1. Background and Review

1. Introduction

In COMP101 the basic elements of the Java programming language were introduced. In the first part of COMP102 we will be concerned with the topic of

Data Structures

both in the sense of the abstract properties, organisation, and operations on the more important of these, and in the context of their specific realisations within the Java environment.

This brief description may raise a few questions:

1. Why consider more involved mechanisms for storing and operating upon data, when it is (presumably) possible to realise solutions to programming design problems using only Primitive Data Types, Strings, and simple Arrays?

   Answers:
   - The initial design of a solution to an application problem is frequently expressed in terms of processing data that is assumed to be stored within some highly structured form. This may happen because the problem suggests such representations in a 'natural' way, e.g. applications involving networks, tables of information, lists of tasks to be processed in a given order, family trees, etc etc. In refining the initial design towards a final implementation it is easier to think in terms of operating on such high-level representations.
   - The choice of data structure can have a significant influence on how efficiently (both in terms of the time and the storage space used) an algorithm solves a particular problem. This will be evident from the course COMP108

2. Even if there is some point to considering such structures, why do we need to be concerned with anything other than how to implement these in Java? In other words, why consider these in a language independent context?

   Answer:

   Because Java is just a single programming language within a specific class (or paradigm) of programming languages. An understanding of the abstract organisation and operations on the more commonly occurring complex data structures provides a basis for both the ability to implement such structures within any sufficiently powerful language, e.g. BASIC, C, Pascal, even assembler; and gives a language independent context for studying algorithms that utilise such structures, cf COMP108 (again).

Over the next lectures the following structures will be examined:

1. Multi-dimensional Arrays.
2. Linked Lists.
3. Stacks and Queues.
4. Trees.

In this lecture, we examine some general aspects of data structures.

2. Representation and Manipulation

If one considers different types of data structure, e.g. Strings which were introduced in COMP101, then it may appear that the only issue one has to be concerned with is how to implement the structure in terms of basic language types: having solved this , the question of how to implement different operations on the structure (and what type of operations to provide) is (in effect) directed by the chosen realisation.

Consider the String class as specified in java.lang.String.

This provides:

1. Mechanisms for creating new Strings, i.e. the Class Constructors.
2. Mechanisms for performing operations involving Strings, e.g. comparing two instances for equality, determining the length (number of characters in) a String instance, etc., i.e. the Class Methods.

There are two important points to notice about this scheme:

1. Both of these are defined within a single package.
2. The manner in which the representation of a String is carried out does not have to be known by applications programs using them.

Java, in common with all genuinely Object-Oriented programming languages, provides an environment for implementing compound data types (such as String) whereby the representation (in terms of Primitive Types and other structures) can only be manipulated by the operations (i.e. Method) that are defined to do so.

(These ideas have already been discussed when Objected-Oriented Programming principles were introduced earlier.)

In many programming languages (particularly older imperative languages such as Pascal, Fortran, C) there is no 'clean' mechanism for enforcing such a regime.

[Note The philosophy that a data type embodies not only its internal organisation but also the operations that can be performed on it - hence data types are realised so that the former is only reachable via the latter - did not originate with Java or even Object-Oriented languages; it in fact was utilised in languages such as ML, and had been proposed by researchers by the late 1970s, e.g. Barbara Liskov's CLU programming language.**]**.

By combining the definitions of

- The implementation of a data structure.
  and
- The operations that act upon the structure.

implementation details can be 'hidden' from programs employing a given structure, e.g. there are several ways in which the String type could be implemented - char[] is one, but such details are unimportant in the context of programs that use the String class. As a result:

1. In applications where the development process is shared among several teams of designers, should Designer X need to use data structures defined by Team Y, all that X has to be aware of are the operations (i.e. the methods) and their definitions as provided by Y: X does not need to know how these are realised in the class that Y implements.
2. Run-time errors are less likely to arise by reason of inconsistencies caused when the internal form of a data structure can be manipulated directly, e.g. if the String type were 'openly' realised as a char array, problems might arise when programmers attempted to exploit this.
3. Provided the operations defined in the class methods remain unchanged, a different implementation of a structure could be substituted without programs using the class having to be rewritten, e.g. in principle one could implement String as an array of int

The organisation of Abstract Data Types (ADTs), as a collection of data and the operations acting upon the data, is known as Encapsulation, and is an important element of Object-Oriented Programming approaches.

3. Arrays of Objects

You have already encountered an important compound data type in the form of Simple Array Structures.

