3. Multi-dimensional Arrays (Continued)

An Example Application

1. Giant Characters Revisited

In the previous lecture, reference was made to a natural setting in which multi-dimensional arrays have potential uses. In this lecture we develop an application program illustrating this particular setting.

Recall that two drawbacks of encoding characters as sequences of String instances were mentioned:

1. The appearance of a ‘character’ was fixed in that only the symbols ’*’ and ’ ’ were used. Thus the foreground (’*’) and background (’ ’) indicators could not be altered (other than by explicitly editing the source code).
2. The print out of individual sequences of characters was performed progressing from the top of a page downwards, rather than the more usual left-to-right style.

The example to be developed presently will address both of these points. First we consider the definition of a class any instance of which will hold representations of the three digits **0**, **1** and **2** stored within a 2-dimensional char array. This class will contain Instance Methods to

1. Set the internal representation of a symbol using any pair of characters as foreground and background symbols.
2. Print out any specified row of the 2-dimensional array storing the depiction of a given digit.
3. Print out the representation of a digit within the instance.

We then describe how this class can be used within a program that converts a given input (non-negative) integer into a ternary representation and prints out this representation using the GIANT_TERNARY class proceeding from left to right.

2. The GIANT_TERNARY Class Definition

Figure 3.1 below shows the Class Diagram for the GIANT_TERNARY class that will be defined.

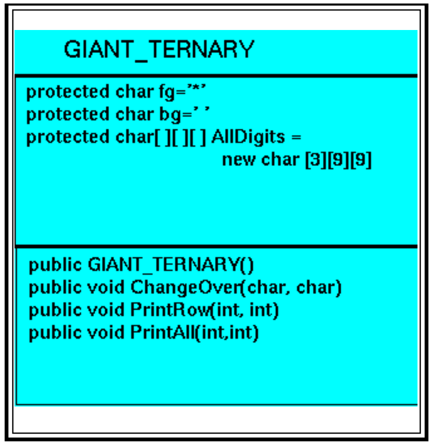

Figure 3.1: GIANT_TERNARY Class Definition.

An instance of GIANT_TERNARY has 3 instance variables: the char types **fg** and **bg** hold, respectively, the symbols that are to be used as foreground and background printing symbols when displaying one of the three digits - **0**, **1**, or **2** - in its giant printing form. The actual representation of these three digits is stored in the 3-dimensional char array **AllDigits**. Recalling the viewpoint of a 3-dimensional array as an simple array of 2-dimensional array, described in the last lecture, the three component (2-dimensional) arrays

- AllDigits[0]
- AllDigits[1]
- AllDigits[2]

store the depiction of the relevant digit (i.e. **0**, **1**, or **2** as indicated by the index) as a pattern within a 2-dimensional char array. The pattern is encoded using the specified foreground and background symbols within 9 rows each of 9 columns.

The single constructor for the class instantiates all 3 of these 9-by-9 grids to contain the default encoding of the giant digit.

