Introduction to Affine Transformations

A rectangular array is often called a matrix. Matrix algebra is an important branch of mathematics, as many common and interesting problems are well expressed and developed using matrices. Let's explore one such example in the field of computer graphics – affine transformations. Figure 1 illustrates a simple house drawn in a graphics window. The house is shown to the right and below the origin.\(^1\) The house could be drawn using the `drawLine()` method of Java's `Graphics` class, as shown in previous programs. Five calls to `drawLine()` are required, each specifying the end points of one of the lines.

<table>
<thead>
<tr>
<th>houseX</th>
<th>houseY</th>
</tr>
</thead>
<tbody>
<tr>
<td>30, 30</td>
<td>60, 40</td>
</tr>
<tr>
<td>30, 40</td>
<td>40, 30</td>
</tr>
<tr>
<td>50, 30</td>
<td>70, 40</td>
</tr>
<tr>
<td>70, 60</td>
<td></td>
</tr>
</tbody>
</table>

An alternate way to draw the house is to store the points in arrays and use the `drawPolygon()` method:

```java
int[] houseX = { 30, 30, 50, 70, 70 };
int[] houseY = { 60, 40, 30, 40, 60 };
g.drawPolygon(houseX, houseY, houseX.length);
```

The `drawPolygon()` method requires three arguments: an array of \(x\) points, an array of \(y\) points, and the number of points. So far, so good. If we want to transform our basic house in some simple yet consistent manner, then we must confront a mathematical problem that is well-suited to matrix algebra, and, hence, multi-dimensional or rectangular arrays. In computer graphics, there are three basic transformations collectively known as affine transformations:

- **Translating** moving an object left, right, up, or down
- **Scaling** enlarging or shrinking an object

\(^1\) The coordinates are consistent with Java's graphics coordinate system.
Rotation \hspace{1cm} \text{rotating an object about a point}

The task of translating, scaling, or rotating our simple graphics house requires transforming each point into a new point such that the new set of points is true to the original set, except transformed in the desired manner. For translating, we simply add a displacement to each $x$ and $y$ coordinate:

\[ x' = x + d_x \quad y' = y + d_y \] \hspace{1cm} (1)

For scaling, we simply multiply each $x$ and $y$ coordinate by a scaling factor:

\[ x' = x \cdot s_x \quad y' = y \cdot s_y \] \hspace{1cm} (2)

For rotating, we transform each point as follows:

\[ x' = x \cdot \cos \theta - y \cdot \sin \theta \quad y' = x \cdot \sin \theta + y \cdot \cos \theta \] \hspace{1cm} (3)

where $\theta$ is the angle of rotation in degrees. Rotating is a bit tricky, so let's explain with a figure. Figure 2 shows a point $x,y$ and the same point $x',y'$ rotated about the origin by $\theta$ degrees. Since rotation is about the origin, the distance from the origin to each point is the same. This is shown as $z$ in the figure.

\[ x = z \cdot \cos \phi \] \hspace{1cm} (4)

\[ y = z \cdot \sin \theta \] \hspace{1cm} (5)

In terms of the new point $x',y'$ we have

\[ x' = z \cdot \cos(\theta + \phi) = z \cdot \cos \phi \cdot \cos \theta - z \cdot \sin \phi \cdot \sin \theta \] \hspace{1cm} (6)
Substituting equations 4 and 5 into equations 6 and 7 yields the equations for $x'$ and $y'$ (equation 3).

So, what does the above discussion have to do with multi-dimensional arrays or matrix algebra? A lot. As it turns out, the three affine transformations, and some variations on these, can all be implemented using the same operation — multiplying a vector by a transformation matrix. The arithmetic is simple, however, as you may be unfamiliar with vector and matrix algebra, a brief review is in order. For the purpose of programming, note that a vector is an array, and a matrix is a two-dimensional rectangular array.

