A
Mathematical Notation

A.1 Symbols

The following symbols are used in the main text primarily with the denotations given below. While some symbols may be used for purposes other than the ones listed, the meaning should always be clear in the particular context.

$\{a, b, c, d, \ldots\}$ — A *set*; i. e., an unordered collection of distinct elements. A particular element x can be contained in a set at most once. A set may also be empty (denoted by { }).

$(a_1, a_2, \ldots a_n)$ — A *vector*; i. e., a fixed-size, ordered collection of elements of the same type. $(a_1, a_2, \ldots a_n)^T$ denotes the *transposed* (i. e., column) vector. In programming, vectors are usually implemented as one-dimensional arrays, with elements being referred to by position (index).

$[c_1, c_2, \ldots c_m]$ — A *sequence* or *list*; i. e., an ordered collection of elements of variable length. Elements can be added to the sequence (inserted) or deleted from the sequence. A sequence may be empty (denoted by [ ]). In programming, sequences are usually implemented with dynamic data structures, such as linked lists. Java's *Collections* framework (see also Appendix B.2.7) provides numerous ready-to-use implementations.

$\langle \alpha_1, \alpha_2, \ldots \alpha_k \rangle$	A *tuple*; i. e., an ordered list of elements, each possibly of a different type. Tuples are typically implemented as *objects* (in Java or C++) or *structures* (in C) with elements being referred to by name.
$*$	Linear convolution operator (Sec. 5.3.1).
$\oplus$	Morphological dilation operator (Sec. 7.2.3).
$\ominus$	Morphological erosion operator (Sec. 7.2.4).
∂	Partial derivative operator (Sec. 6.2.1). For example, $\frac{\partial f}{\partial x}(x, y)$ denotes the *first* derivative of the function $f(x, y)$ along the x variable at position (x, y), $\frac{\partial^2 f}{\partial^2 x}(x, y)$ is the *second* derivative, etc.
∇	Gradient. ∇f is the vector of partial derivatives of a multidimensional function f (Sec. 6.2.1).
$\lfloor x \rfloor$	"Floor" of x, the largest integer $z \in \mathbb{Z}$ smaller than $x \in \mathbb{R}$ (i. e., $z = \lfloor x \rfloor \leq x$). For example, $\lfloor 3.141 \rfloor = 3$, $\lfloor -1.2 \rfloor = -2$.
a	Pixel value (usually $0 \leq a < K$).
$\mathrm{Arctan}(x, y)$	Inverse tangent function, similar to $\arctan\left(\frac{y}{x}\right) = \tan^{-1}\left(\frac{y}{x}\right)$ but with two arguments and returning angles in the range $[-\pi, +\pi]$ (i. e., covering all four quadrants). It corresponds to the Java method `Math.atan2(y,x)` (Secs. 6.3, B.1.6).
$\mathrm{card}\{\ldots\}$	Cardinality (size) of a set, $\mathrm{card}\, A \equiv \lvert A \rvert$ (Sec. 3.1).
$\mathsf{h}(i)$	Histogram of an image at pixel value (or bin) i (Sec. 3.1).
$\mathsf{H}(i)$	Cumulative histogram of an image at pixel value (or bin) i (Sec. 3.6).
$I(u, v)$	Intensity or color value of the image I at (integer) position (u, v).
K	Number of possible pixel values.
M, N	Number of columns (width) and rows (height) of an image ($0 \leq u < M$, $0 \leq v < N$).
mod	Modulus operator: (a mod b) is the remainder of the integer division a/b (Sec. B.1.2).

$\mathsf{p}(i)$	Probability density function (Sec. 4.6.1).
$\mathsf{P}(i)$	Probability distribution function or cumulative probability density (Sec. 4.6.1).
round(x)	Rounding function: rounds x to the nearest integer. round$(x) = \lfloor x + 0.5 \rfloor$.
truncate(x)	Truncation function: truncates x toward zero to the closest integer. For example, truncate$(3.141) = 3$, truncate$(-2.5) = -2$.

A.2 Set Operators

$\|A\|$	The size (number of elements) of the set A (equivalent to card A).
$\forall_x \ldots$	"All" quantifier (for all x, ...).
$\exists_x \ldots$	"Exists" quantifier (there is some x for which ...).
$\cup$	Set union (e. g., $A \cup B$).
$\cap$	Set intersection (e. g., $A \cap B$).
$\bigcup_{\mathcal{R}_i}$	Union over multiple sets $\mathcal{R}_i$.
$\bigcap_{\mathcal{R}_i}$	Intersection over multiple sets $\mathcal{R}_i$.