The Java realisation of the constructor is shown below

```
public class GIANT_TERNARY()
  {
  //
  // This probably looks a bit meaningless, but is easiest
  // mechanism for instantiating a default setting.
```

```
//
// Using fg='*' and bg=' ', the appearance encoded
// is:
//
//    ***       ***      *****
//   *****     * **     **   **
//  **   **   *  **     **   **
//  **   **      **     **   **
//  **   **      **         **
//  **   **      **        **
//  **   **      **       **
//   *****    *******    ******
//    ***     *******   *******
//
// N.B. Layout convention which allows easy checking
// that all elements of the 3-dimensional array have
// been accounted for.
//
final char[][][] TempDigits = {
                                { {bg,bg,bg,fg,fg,fg,bg,bg,bg},
                                  {bg,bg,fg,fg,fg,fg,fg,bg,bg},
                                  {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                  {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                  {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                  {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                  {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                  {bg,bg,fg,fg,fg,fg,fg,bg,bg},
                                  {bg,bg,bg,fg,fg,fg,bg,bg,bg}
                                },
                                { {bg,bg,bg,bg,fg,fg,fg,bg,bg,bg},
                                  {bg,bg,bg,fg,bg,fg,fg,bg,bg,bg},
                                  {bg,fg,bg,bg,fg,fg,bg,bg,bg},
                                  {bg,bg,bg,bg,fg,fg,bg,bg,bg},
                                  {bg,bg,bg,bg,fg,fg,bg,bg,bg},
                                  {bg,bg,bg,bg,fg,fg,bg,bg,bg},
                                  {bg,bg,bg,bg,fg,fg,bg,bg,bg},
                                  {bg,fg,fg,fg,fg,fg,fg,fg,bg},
                                  {bg,fg,fg,fg,fg,fg,fg,fg,bg}
                                },
                                { {bg,bg,fg,fg,fg,fg,fg,bg,bg},
                                  {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                  {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                  {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                  {bg,bg,bg,bg,bg,fg,fg,bg,bg},
                                  {bg,bg,bg,bg,fg,fg,bg,bg,bg},
                                  {bg,bg,bg,fg,fg,bg,bg,bg,bg},
                                  {bg,bg,fg,fg,fg,fg,fg,fg,bg},
                                  {bg,fg,fg,fg,fg,fg,fg,fg,bg}
                                }
                              };
// Now instantiate the appropriate field for the class;
AllDigits = TempDigits;
}
```

Figure 3.2: Default Constructor for GIANT_TERNARY Class

The three instance methods perform the following functions:

1. **ChangeOver**(char fore, char back)

   When invoked this method replaces all of the occurrences of the current foreground symbol (fg) used in representing digits with the character stored in **fore** and, similarly all occurrences of the current background symbol (bg) with the character indicated by **back**. These functions are simply carried out by a nested for loop construction:

```
//*************************************************************
// Change the representation used by default, to employ    *
// the specified foreground symbol (fore) and background   *
// symbol (back).                                           *
//*************************************************************
public void ChangeOver (char fore, char back)
  {
  for (int i=0; i<3; i++)
    for (int j=0; j<9; j++)
      for (int k=0; k<9; k++)
        if (AllDigits[i][j][k]==fg)
          AllDigits[i][j][k]=fore;
        else
          AllDigits[i][j][k]=back;
  fg=fore; bg=back;
  }
```

Figure 3.3: Instance method to alter default representation.

2. PrintRow(int row_value, int ternary_digit)

   The parameter **ternary_digit** indicates which of the 2-dimensional arrays **AllDigits[0]**, **AllDigits[1]**, or **AllDigits[2]** is to be used for the output. The parameter **row_value** references one of the 1-dimensional char arrays within **AllDigits[ternary_digit]**, namely **AllDigits[ternary_digit][row_value]**. This array is then output, no line break occurring after the output is completed.

```
//
// Given a digit (0,1, or 2), output the
// specified row (in row_value) of its
// representation as a String, without a new line being thrown
// after printing.
//
public void PrintRow( int row_value, int ternary_digit )
  {
  System.out.print(String.valueOf(AllDigits[ternary_digit][row_value]));
  }
```

Figure 3.4: Instance method to print a single row of a digit.

3. PrintAll(int ternary_digit)

This instance method outputs the representation of the value in **ternary_digit** as stored in the 2-dimensional char array **AllDigits[ternary_digit]**. The method just invokes the PrintRow method for each of the 9 rows, outputting a newline after each row.

```
//
// Print out the representation of a ternary digit.
//
public void PrintAll (int ternary_digit )
  {
  for (int i=0; i<9; i++)
     {
     PrintRow(i,ternary_digit); System.out.println();
     };
  }
```

Figure 3.5: Instance Method to print an entire digit.