A vector multiplied by another vector yields a single value as a result, for example

\[
\begin{pmatrix} 6 & 5 & 4 \end{pmatrix} \cdot \begin{pmatrix} 9 \\ 8 \\ 7 \end{pmatrix} = 6 \times 9 + 5 \times 8 + 4 \times 7 = 54 + 40 + 28 = 122
\]  

A vector multiplied by matrix yields a vector result, for example

\[
\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \cdot \begin{pmatrix} 4 & 5 & 6 \\ 3 & 2 & 1 \\ 7 & 6 & 5 \end{pmatrix} = \begin{pmatrix} 1 \times 4 + 2 \times 5 + 3 \times 6 \\ 1 \times 3 + 2 \times 2 + 3 \times 1 \\ 1 \times 7 + 2 \times 6 + 3 \times 5 \end{pmatrix} = \begin{pmatrix} 4 + 10 + 18 \\ 3 + 4 + 3 \\ 7 + 12 + 15 \end{pmatrix} = \begin{pmatrix} 32 \\ 10 \\ 34 \end{pmatrix}
\]

We'll put the operation above to use in our next Java program, so have another look if you're not convinced of the arithmetic. For our purpose, the vector represents a point to transform:

\[
P = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}
\]  

The matrix used to transform this point will differ depending on whether we are interested in translating (T), scaling (S), or rotating (R):

\[
T = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \quad S = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad R = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}
\]

So, if we are interested in translating a drawing, we take each point, $P$, and transform it into $P'$ using

\[
P' = P \cdot T
\]

If we want to scale our drawing, each point, $P$, is transformed into $P'$ using

\[
P' = P \cdot S
\]
and for rotating, each point, $P$, is transformed into $P'$ using

$$ P' = P \cdot R $$

(14)

The third value in the point vector is of no use to us, so it is discarded after the transformation. As an example, let's convince ourselves that equation 12 performs translation:

$$ P \cdot T = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} x \times 1 + y \times 0 + 1 \times d_x \\ x \times 0 + y \times 1 + 1 \times d_y \\ x \times 0 + y \times 0 + 1 \times 1 \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \\ 1 \end{pmatrix} = P' $$

(15)

The result above shows the translated point with new coordinates $x + d_x$ and $y + d_y$. This is the effect of translating, as expressed in equation 1 above.

The transformations above are powerful tools for computer graphics. For an example, see DemoAffineTransformations.
import java.awt.*;
import java.applet.*;

public class DemoRotate extends Applet {
  private static final int SIZE = 200;
  private static final double ROTATE = 45.0; // degrees cw

  public void paint(Graphics g) {
    // points for the house polygon
    int[] houseX = {30, 30, 50, 70, 70};
    int[] houseY = {-40, -60, -70, -60, -40};

    // compute rotation angle in radians
    double theta = Math.toRadians(ROTATE);

    // transformation matrix for rotation
    double[][] r = {
      {Math.cos(theta), -Math.sin(theta), 0.0},
      {Math.sin(theta), Math.cos(theta), 0.0},
      {0.0, 0.0, 1.0},
    };

    // draw a rectangle around the graphics window
    g.drawRect(0, 0, SIZE - 1, SIZE - 1);

    // set a new origin in the middle of the graphics window
    g.translate(SIZE / 2, SIZE / 2);

    // draw outline of the original house (in black)
    g.setColor(Color.black);
    g.drawPolygon(houseX, houseY, houseX.length);

    // transform points (use 'rotate' matrix)
    double[] p = new double[3];
    for (int i = 0; i < houseX.length; i++) {
      p[0] = houseX[i];
      p[1] = houseY[i];
      p[2] = 1.0;
      p = vmProduct(p, r);
      houseX[i] = (int)Math.round(p[0]);
      houseY[i] = (int)Math.round(p[1]);
    }

    // draw the transformed house
    g.drawPolygon(houseX, houseY, houseX.length);
  }

  // calculate product of vector 'v' and matrix 'm' (return a new vector)
  public static double[] vmProduct(double[] v, double[][][] m) {
    double[] temp = new double[v.length];
    for (int i = 0; i < v.length; i++)
      temp[i] = v[0] * m[0][i][0] + v[1] * m[1][i][1] + v[2] * m[2][i][2];
    return temp;
  }
}
Figure 3. DemoRotate.java.

This program generates the following output (Figure 4):

The techniques to draw the original house were covered earlier. The interesting part in this program is the implementation of the affine transformation for rotation using matrix algebra. The transformation matrix is initialized in lines 19-23 as a two-dimensional double array named $r$ (for "rotate"). The values in the array are consistent with the transformation matrix for rotation presented earlier (see equation 11). The amount of rotation is declared in the final constant $\text{ROTATE}$ in line 7 as 45°. Since Java's $\text{sin()}$ and $\text{cos()}$ methods expect an angle in radians, the $\text{toRadians()}$ method (line 16) transforms the angle before it is passed on to these methods.