A.3 Algorithmic Complexity and $\mathcal{O}$ Notation

The term "complexity" describes the effort (i. e., computing time or storage) required by an algorithm or procedure to solve a particular problem in relation to the "problem size" n. Often complexity is reported in the literature using "big O" ($\mathcal{O}$) notation [18, Sec. 9.2], as in the following example. Consider a spreadsheet with 20 columns and 30 rows. Obviously, adding up all the entries in the spreadsheet requires performing $30 \cdot 20$ additions. We can be more general by representing the number of columns and rows by M and N, respectively, and saying it requires $M \cdot N$ additions. What if we want to replace each location with the sum of its eight neighbors? Then it would require $M \cdot N \cdot 8$ operations. If we compare these two algorithms, we see that, at their core, both require doing some number of operations $M \cdot N$ times. Since big O notation factors out constants (such as 8), we could say that the complexity of both of these

algorithms is $\mathcal{O}(MN)$.

$\mathcal{O}(MN)$ is an upper bound on the number of operations an algorithm requires on an input of size MN. We can simplify this, since typical images have roughly the same number of rows and columns, by selecting the larger of the rows and columns $n = \max(M, N)$ and replacing it with n. Now, since we know $n \cdot n \geq M \cdot N$ we can say their complexity is $\mathcal{O}(n \cdot n)$ or, more commonly, $\mathcal{O}(n^2)$. Big O notation lets us compare *classes* of algorithms—in this case we discovered that both our algorithms belong to the $\mathcal{O}(n^2)$ class. This tells us that, no matter how much we optimize our code, at the heart our algorithm will require n^2 operations.

Similarly, the direct computation of the linear convolution (Sec. 5.3.1) for an image of size $n \times n$ and a convolution kernel of size $k \times k$ has the time complexity $\mathcal{O}(n^2k^2)$. As another example, the *fast Fourier transform* (FFT, see Vol. 2 [6, Sec. 7.4.2]) of a signal vector of length $n = 2^k$ requires only $\mathcal{O}\big(n \log_2(n)\big)$ time.

Additional details on complexity can be found in any good book on computer algorithms, such as [1, 9].

B
Java Notes

As an undergraduate text for engineering curricula, this book assumes basic programming skills in a procedural language, such as C or Java. The examples in the main text should be easy to understand with the help of some introductory book on Java or one of the many online tutorials. Experience shows, however, that difficulties with some basic Java concepts pertain even at higher levels and frequently cause complications. The following sections aim at resolving some of these typical problem spots.

B.1 Arithmetic

Java is a "strongly typed" programming language, which means in particular that any variable has a fixed type that cannot be altered dynamically. Also, the result of an expression is determined by the types of the involved operands and *not* (in the case of an assignment) by the type of the "receiving" variable.

B.1.1 Integer Division

Division involving integer operands is a frequent cause of errors. If the variables `a` and `b` are both of type `int`, then the expression `(a / b)` is evaluated according to the rules of integer division. The result—the number of times `b` is contained in `a`—is again of type `int`. For example, after the Java statements

```
int a = 2;
int b = 5;
double c = a/b;
```

the value of `c` is *not* 0.4 but 0.0 because the expression `a/b` on the right produces the `int` value 0, which is then automatically converted to the `double` value 0.0.

If we wanted to evaluate `a/b` as a *floating-point* operation (as most pocket calculators do), at least one of the involved operands must be converted to a floating-point value, for example by an explicit type cast `(double)`:

```
double c = (double) a / b;
```

Notice that the type cast `(double)` only applies to the immediately following term (`a`) and not the entire expression `a / b`; i. e., the value of the second operand (`b`) in this division is still of type `int`.

Example

Assume, for example, that we want to scale any pixel value a of an image such that the maximum pixel value $a_{\max}$ is mapped to 255 (see Ch. 4). In mathematical notation, the scaling of the pixel values is simply expressed as

$$c \leftarrow \frac{a}{a_{\max}} \cdot 255,$$

and it may be tempting to convert this 1:1 into Java code, such as

```
int a_max = ip.getMaxValue();
...
int a = ip.getPixel(u,v);
int c = (a / a_max) * 255;  ← PROBLEM!
ip.putPixel(u,v,a);
...
```

As we can easily predict, the resulting image will be all black (zero values), except those pixels whose value was `a_max` originally (they are set to 255). The reason is again the division `(a / a_max)` with two operands of type `int`, where the result is zero whenever the divisor (`a_max`) is greater than the dividend (`a`).