The complete Class implementation is given in Figure 3.6:

```
//
// COMP102
// Example 3: Constrained Multi-dimensional Array
//            Storing and Printing Giant Digits
//
// Paul E. Dunne 12/10/1999
//
public class GIANT_TERNARY
  {
  //
  // Fields
  //
  protected char fg='*';     // Default Foreground Symbol
  protected char bg=' ';     // Default Background Symbol
  protected char[][][] AllDigits= new char[3][9][9];   // Representation
  //
  // Single Constructor
  //
  public GIANT_TERNARY()
     {
     //
     // This probably looks a bit meaningless, but is easiest
     // mechanism for instantiating a default setting.
     //
     // Using fg='*' and bg=' ', the appearance encoded
     // is:
     //
     //    ***       ***      *****
     //   *****     * **     **   **
     //  **   **   *  **     **   **
     //  **   **      **     **   **
     //  **   **      **         **
     //  **   **      **        **
     //  **   **      **       **
     //   *****    *******    ******
     //    ***     *******   *******
     //
     //
     //
     // N.B. Layout convention which allows easy checking
     // that all elements of the 3-dimensional array have
     // been accounted for.
     //
     final char[][][] TempDigits = {
                                   { {bg,bg,bg,fg,fg,fg,bg,bg,bg},
                                     {bg,bg,fg,fg,fg,fg,fg,bg,bg},
                                     {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                     {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                     {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                     {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                     {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                     {bg,bg,fg,fg,fg,fg,fg,bg,bg},
                                     {bg,bg,bg,fg,fg,fg,bg,bg,bg}
                                   },
                                   { {bg,bg,bg,fg,fg,fg,bg,bg,bg},
                                     {bg,bg,fg,bg,fg,fg,bg,bg,bg},
                                     {bg,fg,bg,bg,fg,fg,bg,bg,bg},
                                     {bg,bg,bg,bg,fg,fg,bg,bg,bg},
                                     {bg,bg,bg,bg,fg,fg,bg,bg,bg},
                                     {bg,bg,bg,bg,fg,fg,bg,bg,bg},
                                     {bg,bg,bg,bg,fg,fg,bg,bg,bg},
                                     {bg,fg,fg,fg,fg,fg,fg,fg,bg},
                                     {bg,fg,fg,fg,fg,fg,fg,fg,bg}
                                   },
                                   { {bg,bg,fg,fg,fg,fg,fg,bg,bg},
                                     {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                     {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                     {bg,fg,fg,bg,bg,bg,fg,fg,bg},
                                     {bg,bg,bg,bg,bg,fg,fg,bg,bg},
                                     {bg,bg,bg,bg,fg,fg,bg,bg,bg},
                                     {bg,bg,bg,fg,fg,bg,bg,bg,bg},
                                     {bg,bg,fg,fg,fg,fg,fg,fg,bg},
                                     {bg,fg,fg,fg,fg,fg,fg,fg,bg}
                                   }
                                 };
     // Now instantiate the appropriate field for the class;
     AllDigits = TempDigits;
     }
  //***********************************************************
  // Instance Methods                                         *
  //***********************************************************
  //
  // Change the representation used by default, to employ
  // the specified foreground symbol (fore) and background
  // symbol (back).
  //
  public void ChangeOver (char fore, char back)
    {
    for (int i=0; i<3; i++)
      for (int j=0; j<9; j++)
        for (int k=0; k<9; k++)
          if (AllDigits[i][j][k]==fg)
            AllDigits[i][j][k]=fore;
          else
            AllDigits[i][j][k]=back;
```

```
    fg=fore; bg=back;
    }
  //
  // Given a digit (0,1, or 2), output the
  // specified row (in row_value) of its
  // representation as a String, without a new line being thrown
  // after printing.
  //
  public void PrintRow( int row_value, int ternary_digit )
    {
    System.out.print(String.valueOf(AllDigits[ternary_digit][row_value]));
    } // PrintRow
  //
  // Print out the representation of a ternary digit.
  //
  public void PrintAll (int ternary_digit )
    {
    for (int i=0; i<9; i++)
       {
       PrintRow(i,ternary_digit); System.out.println();
       };
    }
  }
```

Figure 3.6 Implementation of the Class GIANT_TERNARY

3. An Application using the GIANT_TERNARY Class

We develop a Java program that solves the following:

3.1. Requirements

Given user-supplied foreground and background symbols to be used in printing, and a non-negative integer, **n**, say, output the ternary representation of of **n** using a 9-by-9 depiction composed of the foreground and background symbols for each digit. The output should be such that the ternary value of **n** is printed from left to right, i.e. in imitation of the 'normal' method of printing numerical values.