After the original object is drawn, the affine transformation is applied to the array of points (lines 41-50). In turn, each point is read from the original array $\text{houseX}$ and $\text{houseY}$ and assigned to a temporary point array named $p$ (lines 44-46). This point and the transformation matrix $r$ are passed as arguments to the $\text{vmProduct()}$ method in line 47. The $\text{vmProduct()}$ method multiplies the vector ($p$) by the matrix ($r$) and returns a new vector in $p$. The first two values in $p$ are placed back into the $\text{houseX}$ and $\text{houseY}$ array as the newly transformed point (lines 48-49).

The vector-matrix multiplication in the $\text{vmProduct()}$ method is a straightforward implementation of the arithmetic shown earlier in equation 9. After the transformation the rotated house is drawn (line 53).

See Chapter 5 of Foley et al's *Computer graphics: Principles and practice* for more details on 2D transformations for computer graphics.
1.2 Vectors

The size of an array is set when the array is declared, and the array cannot grow or shrink thereafter. This presents a problem when data are read from an external source and the number of data items is not known in advance. We used a work-around earlier by declaring a "larger than necessary" array and then maintaining a variable to hold the "actual" number of elements initialized. Obviously, this is wasteful. It is also possible that someday more data items are read than anticipated. In this case the program simply won't work unless the source code is updated and the program is recompiled. These problems are averted using a dynamic array.

Java's Vector class implements a dynamic array of objects. Like an array, it contains elements accessible using a simple integer index. However, the size of a Vector object can grow or shrink to accommodate adding and removing items on an as-needed basis. Of course, there is behind-the-scenes storage management taking place, but this is transparent to us.

By way of introduction, we'll digress briefly with an example to convince you of the need for dynamic arrays. Let's revisit two programs shown earlier: RandomGen and NumberStats. The RandomGen program outputs random numbers to the standard output stream, and the NumberStats program analyses data read from the standard input stream. That's great because we can connect the two programs using a pipe. (Pipes were presented earlier.) A sample dialogue follows:

```
PROMPT>java RandomGen 100 400.0 600.0 | java NumberStats
N = 100
Minimum = 402.17233767914325
Maximum = 599.6718556474029
Mean = 506.783572075286
Standard deviation = 55.920094678999526
```

In the first part of the command line, the RandomGen program generates 100 random floating-point numbers ranging from 400 to 600. Normally these values are sent to the host system's CRT display; however, the pipe symbol (|) effectively "catches" the output and sends it as input to the NumberStats program (via the java interpreter). The output is not surprising given our understanding of these programs. However, if we are interested in gathering statistics on, say, 150 numbers, then a problem is lurking. Here is another sample dialogue:

```
PROMPT>java RandomGen 150 400.0 600.0 | java NumberStats
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
at NumberStats.main(Compiled Code)
```

The problem lies in the NumberStats program. If you recall, the numbers array was declared with a default size of 100 (see line 13 NumberStats.java). As soon as the 101st element arrived via the pipe, the program crashed with an "array index out of bounds" exception. The most obvious fix is to edit the source code and recompile with yet another "larger than necessary"
array size. But this is a nuisance.²

Dynamic Arrays of Primitive Data Types

An alternative approach, which we will explore now, is to re-work the NumberStats program using the Vector class, instead of arrays. A key difference between Vector objects and arrays is that the former only hold objects. To store primitive data types, like integers, characters, or floating-point numbers, the values must be encoded as wrapper-class objects. (Java’s wrapper classes were presented earlier.)