Java does not limit the use of Array structures to the basic types - int, double, char, boolean, etc, but allows arrays of Object Instances, to be defined and manipulated.

This capability is particularly useful in the context of defining tables of records, for example of the type that would occur in a typical relational database.

As an example, consider the Triangle Recognition Exercise that was set in COMP101.

The code below describes a Class Triangle

```
//
// COMP102
//
// Example 1:
// Triangle Class
//
// Paul E. Dunne  - Monday 13th September 1999
//
// Import Triangle Recognition Package
//    Methods: public double Largest (double s1,s2,s3 )
//             public double Median (double s1,s2,s3 )
//             public double Smallest (double s1,s2,s3 )
//             public String WhatIsIt (double S,M,L )
//
import COMP101_TriangEX;

public class Triangle
  {
  //
  // Fields
  //
  // L : Longest side length.
  // M : Median side length.
  // S : Shortest side length.
  // x[2], y[2], z[2]: Co-ordinates of triangle vertices;
  // TypeOf: Scalene, Isosceles, Equilateral, Not a triangle;
  //
  // N.B. Floating point , so avoid testing Right-angle
  //      since numeric comparison may be inaccurate.
  //
  //
  protected double L=0;                // Default
  protected double M=0;                // Default
  protected double S=0;                // Default
  protected double[] x_corner = {0,0};   // Default
  protected double[] y_corner = {0,0};   // Default
  protected double[] z_corner = {0,0};   // Default
  protected String TypeOf;
  //
  // Constructors
  //
  // 1. Default Constructor:
  //
  public Triangle()
    {
    TypeOf = new String("Not a Triangle");
    }
  //
  // 2. Given 3 side lengths;
  //    Determine type and possible corner co-ordinates
  //    assuming x_corner = {0,0}; y_corner={0,L};
  //

  public Triangle ( double s1,
                    double s2,
                    double s3 )
    {
    L= COMP101_TriangEX.Largest(s1,s2,s3);
    M= COMP101_TriangEX.Median(s1,s2,s3);
    S= COMP101_TriangEX.Smallest(s1,s2,s3);
    TypeOf = new String (COMP101_TriangEX.WhatIsIt(S,M,L));
    if ( !TypeOf.equals("Not a Triangle") )
       {
       // Fix Consistent Set of Co-ordinates;
       y_corner[1] = L;
       z_corner[1] = (M*M-S*S-L*L)/(2*L);
       z_corner[0] = Math.sqrt( M*M - z_corner[1]*z_corner[1] );
       };
    }
   //
   // 3. Given 3 points in the plane: x, y, z
   //    Construct a triangle with side-lengths dist(x,y); dist(x,z); dist(y,z)
   //    and determine its type.
   public Triangle ( double[] x,
```

```
                      double[] y,
                      double[] z )
     {
     double t1;
     double t2;
     double t3;
     x_corner[0] = x[0]; x_corner[1] = x[1];
     y_corner[0] = y[0]; y_corner[1] = y[1];
     z_corner[0] = z[0]; z_corner[1] = z[1];
     t1 = Math.sqrt ( (x[0]-y[0])*(x[0]-y[0]) + (x[1]-y[1])*(x[1]-y[1]) );
     t2 = Math.sqrt ( (x[0]-z[0])*(x[0]-z[0]) + (x[1]-z[1])*(x[1]-z[1]) );
     t3 = Math.sqrt ( (y[0]-z[0])*(y[0]-z[0]) + (y[1]-z[1])*(y[1]-z[1]) );
     L= COMP101_TriangEX.Largest(t1,t2,t3);
     M= COMP101_TriangEX.Median(t1,t2,t3);
     S= COMP101_TriangEX.Smallest(t1,t2,t3);
     TypeOf = new String(COMP101_TriangEX.WhatIsIt(S,M,L));
     }
    //
    // Methods
    //
    public static void PrintTriangle ( Triangle T )
      {
      System.out.println ( "Smallest side : "+ T.S );
      System.out.println ( "Median side : "+ T.M );
      System.out.println ( "Largest side  : "+ T.L );
      System.out.println ("Position : "+ "(" + T.x_corner[0] + " " + T.x_corner[1]
                                       + "(" + T.y_corner[0] + " " + T.y_corner[1]
                                       + "(" + T.z_corner[0] + " " + T.z_corner[1]
      System.out.println("Type Of: "+T.TypeOf);
      }
    public boolean Wellformed()
      {
      return !TypeOf.equals("Not a Triangle");
      }

    public char FormCode()
      {
      return TypeOf.charAt(0);
      }
    public int[] Sides()
      {
      int[] res ={S,M,L}
      return res;
      }
}
```