Of course, the entire operation could be performed in the floating-point domain by converting one of the operands (as shown earlier), but this is not even necessary in this case. Instead, we may simply swap the order of operations and start with the multiplication,

```
int c = a * 255 / a_max;
```

Why does this work? The subexpression `a * 255` is evaluated first,[1] generating large intermediate values that pose no problem for the subsequent (integer) division. In addition, *rounding* should always be considered to obtain more accurate results when computing fractions of integers (see Sec. B.1.5).

[1] In Java, expressions at the same level are always evaluated in left-to-right order, and therefore no parentheses are required in this example (though they would not do any harm either).

B.1.2 Modulus Operator

The result of the modulus operator

$$a \bmod b$$

(used in several places in the main text) is defined [18, p. 82] as the remainder of the integer division a/b,

$$a \bmod b \triangleq \begin{cases} a & \text{for } b = 0 \\ a - b \cdot \left\lfloor \frac{a}{b} \right\rfloor & \text{otherwise.} \end{cases} \tag{B.1}$$

Unfortunately, this type of mod operator (or an equivalent library method) is not available in the standard Java API. Java's native `%` (*remainder*) operator, defined as

$$a \ \texttt{\%} \ b \triangleq a - b \cdot \text{truncate}\left(\frac{a}{b}\right) \quad \text{for } b \neq 0, \tag{B.2}$$

is often used in this context, but produces the same results only for *positive* operands $a \geq 0$ and $b > 0$. For example,

$$\begin{array}{rcrcr} 13 & \bmod & 4 & \to & 1 \\ 13 & \bmod & -4 & \to & -3 \\ -13 & \bmod & 4 & \to & 3 \\ -13 & \bmod & -4 & \to & -1 \end{array} \qquad \begin{array}{rcrcr} 13 & \texttt{\%} & 4 & \to & 1 \\ 13 & \texttt{\%} & -4 & \to & 1 \\ -13 & \texttt{\%} & 4 & \to & -1 \\ -13 & \texttt{\%} & -4 & \to & -1 \end{array}$$

The following Java method implements the mod operation according to the definition in Eqn. (B.1):

```
static int Mod(int a, int b) {
  if (b == 0)
    return a;
  if (a * b >= 0)
    return a - b * (a / b);
  else
    return a - b * (a / b - 1);
}
```

B.1.3 Unsigned Bytes

Most grayscale and indexed images in Java and ImageJ are composed of pixels of type `byte`, and the same holds for the individual components of most color images. A single byte consists of eight bits and can thus represent $2^8 = 256$ different bit patterns or values, usually mapped to the numeric range $0 \ldots 255$. Unfortunately, Java (unlike C and C++) does *not* provide a suitable "unsigned" 8-bit data type. The primitive Java type `byte` is "signed", using one of its eight bits for the $\pm$ sign, and can represent values in the range $-128 \ldots 127$.

Java's `byte` data can still be used to represent the values 0 to 255, but conversions must take place to perform proper arithmetic computation. For example, after execution of the statements

```
int  a = 200;
byte b = (byte) a;
```

the variables `a` (32-bit `int`) and `b` (8-bit `byte`) contain the binary patterns

```
a = 00000000000000000000000011001000
                          b = 11001000
```

respectively. Interpreted as a (signed) `byte` value, with the leftmost bit[2] as the sign bit, the variable `b` has the decimal value −56. Thus, after the statement

```
int a1 = b;              // a1 == -56
```

the value of the new `int` variable `a1` is −56! To (ab-)use signed `byte` data as *unsigned* data, we can circumvent Java's standard conversion mechanism by disguising the content of `b` as a logic (i. e., nonarithmetic) *bit pattern*; e. g., by

```
int  a2 = (0xff & b);     // a2 == 200
```

where `0xff` (in hexadecimal notation) is an `int` value with the binary bit pattern `00000000000000000000000011111111` and `&` is the bitwise AND operator. Now the variable `a2` contains the right integer value (200) and we thus have a way to use Java's (signed) `byte` data type for storing *unsigned* values. Within ImageJ, access to pixel data is routinely implemented in this way, which is considerably faster than using the convenience methods `getPixel()` and `putPixel()`.

B.1.4 Mathematical Functions (Class `Math`)

Java provides the standard mathematical functions as static methods in class `Math`, as listed in Table B.1. The `Math` class is part of the `java.lang` package and thus requires no explicit import to be used. Most `Math` methods accept arguments of type `double` and also return values of type `double`. As a simple example, a typical use of the cosine function $y = \cos(x)$ is

```
double x;
double y = Math.cos(x);
```

Similarly, the `Math` class defines some common numerical constants as static variables; e. g., the value of π could be obtained by

```
double x = Math.PI;
```

[2] Java uses the standard "2s-complement" representation, where a sign bit = 1 stands for a negative value.