An object of the Vector class is instantiated like an object of any other class — using a constructor method. Once instantiated, elements are added and retrieved using instance methods. The Vector class methods are summarized in Table 1.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td><code>Vector()</code> Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero; returns a reference to the new object</td>
</tr>
<tr>
<td>Instance Methods</td>
<td></td>
</tr>
<tr>
<td><code>addElement(Object o)</code></td>
<td>Adds the specified element to the end of this vector, increasing its size by one; returns <code>void</code></td>
</tr>
<tr>
<td><code>add(int idx, Object o)</code></td>
<td>Adds the specified element at the specified position to the this vector; returns <code>void</code></td>
</tr>
<tr>
<td><code>add(Object o)</code></td>
<td>Adds the specified element to the end of this vector; returns a <code>boolean</code></td>
</tr>
<tr>
<td><code>elementAt(int idx)</code></td>
<td>Returns an <code>Object</code> representing the element at the specified index (same as <code>get()</code>)</td>
</tr>
<tr>
<td><code>get(int idx)</code></td>
<td>Returns an <code>Object</code> representing the element at the specified index. (same as <code>elementAt()</code>)</td>
</tr>
<tr>
<td><code>removeElementAt(int idx)</code></td>
<td>Deletes the element at the specified index; returns a <code>void</code></td>
</tr>
<tr>
<td><code>size()</code></td>
<td>Returns an <code>int</code> equal to the number of elements in this vector</td>
</tr>
</tbody>
</table>

² It is possible to create pseudo-dynamic arrays as follows. When the capacity of the original array is reached, a new, larger array is declared and the elements from the old array are copied into the new array. The old array is discarded. If the new array also reaches its capacity, the process is repeated, and so on.
Let's assume we need a dynamic array (a Vector) of integers. The following statement instantiates a Vector object named samples:

```java
Vector samples = new Vector();
```

To add an element to the vector, we use the `addElement()` method. However, as shown in Table 1, the argument must be an Object. Since all Java classes are subclasses of the Object class, we can use any object reference as an argument. We cannot, however, use a primitive data type. If we wish to add an `int` to `samples`, it first must be "wrapped" into an `Integer` object. If, for example, the value of the integer is 5, it is instantiated as an `Integer` object using

```java
Integer x = new Integer(5);
```

and then added to `samples` using the `addElement()` instance method:

```java
samples.addElement(x);
```

Of course, we have no use for the variable `x`, so we can combine the two steps above:

```java
samples.addElement(new Integer(5));
```

Note that the `Integer` constructor also accepts a string representation of an integer as an argument. For example

```java
samples.addElement(new Integer("5"));
```

Additional elements are added to `samples` in a similar manner. The best part is this: We need not concern ourselves with the capacity of `samples`. Extra space is allocated on an as-needed basis.

To retrieve an element from `samples`, we use the `elementAt()` instance method. The argument is a simple integer index specifying the location of the element we wish to retrieve. Once again, we have a minor problem because `samples` is a `Vector` object containing a dynamic array of `Object` objects. We cannot retrieve an element and assign it to an `int` variable:

```java
int y = samples.elementAt(i); // Wrong!
```

The `elementAt()` method returns an Object (actually, a reference to an Object). We can assign it to an `Integer` wrapper class object, but only if the `Object` reference is cast to an `Integer` reference:3

---

3 We did not cast the `Integer` reference to a `Object` reference when using `addElement()` because of the inheritance relationship between the `Integer` (subclass) and `Object` (superclass) classes: an `Integer` object is an `Object` object (always!). However, we must cast the `Object` reference to an `Integer` reference when going the other way using `elementAt()` because an `Object` object is not
Integer temp = (Integer)samples.elementAt(i);

That's half the story. What we really want is to retrieve an element and assign it to an int. So, we follow the statement above with

    int y = temp.intValue();

We can avoid instantiating the Integer object temp by combining the two preceding statements:

    int x = ((Integer)samples.elementAt(i)).intValue();

The extra parentheses are needed to ensure the operations occur in the correct order.

At any time, we can retrieve the number of elements stored in the vector using the size() method:

    for (int i = 0; i < samples.size(); i++)
        ...

So, unlike our experience with arrays, we do not need a separate variable representing the number of initialized entries.

