Algorithms I

Euclid, 300 BC

Agenda

Algorithm analysis

- The algorithm concept
- Estimation of running times
- Big-Oh notation
- Binary search

The Collections API

- Common data structures
- Applications of the data structures
- Organization of the Collections API

Data and information


"How do you want that data?"

Data:
A formalized representation of facts or concepts suitable for communication, interpretation, or processing by people or automated means.

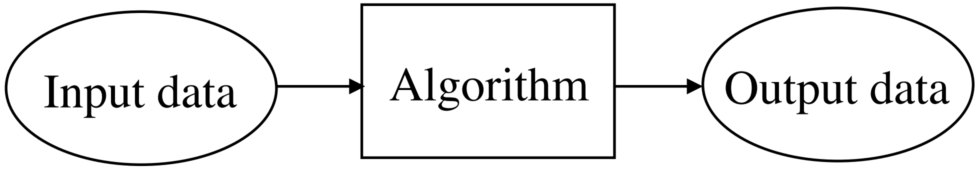
Data on its own carries no meaning.

Information:
The meaning that a human assigns to data by means of known conventions.

What is an algorithm?

An **algorithm** is a step-by-step procedure for solving a problem

Note that it is not a requirement that an algorithm is executed by a computer.


Input data
Algorithm
Output data

"Begin at the beginning," the King said gravely, "and go on until you come to the end; then stop."
Lewis Carroll 1832-1898

Origin of the algorithm concept

The word *algorithm* comes from the name of the 9th century Persian Muslim mathematician Abu Abdullah Muhammad ibn Musa Al-Khwarizmi.

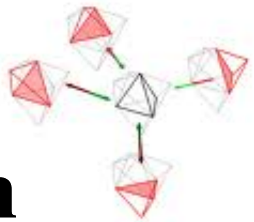
The word *algorism* originally referred only to the rules of performing arithmetic using Hindu-Arabic numerals but evolved via European Latin translation of Al-Khwarizmi’s name into *algorithm* by the 18th century. The use of the word evolved to include all definite procedures for solving problems or performing tasks.

Formal definition of an algorithm

Donald E. Knuth (1968)

An **algorithm** is a *finite, definite,* and *effective procedure*, with some *output*.

- *Finite*:
  there must be an end to it within a reasonable time
- *Definite*:
  precisely definable in clearly understood terms and without ambiguities
- *Effective:*
  it must be possible to actually carry out the steps
- *Procedure*:
  the sequence of specific steps
- *Output*:
  unless there is something coming out of the process, the result will remain unknown!



Desirable properties of an algorithm

(1) It solves the problem **correctly**

(2) It runs (sufficiently) **fast**

(3) It requires (sufficiently) **little storage**

(4) It is **simple**

The last three properties often conflict with each other.

Space-time tradeoff is a way of solving a problem faster by using more storage, or by solving a problem in little storage by spending a long time.

The need for fast algorithms

Why bother about efficiency with today’s fast computers?

Technology increases speed by only a **constant** factor.
Much larger speed-up may often be achieved by careful algorithm design.

A bad algorithm on a supercomputer may run slower than a good one on an abacus.

More powerful computers allow us to solve larger problems, but ...

Suppose an algorithm runs in time proportional to the square of the problem size (time = cn^2, where c is a constant, and n is the problem size). If we buy a new computer that has *10 times as much memory* as the old one, we are able to solve problems that are 10 times larger. However, if the new computer is “only” *10 times faster*, it will take 10 times longer to execute the algorithm.

Euclid's algorithm

One of the first non-trivial algorithms was designed by Euclid (Greek mathematician, 300 BC).

Problem: Find the **largest common divisor** of two positive integers

The largest common divisor of two positive integers is the largest integer that divides both of them without leaving a remainder.

Given	Solution
24 and 32	8
8 and 12	4
7 and 8	1

If gcd(u,v) denotes the greatest common divisor of u and v, the problem may be formulated as follows:

Given two integers $u \geq 1$ and $v \geq 1$, find gcd(u,v).

Solution of the problem is for example relevant to the problem of reducing fractions:

$$\frac{24}{32} = \frac{\dfrac{24}{\gcd(24,32)}}{\dfrac{32}{\gcd(24,32)}} = \frac{\dfrac{24}{8}}{\dfrac{32}{8}} = \frac{3}{4}$$

Two simple algorithms

```
for (int d = 1; d <= u; d++)
    if (u % d == 0 && v % d == 0)
        gcd = d;
```

```
int d = u < v ? u : v;
while (u % d != 0 || v % d != 0)
    d--;
gcd = d;
```

The inefficiency of these algorithms is apparent for large values of `u` and `v`, for instance 461952 and 116298 (where gcd is equal to 18).

Euclid's algorithm

Euclid exploited the following observation to achieve a more efficient algorithm:

If $u \geq v$, and d divides both u and v, then d divides the difference between u and v.

If $u > v$, then $\gcd(u,v) = \gcd(u-v,v)$.

If $u = v$, the $\gcd(u,v) = v$ [$= \gcd(0,v)$]

If $u < v$, then we exploit that $\gcd(u,v) = \gcd(v,u)$ [u and v are exchanged]

Euclid’s algorithm (version 1)

```
while (u > 0) {
    if (u < v)
        { int t = u; u = v; v = t; }
    u = u - v;
}
gcd = v;
```

Example run:

```
u = 461952, v = 18
u = 461934, v = 18
u = 461916, v = 18
.
.
.
u = 18    , v = 18
u = 0     , v = 18
```

461952/18 = **25664** iterations

Euclid's algorithm (version 2)

Is it possible to improve the efficiency?

Yes. The algorithm subtracts v from u until u becomes less than v. But this is exactly the same as diving u by v and setting u equal to the remainder. That is, if $u > v$, then $\text{gcd}(u,v) = \text{gcd}(u\%v,v)$.

```
while (u > 0) {
    if (u < v)
      { int t = u; u = v; v = t; }
    u = u % v;
}
gcd = v;
```

The number of iterations for the previous example is reduced to 1.

Execution of version 2

```
u = 461952, v = 116298
u = 113058, v = 116298
u = 3240,   v = 113058
u = 2898,   v = 3240
u = 342,    v = 2898
u = 162,    v = 342
u = 18,     v = 162
u = 0,      v = 18
```

7 iterations

The algorithm is very efficient, even for large values of u and v.
Its efficiency can be determined by (advanced) algorithm analysis:

maximum number of iterations $\approx 4.8 \log_{10} N - 0.32$
average number of iterations $\approx 1.94 \log_{10} N$

where N is $\max(u, v)$.

[$\log_{10} 461952 \approx 5.66$]

An alternative algorithm

(prime factorization)

A well-known method for reducing fractions:

$$\frac{4400}{7000} = \frac{\cancel{2}\cdot\cancel{2}\cdot\cancel{2}\cdot 2\cdot\cancel{5}\cdot\cancel{5}\cdot 11}{\cancel{2}\cdot\cancel{2}\cdot\cancel{2}\cdot\cancel{5}\cdot\cancel{5}\cdot 5\cdot 7} = \frac{2\cdot 11}{5\cdot 7} = \frac{22}{35}$$

Any positive number u may be expressed as a product of prime factors:

$$u = 2^{u_2}\cdot 3^{u_3}\cdot 5^{u_5}\cdot 7^{u_7}\cdot 11^{u_{11}}\cdot \ldots = \prod_{p \text{ is prime}} p^{u_p}$$

Let u and v be two positive integers. Then gcd(u,v) may be determined as

$$\prod_{p \text{ is prime}} p^{\min(u_p, v_p)}$$

Example: $u = 4400 = 2^4\cdot 3^0\cdot 5^2\cdot 7^0\cdot 11^1$,

$v = 7000 = 2^3\cdot 3^0\cdot 5^3\cdot 7^1\cdot 11^0$

$\gcd(u,v) = 2^3\cdot 3^0\cdot 5^2\cdot 7^0\cdot 11^0 = 2^3\cdot 5^2 = 8\cdot 25 = 200$

Drawback of the alternative algorithm

No efficient algorithm for prime factorization is known.

This fact is exploited in *cryptographic algorithms* (algorithms for information security).

What is algorithm analysis?

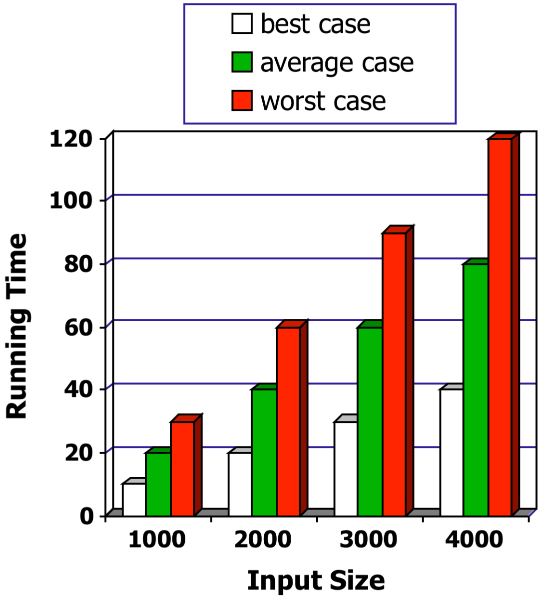
To *analyze an algorithm* is to determine the amount of resources (such as time and storage) necessary to execute it.

Algorithm analysis is a methodology for estimating the resource consumption of an algorithm. It allows us to compare the relative costs of two or more algorithms for solving the same problem.

Algorithm analysis also gives algorithm designers a tool for estimating whether a proposed solution is likely to meet the resource constraints for a problem.

Running time

The running time of an algorithm typically grows with the *input size*.

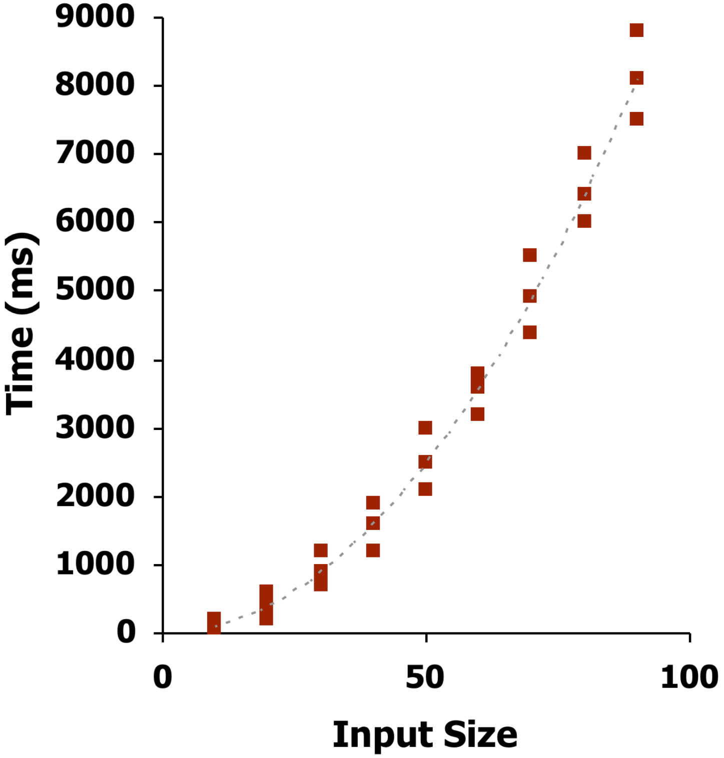


Average case time is often difficult to determine.
We focus on the worst case running time.

- Easier to analyze
- Crucial to applications such as games and robotics

Experimental studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size
- Use a method like `System.currentTimeMillis()` to get an accurate measure of the actual running time
- Plot the results



Limitations of experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of hardware/software environment

Primitive operations

- Basic operations performed by an algorithm
- Each one is assumed to take constant time
- Largely independent from the programming language

Examples:

    Evaluating an expression
    Assigning a value to a variable
    Indexing into an array
    Calling a method
    Returning from a method

Counting primitive operations

By inspecting the code or the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size, *n*.

```
int arrayMax(int[] a) {                          # operations
    int currentMax = a[0];                            2
    for (int i = 1; i < a.length; i++)          1+2(n-1)+n
        if (a[i] > currentMax)                      2(n-1)
            currentMax = a[i];                      2(n-1)
    return currentMax;                                1
}
                                          Total     7n-2
```

Estimating running time for arrayMax

The algorithm *arrayMax* executes $7n\text{-}2$ primitive operations in the **worst case**.

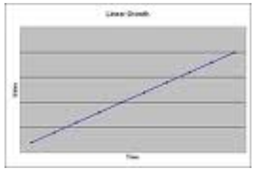
Define:

a = Time taken by the fastest primitive operation
b = Time taken by the slowest primitive operation

Let $T(n)$ be worst-case time of *arrayMax***.** Then

$$a(7n-2) \leq T(n) \leq b(7n-2)$$

Hence, the running time $T(n)$ is bounded by two linear functions. This property, in which running time essentially is directly proportional to the amount of data, is the signature of a *linear* algorithm.



Growth rate of running time

- Changing the hardware/software environment
  - affects $T(n)$ by a constant factor, but
  - does not alter the growth rate of $T(n)$

- The *linear* growth rate of the running time $T(n)$ is an intrinsic property of algorithm *arrayMax.*

Growth rates

A *cubic* function is a function whose dominant term is some constant times N^3. As an example $10N^3+N^2+40N+80$ is a cubic function, since the term $10N^3$ dominates when N is large.

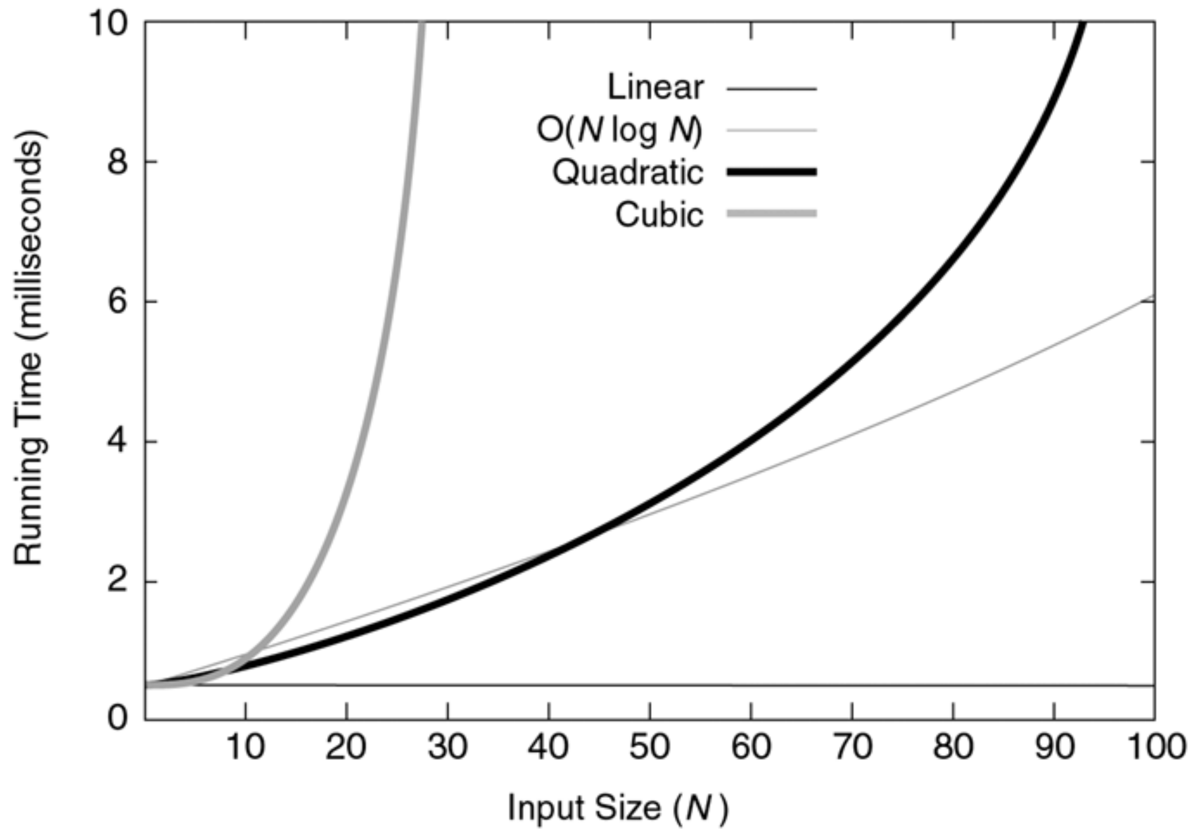
A *quadratic* function is a function whose dominant term is some constant times N^2.

A *linear* function is a function whose dominant term is some constant times N.