Example 1: Triangle Class Realisation

In this class there are 7 data fields:

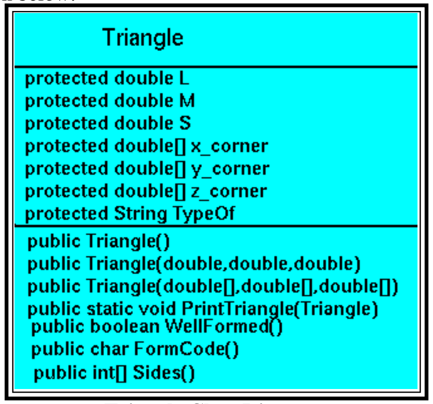
1. 3 double values - **L, M** and **S** - that hold the Largest, Median, and Smallest sides of the Triangle.
2. 3 arrays (each of length 2) - **corner_x, corner_y**, and **corner_z**- which hold Cartesian co-ordinates for the triangle vertices.
3. A String - **TypeOf** - that holds information concerning the category into which the triangle stored fits.

In addition there are 3 Constructors: a default Constructor; one which takes 3 side lengths (in any order) computes the correct field values for these lengths, using the convention that one of the vertices has co-ordinates (0,0) and one has co-ordinates (0,L). The final Constructor computes field values given three points in the plane representing triangle vertices.

There are 4 Methods:

- A Class Method to print the values of each field.
- An Instance Method indicating whether the instance is a well-defined triangle.
- An Instance Method to return the type of triangle as a single character (E, I, or S).
- An Instance Method to return the side lengths ordered by increasing size.

The Class Diagram is shown below:



Triangle Class Diagram

Suppose we wish to define an array of some number, LEN say, Triangle objects. This is, done by a similar method to that used in declaring any other array by:

```
Triangle[] TriArray = new Triangle[LEN];
```

Declaration of Array of Triangle Objects

Of course, since the type of element in TriArray is not one of the basic types, this declaration will not instantiate any of the individual components. In order to this an appropriate Constructor as defined in the Triangle Class must be invoked. For example we could do this by:

```
for (i=0; i< LEN; i++) TriArray[i] = new Triangle();
```

Instantiation of Array of Triangle Objects

4. A Simple Example Program

We illustrate the use of an array of Objects, with respect to a simple example involving the Triangle class introduced above. This example concerns collecting and analysing data pertaining to triangles satisfying certain restrictions.

4.1. Requirements

We are given an integer bound on the length of any side of a triangle: 10 say,

1. Determine the number of distinct valid triangles with integer side lengths at most 10.
2. Output the numbers of equilateral, isosceles (but not equilateral), and scalene triangles of this form,
3. For each category of valid triangle, list all of the valid side lengths.

The qualification 'distinct' means that triangles **S** and **T** with side lengths ($\mathbf{s_1}$, $\mathbf{s_2}$, $\mathbf{s_3}$) and ($\mathbf{t_1}$, $\mathbf{t_2}$, $\mathbf{t_3}$) are counted separately if and only if at least one of the side lengths in **S** differs from all of the side lengths in **T**.

4.2. Analysis

There is no user supplied input necessary (we assume that the maximum side length has been specified as 10). We need to consider the following:

1. How to enumerate all possible distinct cases, i.e. distinct triples of integer values between 1 and 10.
2. The process by which the total numbers of each valid triangle form is calculated.
3. The mechanisms for printing the results required.

Since the defining condition for three lengths (S,M,L) (with S<=M<=L) to be valid is that S+M> L, the simplest method to calculate the number of valid cases is:

```
final static int LEN_UPB = 10;   // Longest Side Length
static int MAX_POSS = 0;
private static void Compute_Max ()
    {
    for (int i=1; i<=LEN_UPB; i++)
      for (int j=i; j<=LEN_UPB; j++)
        for (int k=j; k<=LEN_UPB; k++)
          if (i+j>k) MAX_POSS++;
    }
```

Figure 1.1: Total number of Valid Triangles Computation

Once this number has been computed we know how many elements the array of Triangle instances needs to hold. Thus, having defined the identifier of the array the number of elements is set using:

```
static Triangle[] TriArray;      // The array of Triangle Objects
                                 // to be instantiated in main()
    ....
public static void main( String[] args )
    {
    Compute_Max();               // Calculate number of valid triangles.
    TriArray = new Triangle[MAX_POSS];  // Initiate array;
       ...
    }
```

Figure 1.2: Instantiation of Triangle Array - i) Number of Elements

In the computational processing part, it remains to instantiate the individual elements of the Triangle array, and to realise the counting of each different type of triangle.

