遞迴 (Recursive)

- 假設，要設計一個**階層運算** (Factorial) 的程式
  - 計算式為 $x! = x * x\text{-}1 * x\text{-}2 * \ldots * 1$
  - 調整一下式子得 $x! = x * (x\text{-}1)!$
  - ★ **注意**★ 當 $x = 0$ 時 $x! = 1$
  - 直覺的寫法

遞迴 (Recursive) (Cont.)

```java
/* program name: RecursiveTest.java
 * Author: Yung-Chen Chou
 * Date: Apr. 21, 2009
 */
import java.io.*;
public class RecursiveTest{
    public static void main(String[] argv) throws IOException{
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("請輸入欲計算的階層數:");
        int levelNum = Integer.parseInt(br.readLine());
        int facVal = levelNum;
        for(int i=levelNum-1;i>0;i--){
            facVal = facVal * i;
        }
        System.out.println("結果爲 -> "+facVal);
    }
}
```

```
<已終止> RecursiveTest [Java 應用程
請輸入欲計算的階層數:6
結果爲 -> 720
```

遞迴 (Recursive) (Cont.)

- 另一種高桿的寫法—**遞迴**
- 遞迴的意思是在**方法**中自己呼叫自己

```
5  import java.io.*;
6  class Calculator{
7      int factorial(int levelNum){
8          if(levelNum<=1){
9              return 1;
10         }else{
11             System.out.println("in rec fac levelNum = "+levelNum);
12             return levelNum * factorial(levelNum - 1);
13         }
14     }
15 }
16 public class RecursiveTest{
17     public static void main(String[] argv) throws IOException{
18         BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
19         Calculator cal = new Calculator();
20         System.out.print("請輸入欲計算的階層數:");
21         int levelNum = Integer.parseInt(br.readLine());
22         /*
23         int facVal = levelNum;
24         for(int i=levelNum-1;i>0;i--){
25             facVal = facVal * i;
26         }
27         */
28         System.out.println("結果爲 -> "+cal.factorial(levelNum));
29     }
30 }
```

```
請輸入欲計算的階層數:6
in rec fac levelNum = 6
in rec fac levelNum = 5
in rec fac levelNum = 4
in rec fac levelNum = 3
in rec fac levelNum = 2
結果爲 -> 720
```

遞迴 (Recursive) (Cont.)

- ★重要★ 使用遞迴時, 最重要的一點就是**結束遞迴的條件**, 也就是不再呼叫自己的條件
- 以 RecursiveTest.java 為例，在 class Calculator 中的 factorial method 中的中止條件是
  - if (levelNum<=1)
- 缺少中止條件的情況：程式不斷呼自己，無法結束程式，跟迴圈沒有加結束迴圈的終止條件一樣形成無窮迴圈

各個擊破 (Divide and Conquer)

- 春秋戰國，張儀提連橫策略，分裂魏、燕、齊 ... 等六國，再**各個擊破**
- 遞迴的概念同各個擊破，先將大問題分解成數個小問題，把小問題解決後統合其結果，便解了大問題
- 以階層計算為例，每呼叫自己一次都將問題縮小 1，直到最後可以取得結果 (i.e. levelNum <=1) 時再回頭一步步計算總結果
- 二分搜尋法能否也用各個擊破？

各個擊破 (Divide and Conquer) (Cont.)

- 二分搜尋法 (Binary search)
  1. 將資料依遞增方式排序
  2. 如果中間元素即是要找的資料，則輸出結果
  3. 如果中間元素的值比要找的資料小，則很右側找
  4. 否則，往左側找
- 能否改用遞迴？
  - 能將大問題分解成小問題嗎？
  - 中止條件是什麼？

各個擊破 (Divide and Conquer) (Cont.)