The *logarithm* is a slowly growing function. For instance, the logarithm of 1,000,000 (with the typical base 2) is only 20.

figure 5.1

Running times for small inputs



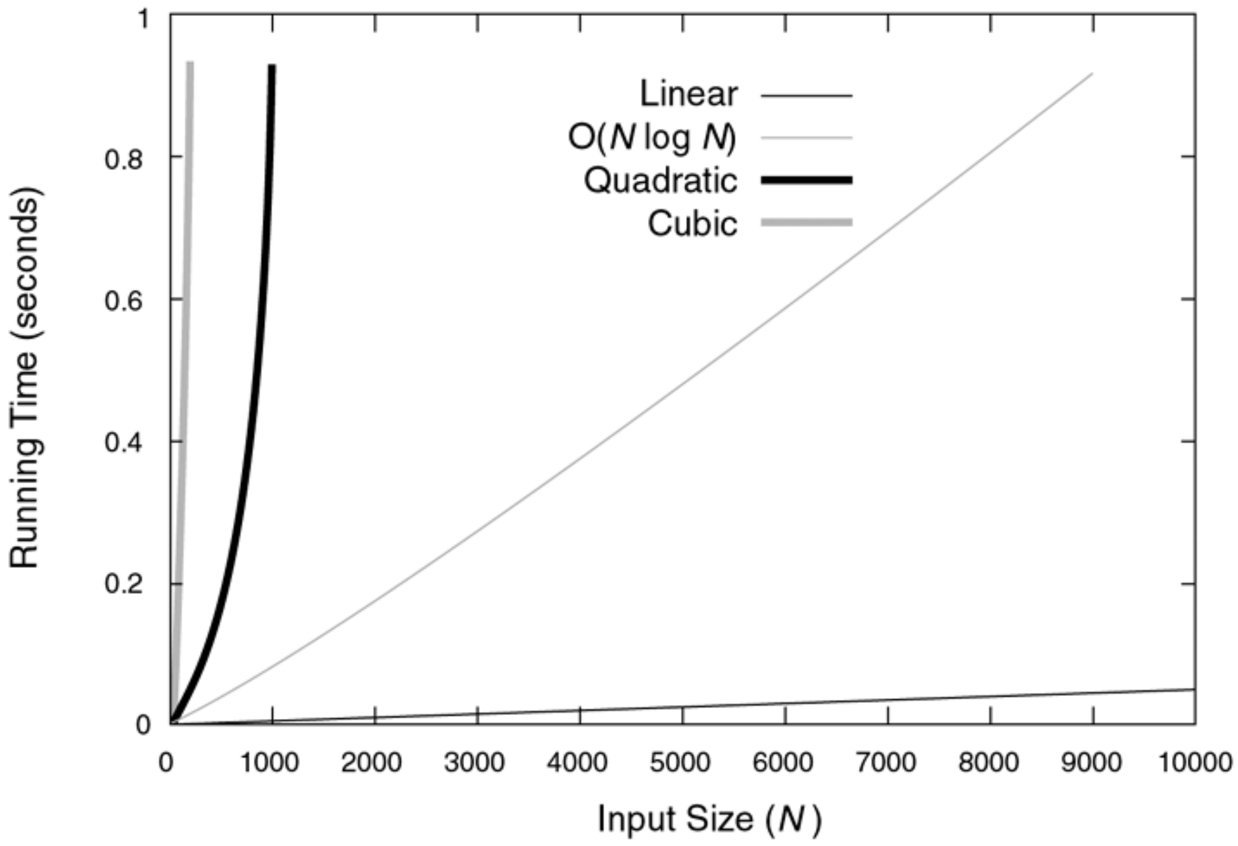
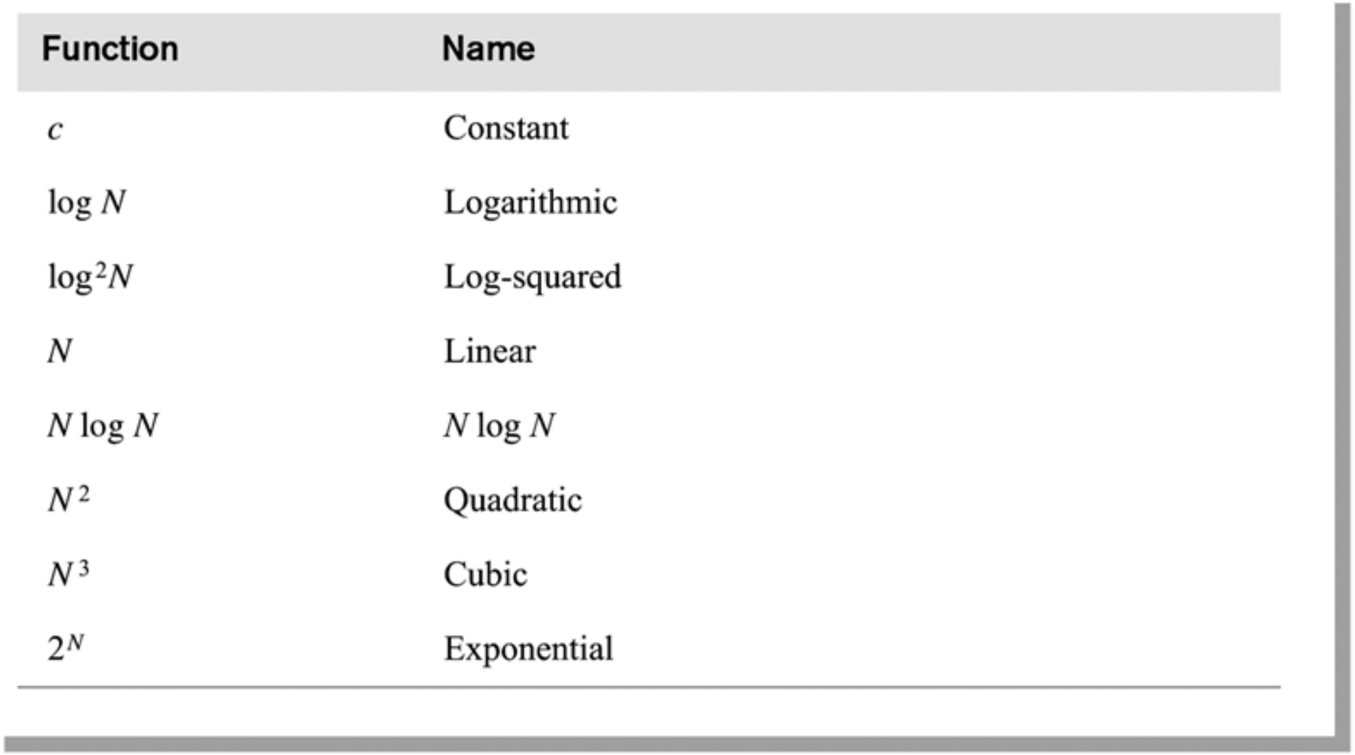


figure 5.2

Running times for moderate inputs



Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	$N \log N$
N^2	Quadratic
N^3	Cubic
2^N	Exponential

figure 5.3

Functions in order of increasing growth rate

$N \log N$ is sometimes called *linearithmic*, *loglinea*r, or *quasilinear.*

Big-Oh notation

We use Big-Oh notation to capture the most dominant term in a function and to represent growth rate.

For instance, the running time of a quadratic algorithm is specified as $O(N^2)$ (pronounced “*order en-squared*”).

Even the most clever programming tricks cannot make an inefficient algorithm fast. Thus, before attempting to optimize code, we need to optimize the algorithm.

Examples of algorithm running times

Minimum element in an array
Given an array of N elements, find the minimum element.

Closest pair of points in the plane
Given N points in the plane (that is, in the x-y coordinate system), find the pair that are closest together.

Colinear points in the plane
Given N points in the plane, determine if any three form a straight line.

Simple algorithms for these problems run in $O(N)$, $\mathrm{O}(N^2)$, and $\mathrm{O}(N^3)$ time, respectively. For the last two problems more efficient algorithms have been developed.

The maximum contiguous subsequence sum problem

Given (possible negative) integers $A_1, A_2, ..., A_N$, find (and identify the sequence corresponding to) the maximum value of

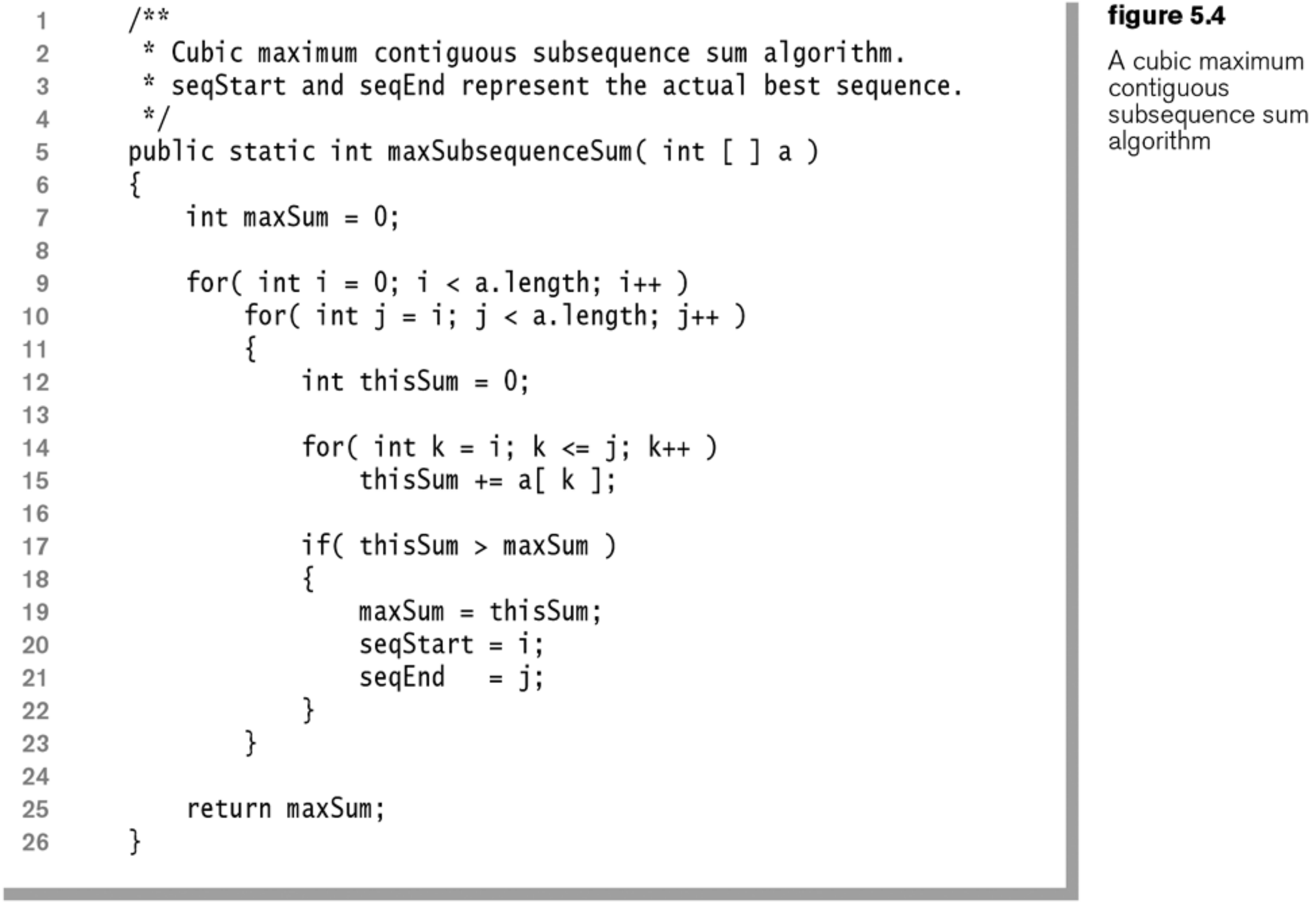
$$\sum_{k=i}^{j} A_k$$

The maximum contiguous subsequence sum is zero if all the integers are negative.

Example. If the input is (-2, 11, -4, -1, 13, -5, 2), then the answer is 19, which represents the sum of the contiguous subsequence (11, -4, -1, 13).

If all elements are positive, then the sequence itself is maximal.

An obvious $O(N^3)$ algorithm



```
/**
 * Cubic maximum contiguous subsequence sum algorithm.
 * seqStart and seqEnd represent the actual best sequence.
 */
public static int maxSubsequenceSum( int [ ] a )
{
    int maxSum = 0;

    for( int i = 0; i < a.length; i++ )
        for( int j = i; j < a.length; j++ )
        {
            int thisSum = 0;

            for( int k = i; k <= j; k++ )
                thisSum += a[ k ];

            if( thisSum > maxSum )
            {
                maxSum = thisSum;
                seqStart = i;
                seqEnd   = j;
            }
        }

    return maxSum;
}
```

figure 5.4

A cubic maximum contiguous subsequence sum algorithm

Analysis of running time

Running time of the algorithm is entirely dominated by the innermost loop (lines 14 and 15).

The number of times line 15 is executed is exactly equal to the number of triplets (i, j, k) that satisfy $1 \le i \le k \le j \le N$, which is $N(N+1)(N+2)/6$.

Consequently, the algorithm runs in $O(N^3)$ time.

Note that three consecutive (nonnested) loops exhibit linear behavior; it is nesting that leads to a combinatoric explosion. Consequently, to improve the algorithm we need to remove a loop.

An improved $O(N^2)$ algorithm

Suppose we have just calculated the sum for the subsequence $A_i, .., A_{j-1}$. Then computing the sum for the subsequence $A_i, .., A_j$ should not take long because we need only one more addition (of A_j). However, the cubic algorithm throws away this information.

Using this observation, we obtain the following algorithm.

An improved $O(N^2)$ algorithm

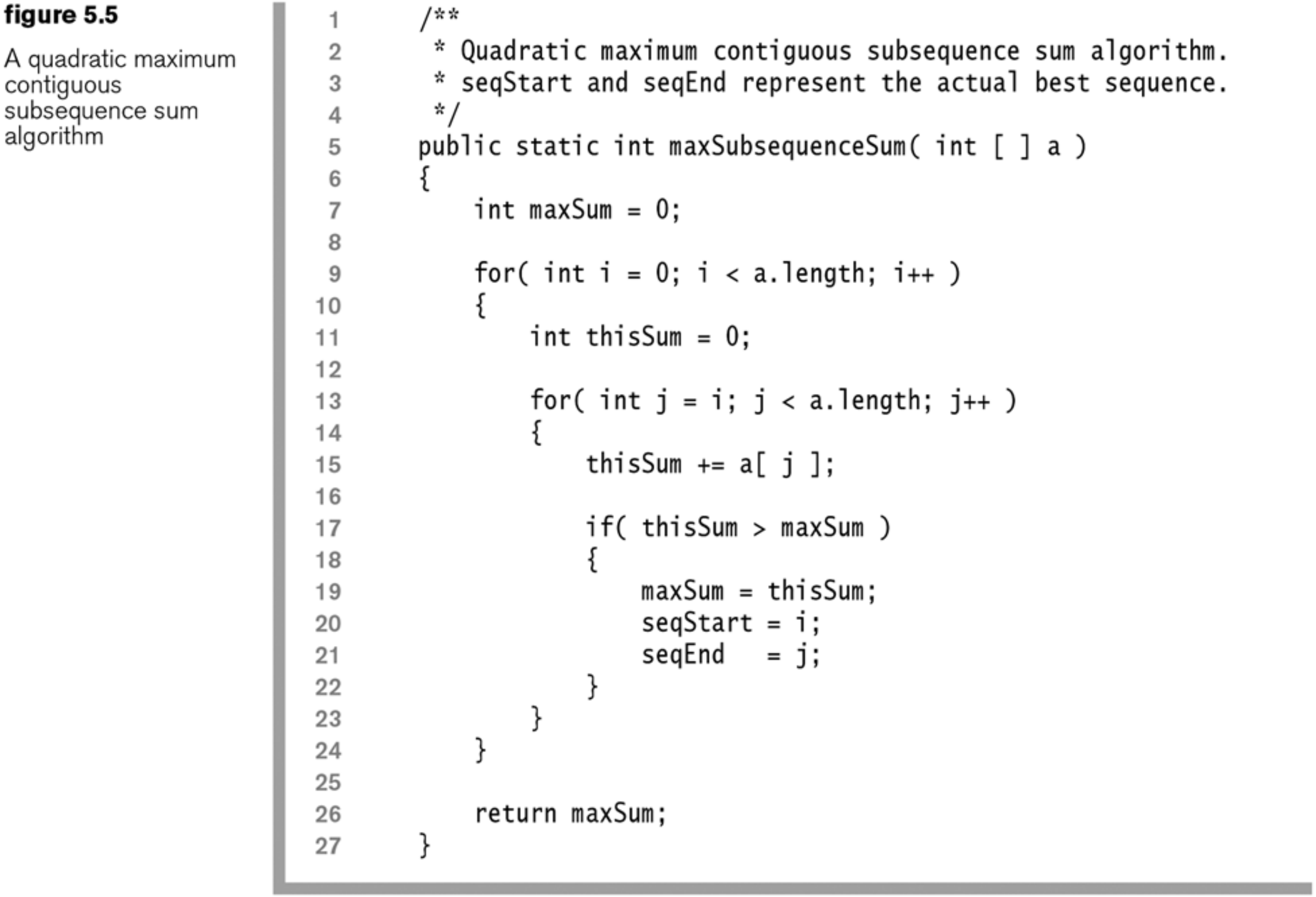


figure 5.5

A quadratic maximum contiguous subsequence sum algorithm

```java
/**
 * Quadratic maximum contiguous subsequence sum algorithm.
 * seqStart and seqEnd represent the actual best sequence.
 */
public static int maxSubsequenceSum( int [ ] a )
{
    int maxSum = 0;

    for( int i = 0; i < a.length; i++ )
    {
        int thisSum = 0;

        for( int j = i; j < a.length; j++ )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
            {
                maxSum = thisSum;
                seqStart = i;
                seqEnd   = j;
            }
        }
    }

    return maxSum;
}
```

Analysis of running time

Running time of the algorithm is entirely dominated by the statement block in the innermost loop (lines 14-23).

The number of times this block is executed is $N + (N-1) + (N-2) + ... + 2 + 1 = N(N+1)/2$.

Consequently, the algorithm runs in $O(N^2)$ time.

To move from a quadratic algorithm to a linear algorithm we need to remove another loop.

A linear algorithm

The previous algorithms are *exhaustive*, i.e., they examine all subsequences. The only way we can attain a subquadratic bound is to find a clever way to eliminate from consideration a large number of subsequences without actually computing their sum and testing to see if that sum is a new maximum.

We may use the following two observations:

(1) The best subsequence cannot start with a negative number.
(2) More generally, the best subsequence cannot start with a negative sum (Theorem 5.2).

Theorem 5.2

Let $A_{i,j}$ be the subsequence encompassing elements from i to j, and let $S_{i,j}$ be its sum.

Theorem 5.2 Let $A_{i,j}$ be any subsequence with $S_{i,j} < 0$. If $q > j$, then $A_{i,q}$ is not a maximum contiguous subsequence.

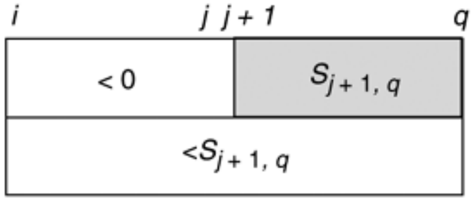


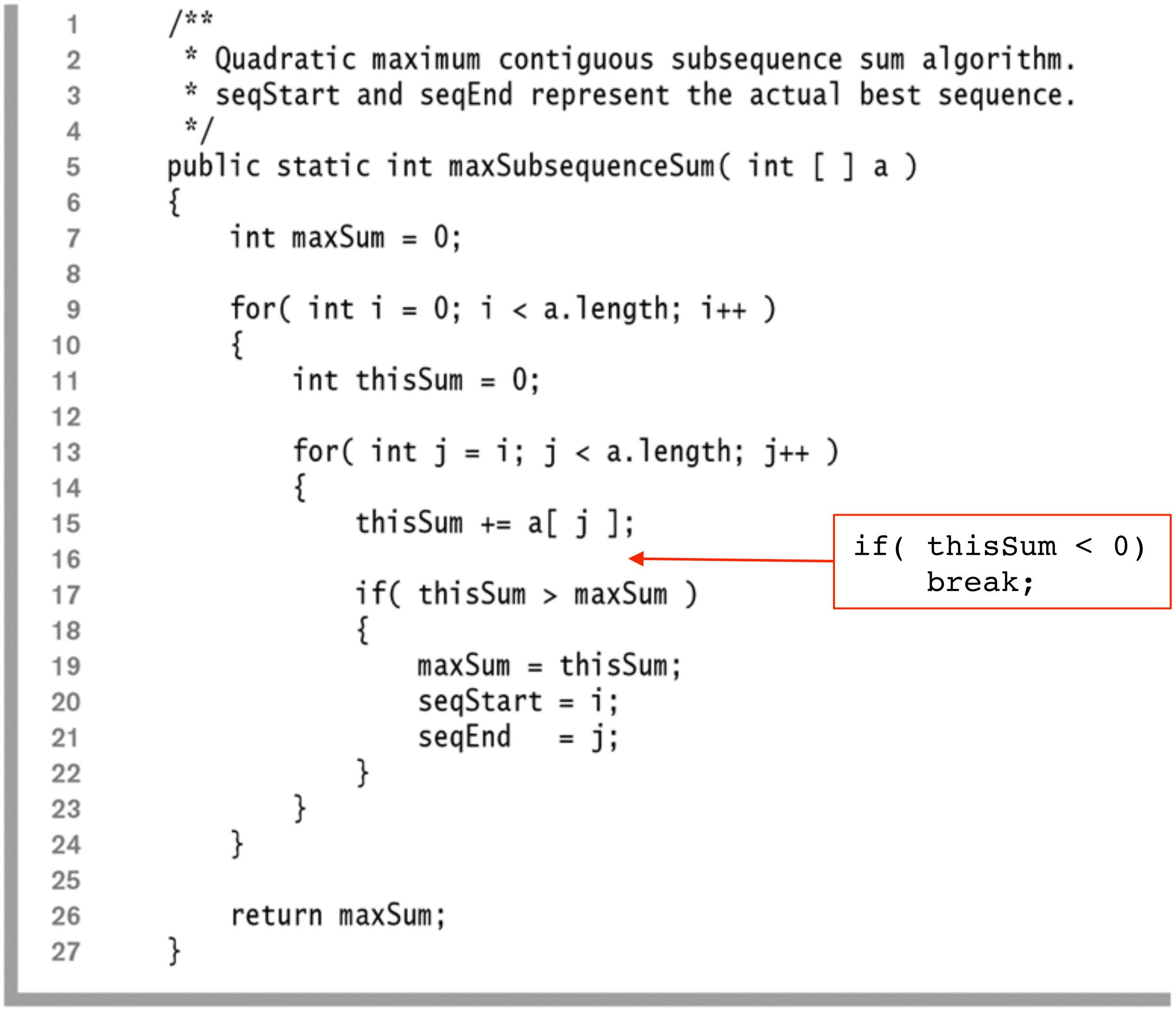
figure 5.6

The subsequences used in Theorem 5.2

This observation by itself is not sufficient to reduce the running time below quadratic.