Since we have three types of triangle - equilateral, isosceles, and scalene - we use three different integers to maintain these:

```
static int equi_tot=0;
static int iso_tot=0;
static int scal_tot=0;
```

Figure 1.3: Counters For Different Triangle Types

The process of updating these and instantiating the individual Triangle array elements can be carried out within the same nested for loop structure: we know that a given triple of integer values - (i,j,k) - in which i<=j<=k, can only be a valid triangle if i+j> k: - recall that this is the method by which the total number of elements to allow in the array was determined. We can therefore proceed by generating (i,j,k) (with i<=j<=k) in turn; testing if i+j> k; and instantiating the next element of our array using the Constructor Triangle(double,double,double) provided by the class Triangle, i.e. this one. Hence

```
static int IsTriangle = 0;
  ...
public static void main( String[] args )
   {
   for (int i=1; i<=LEN_UPB; i++)
     for (int j=i; j<=LEN_UPB; j++)
       for (int k=j; k<=LEN_UPB; k++)
         if (i+j>k)
           {
           TriArray[IsTriangle] = new Triangle(i,j,k);   // Instantiate specific el
            //
            //  Determine which counter should increase
            //
           IsTriangle++;           // Increase count of number of valid cases store
           };
```

Figure 1.4: Instantiation of Triangle Array - ii) Individual Elements

The maintenance of the counters defined in Figure 1.3 can be carried out whenever a valid triple (i,j,k) of side lengths is found, via a switch statement triggered on the result of the Instance method FormCode() associated with the Triangle class. Thus,

```
switch(TriArray[IsTriangle].FormCode())
              {
              case 'E': equi_tot++; break;
              case 'I': iso_tot++; break;
              case 'S': scal_tot++;
              };
```

Figure 1.5: Updating Triangle Type Counters

Finally we have to deal with the output requirements. There are two stages to this: print out the statistics concerning the total numbers; and the details of the specific triangles of each type

```
    //
    // Output Statistics
    //
    System.out.println("Triangles with integer side lengths at most " + LEN_UPB);
    System.out.println("##################################################");
    System.out.println("Well formed | " + IsTriangle);
    System.out.println("Equilateral      | " + equi_tot);
    System.out.println("Isosceles        | " + iso_tot);
    System.out.println("Scalene          | " + scal_tot);
    System.out.println("##################################################");
    System.out.println();
```

Figure 1.6: Output Total Numbers

For the second part we use a method within the application class that is parameterised by a character describing the triangle form of interest:

```
//
// Print the sides of all triangles of type
// EIS, where EIS is one of 'E'(quilateral) 'I'(sosceles) or 'S'(calene).
// Triangle output as (S,M,L): 5 to a line;
//
private static void ListTypes ( final char EIS, final int MAX, Triangle[] T )
  {
  int per_line = 0;
  for (int i=0; i< MAX; i++)
    {
    if (T[i].FormCode()==EIS)
      {
      System.out.print("("+T[i].Sides()[0]+","+
                           T[i].Sides()[1]+","+
                           T[i].Sides()[2]+") ; ");
      per_line++;
      if (per_line==5)
        {
        per_line=0; System.out.println();
        };
      };
    };
  }
```

Figure 1.7: Output Different Cases for the Specified Type of Triangle

This is invoked in the main() method by:

```
    //
    // List each found in the different categories.
    //
    System.out.println("Equilateral Triangles with Integer Side Length at most " +
    System.out.println("##########################################################
    ListTypes('E',IsTriangle,TriArray);
    System.out.println();
    System.out.println("Isosceles Triangles with Integer Side Length at most " + L
    System.out.println("##########################################################
    ListTypes('I',IsTriangle,TriArray);
    System.out.println();
    System.out.println("Scalene Triangles with Integer Side Length at most " + LEN
    System.out.println("##########################################################
    ListTypes('S',IsTriangle,TriArray);
    System.out.println();
```