Let's put these pieces together in a new version of the NumberStats program that uses dynamic arrays. The program NumbersStats2 inputs any number of floating point numbers from the standard input and outputs some simple statistics on these numbers (see Figure 5).
import java.io.*;
import java.util.*;

public class NumberStats2
{
  public static void main(String[] args) throws IOException
  {
    BufferedReader stdin =
      new BufferedReader(new InputStreamReader(System.in), 1);

    // declare 'v' as a dynamic array (a Vector object)
    Vector v = new Vector();

    // input data and put into dynamic array
    String line;
    while ((line = stdin.readLine()) != null)
    {
      // prepare to tokenize line
      StringTokenizer st = new StringTokenizer(line, " ,\t");

      // process tokens in line
      while (st.hasMoreTokens())
      {
        String s = st.nextToken();   // get a token
        Double d = new Double(s);    // convert and wrap as Double
        v.addElement(d);             // add to dynamic array
      }
    }

    // declare a double array with exactly the size needed
    double[] numbers = new double[v.size()];

    // copy and convert elements of Vector object into double array
    for (int i = 0; i < v.size(); i++)
      numbers[i] = ((Double)v.elementAt(i)).doubleValue();

    // output some statistics on the data
    System.out.println("N = " + numbers.length);
    System.out.println("Minimum = " + min(numbers));
    System.out.println("Maximum = " + max(numbers));
    System.out.println("Mean = " + mean(numbers));
    System.out.println("Standard deviation = " + sd(numbers));
  }

  // find the minimum value in an array
  public static double min(double[] n)
  {
    double min = n[0];
    for (int j = 1; j < n.length; j++)
      if (n[j] < min)
        min = n[j];
    return min;
  }

  // find the maximum value in an array
  public static double max(double[] n)
  {
    double max = n[0];
    for (int j = 1; j < n.length; j++)
      if (n[j] > max)
        max = n[j];
    return max;
Now we can do what our original version of NumberStats could not:

```
PROMPT>java RandomGen 150 400.0 600.0 | java NumberStats2
N = 150
Minimum = 400.4034372537817
Maximum = 598.4001802690623
Mean = 498.7361024890425
Standard deviation = 55.27611100869566
```

A `Vector` object named `v` is declared in line 12. Values are read from the standard input, as before, except now they are placed in the dynamic array `v` instead of in an over-sized `double` array. This involves converting the inputted `String` to a `Double` object (line 25) and then inserting it into the dynamic array using the `addElement()` method of the `Vector` class (line 26). Once all the elements are read, we can declare a `double` array with precisely the space need. This is done in line 31 using the `size()` method of the `Vector` class to specified the size of a `double` array named `numbers`.

For serious number crunching, it is much more efficient to work with a `double` array than with a vector containing a dynamic array of `Double` objects. So, the elements of the `Vector` object are retrieved, converted, and copied into the `double` array `numbers` (lines 34-35). The expression

```
((Double)v.elementAt(i)).doubleValue()
```

performs the convoluted task of retrieving the element at position `i`, casting it to a `Double` object, and converting it to a `double`. The result is assigned to position `i` of the `double` array `numbers`. 

Figure 5. NumberStats2.java.
At this point, it appears we are in basically the same position as in the original program, NumberStats. We have a double array of values, and we wish to calculate some simple statistics on the values. In fact, we are in a much better position. Because numbers is precisely the correct size, we do not need a separate variable for the "actual" number of initialized elements in the array: All elements are initialized! We can now work with the length field of the array — a much more elegant approach. All the statistics methods are modified slightly with this improvement. It is no longer necessary to pass the "size" of the array to the statistics methods. We simply pass the array reference and the for loops are setup using the length field of the array.

**Dynamic Arrays of Objects**

In the preceding example, the Vector class came to the rescue. We inputted an unspecified number of floating point values and created a double array with precisely the space needed. A dynamic array of Vector objects was used as an intermediate step, between inputting the values and creating the double array. However, the values had to be "wrapped" and "unwrapped" because the Vector class only works with objects of the Object class. This was inconvenient, but it was well worth the extra effort.

If the data of interest are objects (as opposed to primitive data types), then the Vector class is simple and natural to work with. This is illustrated in the following example working with String objects. The program MedalWinners uses a Vector object to hold the names of the medal winners in an Olympic event (see Figure 6).
import java.util.*;

public class MedalWinners
{
    private static final String[] MEDAL = { "GOLD", "SILVER", "BRONZE" };

    public static void main(String[] args)
    {
        // declare 'topThree' as a Vector object (a dynamic array)
        Vector topThree = new Vector();

        // add entries (top three finishers in men's 100 meter event)
        topThree.addElement("Ben Johnson (CAN), 9.83 s");
        topThree.addElement("Carl Lewis (USA), 9.92 s");
        topThree.addElement("Linford Christie (GBR), 9.97 s");