A quadratic maximum contiguous subsequence sum algorithm

(improved using
Theorem 5.2)

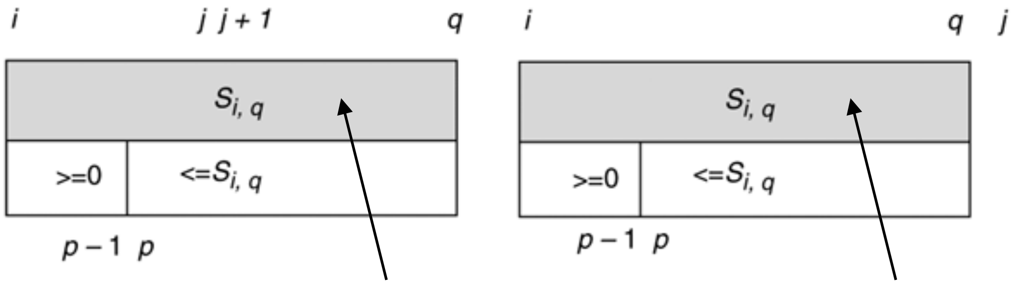


```java
/**
 * Quadratic maximum contiguous subsequence sum algorithm.
 * seqStart and seqEnd represent the actual best sequence.
 */
public static int maxSubsequenceSum( int [ ] a )
{
    int maxSum = 0;

    for( int i = 0; i < a.length; i++ )
    {
        int thisSum = 0;

        for( int j = i; j < a.length; j++ )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
            {
                maxSum = thisSum;
                seqStart = i;
                seqEnd   = j;
            }
        }
    }

    return maxSum;
}
```

```java
if( thisSum < 0)
    break;
```

Theorem 5.3

Theorem 5.3 For any i, let $A_{i,j}$ be the first sequence with $S_{i,j} < 0$. Then for any $i \le p \le j$ and $p \le q$, $\mathrm{A}_{p,q}$ either is not a maximum contiguous subsequence or is equal to an already seen maximum contiguous subsequence.



Not maximum (by Theorem 5.2)

Already seen

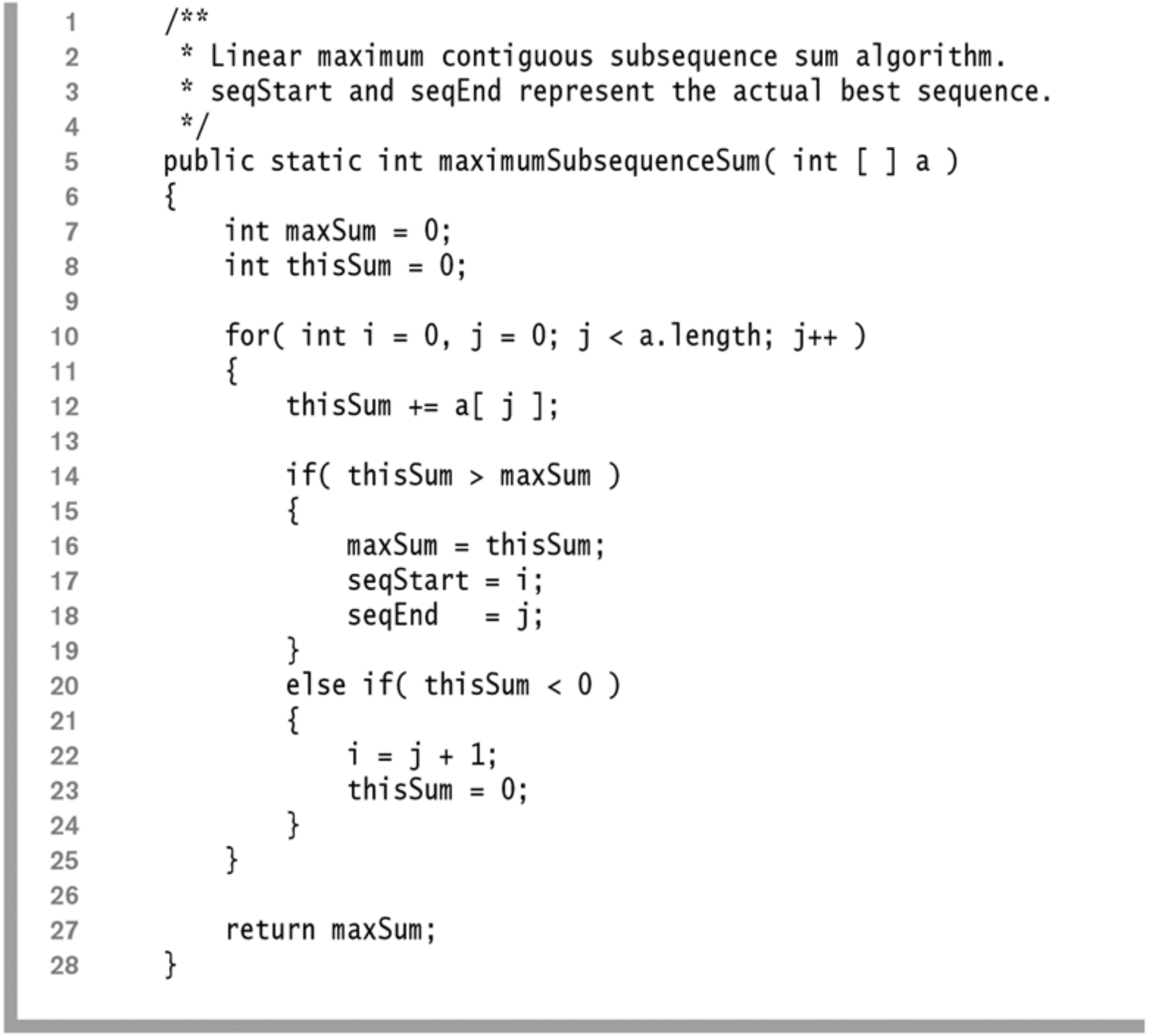
figure 5.7

The subsequences used in Theorem 5.3. The sequence from p to q has a sum that is, at most, that of the subsequence from i to q. On the left-hand side, the sequence from i to q is itself not the maximum (by Theorem 5.2). On the right-hand side, the sequence from i to q has already been seen.

Theorem 5.3 tells us that when a negative sum is detected, not only can we `break` the loop, but we can also advance `i` to `j+1`.

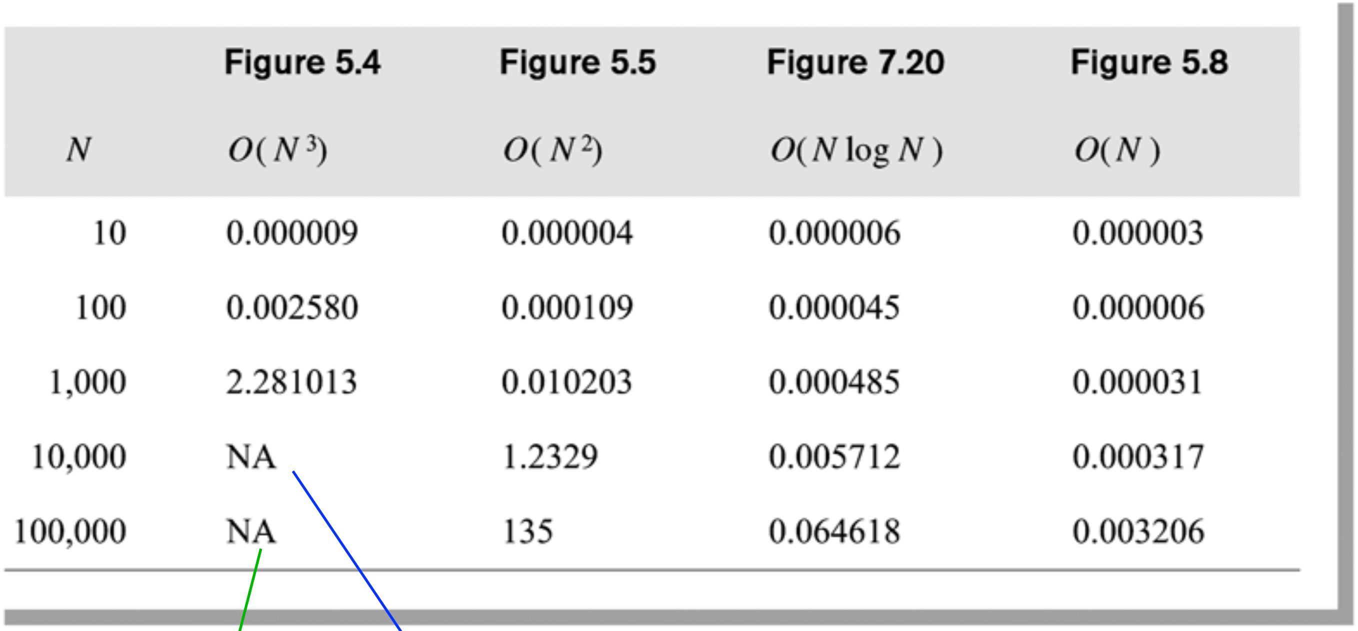
figure 5.8

A linear maximum contiguous subsequence sum algorithm



```
/**
 * Linear maximum contiguous subsequence sum algorithm.
 * seqStart and seqEnd represent the actual best sequence.
 */
public static int maximumSubsequenceSum( int [ ] a )
{
    int maxSum = 0;
    int thisSum = 0;

    for( int i = 0, j = 0; j < a.length; j++ )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
        {
            maxSum = thisSum;
            seqStart = i;
            seqEnd   = j;
        }
        else if( thisSum < 0 )
        {
            i = j + 1;
            thisSum = 0;
        }
    }

    return maxSum;
}
```

Empirical study



N	Figure 5.4 $O(N^3)$	Figure 5.5 $O(N^2)$	Figure 7.20 $O(N \log N)$	Figure 5.8 $O(N)$
10	0.000009	0.000004	0.000006	0.000003
100	0.002580	0.000109	0.000045	0.000006
1,000	2.281013	0.010203	0.000485	0.000031
10,000	NA	1.2329	0.005712	0.000317
100,000	NA	135	0.064618	0.003206

figure 5.10

Observed running times (in seconds) for various maximum contiguous subsequence sum algorithms

≈ 2281 seconds ≈ 38 minutes

≈ 2281000 seconds ≈ 26 days

Big-Oh and its relatives

Definition: (Big-Oh) $T(N)$ is $\boldsymbol{O}(F(N))$ if there are positive constants c and N_0 such that $T(N) \leq cF(N))$, when $N \geq N_0$.

$\leq$

Definition: (Big-Omega) $T(N)$ is $\boldsymbol{\Omega}(F(N))$ if there are positive constants c and N_0 such that $T(N) \geq cF(N))$, when $N \geq N_0$.

$\geq$

Definition: (Big-Theta) $T(N)$ is $\boldsymbol{\Theta}(F(N))$ if and only if $T(N)$ is $O(F(N))$ and $T(N)$ is $\Omega(F(N))$.

$=$

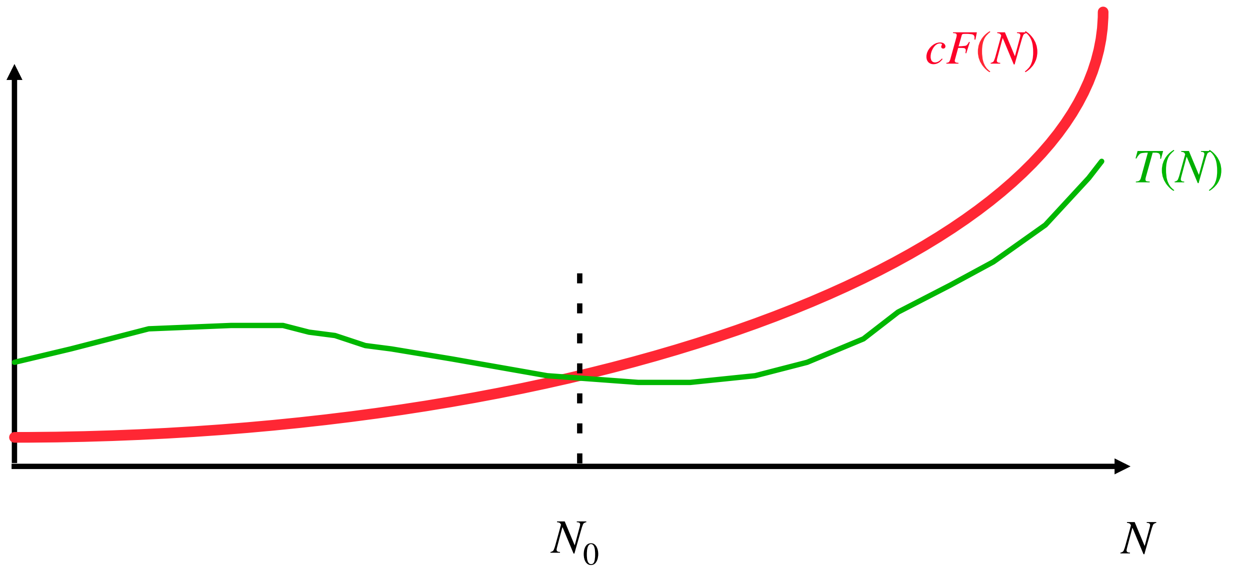
Definition: (Little-Oh) $T(N)$ is $\boldsymbol{o}(F(N))$ if and only if $T(N)$ is $O(F(N))$ and $T(N)$ is not $\Theta(F(N))$.

$<$

Definition: (Little-Omega) $T(N)$ is $\boldsymbol{\omega}(F(N))$ if and only if $T(N)$ is $\Omega(F(N))$ and $T(N)$ is not $\Theta(F(N))$.

$>$

Illustration of the statement "$T(N)$ is $O(F(N))$"



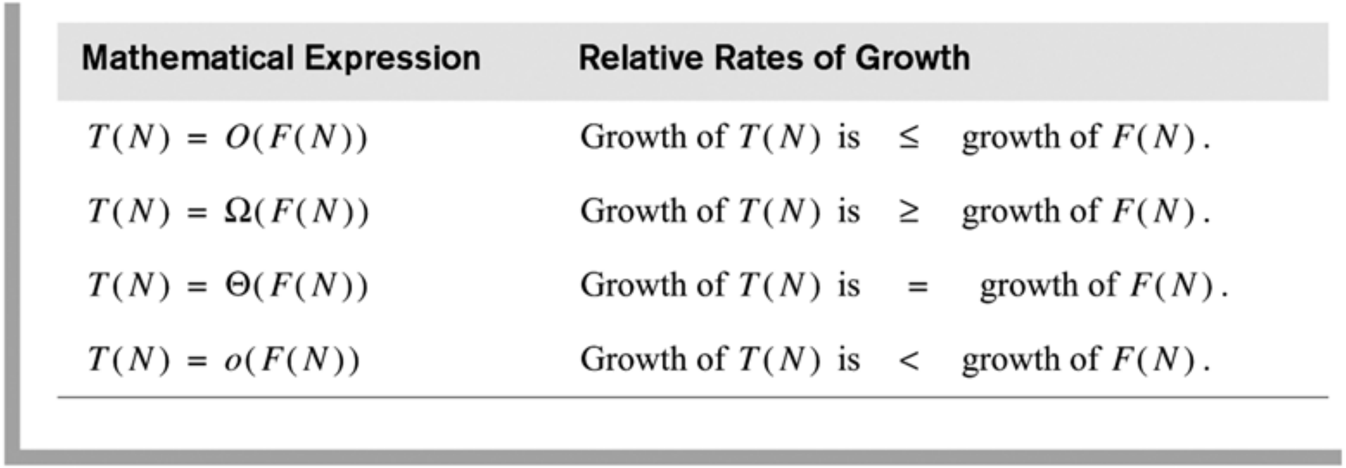


figure 5.9

Meanings of the various growth functions

Mathematical Expression	Relative Rates of Growth
$T(N) = O(F(N))$	Growth of $T(N)$ is $\leq$ growth of $F(N)$.
$T(N) = \Omega(F(N))$	Growth of $T(N)$ is $\geq$ growth of $F(N)$.
$T(N) = \Theta(F(N))$	Growth of $T(N)$ is $=$ growth of $F(N)$.
$T(N) = o(F(N))$	Growth of $T(N)$ is $<$ growth of $F(N)$.

General Big-Oh rules

- If $T(N)$ is a polynomial of degree d, then $T(N)$ is $O(N^d)$, i.e.,
    1. Drop lower-order terms
    2. Drop constant factors

- Use the smallest possible class of functions:
  Say "$2N$ is $O(N)$" instead of "$2N$ is $O(N^2)$

- Use the simplest expression of the class
  Say "$3N + 5$ is $O(N)$" instead of "$3N + 5$ is $O(3N)$"
  Say "$75 + 25$ is $O(1)$" instead of "$75 + 25$ is $O(100)$"

The logarithm

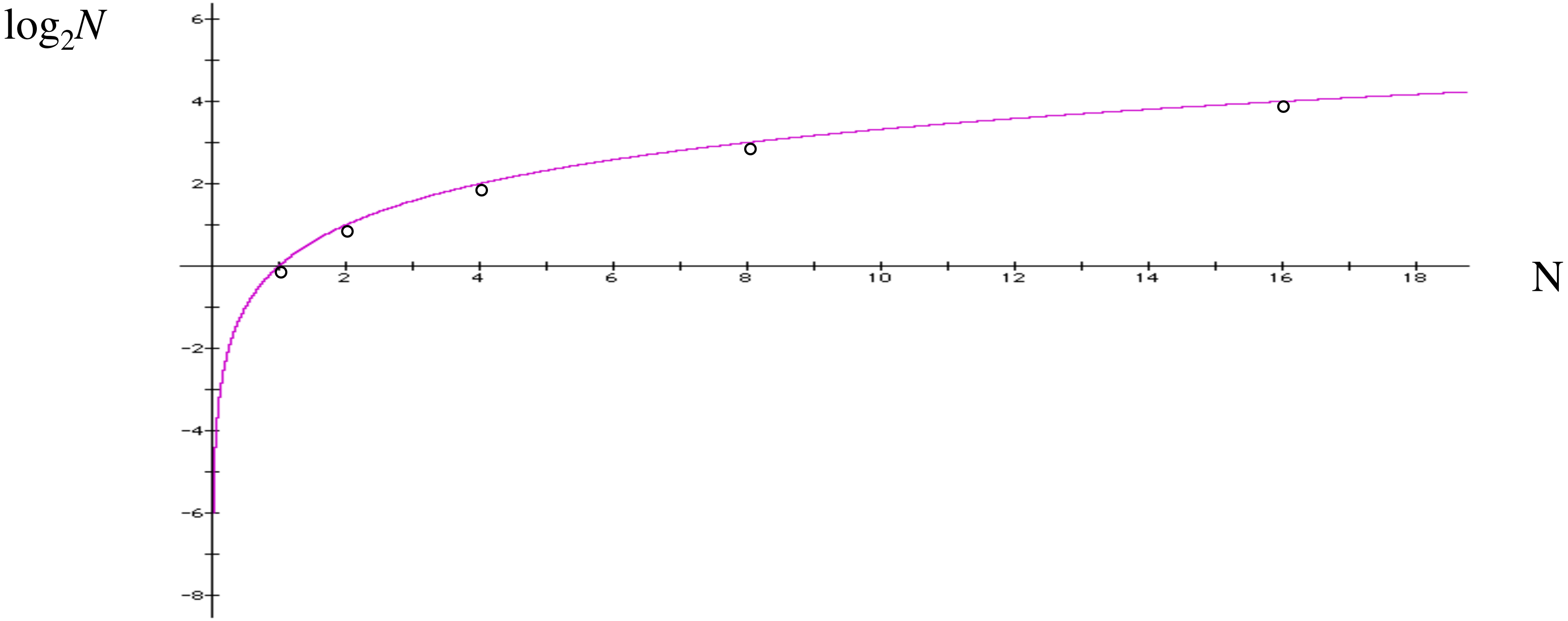
Definition: For any $B, N > 0$, $\log_B N = K$ if $B^K = N$.