Figure 1.8: Final Output Stage

The complete implementation is given below:

```
//
// COMP102
// Example 2: Array of Object Instances
```

```
//            Triangle Data
//
// Paul E. Dunne 15/9/1999
//
import Triangle;                   // The Triangle Class
import java.io.*;
class TriConApp
  {
  // Fields
  final static int LEN_UPB = 10;   // Longest Side Length
  static int MAX_POSS = 0;
  static int equi_tot=0;
  static int iso_tot=0;
  static int scal_tot=0;
  static int IsTriangle = 0;
  static Triangle[] TriArray;      // The array of Triangle Objects
                                   // to be instantiated in main()
  //
  // Calculate how many possible triangles of side
  // length at most LEN_UPB.
  // Could do this directly, but would require opaque
  // algebraic screed in initiating MAX_POSS.
  //
  private static void Compute_Max ()
    {
    for (int i=1; i<=LEN_UPB; i++)
      for (int j=i; j<=LEN_UPB; j++)
        for (int k=j; k<=LEN_UPB; k++)
          if (i+j>k) MAX_POSS++;
    }
  //
  // Print the sides of all triangles of type
  // EIS, where EIS is one of 'E'(quilateral) 'I'(sosceles) or 'S'(calene).
  // Triangle output as (S,M,L): 5 to a line;
  //
  private static void ListTypes ( final char EIS, final int MAX, Triangle[] T )
    {
    int per_line = 0;
    for (int i=0; i< MAX; i++)
      {
      if (T[i].FormCode()==EIS)
        {
        System.out.print("("+T[i].Sides()[0]+","+
                             T[i].Sides()[1]+","+
                             T[i].Sides()[2]+") ; ");
        per_line++;
        if (per_line==5)
          {
          per_line=0; System.out.println();
          };
        };
      };
    }
   public static void main( String[] args )
     {
     for (int i=1; i<=LEN_UPB; i++)
       for (int j=i; j<=LEN_UPB; j++)
         for (int k=j; k<=LEN_UPB; k++)
           if (i+j>k)
             {
             TriArray[IsTriangle] = new Triangle(i,j,k);   // Instantiate specific
```

```
               //
               //  Determine which counter should increase
               //
               switch(TriArray[IsTriangle].FormCode())
                 {
                 case 'E': equi_tot++; break;
                 case 'I': iso_tot++; break;
                 case 'S': scal_tot++;
                 };
               IsTriangle++;           // Increase count of number of valid cases st
               };
     //
     // Output Statistics
     //
     System.out.println("Triangles with integer side lengths at most " + LEN_UPB);
     System.out.println("##################################################");
     System.out.println("Well formed      | " + IsTriangle);
     System.out.println("Equilateral      | " + equi_tot);
     System.out.println("Isosceles        | " + iso_tot);
     System.out.println("Scalene          | " + scal_tot);
     System.out.println("##################################################");
     System.out.println();
     //
     // List each found in the different categories.
     //
     System.out.println("Equilateral Triangles with Integer Side Length at most "
     System.out.println("##########################################################
     ListTypes('E',IsTriangle,TriArray);
     System.out.println();
     System.out.println("Isosceles Triangles with Integer Side Length at most " +
     System.out.println("##########################################################
     ListTypes('I',IsTriangle,TriArray);
     System.out.println();
     System.out.println("Scalene Triangles with Integer Side Length at most " + LE
     System.out.println("##########################################################
     ListTypes('S',IsTriangle,TriArray);
     System.out.println();
     }
  }
```