Table B.1 Methods and constants defined by Java's `Math` class.

`double abs(double a)`	`double max(double a, double b)`
`int abs(int a)`	`float max(float a, float b)`
`float abs(float a)`	`int max(int a, int b)`
`long abs(long a)`	`long max(long a, long b)`
`double ceil(double a)`	`double min(double a, double b)`
`double floor(double a)`	`float min(float a, float b)`
`double rint(double a)`	`int min(int a, int b)`
`long round(double a)`	`long min(long a, long b)`
`int round(float a)`	`double random()`
`double toDegrees(double rad)`	`double toRadians(double deg)`
`double sin(double a)`	`double asin(double a)`
`double cos(double a)`	`double acos(double a)`
`double tan(double a)`	`double atan(double a)`
`double atan2(double y, double x)`	
`double log(double a)`	`double exp(double a)`
`double sqrt(double a)`	`double pow(double a, double b)`
`double E`	`double PI`

B.1.5 Rounding

Java's `Math` class (confusingly) offers three different methods for rounding floating-point values:

```
double rint (double x)
long  round (double x)
int   round (float x)
```

For example, a `double` value `x` can be rounded to `int` in one of the following ways:

```
double x; int k;
k = (int) Math.rint(x);
k = (int) Math.round(x);
k = Math.round((float)x);
```

If the operand `x` is known to be positive (as is typically the case with pixel values) rounding can be accomplished without using any method calls by

```
k = (int) (x + 0.5); // works for x ≥ 0 only!
```

In this case, the expression `(x + 0.5)` is first computed as a floating-point (`double`) value, which is then truncated (toward zero) by the explicit `(int)` typecast.

B.1.6 Inverse Tangent Function

The inverse tangent function $\varphi = \tan^{-1}(a)$ or $\varphi = \arctan(a)$ is used in several places in the main text. This function is implemented by the method `atan(double a)` in Java's `Math` class (Table B.1). The return value of `atan()` is in the range $[-\frac{\pi}{2} \ldots \frac{\pi}{2}]$ and thus restricted to only two of the four quadrants. Without any additional constraints, the resulting angle is ambiguous. In many practical situations, however, a is given as the ratio of two catheti $(\Delta x, \Delta y)$ of a right-angled triangle in the form

$$\varphi = \tan^{-1}\Big(\frac{\Delta y}{\Delta x}\Big),$$

for which we used the (self-defined) two-parameter function

$$\varphi = \mathrm{Arctan}(\Delta y, \Delta x)$$

in the main text. The function $\mathrm{Arctan}(\Delta y, \Delta x)$ is implemented by the static method `atan2(dy,dx)` in Java's `Math` class and returns an unambiguous angle φ in the range $[-\pi \ldots \pi]$; i. e., in any of the four quadrants of the unit circle.[3]

B.1.7 `Float` and `Double` (Classes)

The representation of floating-point numbers in Java follows the IEEE standard, and thus the types `float` and `double` include the values

`POSITIVE_INFINITY`
`NEGATIVE_INFINITY`
`NaN` ("not a number")

These values are defined as constants in the corresponding wrapper classes `Float` and `Double`, respectively. If such a value occurs in the course of some computation (e. g., `POSITIVE_INFINITY` as the result of dividing by zero),[4] Java continues without raising an error.

B.2 Arrays and Collections

B.2.1 Creating Arrays

Unlike in most traditional programming languages (such as FORTRAN or C), arrays in Java can be created *dynamically*, meaning that the size of an array can be specified at runtime using the value of some variable or arithmetic expression. For example:

[3] The function `atan2(dy,dx)` is available in most current programming languages, including Java, C, and C++.

[4] In Java, this only holds for floating-point operations. Integer division by zero still causes an *exception*.

```
int N = 20;
int[] A = new int[N];
int[] B = new int[N*N];
```

Once allocated, however, the size of any Java array is fixed and cannot be subsequently altered. For additional variability, Java provides a number of universal container classes (e. g., the class `Vector`) for a wide range of applications.

After its definition, an array variable can be assigned any other compatible array or the constant value `null`; e. g.,

```
A = B;     // A now points to B's data
B = null;
```

Through the assignment `A = B` above, the array initially referenced by `A` becomes unaccessible and thus turns into *garbage*. In contrast to C and C++, where unnecessary storage needs to be *de*allocated explicitly, this is taken care of in Java by its built-in "garbage collector". It is also convenient that newly created arrays of numerical element types (`int`, `float`, `double`, etc.) are automatically initialized to zero.