In this definition, B is the base. In computer science, when the base is omitted, it defaults to 2.

As far as Big-Oh is concerned, the base is unimportant since all logarithm functions are proportional.

Example: $\log_2 N = c \log_{10} N$, where $c = 1/\log_{10} 2 \approx 3.3$

Growth of $\log_2 N$



Some uses of $\log_2 N$

Bits in a binary number
How many bits are required to represent N consecutive integers?

Answer: $\lceil \log N \rceil$.

Here $\lceil X \rceil$ is the *ceiling* function and represents the smallest integer that is at least as large as X. Example: $\lceil 3.2 \rceil = 4$.

The corresponding *floor* function $\lfloor X \rfloor$ represents the largest integer that is at least as small as X. Example: $\lfloor 3.2 \rfloor = 3$.

to be cont'd

Follow Me

Doubling and Halving

Repeated doubling

Starting from X=1, how many times should X be doubled before it is at least as large as N?

Answer: $\lceil \log N \rceil$.

Repeated halving

Starting from X=N, if N is repeatedly halved, how many iterations must be applied to make N smaller than or equal to 1.

Answer: $\lceil \log N \rceil$ if divisions rounds up; $\lfloor \log N \rfloor$ if division rounds down (as in Java).

Search

Search is the problem of determining whether or not any of a sequence of objects appear among a set of previously stored objects.

Search is the most time consuming part of most programs. Replacing an inefficient search algorithm with a more efficient one will often lead to substantial increase in performance.

Static searching problem

Static searching problem
Given an object X and an array A, return the position of X in A or an indication that it is not present. If X occurs more than once, return any occurrence. The array A is never altered.

Example: Looking up a person in the telephone book.

The efficiency of a static searching algorithm depends on whether the array being searched is sorted.

Linear sequential search

Step through the array until a match is found.

```
public static final int NOT_FOUND = -1;

public static int sequentialSearch(Object[] a, Object x) {
    for (int i = 0; i < a.length; i++)
        if (a[i].equals(x))
            return i;
    return NOT_FOUND;
}
```

Worst-case running time is $O(N)$.
Average-case running time is $O(N)$.

Binary search

Requires that the input array is **sorted**.

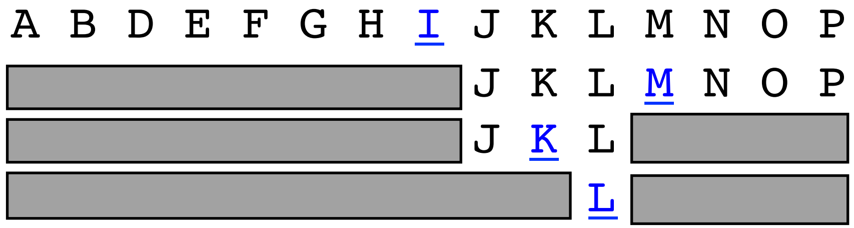
Algorithm:

Split the array into two parts of (almost) equal size.

Determine which part may contain the search item.

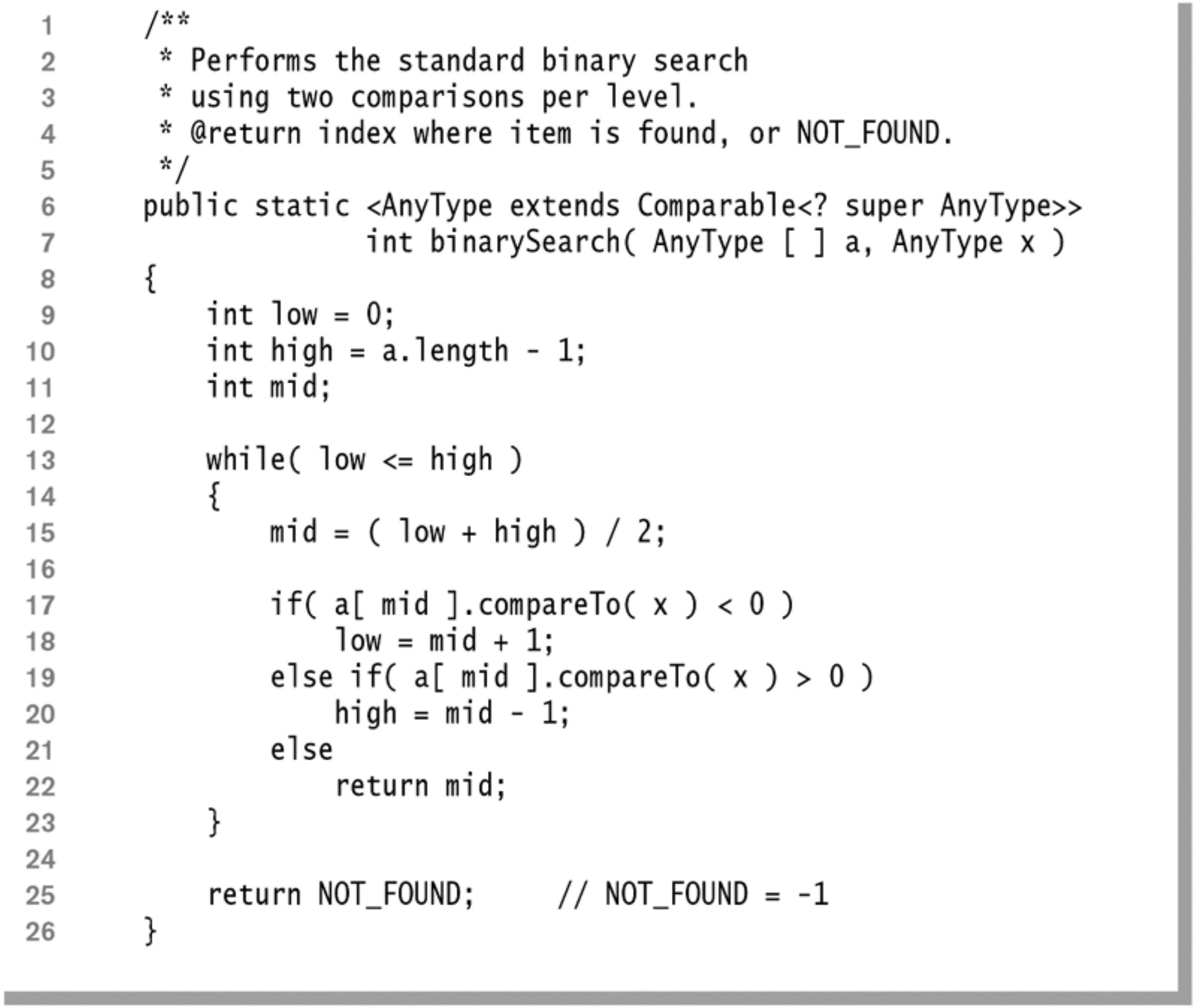
Continue the search in this part in the same fashion.

Example: Searching for **`L`**.



Worst-case running time is *O*(log *N*).
Average-case running time is *O*(log *N*).

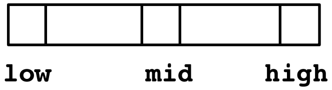
Binary search (three way-comparisons)



```java
/**
 * Performs the standard binary search
 * using two comparisons per level.
 * @return index where item is found, or NOT_FOUND.
 */
public static <AnyType extends Comparable<? super AnyType>>
              int binarySearch( AnyType [ ] a, AnyType x )
{
    int low = 0;
    int high = a.length - 1;
    int mid;

    while( low <= high )
    {
        mid = ( low + high ) / 2;

        if( a[ mid ].compareTo( x ) < 0 )
            low = mid + 1;
        else if( a[ mid ].compareTo( x ) > 0 )
            high = mid - 1;
        else
            return mid;
    }

    return NOT_FOUND;     // NOT_FOUND = -1
}
```

figure 5.11

Basic binary search that uses three-way comparisons



Binary search (two way-comparisons)

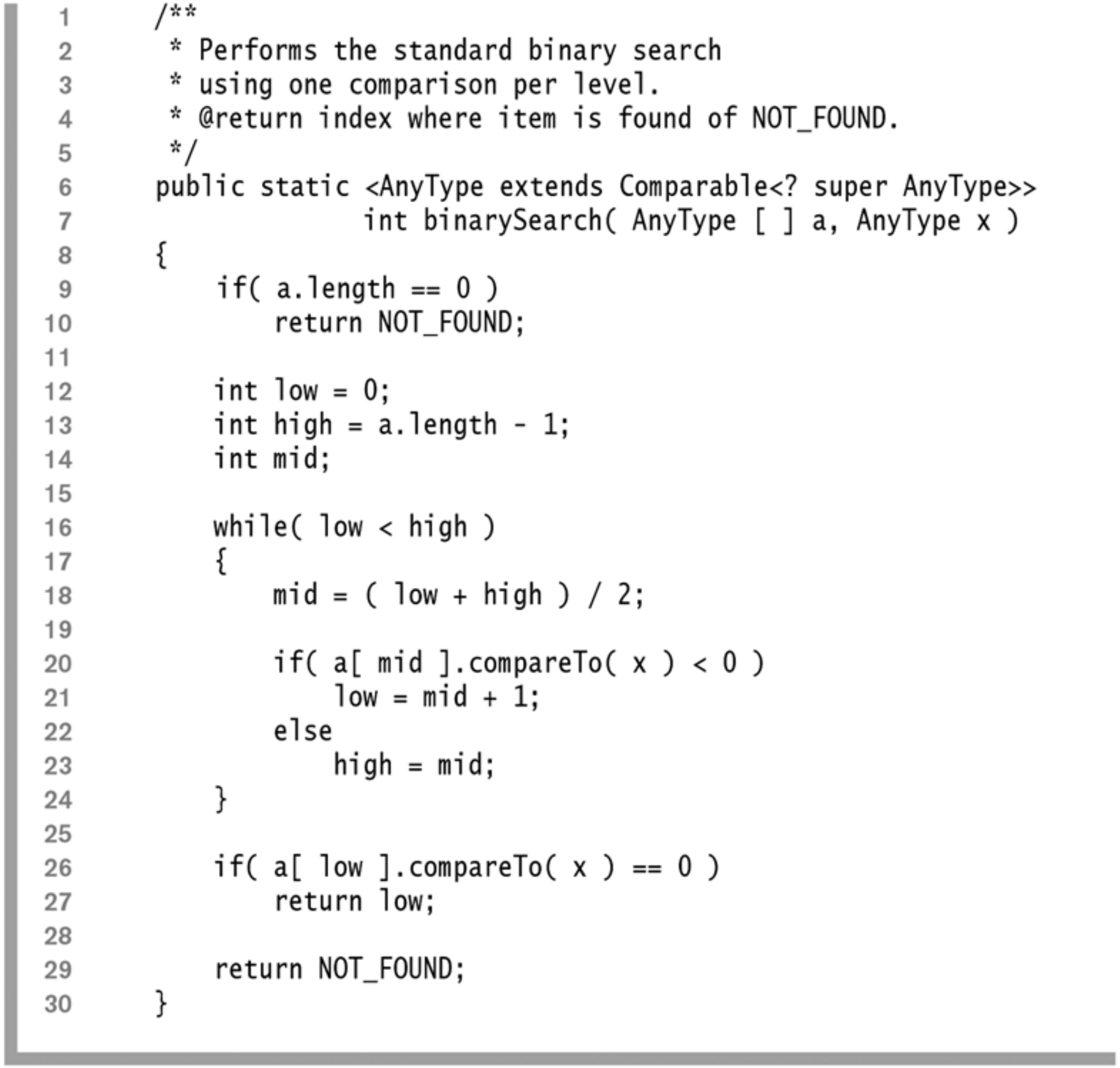
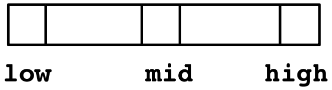


figure 5.12

Binary search using two-way comparisons

```
/**
 * Performs the standard binary search
 * using one comparison per level.
 * @return index where item is found of NOT_FOUND.
 */
public static <AnyType extends Comparable<? super AnyType>>
              int binarySearch( AnyType [ ] a, AnyType x )
{
    if( a.length == 0 )
        return NOT_FOUND;

    int low = 0;
    int high = a.length - 1;
    int mid;

    while( low < high )
    {
        mid = ( low + high ) / 2;

        if( a[ mid ].compareTo( x ) < 0 )
            low = mid + 1;
        else
            high = mid;
    }

    if( a[ low ].compareTo( x ) == 0 )
        return low;

    return NOT_FOUND;
}
```



Are you one of the 10% of programmers that can write a binary search?

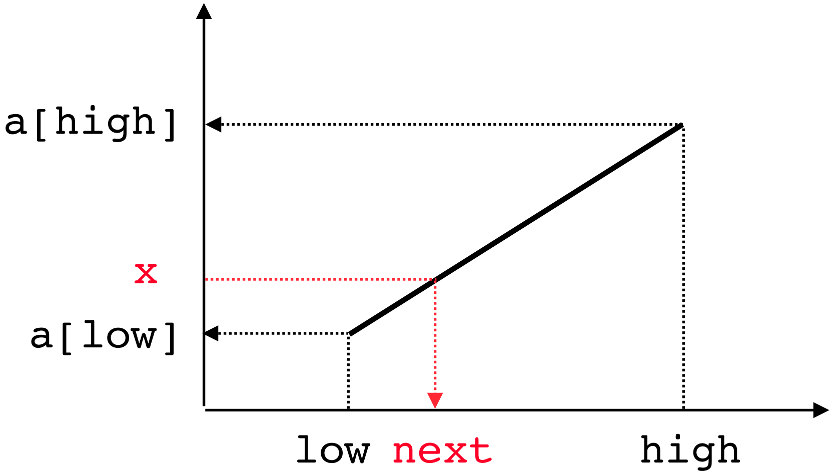
While the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.

In 1986 Jon Bentley found that 90% of the professional programmers that followed his courses could not write an error-free version in two hours.

Interpolation search

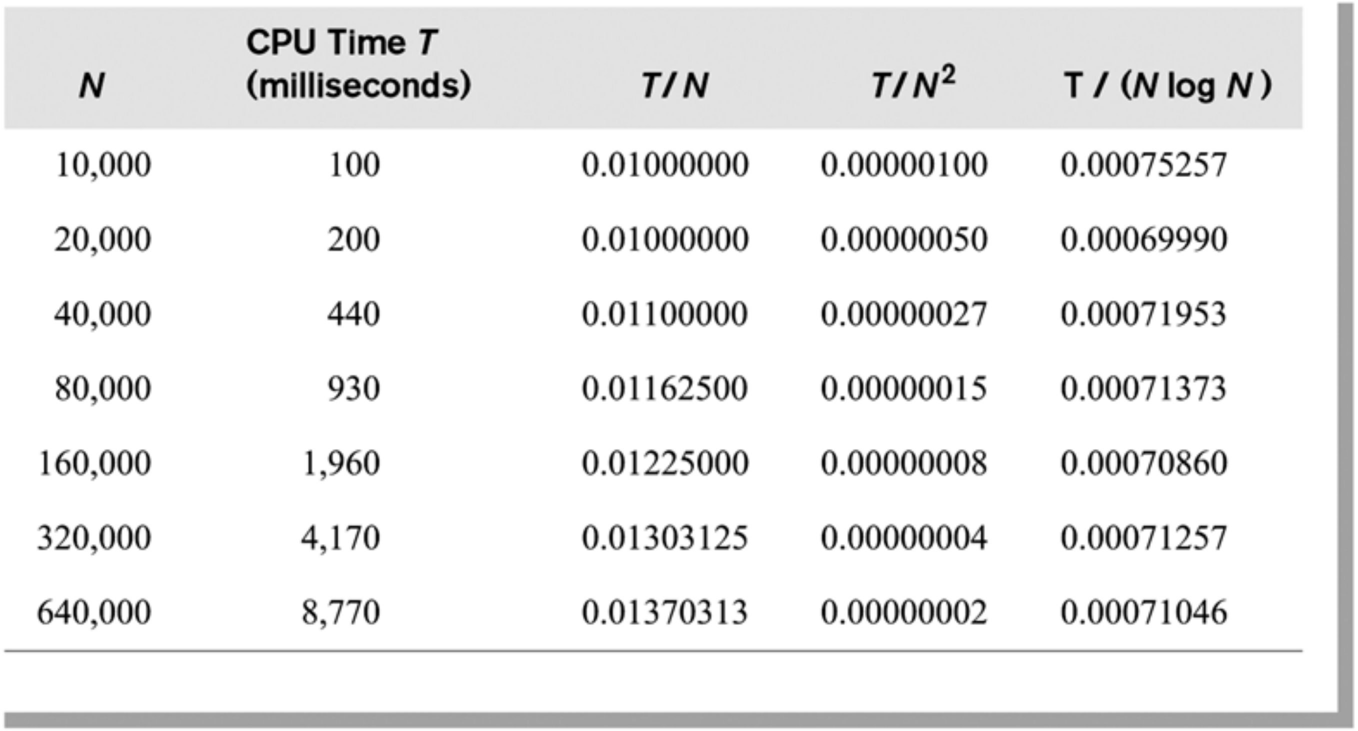
One improvement that is possible to binary search is to *guess* where the search key falls within the current interval of interest. In binary search we replace `mid` with `next`, and `mid = (low + high) / 2` with

`next = low + (x - a[low]) / (a[high] - a[low]) *(high - low)`



If the elements are uniformly distributed, the average number of comparisons has shown to be $O(\log \log N)$.
For $N = 4{,}000{,}000{,}000$, $\log \log N$ is about 5.

