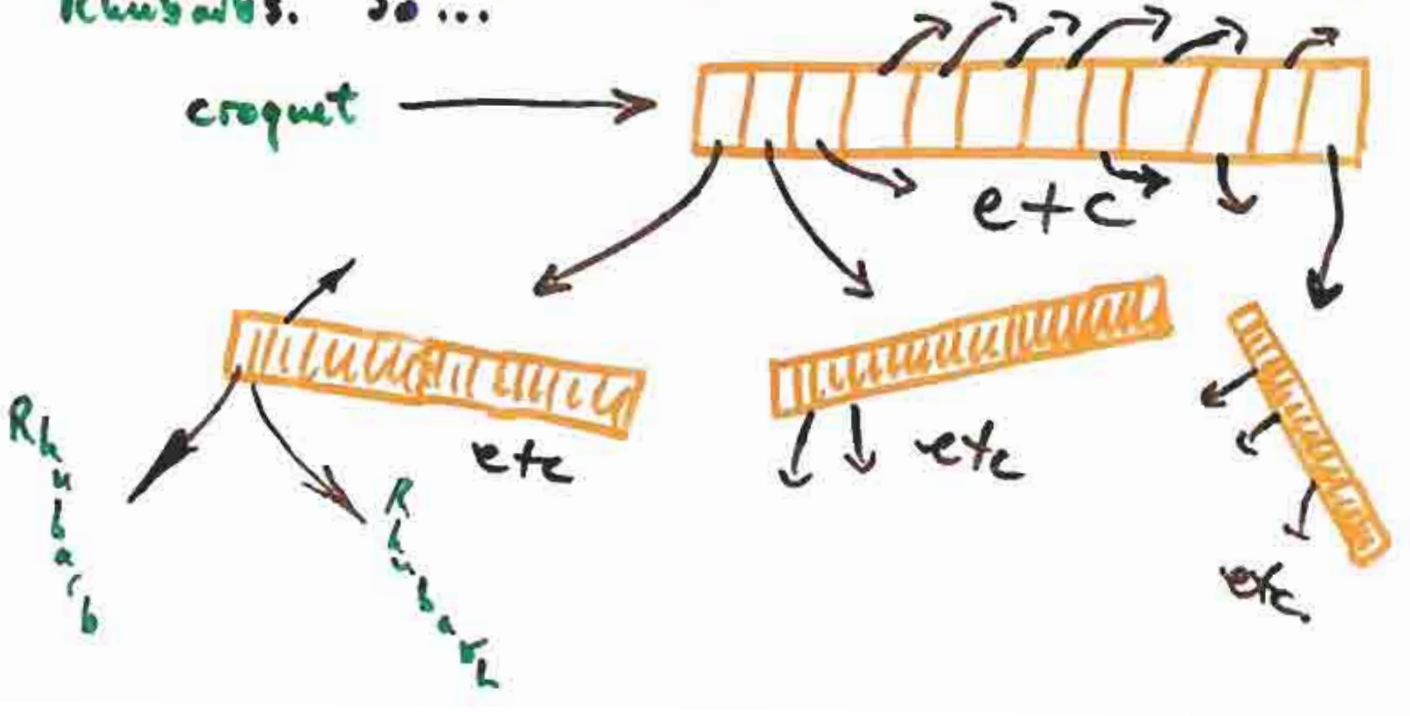
```
Rhubarb [][] croquet;
```

declares croquet to be the reference of a two dimensional array of Rhubarbs, and then

```
croquet = new Rhubarb[12][24];
```

creates an array prepared to hold two gross of Rhubarb, and then it can be filled with any actually created (using new) actual Rhubarbs.

It's worth noting that the above two dimensional array croquet is really treated as if it were a one dimensional array holding 12 objects, where each of these objects is a one dimensional array holding 24 actual addresses of Rhubarbs. So ...



This means we can write...

```
croquet = new Rhubarb[10][];
```

which makes croquet the reference of a two dimensional array of Rhubarbs, which can be thought of as having 10 rows of as yet undetermined length. Of course, the length of each row will have to be fixed (using new) before the rows can be filled...

```
for (i=0; i< croquet.length; i++)
 { croquet[i] = new Rhubarb[i*i+1];
   for (j=0; j< croquet[i].length; j++)
     croquet[i][j] = new Rhubarb(i+j);
 }
```

assumes that makes sense for Rhubarbs

Using this and similar tricks, we can size our arrays dynamically, though always being careful to remember that we're working with references. The concrete size of the array must be fixed before it's used, so really dynamic resizing is an exercise in continual re-copying.