Reminder: You will be familiar, by now, with the concept of representing a non-negative integer value using **binary** (or base 2), **octal** (base 8) and **hexadecimal** (base 16), cf. COMP103. Obviously, such systems can be defined to encompass representing integer values with respect to any base integer. The ternary number system is that which uses the number **3** as its base, so that only the digits **0**, **1**, **2** are employed and successive digits (moving from right-to-left) indicate the contribution to the value of increasing powers of **three**, e.g. the integer values 1,3,9,27,81 are written in ternary as 1,10,100,1000,10000; the integer value 73 is written in ternary as 2201 since

$$73 = 2*3^3 + 2*3^2 + 0*3^1 + 1*3^0$$
$$= 2*27 + 2*9 + 0 + 1$$

3.1. Analysis

We can view the solution program as involving the stages below:

1. Obtain the user-specified foreground and background printing symbols.
2. Customise an instance of a GIANT_TERNARY object to use these symbols.
3. Obtain the non-negative integer value that is to be printed using GIANT_TERNARY digits.
4. Convert this integer to an equivalent ternary representation.
5. Output the ternary representation using the corresponding GIANT_TERNARY digit for each digit, with the output format respecting the requirements specified, i.e. the digits are output from left-to-right.

Assuming that the relevant input stream has been configured and that the two characters required are to be held in variables identified as **fore** and **back** respectively, accomplishing the first part could be carried out by by prompting the user for the required information, and then taking the first character of the String input as its value, i.e.

```
public static void Get_fore_and_back() throws IOException
  {
  String input_line;
  System.out.print("Foreground Character:");
  fore = input_line.charAt(0);
  System.out.print("Background Character:");
  back = input_line.charAt(0);
  }
```

'Naive' Input Harness for Foreground and Background Data

The main drawback of this approach, however, is that if the user simply hits the **< return>** in reply then a run-time error will result since the String **input_line** will contain no characters (and will not have a 'first' character). Of course it is trivial to test for this, and rather than iterate until a non-empty String is supplied, we can use such a response as indicating that the default setting be used. So, the input mechanism now becomes,

```
public static void Get_fore_and_back() throws IOException
  {
  String input_line;
  System.out.print("Foreground Character (hit return for default value):");
  input_line = keyboardInput.readLine();
  if (input_line.length()==0)   // Empty String read: use default.
    {
    fore='*';
    }
  else
    {
    fore = input_line.charAt(0);
    };
  System.out.print("Background Character (hit return for default value):");
  if (input_line.length()==0)   // Empty String read: use default.
    {
    back=' ';
    }
  else
    {
    back = input_line.charAt(0);
    };
  }
```

Improved Input Harness for Foreground and Background Data

For the second part we use the instance method

```
ChangeOver(char,char)
```

provided by the GIANT_TERNARY Class. Since this is an instance method a variable of the appropriate type must be instantiated. Hence,

```
static GIANT_TERNARY NumberForm=new GIANT_TERNARY();
static char fore;       // For user-defined foreground symbol
static char back;       // For user-defined background symbol

...

public static void main( String[] args ) throws IOException
    {
    Get_fore_and_back();
    NumberForm.ChangeOver(fore,back);  // Modify the instantiation to
                                       // use the new foreground/background
                                       // symbols.
    }
```

The final item of input is the integer value to be printed in ternary. This will be held in an int **InputValue**. No data validation, i.e. a test that this value is non-negative is performed. Following the instantiation of the GIANT_TERNARY Object, the required value is obtained using

```
System.out.print("Enter non-negative integer value:");
InputValue = new Integer(keyboardInput.readLine()).intValue();
```

The remaining stages - (4) and (5) - constitute the bulk of the processing that is required of the application.

Consider the first of these - converting the integer value stored in **InputValue** to an equivalent ternary representation. We have to consider,

1. In what form the sequence of ternary digits will be stored.
2. How to construct this from from any given non-negative integer value.