B.2.2 Array Size

Since an array may be created dynamically, it is important that its actual size can be determined at runtime. This is done by accessing the `length` attribute[5] of the array:

```
int k = A.length; // number of elements in A
```

It may be surprising that Java arrays may have *zero* (not `null`) elements! If an array has more than one dimension, the size (`length`) along every dimension must be derived separately. The size is a property of the array itself and can therefore be obtained inside any method from array arguments passed to it. Thus (unlike in C, for example) it is not necessary to pass the size of an array as a separate function argument.

B.2.3 Accessing Array Elements

In Java, the index of the first array element is always 0 and the index of the last element is $N-1$ for an array with a total of N elements. To iterate through a one-dimensional array `A` of arbitrary size, one would typically use a construct like

```
for (int i = 0; i < A.length; i++) {
    // do something with A[i]
}
```

[5] Notice that the `length` attribute of an array is not a method!

Since images in Java and ImageJ are stored as one-dimensional arrays (accessible through the `ImageProcessor` method `getPixels()`), most point operations can be efficiently implemented in this way.[6]

B.2.4 Two-Dimensional Arrays

Multidimensional arrays are a common cause of misunderstanding. In Java, all arrays are one-dimensional, and multidimensional arrays are implemented as one-dimensional arrays of subarrays (Fig. B.1). If, for example, the 3×3 matrix

$$\boldsymbol{A} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} \\ A_{1,0} & A_{1,1} & A_{1,2} \\ A_{2,0} & A_{2,1} & A_{2,2} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \tag{B.3}$$

with elements a_{ij} (i being the *row* and j being the *column* index) is represented as a two-dimensional floating-point array,

```
double[][] A = {{1,2,3},
                {4,5,6},
                {7,8,9}};
```

then `A` is really a *one*-dimensional array containing three items, each of which is again a one-dimensional array of type `double` (see Fig. B.1).

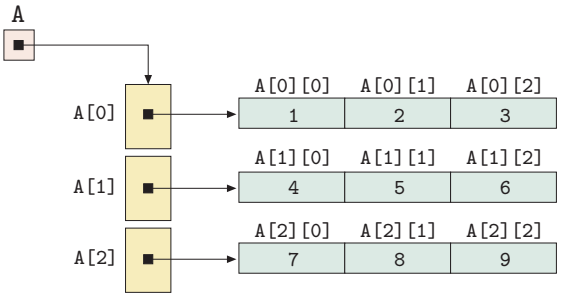

Figure B.1 Multidimensional arrays are implemented in Java as *one*-dimensional arrays whose elements are again one-dimensional arrays.

The usual assumption is that the array elements are arranged in *row-first* ordering, as illustrated in Fig. B.1. The first index thus corresponds to the row number *row* and the second index corresponds to the column number *col*,

$$a_{row,col} \equiv \texttt{A[row][col]} \qquad \text{or} \qquad a_{i,j} \equiv \texttt{A[i][j]}.$$

[6] See Prog. 7.1 in Sec. 7.6 of the ImageJ Short Reference [5] for an example.

So here the first array index runs downwards in the matrix and the second index runs to the right. This is quite convenient, because the array initialization in the code segment above looks exactly the same as the original matrix in Eqn. (B.3).

However, if the matrix represents an *image* or *filter kernel*, we usually associate the row index with the *vertical* coordinate v (or j) and the column index with the *horizontal* coordinate u (or i)—so the ordering of indices is reversed! For example, if we represent the filter kernel

$$H(i,j) = \begin{bmatrix} H(0,0) & H(1,0) & H(2,0) \\ H(0,1) & H(1,1) & H(2,1) \\ H(0,2) & H(1,2) & H(2,2) \end{bmatrix} = \begin{bmatrix} -1 & -2 & 0 \\ -2 & 0 & 2 \\ 0 & 2 & 1 \end{bmatrix}$$

(with i, j denoting the horizontal and vertical coordinate, respectively) as a two-dimensional Java array,

```
double[][] H = {{-1,-2, 0},
                {-2, 0, 2},
                { 0, 2, 1}};
```

then the indices must be *reversed* in order to access the right elements. In this particular case,

$$H(i,j) \;\equiv\; \texttt{H[j][i]}.$$

This scheme was used, for example, for implementing the 3×3 filter plugin in Prog. 5.2 (p. 105).