You must have wondered what the array args is used for in ...

```
public static void main (String [] args)
```

The answer is that it's designed to allow the use of command line arguments. So the program.

```
public class Average
{
  public static void main (String [] args)
  {
    double avg = 0.0;

    if (args.length != 0)
    {for (int i=0; i < args.length; i++)
      avg += Integer.parseInt (args[i]);
     avg = avg / args.length;
     System.out.println ("The average "
          + "is " + avg);
    }
    else
     System.out.println ("You didn't "
          + "give me any numbers.");

  }
}
```

This could be invoked by typing ...

Average | 3 | 17 | 4 | 9684 | 18 |

forms 'args' a array of strings

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

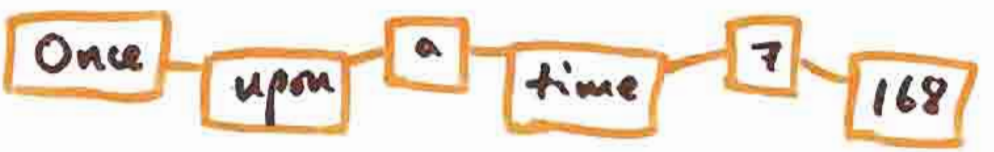
```
import java.util.*;
```

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..

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 isr = new InputStreamReader (System.in);
    BufferedReader kbd = new BufferedReader (isr);
    PrintWriter out = new PrintWriter (System.out, true);

    int x;
    String s;

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

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

runs this section

catch is invoked

goes down here only if exception occurs

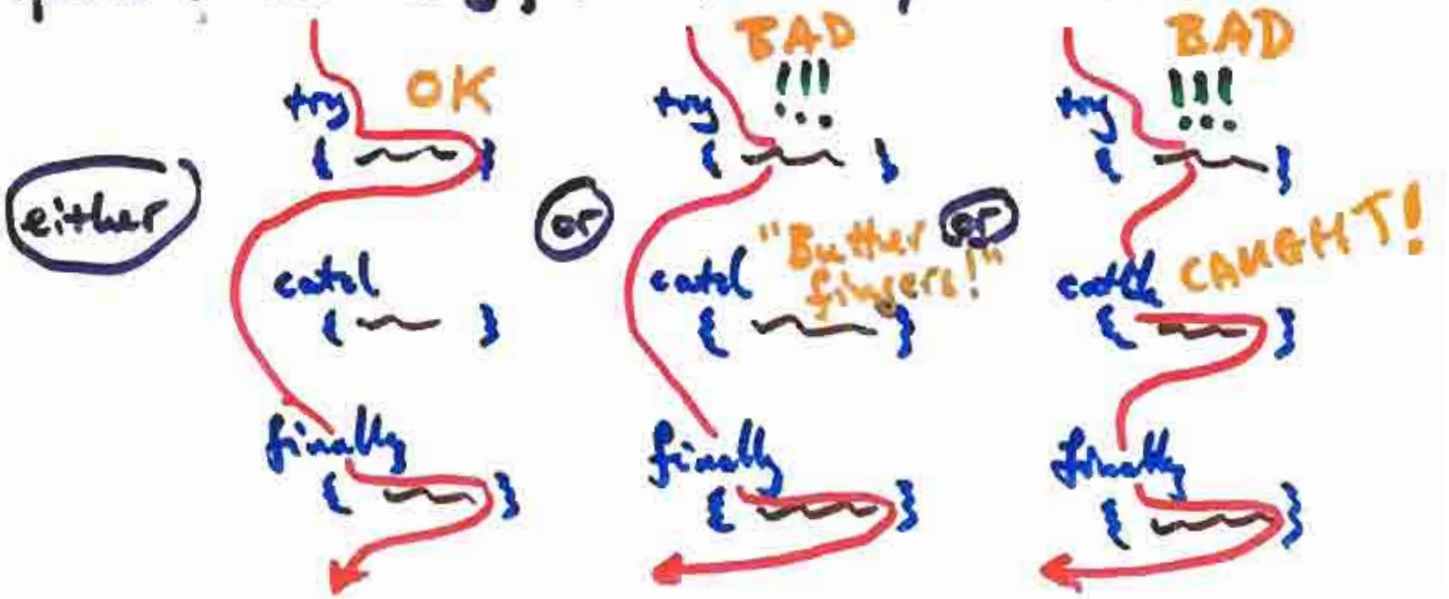
could generate an IOException

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", eg. public static int Gcd(int x) throws IOException
{ ... }