Checking an algorithm analysis



N	CPU Time T (milliseconds)	T/N	T/N^2	$T/(N \log N)$
10,000	100	0.01000000	0.00000100	0.00075257
20,000	200	0.01000000	0.00000050	0.00069990
40,000	440	0.01100000	0.00000027	0.00071953
80,000	930	0.01162500	0.00000015	0.00071373
160,000	1,960	0.01225000	0.00000008	0.00070860
320,000	4,170	0.01303125	0.00000004	0.00071257
640,000	8,770	0.01370313	0.00000002	0.00071046

figure 5.13

Empirical running time for N binary searches in an N-item array

Limitations of Big-Oh analysis

Big-Oh analysis is not appropriate for small amounts of input. For small amounts of input use the simplest algorithm.

Large constants can come into play when an algorithm is excessively complex.

The analysis assumes infinite memory, but in applications involving large data sets, lack of sufficient memory can be a severe problem.

For many complicated algorithms the worst-case bound is achievable only by some bad input, but in practice it is usually an overestimate.

Quote

“People who analyze algorithms have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically.”

Donald E. Knuth

The Collections API

Data structures

Data structure: a particular way of storing and organizing data in a computer so that it can be used efficiently.

Each data structure allows arbitrary insertion but differs in how it allows access to items. Some data structures allow arbitrary access and deletions, whereas others impose restrictions, such as allowing access to only the most recently or least recently inserted items. Some data structures allow duplicates; others do not.

The **Collections API** provides a number of useful of data structures. It also provides some generic algorithms, such as sorting.

Collections

A **collection** is an object that *contains* other objects, which are called the **elements** of the collection.

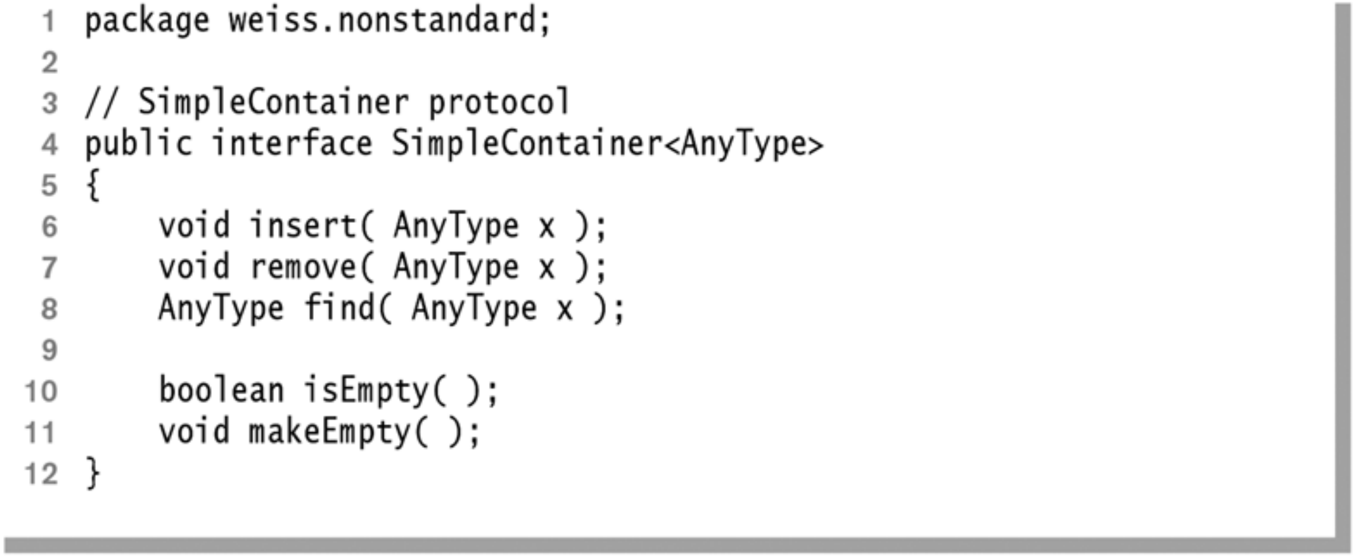
Java Collections is a set of interfaces and classes that support storing and retrieving elements in collections.

Collections are implemented by a variety of data structures and algorithms (with different time-space complexities).

The Collection API frees you from reinventing the wheel. This is the essence of *reuse*.

We do not need to know *how* something is implemented, so long as we know *what* is implemented. This is the essence of *data abstraction*.

A generic protocol for collections



```
package weiss.nonstandard;

// SimpleContainer protocol
public interface SimpleContainer<AnyType>
{
    void insert( AnyType x );
    void remove( AnyType x );
    AnyType find( AnyType x );

    boolean isEmpty( );
    void makeEmpty( );
}
```

figure 6.1

A generic protocol for many data structures

Many data structures tend to follow this protocol.
However, we do not use this protocol directly in any code.

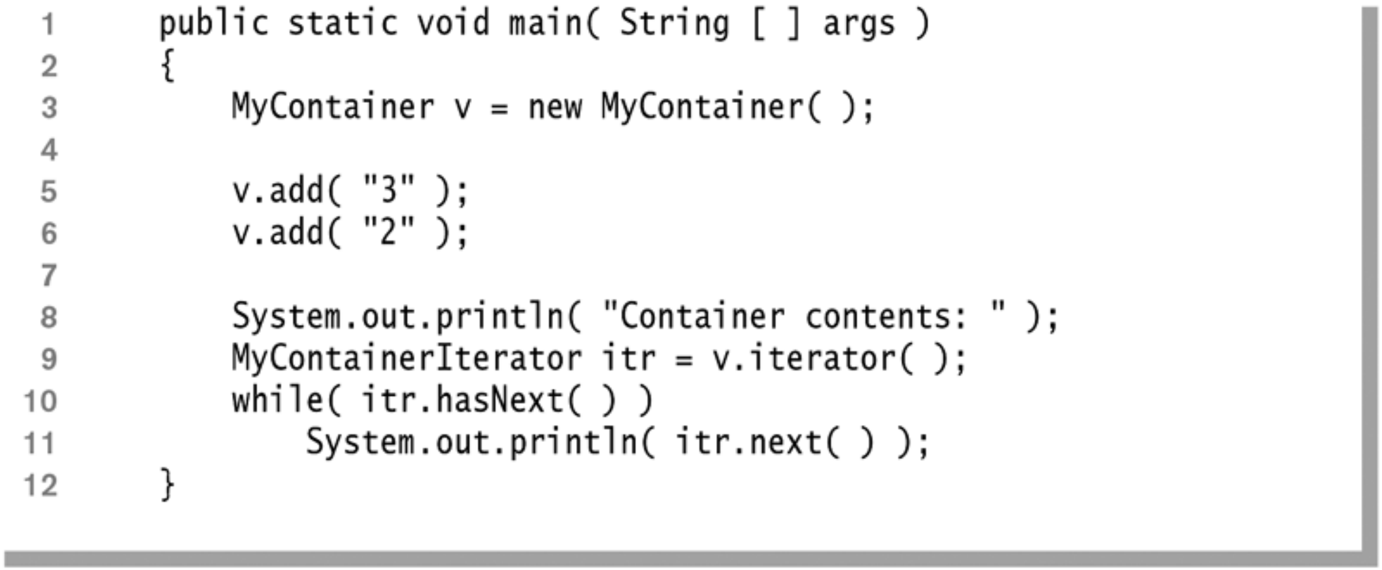
The iterator pattern

Enumerating all elements in a collection is one of the most common operations on any collection.

The *iterator design pattern* makes this possible without exposing the underlying data structure.

An **iterator** is an object that allows traversal of all elements in a collection, regardless of its specific implementation. Thus, if the implementation changes, code that uses the iterator does not need to be changed. This is an example of *abstract coupling*.

`main` for design 1

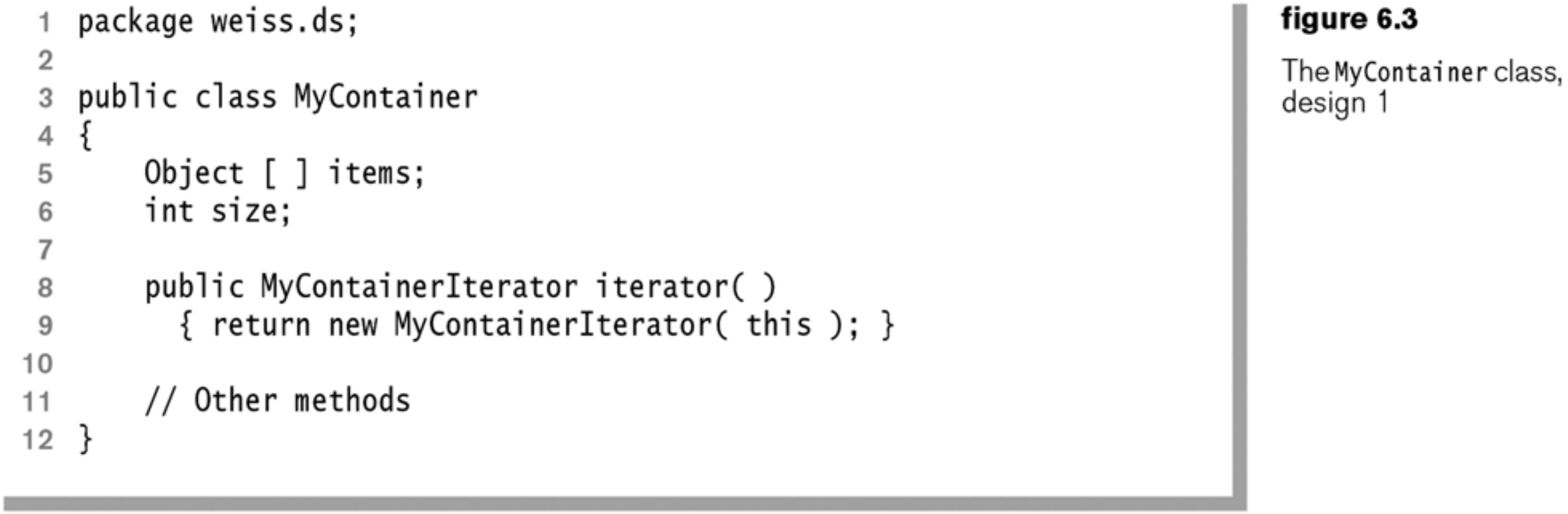


```
public static void main( String [ ] args )
{
    MyContainer v = new MyContainer( );

    v.add( "3" );
    v.add( "2" );

    System.out.println( "Container contents: " );
    MyContainerIterator itr = v.iterator( );
    while( itr.hasNext( ) )
        System.out.println( itr.next( ) );
}
```

figure 6.2

A `main` method, to illustrate iterator design 1

`MyContainer` for design 1 (array-based collection)



```
package weiss.ds;

public class MyContainer
{
    Object [ ] items;
    int size;

    public MyContainerIterator iterator( )
      { return new MyContainerIterator( this ); }

    // Other methods
}
```

figure 6.3

The `MyContainer` class, design 1

`MyContainerIterator` for design 1

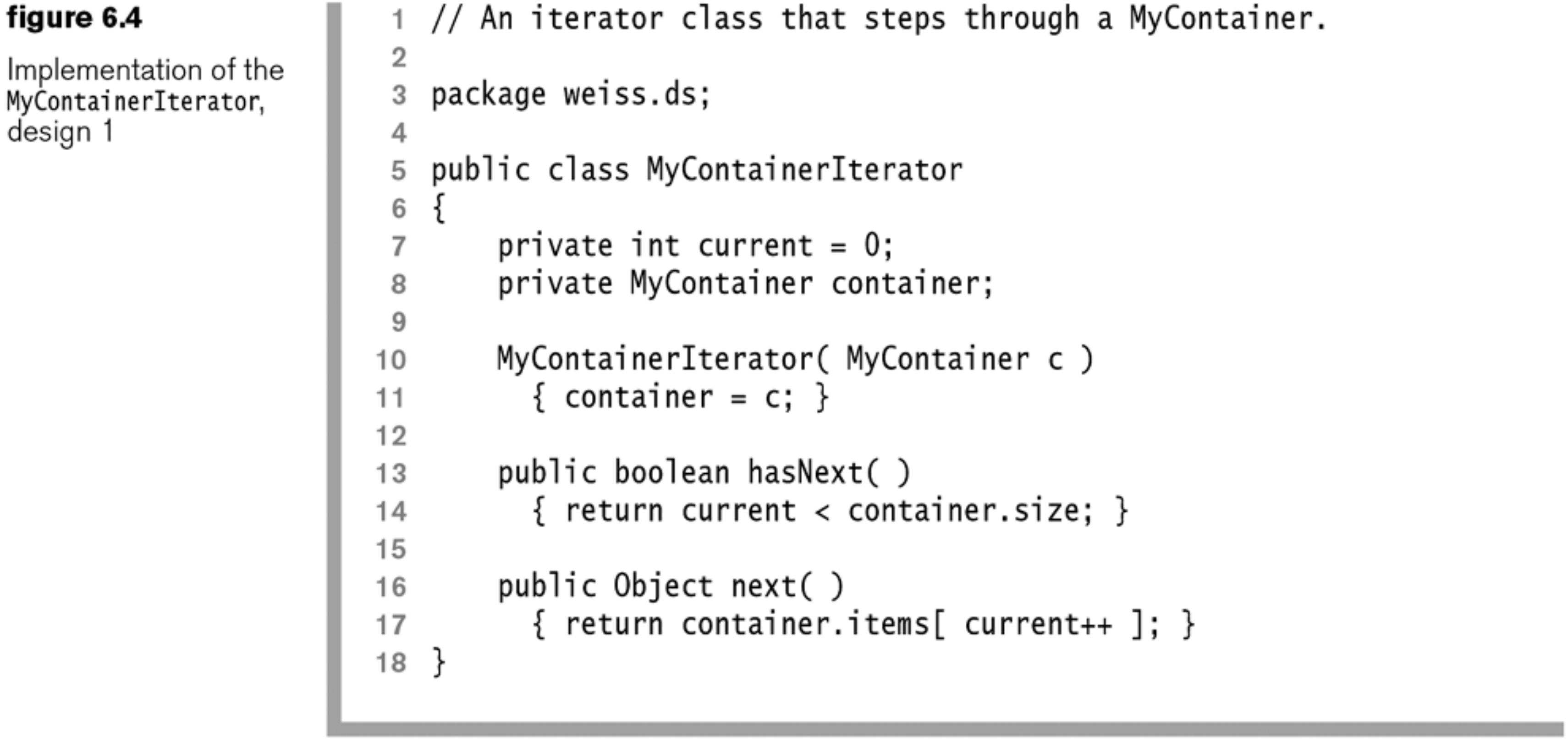


figure 6.4

Implementation of the `MyContainerIterator`, design 1

```
// An iterator class that steps through a MyContainer.

package weiss.ds;

public class MyContainerIterator
{
    private int current = 0;
    private MyContainer container;

    MyContainerIterator( MyContainer c )
      { container = c; }

    public boolean hasNext( )
      { return current < container.size; }

    public Object next( )
      { return container.items[ current++ ]; }
}
```

Drawback of design 1

Change from an array-based collection to something else requires that we change all declarations of the iterator.

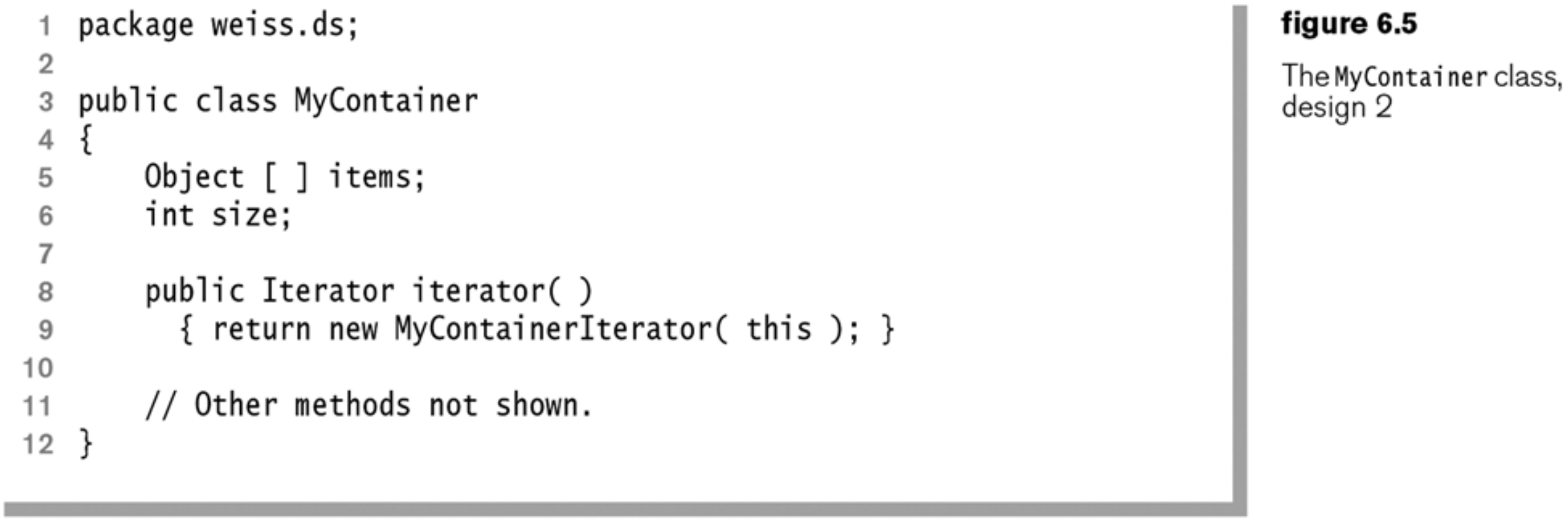
For instance, in the `main` method we need to change the line:

```
MyContainerIterator itr = ...
```

This drawback may be removed by defining an interface, `Iterator`, that is an abstraction of the capabilities of iterators.

Clients (in this case, `main`) will deal only with the abstract iterator and need no knowledge about the concrete iterator.