- 在遞迴的二分搜尋法中，**終止條件**為何？
  - 當資料找到時 (i.e. data[middle] == item)
  - 當資料不在陣列中時 (i.e. low > high)
- 使用遞迴的優點
  - 程式簡潔
  - 節省記憶體空間
- 使用遞迴的缺點
  - 費時，因每呼叫自己一次就必須做參數傳遞

各個擊破 (Divide and Conquer) (Cont.)

```
71 class Searcher{
72     int[] data;
73     int item;
74     int binsearch(int low, int high){
75         if(low > high){ // 找不到資料
76             return -1;
77         }
78         int middle = (low+high) / 2;
79         if(data[middle] == item){
80             return middle;
81         }else if(data[middle] > item){
82             return binsearch(low, middle -1);
83         }else{
84             return binsearch(middle+1, high);
85         }
86     }
87 }
88
89 public class OOPTest{
90     public static void main(String[] argv) throws IOException{
91         BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
92         Sortor st = new Sortor();
93         Searcher sc = new Searcher();
94         System.out.print("請輸入資料個數: ");
95         int eleNum = Integer.parseInt(br.readLine());
96         //System.out.print("遞增(a) or 遞減(d) 排序? ");
97         //String flag = br.readLine();
98         int[] data = new int[eleNum];
99         for(int i=0;i<eleNum;i++){
100            System.out.print("data["+i+"]->");
101            data[i] = Integer.parseInt(br.readLine());
102        }
103        data = st.bubbleSort(data, "a");
104        for(int i=0;i<data.length;i++){
105            System.out.print(data[i]+"\t");
106        }
107        System.out.print("\n請輸入要查找的資料: ");
108        sc.data = data;
109        sc.item = Integer.parseInt(br.readLine());
110        int res = sc.binsearch(0, data.length);
111        if(res== -1){
112            System.out.println("Sorry, 您要找的資料 "+sc.item+" 不在陣列中");
113        }else{
114            System.out.println("您要找的資料 "+sc.item+" 在陣列中的第 "+res+" 號位置");
115        }
```

```
請輸入資料個數: 7
data[0]->56
data[1]->54
data[2]->24
data[3]->12
data[4]->89
data[5]->101
data[6]->14
12      14      24      54      56      89      101
請輸入要查找的資料: 39
Sorry, 您要找的資料 39 不在陣列中
```

```
請輸入資料個數: 7
data[0]->56
data[1]->65
data[2]->85
data[3]->42
data[4]->13
data[5]->98
data[6]->3
3       13      42      56      65      85      98
請輸入要查找的資料: 98
您要找的資料 98 在陣列中的第 6 號位置
```

方法的多重定義(Overloading)

- 在呼叫某類別的方法時必須要與該方法所要傳入的參數個數及型態一致，才能順利運作
  - 但 System.out.println(); 可以
  - 又 System.out.println("^_^"); 也可以，為何會這樣？
  - 而且不管在其中傳入什樣資料型態都能成功印出資料
- 達到上面的要求，關鍵在於方法可以進行多重定義

方法的多重定義(Overloading) (Cont.)

- 遇類似意義的動作，但處理對象型別不同而有差異時可用多重定義解決
- 多重定義：同樣一個方法名，但傳入的參數數量及型別可以不一樣
- Java 除了會辨識方法名稱之外，還會加入傳入參數的數量及型別了以整理後產生**方法簽名** (method signature)
- Java 在進行方法呼叫時是依**方法簽名**來找到正確該執行的方法

方法的多重定義(Overloading) (Cont.)

程式 Overloading.java 以多種方式表示矩形

```
01 class Test {
02
03  // 1號版本：使用寬與高
04  int rectangleArea(int width,int height) {
05      return width * height;
06   }
07
08  // 2號版本：使用座標
09  int rectangleArea(int top,int left,int bottom,int right) {
10      return (right - left) * (bottom - top);
11   }
12 }
13
```

```
14 public class Overloading {
15
16    public static void main(String[] argv){
17      Test a = new Test();
18      int area;
19
20      area = a.rectangleArea(10,20);
21      System.out.println("矩形面積：" + area);
22
23      area = a.rectangleArea(5,5,15,25);
24      System.out.println("矩形面積：" + area);
25    }
26 }
```

執行結果

```
矩形面積：200
矩形面積：200
```

方法的多重定義(Overloading) (Cont.)

- 注意事項：
  - 要讓同名方法具有不同的**方法簽名** (Method signature)，必須要傳入參數的**數量**或**資料型態**不同才行
  - 傳入參數的名稱及回傳值的資料型別不是產生**方法簽名** (method signature) 的依據