Figure 1.9: Full Implementation: Array of Objects Example Program

When executed the output below results:

```
Triangles with integer side lengths at most 10
##################################################
Well formed     | 125
Equilateral     | 10
Isosceles       | 65
Scalene         | 50
##################################################

Equilateral Triangles with Integer Side Length at most 10
############################################################
(1,1,1) ; (2,2,2) ; (3,3,3) ; (4,4,4) ; (5,5,5) ;
(6,6,6) ; (7,7,7) ; (8,8,8) ; (9,9,9) ; (10,10,10) ;

Isosceles Triangles with Integer Side Length at most 10
############################################################
(1,2,2) ; (1,3,3) ; (1,4,4) ; (1,5,5) ; (1,6,6) ;
(1,7,7) ; (1,8,8) ; (1,9,9) ; (1,10,10) ; (2,2,3) ;
(2,3,3) ; (2,4,4) ; (2,5,5) ; (2,6,6) ; (2,7,7) ;
(2,8,8) ; (2,9,9) ; (2,10,10) ; (3,3,4) ; (3,3,5) ;
(3,4,4) ; (3,5,5) ; (3,6,6) ; (3,7,7) ; (3,8,8) ;
(3,9,9) ; (3,10,10) ; (4,4,5) ; (4,4,6) ; (4,4,7) ;
(4,5,5) ; (4,6,6) ; (4,7,7) ; (4,8,8) ; (4,9,9) ;
(4,10,10) ; (5,5,6) ; (5,5,7) ; (5,5,8) ; (5,5,9) ;
(5,6,6) ; (5,7,7) ; (5,8,8) ; (5,9,9) ; (5,10,10) ;
(6,6,7) ; (6,6,8) ; (6,6,9) ; (6,6,10) ; (6,7,7) ;
(6,8,8) ; (6,9,9) ; (6,10,10) ; (7,7,8) ; (7,7,9) ;
(7,7,10) ; (7,8,8) ; (7,9,9) ; (7,10,10) ; (8,8,9) ;
(8,8,10) ; (8,9,9) ; (8,10,10) ; (9,9,10) ; (9,10,10) ;

Scalene Triangles with Integer Side Length at most 10
############################################################
(2,3,4) ; (2,4,5) ; (2,5,6) ; (2,6,7) ; (2,7,8) ;
(2,8,9) ; (2,9,10) ; (3,4,5) ; (3,4,6) ; (3,5,6) ;
(3,5,7) ; (3,6,7) ; (3,6,8) ; (3,7,8) ; (3,7,9) ;
(3,8,9) ; (3,8,10) ; (3,9,10) ; (4,5,6) ; (4,5,7) ;
(4,5,8) ; (4,6,7) ; (4,6,8) ; (4,6,9) ; (4,7,8) ;
(4,7,9) ; (4,7,10) ; (4,8,9) ; (4,8,10) ; (4,9,10) ;
(5,6,7) ; (5,6,8) ; (5,6,9) ; (5,6,10) ; (5,7,8) ;
(5,7,9) ; (5,7,10) ; (5,8,9) ; (5,8,10) ; (5,9,10) ;
(6,7,8) ; (6,7,9) ; (6,7,10) ; (6,8,9) ; (6,8,10) ;
(6,9,10) ; (7,8,9) ; (7,8,10) ; (7,9,10) ; (8,9,10) ;
```

Figure 1.10: Output from Array of Triangle Objects Example Program

5. Summary

- An Abstract Data Type (ADT) is defined by a collection of data and a set of operations manipulating it. The principle of Encapsulation states that Data and Operations on the data should be defined together as part of a single 'unit'
- In Java ADTs can be realised as new Objects using Class definitions. The data elements are defined within the class fields and can consist of Primitive Types, together with Compound Structures (e.g. Arrays) and instances of previously defined Objects. Creation of new instances is achieved through the Class Constructors and the operations available specified through the Methods defined . Recall that such methods can be in one of two forms:
    - Class methods, which are instantiated exactly once, and indicated by the static modifier in the method header, e.g. the method PrintTriangle in the Triangle Class. A Class Method is called by a statement of the form

      NameOfClass.NameOfMethodInClass(parameters);

    - Instance methods, a copy of which is instantiated for each instance created. Instance Methods provide a 'safe' mechanism for accessing (instance) fields of a class, e.g. the method FormCode() in the Triangle Class which extracts data from the String field TypeOf. Instance Methods are invoked by a statement of the form:

      NameOfInstanceOfClass.NameOfMethodInClass(parameters);

- Encapsulation and Data Hiding (so that data fields are accessed only through prescribed methods) provide a number of advantages in the development of applications programs:
    - The organisation of data within an ADT can be changed without programs using the ADT having to be aware of it (and thence re-written).
    - The risk of errors in data integrity is greatly reduced, since only permissible methods can alter data.
    - Documentation can be organised in more tightly structured fashion, cf. the documentation of Java Package definitions here.
- In order to support the ideas of Data Hiding, data fields in a class definition are normally declared as private or protected.
- ADTs are a fundamental study within software and algorithm design (in all programming language paradigms, i.e. not simply Object-Oriented approaches). Careful design of an appropriate ADT in a given application can show considerable benefits in clarity, maintenance, and efficiency of the final implementation.

(Notes prepared and maintained by Paul E. Dunne, October 1999)