`MyContainer` for design 2

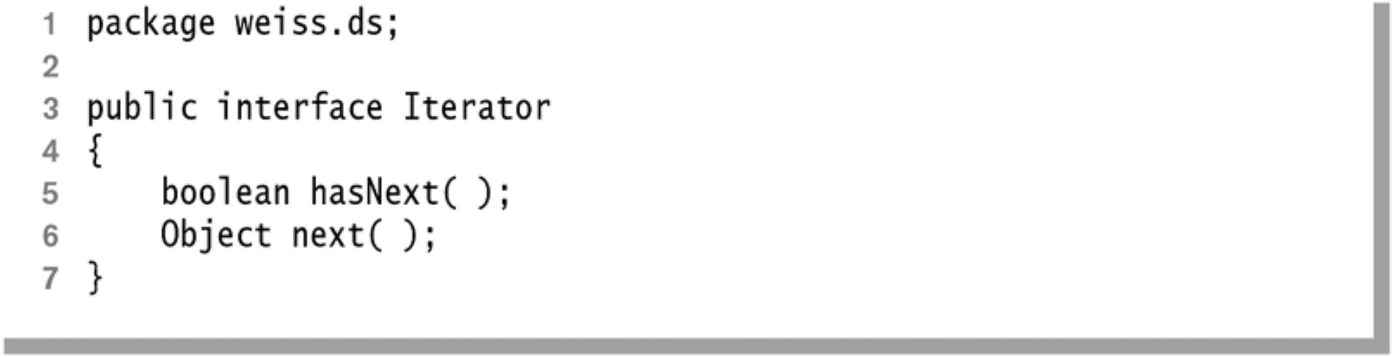


```
package weiss.ds;

public class MyContainer
{
    Object [ ] items;
    int size;

    public Iterator iterator( )
      { return new MyContainerIterator( this ); }

    // Other methods not shown.
}
```

figure 6.5

The `MyContainer` class, design 2

The `Iterator` interface for design 2



```
package weiss.ds;

public interface Iterator
{
    boolean hasNext( );
    Object next( );
}
```

figure 6.6

The `Iterator` interface, design 2

`MyContainerIterator` for design 2

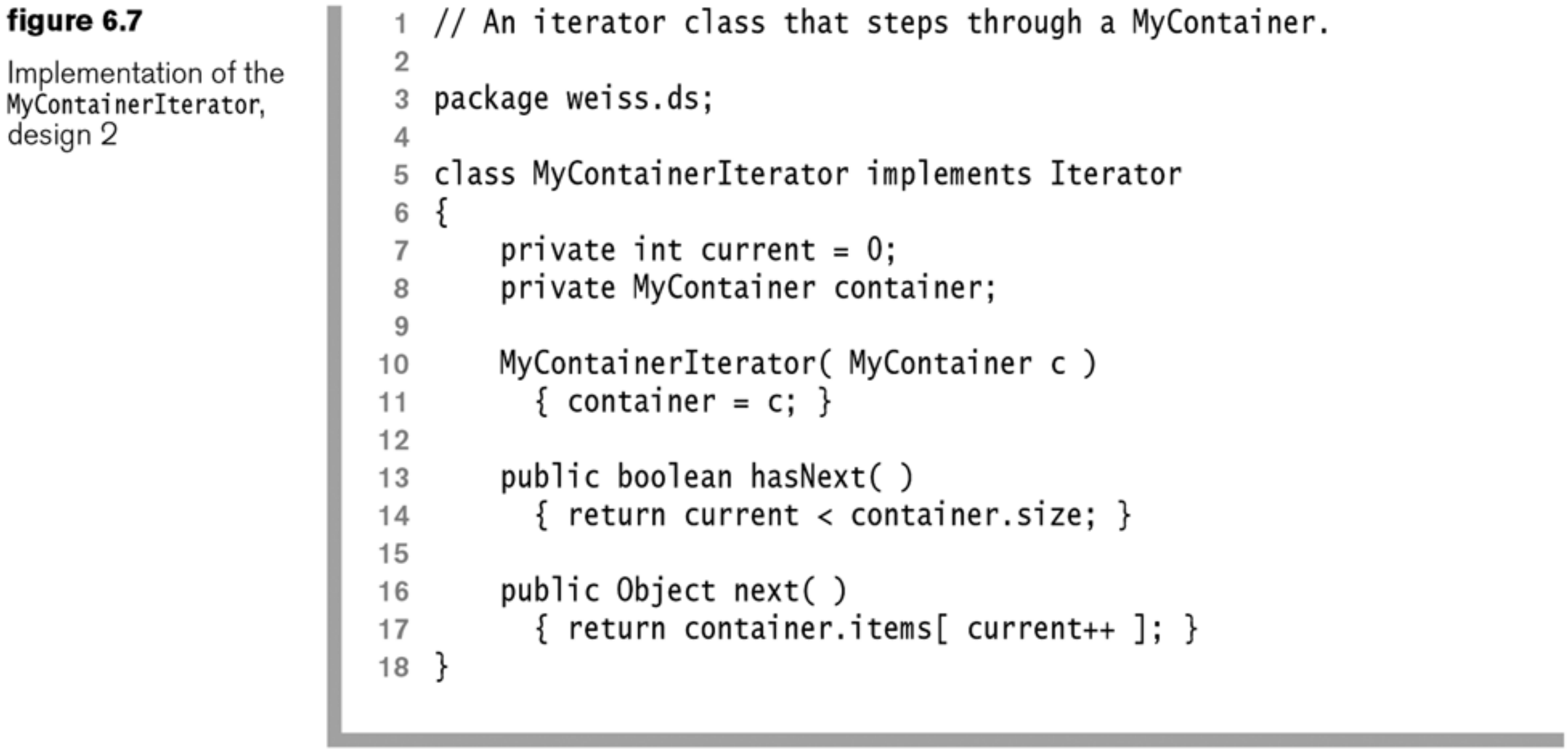


figure 6.7

Implementation of the `MyContainerIterator`, design 2

```
// An iterator class that steps through a MyContainer.

package weiss.ds;

class MyContainerIterator implements Iterator
{
    private int current = 0;
    private MyContainer container;

    MyContainerIterator( MyContainer c )
      { container = c; }

    public boolean hasNext( )
      { return current < container.size; }

    public Object next( )
      { return container.items[ current++ ]; }
}
```

`main` for design 2

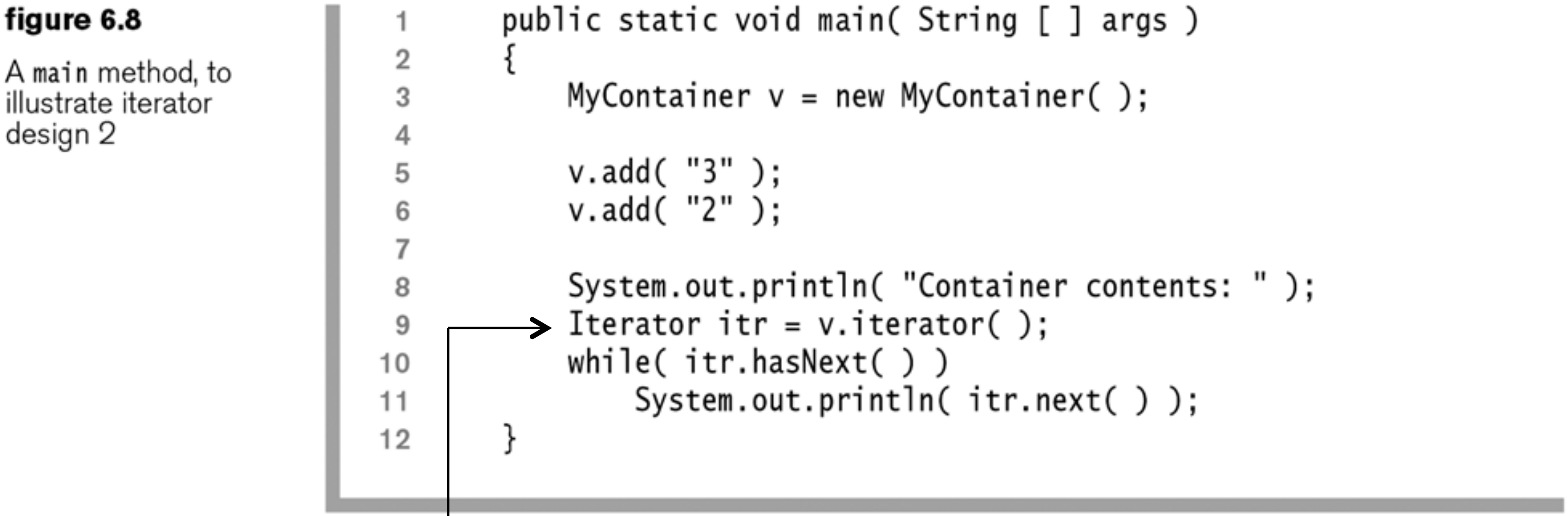


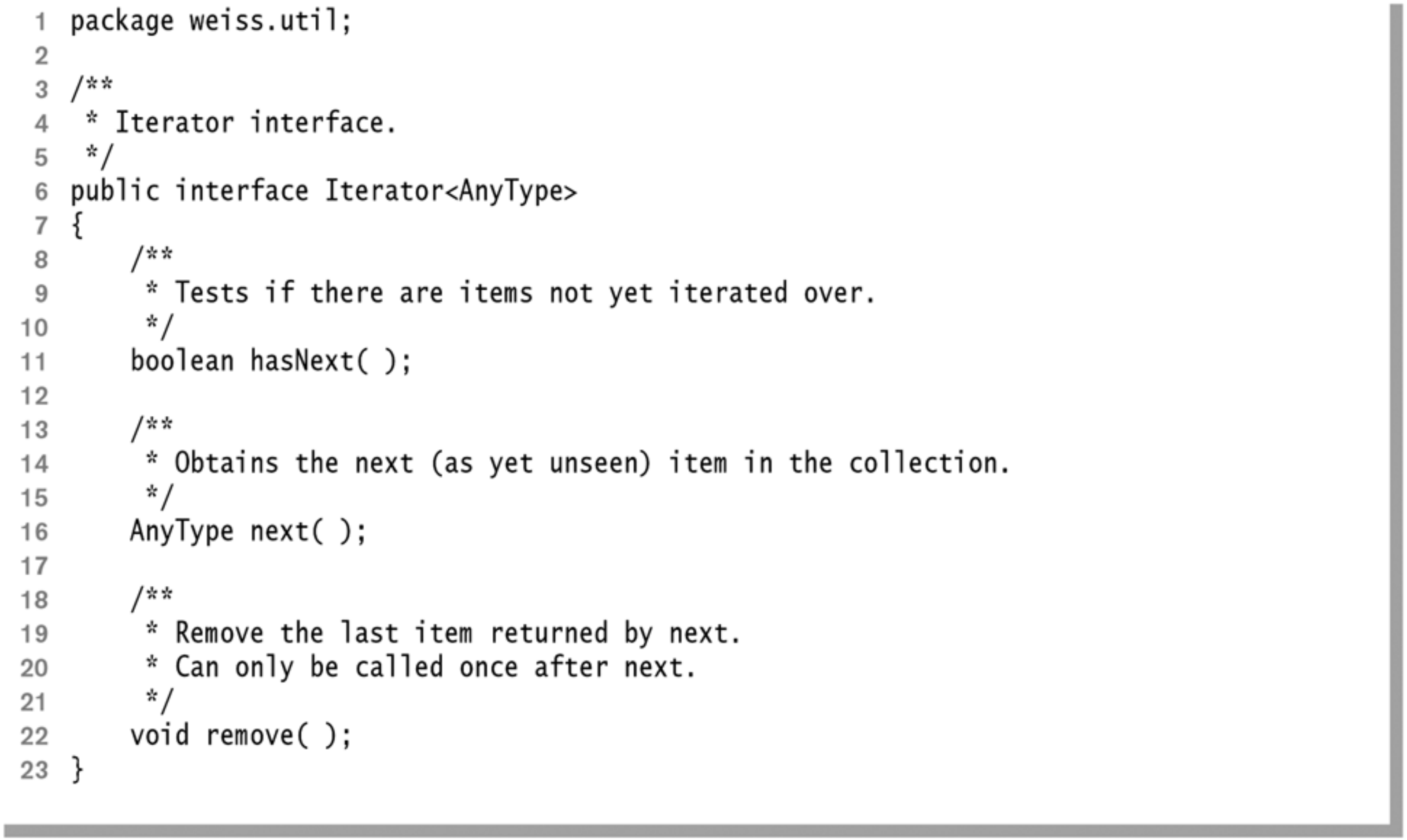
figure 6.8

A `main` method, to illustrate iterator design 2

```java
public static void main( String [ ] args )
{
    MyContainer v = new MyContainer( );

    v.add( "3" );
    v.add( "2" );

    System.out.println( "Container contents: " );
    Iterator itr = v.iterator( );
    while( itr.hasNext( ) )
        System.out.println( itr.next( ) );
}
```

Programming to an interface

A sample specification of `Iterator`



```
package weiss.util;

/**
 * Iterator interface.
 */
public interface Iterator<AnyType>
{
    /**
     * Tests if there are items not yet iterated over.
     */
    boolean hasNext( );

    /**
     * Obtains the next (as yet unseen) item in the collection.
     */
    AnyType next( );

    /**
     * Remove the last item returned by next.
     * Can only be called once after next.
     */
    void remove( );
}
```

figure 6.10

A sample specification of `Iterator`

Printing the contents of any `Collection`

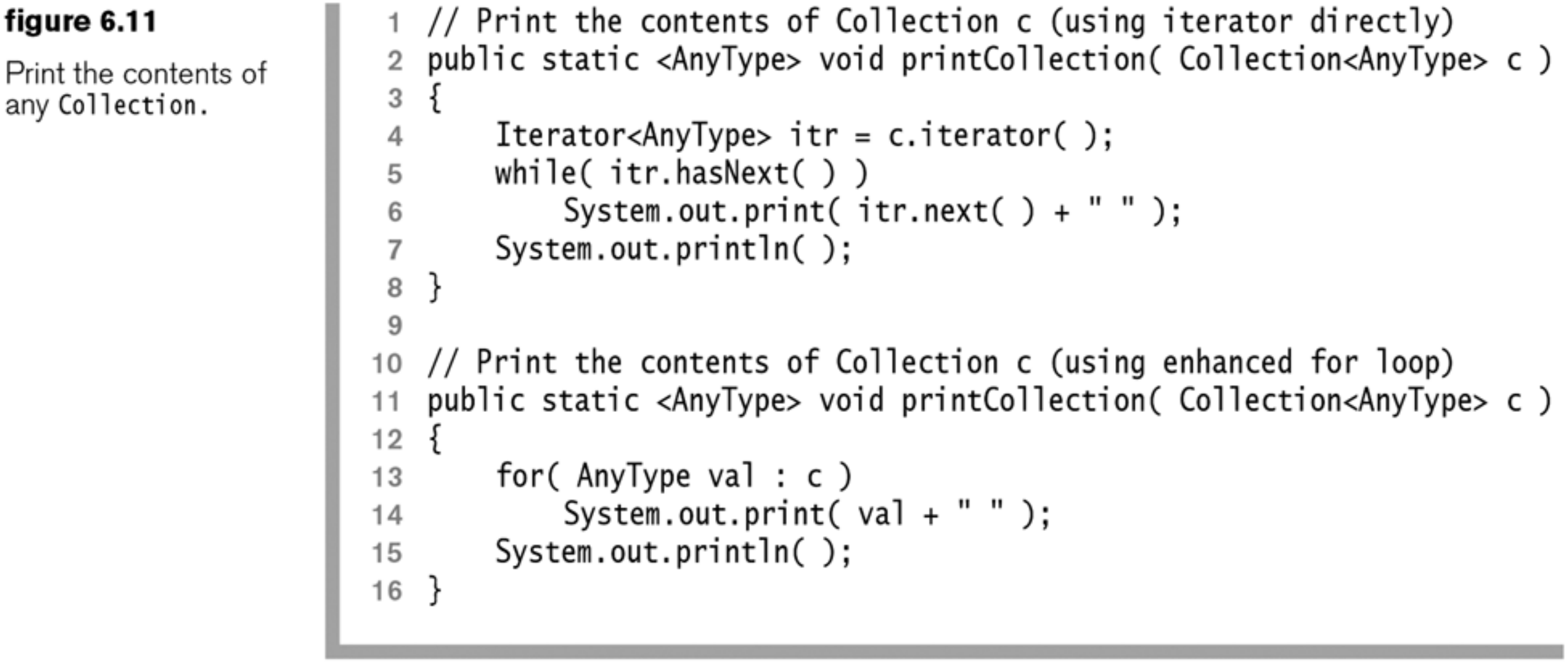


figure 6.11

Print the contents of any `Collection`.

```
// Print the contents of Collection c (using iterator directly)
public static <AnyType> void printCollection( Collection<AnyType> c )
{
    Iterator<AnyType> itr = c.iterator( );
    while( itr.hasNext( ) )
        System.out.print( itr.next( ) + " " );
    System.out.println( );
}

// Print the contents of Collection c (using enhanced for loop)
public static <AnyType> void printCollection( Collection<AnyType> c )
{
    for( AnyType val : c )
        System.out.print( val + " " );
    System.out.println( );
}
```

The enhanced for loop is simply a compiler substitution.

Abstract collections

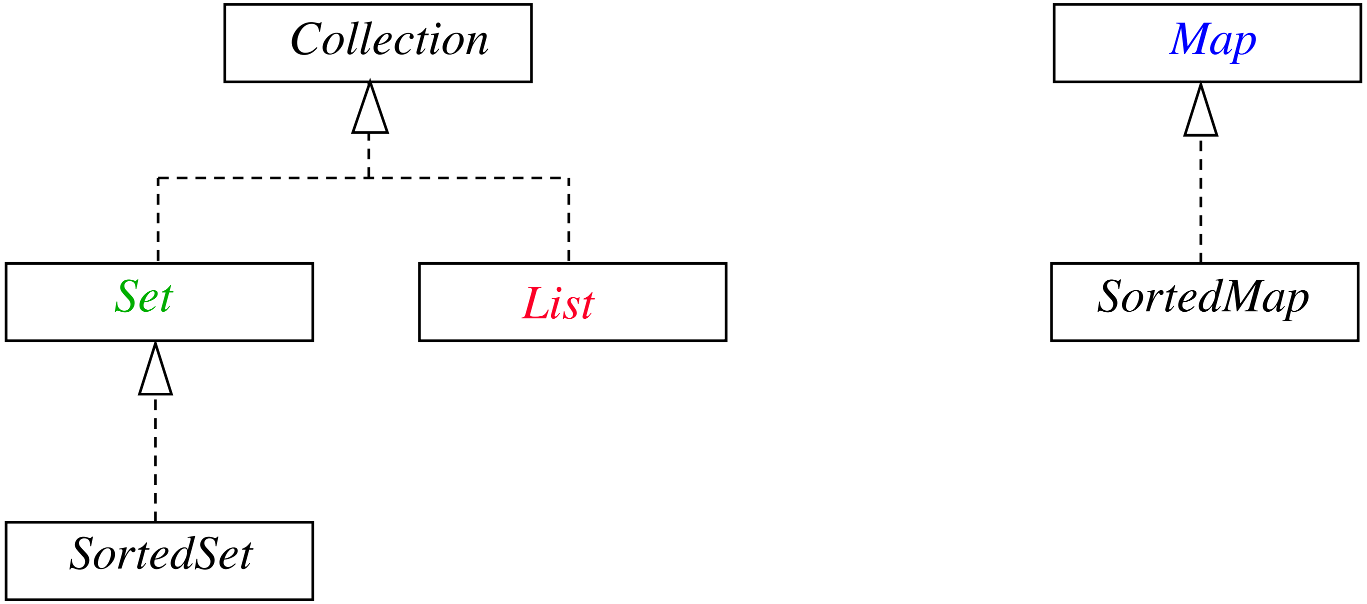
A **set** is an unordered collection of elements.
No duplicates are allowed.

A **list** is an ordered collection of elements.
Duplicates are allowed.
Lists are also known an *sequences*.