Since, when the final output stage is being performed, it will be necessary to access each digit in the ternary representation repeatedly (in order to meet the requirement that the GIANT_TERNARY encoding is printed from left to right) we can choose between an 'explicit' array representation **or** use a String each element of which will be one of the characters **0**, **1**, or **2**,

Advantages of using an 'explicit' array

1. If the elements are of type int then each element will have an integer value **0**, **1**, or **2**. This can then be used directly to access the appropriate 2-dimensional array describing the printing form of the digit within the 3-dimensional array comprising the instance NumberForm.
2. Similarly, it will be possible to store each digit directly into an int[] without having to convert to a char equivalent.

Disadvantages of using an 'explicit' array

1. We cannot know in advance of reading **InputValue** the exact number of elements such an array ought to have allocated to it. Of course, once this value is known then (cf. binary representation) at most

   $$\mathbf{1 + \log_3 InputValue}$$

   elements suffice (this value being rounded-up to the nearest integer).
2. The conversion may result in extraneous 'leading 0s' being stored, e.g. **26** would be stored in a 4 element array that would be printed as **0222** rather than as (the minimal) **222**. Thus some additional processing would be needed to detect if this had occurred and the necessary measures taken to commence the output from the first significant (non-zero) digit.

Advantages of using a String

1. The appropriate String instance can be formed without explicit reference to how long this has to be (using the concatentation operator **+**).
2. The problem is leading 0s can be avoided in a simple manner.

Disadvantage of using a String

Neither the storage of single digits in the String form nor the determination of the correct 2-dimensional form to access when printing a giant digit, can be carried out directly: the some translation from a character to an (integer) index has to be made.

It is clear that the gains made in one form of representation translate into problems in the alternative form. As is often the case in program development, neither choice is self-evidently ‘correct’ or ‘incorrect’.

We choose to develop our solution in terms of storing the ternary representation using Strings, mainly to avoid certain opacities that arise in the dealing with the alternative form. Thus,

String **TernaryValue**;

will be the variable used to hold the ternary representation of **InputValue**.

Turning to the actual conversion process, this is handled by a method

```
public static String IntegerToTernary( final int n )
```

The realisation of this is given below:

```
//************************************************************
// Convert integer to ternary representation returning String *
// of the encoding digits as the result.                      *
//************************************************************
public static String IntegerToTernary( final int n )
  {
  String res = new String();
  int tn = n;           // Save value of input parameter.
  int next_digit;       // The next ternary digit to be stored in res.
  if (tn==0)            // Can convert 0 immediately.
    {
    res= "0"+res;
    }
  else
    {
    while (tn> 0)
      {
      next_digit = tn%3;   // Find the least significant ternary digit left;
      tn=tn/3;             // and ‘shift’ this off the current value of tn.
      //
      // Append the correct symbol to the result String.
      // We could do this directly (i.e. avoiding case statement)
      // by using an integer-character encoding, however, this
      // assumes a specific encoding scheme (ASCII say) and
      //is a bit opaque.
      //
      switch(next_digit)
        {
        case 0: res = "0"+res; break;
        case 1: res = "1"+res; break;
        case 2: res = "2"+res;
        };
      };
    };
  return res;
  }
```

Figure 3.7 Method converting integer value (n) to String of ternary digits

The final stage is the actual output of the data in **TernaryValue** as a sequence of GIANT_TERNARY templates.

This is carried out within the **main()** method.