Size of Multi-Dimensional Arrays

The size of a multidimensional array can be obtained by querying the size of its subarrays. For example, given the following three-dimensional array with dimensions $P \times Q \times R$,

```
int[][][] B = new int[P][Q][R];
```

the size of `B` along its three dimensions is obtained by the statements

```
int p = B.length;          // = P
int q = B[0].length;       // = Q
int r = B[0][0].length;    // = R
```

At least this works for "rectangular" Java arrays, i. e., multidimensional arrays with all subarrays at the same level having *identical* length. If this is not the case, the length of each (one-dimensional) subarray must be determined individually to avoid "index-out-of-bounds" errors. Thus a "bullet-proof" iteration over all elements of a three-dimensional—potentially "non-rectangular"—array `C` could be implemented as follows:

```
1 import java.lang.reflect.Array;
2
3 public static Object duplicateArray(Object orig) {
4   Class origClass = orig.getClass();
5   if (!origClass.isArray())
6     return null; // no array to duplicate
7   Class compType = origClass.getComponentType();
8   int n = Array.getLength(orig);
9   Object dup = Array.newInstance(compType, n);
10  if (compType.isArray()) // array elements are arrays again:
11    for (int i = 0; i < n; i++)
12      Array.set(dup, i, duplicateArray(Array.get(orig, i)));
13  else // array elements are objects or primitives:
14    System.arraycopy(orig, 0, dup, 0, n);
15  return dup;
16 }
```

Program B.1 Utility method `duplicateArray()` for cloning arrays of any element type and dimensionality. Objects inside the array are not duplicated.

```
for (int i = 0; i < C.length; i++) {
  for (int j = 0; j < C[i].length; j++) {
    for (int k = 0; k < C[i][j].length; k++) {
      // do something with C[i][j][k]
    }
  }
}
```

B.2.5 Cloning Arrays

Java arrays implement the standard `java.lang.Cloneable` interface and provide `clone()` methods to perform a single-level ("shallow") form of duplication; i. e., to make a copy of the top-level structure of the array. Applied to a one-dimensional array of primitive element type, e. g.,

```
int[] A1 = {1,2,3,4};
int[] A2 = (int[]) A1.clone();
```

the result `A2` is an exact and independent copy of the array `A1`, as one would expect. If the original array contains real (i. e., nonprimitive) Java *objects*, `clone()` does *not* duplicate the individual objects themselves, but the cells of both arrays refer to the same original objects.

Similarly, applying `clone()` to a two-dimensional (or multidimensional) array duplicates only the top-level structure of that array but none of its subarrays. Java has no standard method for doing a *full-depth* duplication of multidimensional arrays. The (nontrivial) method `duplicateArray()` in Prog. B.1 shows how this could be accomplished recursively for arrays of any element type and dimensionality.

B.2.6 Arrays of Objects, Sorting

In Java, as mentioned earlier, we can create arrays dynamically; i. e., the size of an array can be specified during execution. This is convenient because we can adapt the size of the arrays to the actual problem. For example, we could write

```
Corner[] cornerArray = new Corner[n];
```

to create an array that can hold `n` objects of type `Corner` (as defined in Vol. 2 [6, Sec. 4.3]). But be aware that the new array is not filled with corners yet but initialized with `null` (i. e., empty references), so the array is really empty. We can insert a `Corner` object into its first (or any other) cell by

```
cornerArray[0] = new Corner(10,20,6789.0f);
```

Arrays can be sorted quickly using the static utility methods in the `java.util.Arrays` class,

```
Arrays.sort(type[] arr)
```

where `arr` can be any array of primitive *`type`* (`int`, `float`, etc.) or an array of objects. In the latter case, the array may not have `null` entries. Also, the class of every contained object must implement the `Comparable` interface, i. e., provide a public method

```
int compareTo(Object obj)
```

that must return an `int` value of -1, 0, or 1, depending upon the intended order relation to the other object `obj`. For example, within the `Corner` class, the `compareTo()` method could be defined as follows:

```
public int compareTo (Object obj){   // in class Corner
  Corner c2 = (Corner) obj;
  if (this.q > c2.q) return -1;
  if (this.q < c2.q) return 1;
  else return 0;
}
```

which implicitly assumes that objects of class `Corner` need never be compared with any other type of object.[7]

In summary, arrays are highly efficient data structures that allow fast searching and sorting and therefore should be used whenever fixed size is not a problem.

[7] Note that the typecast `(Corner)obj` (line 2 in method `compareTo`) is potentially dangerous and will create a runtime exception if `obj` is not of type `Corner`.

B.2.7 Collections

Once created, arrays in Java are of fixed size and cannot be expanded or shrunk. To use an array for collecting the corners detected in an image may thus not be a good idea because we do not know a priori how many corners the image contains. If we make the initial array too small, we will run out of space during the process. If we make the array as large as possibly needed, we will probably waste a lot of memory most of the time.