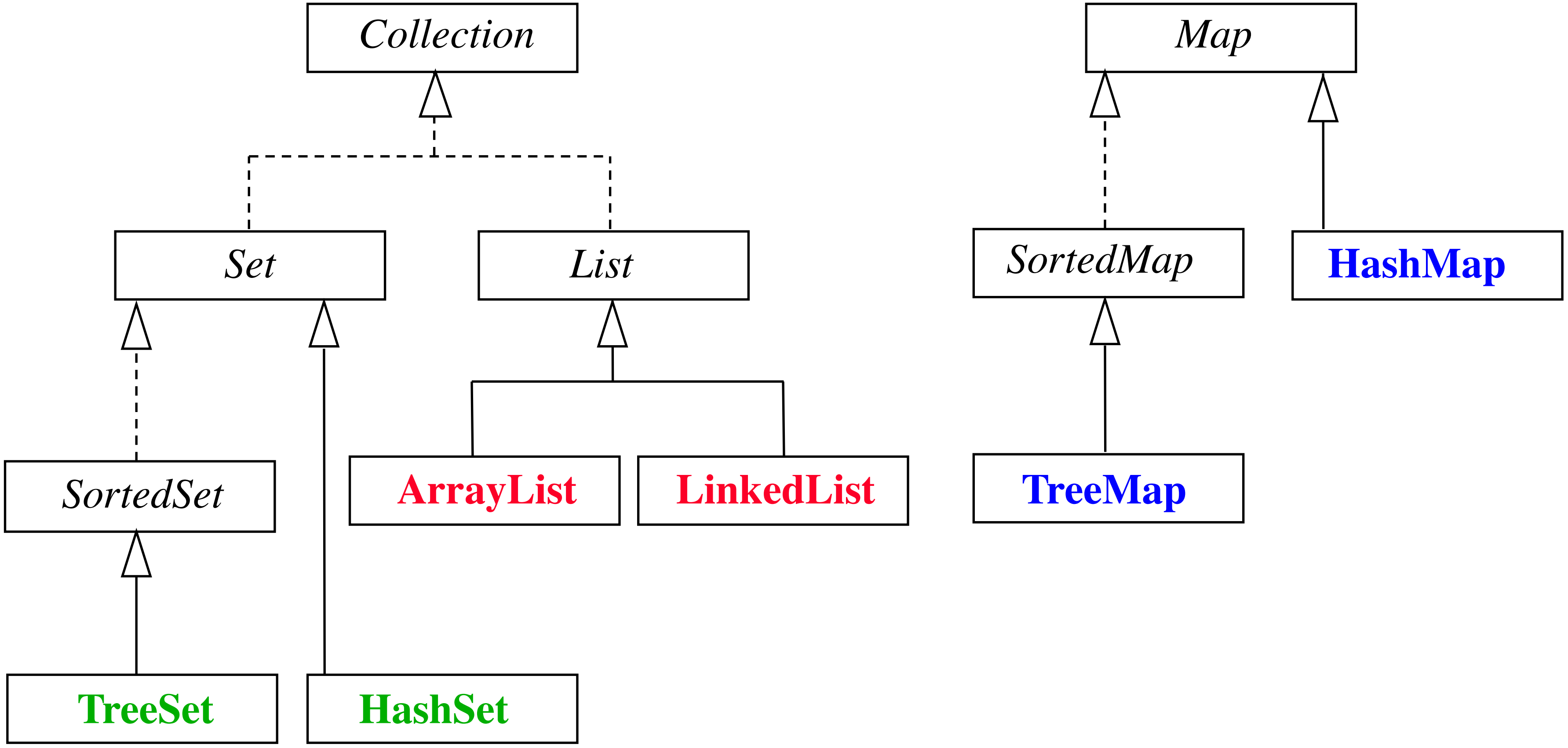
A map is an unordered collection of key-value pairs.
The keys must be unique.
Maps are also known as *dictionaries*.

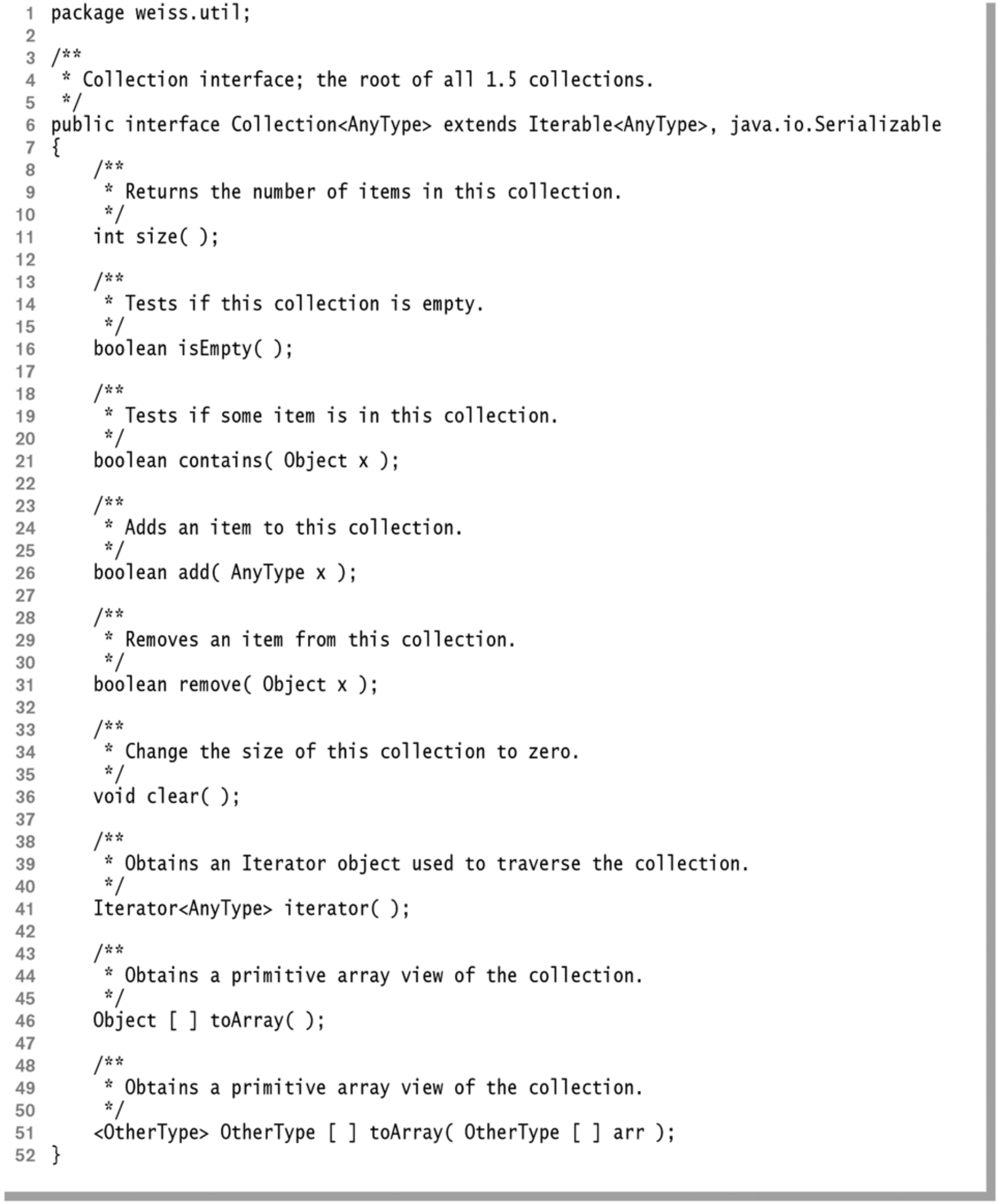
Interfaces for collections

java.util.*



Concrete collections





```
package weiss.util;

/**
 * Collection interface; the root of all 1.5 collections.
 */
public interface Collection<AnyType> extends Iterable<AnyType>, java.io.Serializable
{
    /**
     * Returns the number of items in this collection.
     */
    int size( );

    /**
     * Tests if this collection is empty.
     */
    boolean isEmpty( );

    /**
     * Tests if some item is in this collection.
     */
    boolean contains( Object x );

    /**
     * Adds an item to this collection.
     */
    boolean add( AnyType x );

    /**
     * Removes an item from this collection.
     */
    boolean remove( Object x );

    /**
     * Change the size of this collection to zero.
     */
    void clear( );

    /**
     * Obtains an Iterator object used to traverse the collection.
     */
    Iterator<AnyType> iterator( );

    /**
     * Obtains a primitive array view of the collection.
     */
    Object [ ] toArray( );

    /**
     * Obtains a primitive array view of the collection.
     */
    <OtherType> OtherType [ ] toArray( OtherType [ ] arr );
}
```

figure 6.9

A sample specification of the `Collection` interface

`interface` **`Collection<E>`**

```
boolean add(E o)
boolean addAll(Collection<? extends E> c)
void clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean isEmpty()
Iterator<E> iterator()
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
int size()
Object[] toArray()
<T> T[] toArray(T[] a)
```

`interface` **`Set<E>`** `extends Collection<E>`

No new methods are introduced. However, the contracts for

```
boolean add(E o)
boolean addAll(Collection<? extends E> c)
```

are changed owing to the “no duplicates” restriction of sets (checked by calls to `equals`).

Example of using `Set`

```
Set<String> set = new HashSet<String>();
set.add("cat");
set.add("dog");
int n = set.size();
System.out.println("The set contains " + n + " elements");
if (set.contains("dog"))
    System.out.println("dog is in the set");
```

Type inference. The diamond operator:
In Java 7, you can write

```
Set<String> set = new HashSet<>();
```

`interface` **`List<E>`** `extends Collection<E>`

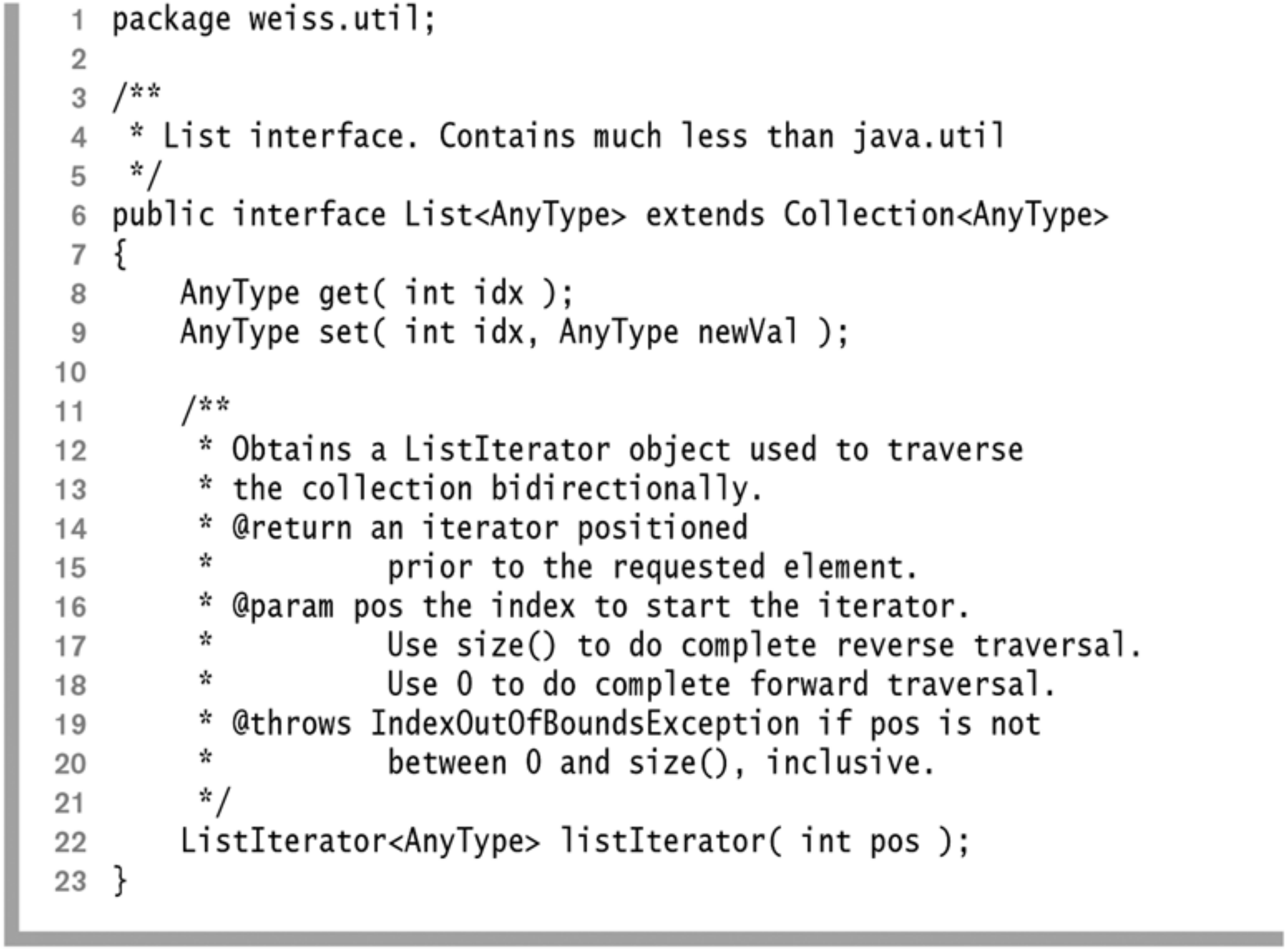
New methods:

```
void add(int index, E element)
E get(int index)
int indexOf(Object o)
int lastIndexOf(Object o)
ListIterator<E> listIterator()
ListIterator<E> listIterator(int index)
E remove(int index)
E set(int index, E element)
List subList(int fromIndex, int toIndex)
```

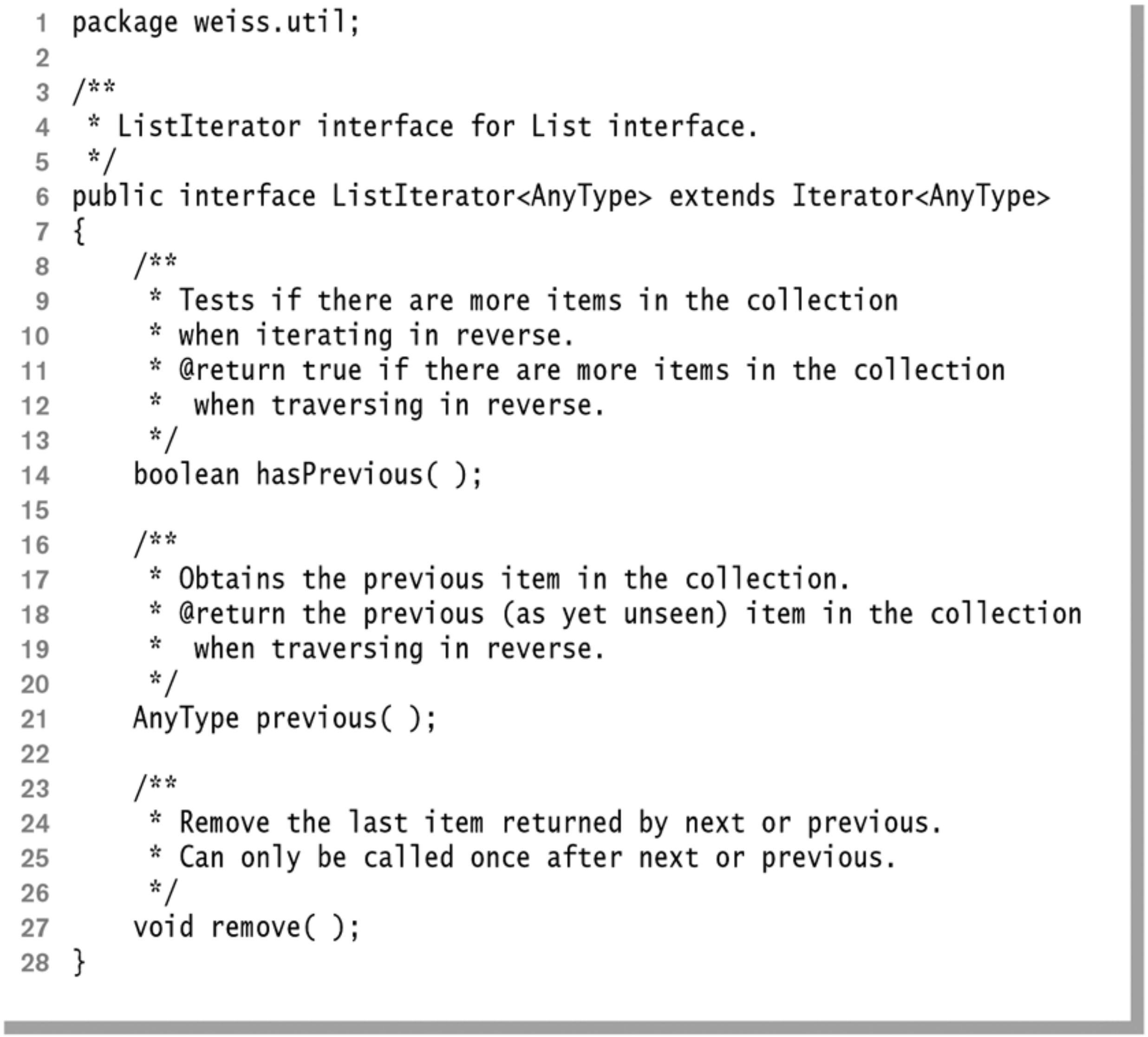
The contracts of `add(o)` and `addAll(c)` are changed because of the ordering imposed on lists.