        // print results
        System.out.println("1988 Seoul Olympics, Men's 100 M");
        for (int i = 0; i < topThree.size(); i++)
            System.out.println(MEDAL[i] + ":\t" + topThree.elementAt(i));

        // remove Ben Johnson (disqualified)
        topThree.removeElementAt(0);

        // add new bronze medal winner
        topThree.addElement("Calvin Smith (USA), 9.99 s");

        // print revised results
        System.out.println("1988 Seoul Olympics, Men's 100 M (revised)");
        for (int i = 0; i < topThree.size(); i++)
            System.out.println(MEDAL[i] + ":\t" + topThree.elementAt(i));
    }
}

Figure 6. MedalWinners.java.

This program generates the following output:

1988 Seoul Olympics, Men's 100 M
GOLD: Ben Johnson (CAN), 9.83 s
SILVER: Carl Lewis (USA), 9.92 s
BRONZE: Linford Christie (GBR), 9.97 s

1988 Seoul Olympics, Men's 100 M (revised)
GOLD: Carl Lewis (USA), 9.92 s
SILVER: Linford Christie (GBR), 9.97 s
BRONZE: Calvin Smith (USA), 9.99 s

And the rest is history!

A Vector class object — a dynamic array — named topThree is declared in line 10. Three String objects are added to the array in lines 13-15. The addElement() method requires an Object as an argument, however an object of any class is a valid argument because all classes are subclasses of Object. As each element is added, the size of the dynamic array increases by one. The three elements are printed in lines 19-20, and you can see the effect in the first part of the
program's output above.

In the print statement in line 18, the following expression appears:

```java
topThree.elementAt(i)
```

The `elementAt()` method returns an `Object` reference; however, the `Object` is automatically converted to a string because of the concatenation operator. Be aware, however, that the following attempt to retrieve the gold-medal winner would generate a compile error:

```java
String goldMedalWinner = topThree.elementAt(0); // Wrong!
```

The `Object` returned by the `elementAt()` method cannot be assigned to a `String` object unless it is cast:

```java
String goldMedalWinner = (String)topThree.elementAt(0); // OK!
```

The statement above is good, clean Java code. Unfortunately, Mr. Johnson was not as clean. His use of metabolic steroids necessitated his removal from the Olympic record book. And so, too, we remove his entry from the `topThree` array in line 23 using the `removeElementAt()` method of the `Vector` class. The size of the array is automatically reduced by one. The previous Silver and Bronze medal winners are "bumped up" by one, and a new Bronze medal winner is added at the end of the array in line 26. The size of the array is then automatically increased by one. The final standings are outputted and appear in the last part of the program's output.

**Key Points – Arrays and Vectors**

We have explored a variety of new ways to work with data — namely through array data structures or using Java's `Vector` class. Here are the key points we have learned:

- Arrays are useful data structures to store a collection of data of the same type.
- Java supports arrays of objects (actually object references) as well as arrays of primitive data types.
- Elements in an array are accessed through an integer index. (For example, if `inventory` is an array and `i` is an integer, `inventory[i]` is the `i`th element in the array.)
- For each array a public data field named `length` holds the length of the array. (For example, if `gizmo` is an array, `gizmo.length` is the length of the array.)
- A common error in using arrays is attempting to access an element that does not exist. This error is not caught by the compiler; rather, it causes an "array index out of bounds" run-time error.
- Valid array indices range from 0 to "one less than" the length of the array. (For example,
An array can be initialized when it is declared using an *initialization list*. (For example, `String[] medals = { "GOLD", "SILVER", "BRONZE" };`.)

An array is an object, however its implementation as an object is through the compiler and the run-time interpreter.

Java supports multi-dimensional arrays, for example, to represent matrices.

An array is a *static* data structure. Once an array is declared, its size cannot change. Java supports *dynamic* arrays through its `Vector` class.

A `Vector` class object is a dynamic array of objects. Its size increases and decreases automatically as elements are removed or added.

Objects stored in or retrieved from a `Vector` object are of the `Object` class.

An object retrieved from a dynamic array (a `Vector` object) must be cast before it can be assigned to an object variable (unless the object variable is of the `Object` class).