```
        return at;
    }

    public static Affine3DMatrix Translation(double dx, double dy,
          double dz) {
        Affine3DMatrix at = new Affine3DMatrix(null);
        double[][] m = at.matrix;
        // matrix elements not listed are zero
        m[0][0] = 1;
        m[0][3] = dx;
        m[1][1] = 1;
        m[1][3] = dy;
        m[2][2] = 1;
        m[2][3] = dz;
        m[3][3] = 1;
        return at;
    }

    public Object clone() {
        return new Affine3DMatrix(matrix);
    }

    public double[] direct(double[] point) {
        int n = point.length;
        double[] tempPoint = new double[n];
        System.arraycopy(point, 0, tempPoint, 0, n);
        for(int i = 0;i<n;i++) {
            point[i] = 0;
            for(int j = 0;j<n;j++) {
                point[i] += matrix[i][j]*tempPoint[j];
            }
        }
        return point;
    }

    public double[] inverse(double[] point) throws
          UnsupportedOperationException {
        if(!inverted) {
            calcInverse(); // computes inverse using LU decompostion
        }
        if(inverse==null) { // inverse does not exist
            throw new UnsupportedOperationException("inverse matrix does
                  not exist");
        }
        int n = point.length;
        double[] pt = new double[n];
        System.arraycopy(point, 0, pt, 0, n);
        for(int i = 0;i<n;i++) {
            point[i] = 0;
            for(int j = 0;j<n;j++) {
                point[i] += inverse[i][j]*pt[j];
            }
        }
        return point;
    }
```



```
    // calculates inverse using Lower-Upper decomposition
    private void calcInverse() {
        LUPDecomposition lupd = new LUPDecomposition(matrix);
        inverse = lupd.inverseMatrixComponents();
        // signal that the inverse computation has been performed
        inverted = true;
    }
}
```

Exercise 17.21 Transformation interface
Write a simple program to test the `Affine3DMatrix` class. Show that the direct and inverse transformations reverse the mappings. ■

APPENDIX 17B: CONVERSIONS

Physicists use Euler angles because they are useful for analytically solving Euler's rigid body equations of motion in a small number of special cases. Because the quaternion representation is unfamiliar and is not taught in standard texts, we present conversion formulas between quaternions, rotation matrices, and Euler angles. See Shoemake for a more complete discussion of these conversions.

Quaternion to matrix. For a quaternion $q = (q_0, q_1, q_2, q_3)$ that satisfies the normalization condition $1 = q_0^2 + q_1^2 + q_2^2 + q_3^2$, the rotation matrix is

$$\mathcal{R} = 2\begin{bmatrix} \frac{1}{2} - q_2^2 - q_3^2 & q_1q_2 + q_0q_3 & q_1q_3 - q_0q_2 \\ q_1q_2 - q_0q_3 & \frac{1}{2} - q_1^2 - q_3^2 & q_2q_3 + q_0q_1 \\ q_1q_3 + q_0q_2 & q_2q_3 - q_0q_1 & \frac{1}{2} - q_1^2 - q_2^2 \end{bmatrix}. \tag{17.51}$$

Matrix to quaternion. The quaternion components can be computed using linear combinations of the rotation matrix elements of the (3×3) matrix $\mathcal{R} = [r_{i,j}]_{3\times3}$. To avoid the pitfall of dividing by a small number ϵ (the machine precision), we compute quaternion components using if statements:

(i) Compute $w^2 = (1 + r_{0,0} + r_{1,1} + r_{2,2})/4$. If $w^2 > \epsilon$, then

$$q_0 = \sqrt{w^2} \tag{17.52a}$$

$$q_1 = (r_{1,2} - r_{2,1})/4q_0 \tag{17.52b}$$

$$q_2 = (r_{2,0} - r_{0,2})/4q_0 \tag{17.52c}$$

$$q_3 = (r_{0,1} - r_{1,0})/4q_0, \tag{17.52d}$$

else compute $x^2 = -1/2(r_{1,1} + r_{2,2})$.