When we try to extract entities (e. g., corner points) from images, we do not know in advance how many of them we are going to find. Also, the properties of these items of interest may vary. This is a frequent situation, and while most simple processes in digital imaging are done with fixed-sized arrays of numbers, dynamic data structures are often needed for advanced tasks. Incidentally, this is also one of Java's strongest aspects. In fact, Java provides a complete collection framework with several convenient data structures that would be complicated to implement by oneself.

A "collection" represents a group of objects, known as its elements. So arrays, which we have been using over and over again, are of course collections. The Java collections framework is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation. It reduces programming effort while delivering high performance. It allows for interoperability among unrelated APIs, reduces effort in designing and learning new APIs, and fosters software reuse. The framework is based on six collection interfaces. It includes implementations of these interfaces and algorithms to manipulate them. Some types of collections allow duplicate elements and others do not, and some collections are ordered and others unordered.

The Java SDK does not provide any *direct* implementations of this interface but implements more specific subinterfaces such as `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired. Concrete implementations of the `Collection` interface include the classes `Vector` and `ArrayList`, as well as `HashSet` for the convenient construction of hash tables.

Additional details and application examples can be found in the Java SDK documentation[8] and the Java Collections tutorial.[9] For general hints on effective programming in Java, the classic book by Bloch [4] is a particularly valuable source.

[8] http://java.sun.com/javase/reference/
[9] http://java.sun.com/docs/books/tutorial/collections/

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. "The Design and Analysis of Computer Algorithms". Addison-Wesley, Reading, MA (1974).

[2] K. Arnold, J. Gosling, and D. Holmes. "The Java Programming Language". Addison-Wesley, Reading, MA, fourth ed. (2005).

[3] W. Bailer. "Writing ImageJ Plugins—A Tutorial" (2003). http://www.imagingbook.com.

[4] J. Bloch. "Effective Java Programming Language Guide". Addison-Wesley, Reading, MA (2001).

[5] W. Burger and M. J. Burge. "ImageJ Short Reference for Java Developers" (2008). http://www.imagingbook.com.

[6] W. Burger and M. J. Burge. "Principles of Image Processing—Core Algorithms". Springer, New York (2009).

[7] P. J. Burt and E. H. Adelson. The Laplacian pyramid as a compact image code. *IEEE Transactions on Communications* **31**(4), 532–540 (1983).

[8] J. F. Canny. A computational approach to edge detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence* **8**(6), 679–698 (1986).

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. "Introduction to Algorithms". MIT Press, Cambridge, MA, second ed. (2001).

[10] L. S. Davis. A survey of edge detection techniques. *Computer Graphics and Image Processing* **4**, 248–270 (1975).

[11] B. Eckel. "Thinking in Java". Prentice Hall, Englewood Cliffs, NJ, fourth ed. (2006). Earlier versions available online.

[12] N. Efford. "Digital Image Processing—A Practical Introduction Using Java". Pearson Education, Upper Saddle River, NJ (2000).

[13] D. Flanagan. "Java in a Nutshell". O'Reilly, Sebastopol, CA, fifth ed. (2005).

[14] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. "Computer Graphics: Principles and Practice". Addison-Wesley, Reading, MA, second ed. (1996).

[15] A. Ford and A. Roberts. "Colour Space Conversions" (1998). http://www.poynton.com/PDFs/coloureq.pdf.

[16] A. S. Glassner. "Principles of Digital Image Synthesis". Morgan Kaufmann Publishers, San Francisco (1995).

[17] R. C. Gonzalez and R. E. Woods. "Digital Image Processing". Addison-Wesley, Reading, MA (1992).

[18] R. L. Graham, D. E. Knuth, and O. Patashnik. "Concrete Mathematics: A Foundation for Computer Science". Addison-Wesley, Reading, MA, second ed. (1994).

[19] R. W. G. Hunt. "The Reproduction of Colour". Wiley, New York, sixth ed. (2004).

[20] International Telecommunications Union, ITU, Geneva. "ITU-R Recommendation BT.709-3: Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange" (1998).

[21] International Telecommunications Union, ITU, Geneva. "ITU-R Recommendation BT.601-5: Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios" (1999).

[22] K. Jack. "Video Demystified—A Handbook for the Digital Engineer". LLH Publishing, Eagle Rock, VA, third ed. (2001).

[23] B. Jähne. "Practical Handbook on Image Processing for Scientific Applications". CRC Press, Boca Raton, FL (1997).

[24] B. Jähne. "Digitale Bildverarbeitung". Springer-Verlag, Berlin, fifth ed. (2002).

[25] A. K. Jain. "Fundamentals of Digital Image Processing". Prentice Hall, Englewood Cliffs, NJ (1989).

[26] J. King. Engineering color at Adobe. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 15, pp. 341–369. Wiley, New York (2002).