Recall that the GIANT_TERNARY class provides an Instance method that outputs a single specified row of the 9-by-9 matrix of a given digit encoding **without** throwing a line-break after printing. Using this method, all that needs to be done is

print a new_line
Print the first row of each digit in **TernaryValue**
print a new_line
Print the second row of each digit in **TernaryValue**

print a new_line
Print the third row of each digit in **TernaryValue**
print a new_line
....
Print the last row of each digit in **TernaryValue**
print a new_line

i.e.

```
//
// Output Stage
//
// Proceeding from left-to-right output
// the current (i'th) row of the GIANT_TERNARY digit
// corresponding to each digit in the ternary representation
// of the integer InputValue
//
System.out.println();  // Start printing on a new line.
for (int i=0; i< 9; i++)
  {
  for (int j=0; j<TernaryValue.length(); j++)
    {
    //
    // As with IntegerToTernary conversion, the case
    // statement could be replaced by a direct calculation.
    //
    switch(TernaryValue.charAt(j))
      {
      case '0': NumberForm.PrintRow(i,0); break;
      case '1': NumberForm.PrintRow(i,1); break;
      case '2': NumberForm.PrintRow(i,2);
      };
    };
    System.out.println();        // Throw a new line after one row completed.
   };
```

Figure 3.8: Main Output Stage

The complete implementation is given in Figure 3.9:

```
//
// COMP102
// Example 4: Printing GIANT_TERNARY representation
//
// Paul E. Dunne 13/10/99
//
import java.io.*;
import GIANT_TERNARY;  // Ternary Digit Class
//
public class GiantTernaryOut
  {
```

```
// Instantiate Input streams
public static InputStreamReader input = new InputStreamReader(System.in);
public static BufferedReader   keyboardInput = new BufferedReader(input);
//
static GIANT_TERNARY NumberForm=new GIANT_TERNARY();
static char fore;     // For user-defined foreground symbol
static char back;     // For user-defined background symbol
static int InputValue;  // User supplied integer to be printed in ternary
                        // using GIANT_TERNARY representation.
static String TernaryValue;  // This will contain the ternary value of InputValue
                             // as a String of {0,1,2} symbols.
//*************************************************************
// Read user-defined foreground and background                *
// symbols.                                                   *
//*************************************************************
public static void Get_fore_and_back() throws IOException
  {
  String input_line;
  System.out.print("Foreground Character (hit return for default value):");
  input_line = keyboardInput.readLine();
  if (input_line.length()==0)   // Empty String read: use default.
   {
   fore='*';
   }
  else
    {
    fore = input_line.charAt(0);  // otherwise take first character.
    };
  System.out.print("Background Character (hit return for default value):");
  input_line = keyboardInput.readLine();
  if (input_line.length()==0)   // Empty String read: use default.
   {
   back=' ';
   }
  else
    {
    back = input_line.charAt(0);  // otherwise take first character.
    };
  }  // Get_fore_and_back
//***********************************************************
// Convert integer to ternary representation returning String *
// of the encoding digits as the result.                    *
//***********************************************************
public static String IntegerToTernary( final int n )
  {
  String res = new String();
  int tn = n;           // Save value of input parameter.
  int next_digit;       // The next ternary digit to be stored in res.
  if (tn==0)            // Can convert 0 immediately.
    {
    res= "0"+res;
    }
  else
    {
    while (tn> 0)
      {
      next_digit = tn%3;   // Find the least significant ternary digit left;
      tn=tn/3;             // and 'shift' this off the current value of tn.
      //
      // Append the correct symbol to the result String.
      // We could do this directly (i.e. avoiding case statement)
```

```
        // by using an integer-character encoding, however, this
        // assumes a specific encoding scheme (ASCII say) and
        //is a bit opaque.
        //
        switch(next_digit)
          {
          case 0: res = "0"+res; break;
          case 1: res = "1"+res; break;
          case 2: res = "2"+res;
          };
        };
      };
    return res;
    }   // IntegerToTernary
  //
  // Main Method
  //
  public static void main( String[] args ) throws IOException
    {
    Get_fore_and_back();
    NumberForm.ChangeOver(fore,back);  // Modify the instantiation to
                                       // use the new foreground/background
                                       // symbols.
    System.out.print("Enter non-negative integer value:");
    InputValue = new Integer(keyboardInput.readLine()).intValue();
    TernaryValue = IntegerToTernary(InputValue);
    //
    // Output Stage
    //
    // Proceeding from left-to-right output
    // the current (i'th) row of the GIANT_TERNARY digit
    // corresponding to each digit in the ternary representation
    // of the integer InputValue
    //
    System.out.println();   // Start printing on newline.
    for (int i=0; i< 9; i++)
      {
      for (int j=0; j<TernaryValue.length(); j++)
        {
        //
        // As with IntegerToTernary conversion, the case
        // statement could be replaced by a direct calculation.
        //
        switch(TernaryValue.charAt(j))
          {
          case '0': NumberForm.PrintRow(i,0); break;
          case '1': NumberForm.PrintRow(i,1); break;
          case '2': NumberForm.PrintRow(i,2);
          };
        };
        System.out.println();        // Throw a new line after one row completed.
       };
    }  // main
  }
```