[27] R. A. Kirsch. Computer determination of the constituent structure of biological images. *Computers in Biomedical Research* **4**, 315–328 (1971).

[28] T. Lindeberg. Feature detection with automatic scale selection. *International Journal of Computer Vision* **30**(2), 77–116 (1998).

[29] D. Marr and E. Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London, Series B* **207**, 187–217 (1980).

[30] J. Miano. "Compressed Image File Formats". ACM Press, Addison-Wesley, Reading, MA (1999).

[31] P. A. Mlsna and J. J. Rodriguez. Gradient and laplacian-type edge detection. In A. Bovik, editor, "Handbook of Image and Video Processing", pp. 415–431. Academic Press, New York (2000).

[32] J. D. Murray and W. VanRyper. "Encyclopedia of Graphics File Formats". O'Reilly, Sebastopol, CA, second ed. (1996).

[33] T. Pavlidis. "Algorithms for Graphics and Image Processing". Computer Science Press / Springer-Verlag, New York (1982).

[34] W. S. Rasband. "ImageJ". U.S. National Institutes of Health, MD (1997–2007). http://rsb.info.nih.gov/ij/.

[35] I. E. G. Richardson. "H.264 and MPEG-4 Video Compression". Wiley, New York (2003).

[36] L. G. Roberts. Machine perception of three-dimensional solids. In J. T. Tippet, editor, "Optical and Electro-Optical Information Processing", pp. 159–197. MIT Press, Cambridge, MA (1965).

[37] J. C. Russ. "The Image Processing Handbook". CRC Press, Boca Raton, FL, third ed. (1998).

[38] Y. Schwarzer, editor. "Die Farbenlehre Goethes". Westerweide Verlag, Witten (2004).

[39] N. Silvestrini and E. P. Fischer. "Farbsysteme in Kunst und Wissenschaft". DuMont, Cologne (1998).

[40] M. Stokes and M. Anderson. "A Standard Default Color Space for the Internet—sRGB". Hewlett-Packard, Microsoft, www.w3.org/Graphics/Color/sRGB.html (1996).

[41] A. Watt. "3D Computer Graphics". Addison-Wesley, Reading, MA, third ed. (1999).

[42] A. Watt and F. Policarpo. "The Computer Image". Addison-Wesley, Reading, MA (1999).

[43] G. Wolberg. "Digital Image Warping". IEEE Computer Society Press, Los Alamitos, CA (1990).

[44] T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Communications of the ACM* **27**(3), 236–239 (1984).

Index

C

G

H

I

N

O

P

Q

R

S

T

U

V

W

About the Authors

Wilhelm Burger received a Master's degree in Computer Science from the University of Utah (Salt Lake City) and a doctorate in Systems Science from Johannes Kepler University in Linz, Austria. As a post-graduate researcher at the Honeywell Systems & Research Center in Minneapolis and the University of California at Riverside, he worked mainly in the areas of visual motion analysis and autonomous navigation. In the Austrian research initiative on digital imaging, he was engaged in projects on generic object recognition and biometric identification. Since 1996, he has been the director of the Digital Media degree programs at the Upper Austria University of Applied Sciences at Hagenberg. Personally the author appreciates large-engine vehicles and (occasionally) a glass of dry "Veltliner".

Mark J. Burge received a BA degree from Ohio Wesleyan University, a MSc in Computer Science from the Ohio State University, and a doctorate from Johannes Kepler University in Linz, Austria. He spent several years as a researcher in Zürich, Switzerland at the Swiss Federal Institute of Technology (ETH), where he worked in computer vision and pattern recognition. As a post-graduate researcher at the Ohio State University, he was involved in the "Image Understanding and Interpretation Project" sponsored by the NASA Commercial Space Center. He earned tenure within the University System of Georgia as an associate professor in computer science and served as a Program Director at the National Science Foundation. Currently he is a Principal at Noblis (Mitretek) in Washington D.C. Personally, he is an expert on classic Italian espresso machines.

About this Book Series

The complete manuscript for this book was prepared by the authors "camera-ready" in LaTeX using Donald Knuth's Computer Modern fonts. The additional packages algorithmicx (by Szász János) for presenting algorithms, listings (by Carsten Heinz) for listing progam code, and psfrag (by Michael C. Grant and David Carlisle) for replacing text in graphics were particularly helpful in this task. Most illustrations were produced with Macromedia Freehand (now part of Adobe), function plots with Mathematica, and images with ImageJ or Adobe Photoshop. All book figures, test images in color and full resolution, as well as the Java source code for all examples are available at the book's support site: www.imagingbook.com.