Figure 3.9 Printing GIANT_TERNARY Digits

The alternative representation, using int[] rather than String can be found here.

3.2. Testing

a) Black Box Testing

Limit Tests

The cases we have to consider here are: the three possible combinations of at least one foreground/background symbol being the default; and cases when an integer value of **0** is to be printed. The table summarises the expected output for such cases.

<table>
<tr><th>Response to foreground symbol prompt</th><th>Response to background symbol prompt</th><th>Expected output on integer 0</th></tr>
<tr><td>< Return ></td><td>< Return ></td><td><pre>
   ***
  *****
 **   **
 **   **
 **   **
 **   **
 **   **
  *****
   ***
</pre></td></tr>
<tr><td>< Return ></td><td>.</td><td><pre>
...***...
..*****..
.**...**.
.**...**.
.**...**.
.**...**.
.**...**.
..*****..
...***...
</pre></td></tr>
<tr><td>@</td><td>< Return ></td><td><pre>
   @@@
  @@@@@
 @@   @@
 @@   @@
 @@   @@
 @@   @@
 @@   @@
  @@@@@
   @@@
</pre></td></tr>
</table>

b) White Box Testing

Path Testing

The only paths that have not been tested through Limit value testing are those concerned with ensuring that all of the possible ternary digits are stored and printed in the correct fashion. We can test for these simply by supplying as an input positive integer, one whose ternary representation requires the use of all 3 digits, e.g. **11** whose ternary form is **102**. We should also test (given one of the reasons for using String as the data type in which to store a ternary value) that leading 0s are not output. Any positive integer, however, which is **not** of the form

$$3^k \text{ or } 2*(3^k)$$

will be suitable, and since the ternary form of such numbers involves only the digits **1** and **0** (in the former case) or **2** and **0** (in the latter), it follows that any integer value involving all three digits in its ternary form will provide a suitable test case.

Foreground	Background	Input Integer	Expected output

#	,	11	`,,,###,,,,,,###,,,,,#####,,`
			`,,#,##,,,,,#####,,,##,,,##,`
			`,#,,##,,,,##,,,##,,##,,,##,`
			`,,,,##,,,,##,,,##,,##,,,##,`
			`,,,,##,,,,##,,,##,,,,,,##,,`
			`,,,,##,,,,##,,,##,,,,,##,,,`
			`,,,,##,,,,##,,,##,,,,##,,,,`
			`,#######,,,#####,,,,######,`
			`,#######,,,,###,,,,#######,`

4. Summary

The constrained multi-dimensional array used in the example application developed here, illustrates several of the points referred to in the previous lecture:

1. The initiation of such structures is handled in a manner that is similar to that used for 1-dimensional arrays, cf. the Constructor definition for the GIANT_TERNARY class
2. The interpretation of a **k**-dimensional array as an ordered (i.e. 1-dimensional array) collection of **(k-1)**-dimensional arrays, plays an important role in the analysis of the problem considered: although the 3-dimensional structure of char[3][9][9] that underpins the symbols representation may, superficially, appear complicated, the instantiation of this (in the instance field **AllDigits**) shows that this provides a simple device for referring to the composition of a single specific digit - **t** which is one of **0**, **1**, or **2** - as the 2-dimensional char array, **AllDigits [t]**.
3. In the same way, as was seen in the realisation of the main output process, viewing a 2-dimensional array as an ordered collection of 1-dimensional arrays, provides a straightforward solution to the problem of producing the output of a sequence of values in the required format.

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