figure 6.16

A sample `List` interface



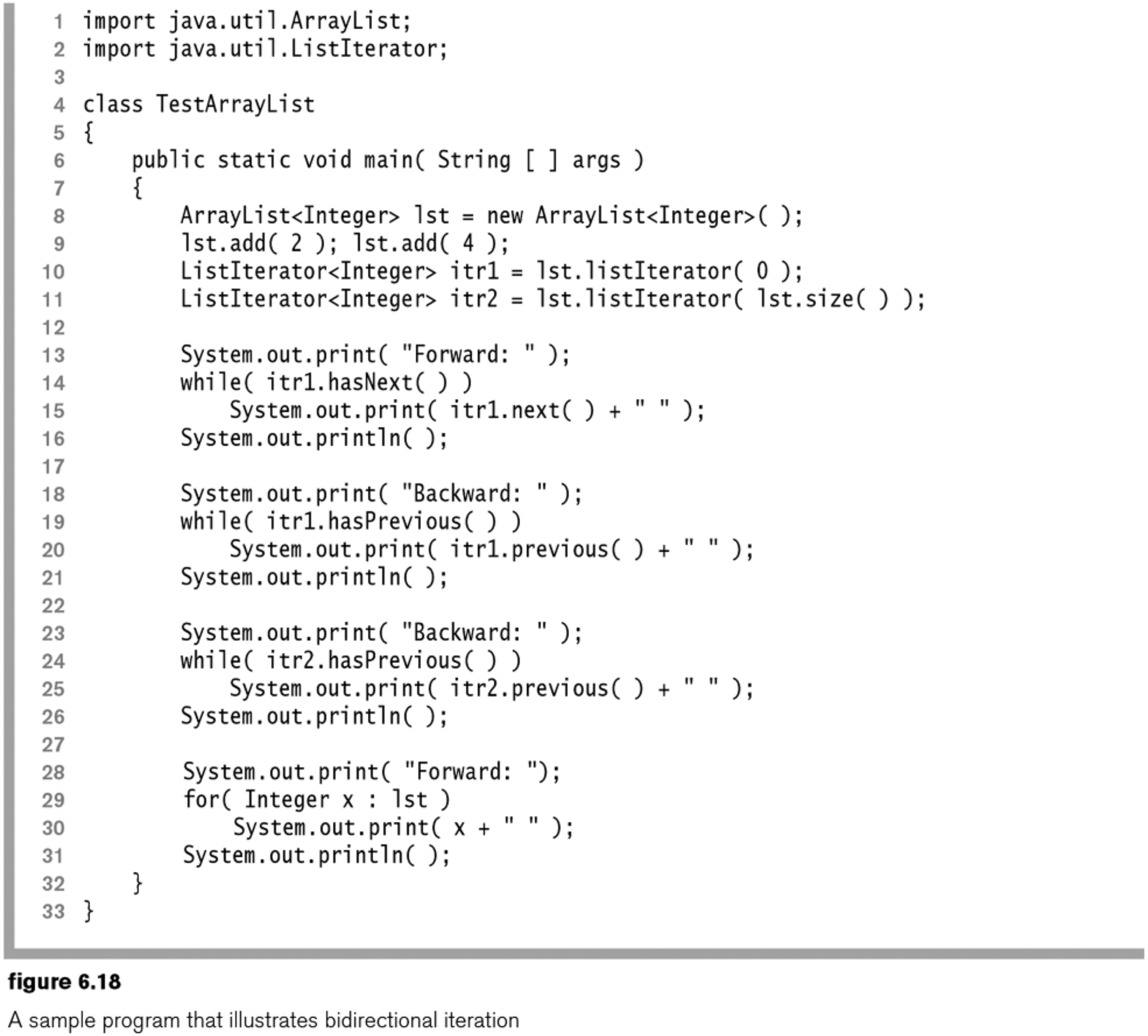
```
package weiss.util;

/**
 * List interface. Contains much less than java.util
 */
public interface List<AnyType> extends Collection<AnyType>
{
    AnyType get( int idx );
    AnyType set( int idx, AnyType newVal );

    /**
     * Obtains a ListIterator object used to traverse
     * the collection bidirectionally.
     * @return an iterator positioned
     *         prior to the requested element.
     * @param pos the index to start the iterator.
     *         Use size() to do complete reverse traversal.
     *         Use 0 to do complete forward traversal.
     * @throws IndexOutOfBoundsException if pos is not
     *         between 0 and size(), inclusive.
     */
    ListIterator<AnyType> listIterator( int pos );
}
```



```
package weiss.util;

/**
 * ListIterator interface for List interface.
 */
public interface ListIterator<AnyType> extends Iterator<AnyType>
{
    /**
     * Tests if there are more items in the collection
     * when iterating in reverse.
     * @return true if there are more items in the collection
     *  when traversing in reverse.
     */
    boolean hasPrevious( );

    /**
     * Obtains the previous item in the collection.
     * @return the previous (as yet unseen) item in the collection
     *  when traversing in reverse.
     */
    AnyType previous( );

    /**
     * Remove the last item returned by next or previous.
     * Can only be called once after next or previous.
     */
    void remove( );
}
```

figure 6.17

A sample `ListIterator` interface



```
import java.util.ArrayList;
import java.util.ListIterator;

class TestArrayList
{
    public static void main( String [ ] args )
    {
        ArrayList<Integer> lst = new ArrayList<Integer>( );
        lst.add( 2 ); lst.add( 4 );
        ListIterator<Integer> itr1 = lst.listIterator( 0 );
        ListIterator<Integer> itr2 = lst.listIterator( lst.size( ) );

        System.out.print( "Forward: " );
        while( itr1.hasNext( ) )
            System.out.print( itr1.next( ) + " " );
        System.out.println( );

        System.out.print( "Backward: " );
        while( itr1.hasPrevious( ) )
            System.out.print( itr1.previous( ) + " " );
        System.out.println( );

        System.out.print( "Backward: " );
        while( itr2.hasPrevious( ) )
            System.out.print( itr2.previous( ) + " " );
        System.out.println( );

        System.out.print( "Forward: ");
        for( Integer x : lst )
            System.out.print( x + " " );
        System.out.println( );
    }
}
```

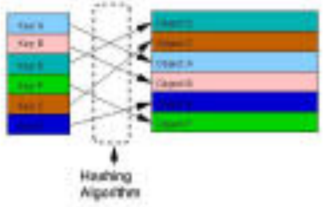
figure 6.18

A sample program that illustrates bidirectional iteration

Example of using `List`

```
List<String> list = new ArrayList<String>();
list.add("cat");
list.add("dog");
list.add("cat");
for (String s : list)
    System.out.println(s);
System.out.println("The first element is " + list.get(0));
```

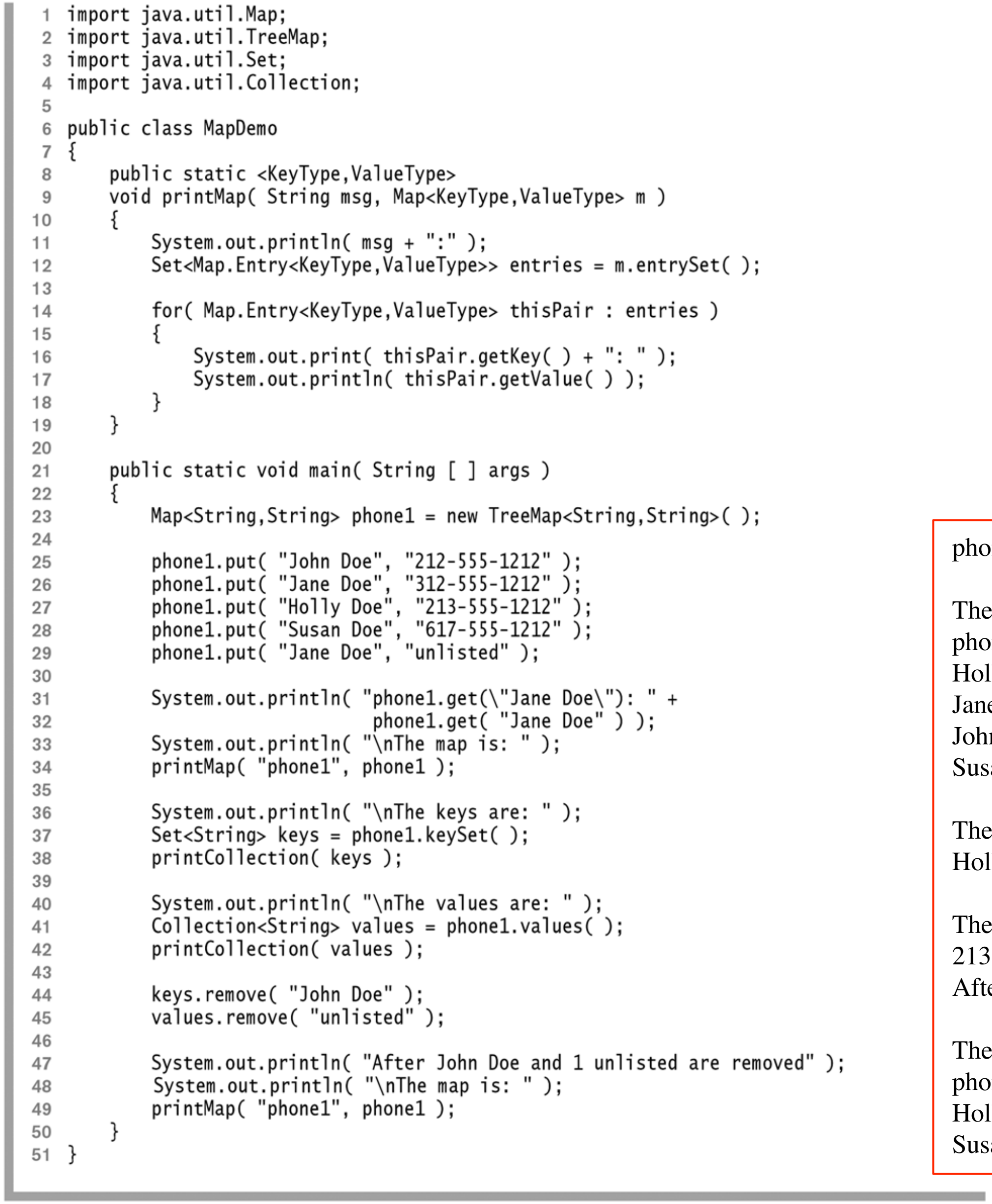
interface **Map<K,V>**



```
V put(K key, V value)
V get(Object key)
V remove(Object key)

void clear()
boolean containsKey(Object key)
boolean containsValue(Object value)
boolean isEmpty()
void putAll(Map<? extends K,? extends V>> map)
int size()

Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K,V>> entrySet()
```

Example of using `Map`

```
Map<String,String> map = new HashMap<String,String>();
map.put("cat", "kat");
map.put("dog", "hund");
String val = map.get("dog");  // val is "hund"
map.remove("cat");
map.put("dog", "køter");
val = map.get("dog");         // val is "køter"
```



```java
import java.util.Map;
import java.util.TreeMap;
import java.util.Set;
import java.util.Collection;

public class MapDemo
{
    public static <KeyType,ValueType>
    void printMap( String msg, Map<KeyType,ValueType> m )
    {
        System.out.println( msg + ":" );
        Set<Map.Entry<KeyType,ValueType>> entries = m.entrySet( );

        for( Map.Entry<KeyType,ValueType> thisPair : entries )
        {
            System.out.print( thisPair.getKey( ) + ": " );
            System.out.println( thisPair.getValue( ) );
        }
    }

    public static void main( String [ ] args )
    {
        Map<String,String> phone1 = new TreeMap<String,String>( );

        phone1.put( "John Doe", "212-555-1212" );
        phone1.put( "Jane Doe", "312-555-1212" );
        phone1.put( "Holly Doe", "213-555-1212" );
        phone1.put( "Susan Doe", "617-555-1212" );
        phone1.put( "Jane Doe", "unlisted" );

        System.out.println( "phone1.get(\"Jane Doe\"): " +
                             phone1.get( "Jane Doe" ) );
        System.out.println( "\nThe map is: " );
        printMap( "phone1", phone1 );

        System.out.println( "\nThe keys are: " );
        Set<String> keys = phone1.keySet( );
        printCollection( keys );

        System.out.println( "\nThe values are: " );
        Collection<String> values = phone1.values( );
        printCollection( values );

        keys.remove( "John Doe" );
        values.remove( "unlisted" );

        System.out.println( "After John Doe and 1 unlisted are removed" );
        System.out.println( "\nThe map is: " );
        printMap( "phone1", phone1 );
    }
}
```

figure 6.33

An illustration using the `Map` interface

```
phone1.get("Jane Doe"): unlisted

The map is:
phone1:
Holly Doe: 213-555-1212
Jane Doe: unlisted
John Doe: 212-555-1212
Susan Doe: 617-555-1212

The keys are:
Holly Doe Jane Doe John Doe Susan Doe

The values are:
213-555-1212 unlisted 212-555-1212 617-555-1212
After John Doe and 1 unlisted are removed

The map is
phone1:
Holly Doe: 213-555-1212
Susan Doe: 617-555-1212
```

A typical use of `Map`

```
public static void listDuplicates(List<String> coll) {
    Map<String,Integer> count = new TreeMap<String,Integer>();

    for (String word : coll) {
        Integer occurs = count.get(word);
        if (occurs == null)
            count.put(word, 1);
        else
            count.put(word, occurs + 1);
    }

    for (Map.Entry<String,Integer> e : count.entrySet())
        if (e.getValue() >= 2)
            System.out.println(e.getKey() + " (" +
                               e.getValue() + ")");
}
```

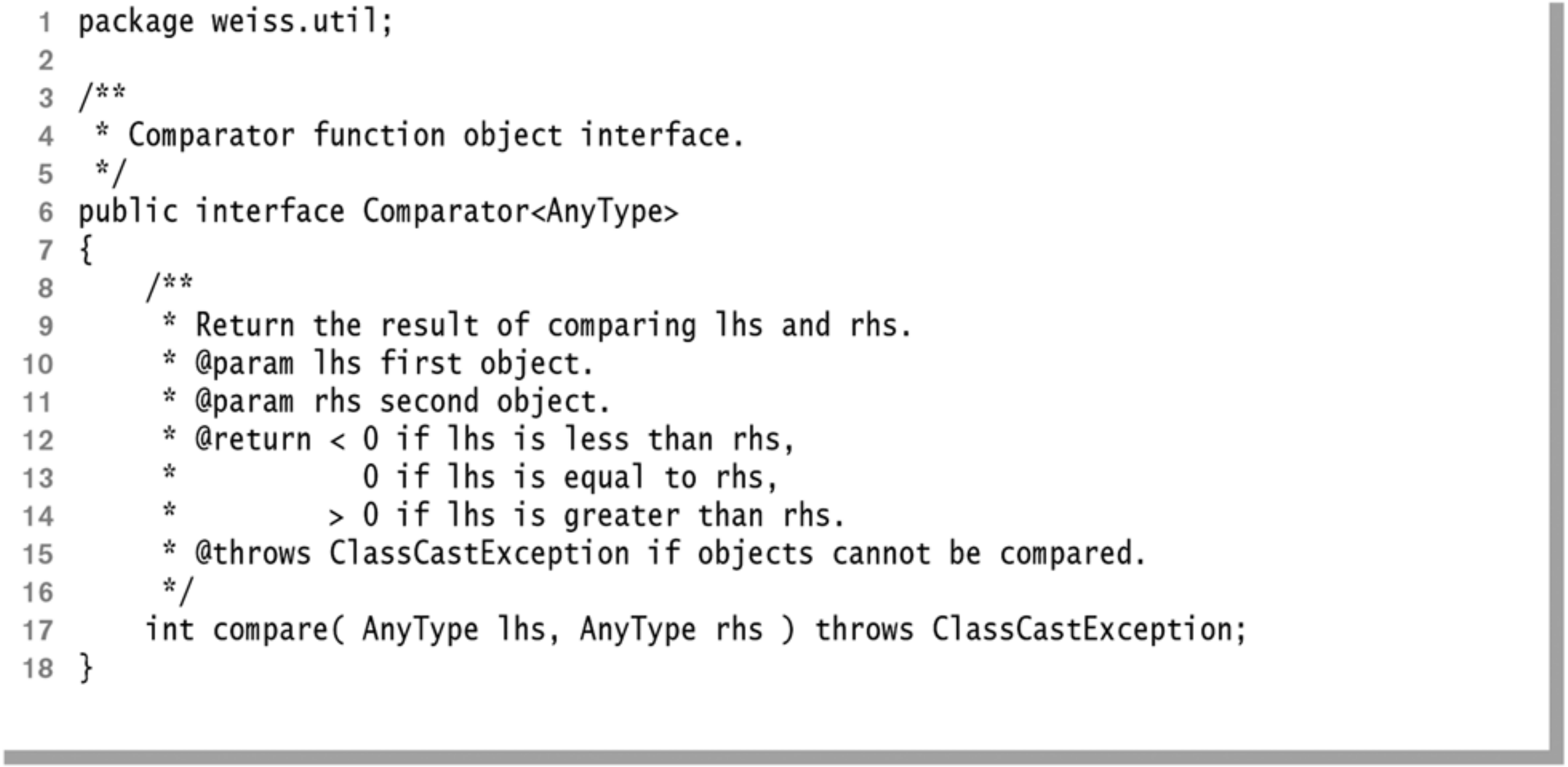
Ordering and sorting

There are two ways to define ordering of objects:

(1) Each class can define a *natural order* among its instances by implementing the `Comparable` interface.

(2) Arbitrary orders among different objects can be defined by *comparators*, or classes that implement the `Comparator` interface.

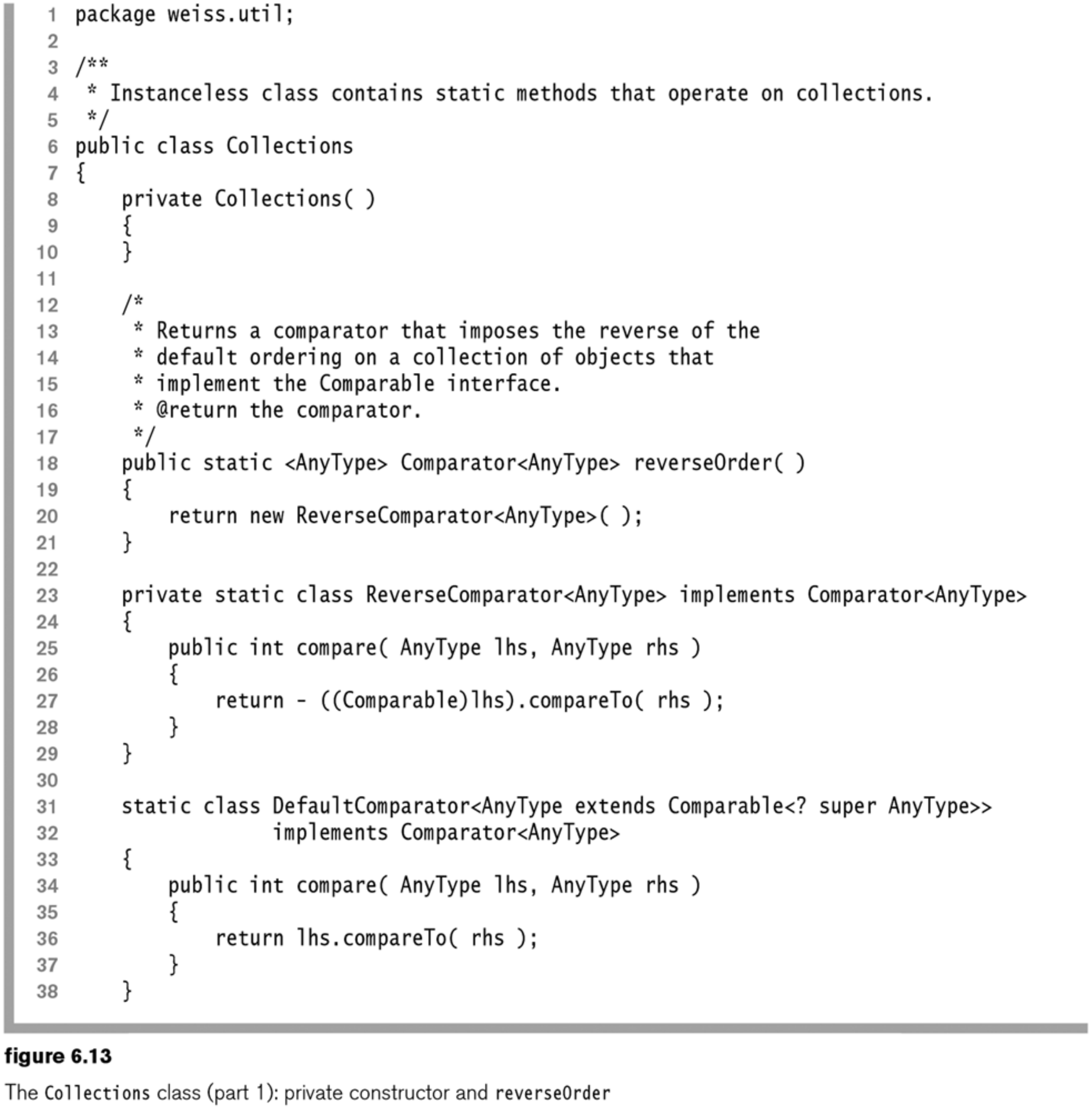
The `Comparator` interface



```
package weiss.util;

/**
 * Comparator function object interface.
 */
public interface Comparator<AnyType>
{
    /**
     * Return the result of comparing lhs and rhs.
     * @param lhs first object.
     * @param rhs second object.
     * @return < 0 if lhs is less than rhs,
     *           0 if lhs is equal to rhs,
     *         > 0 if lhs is greater than rhs.
     * @throws ClassCastException if objects cannot be compared.
     */
    int compare( AnyType lhs, AnyType rhs ) throws ClassCastException;
}
```

figure 6.12

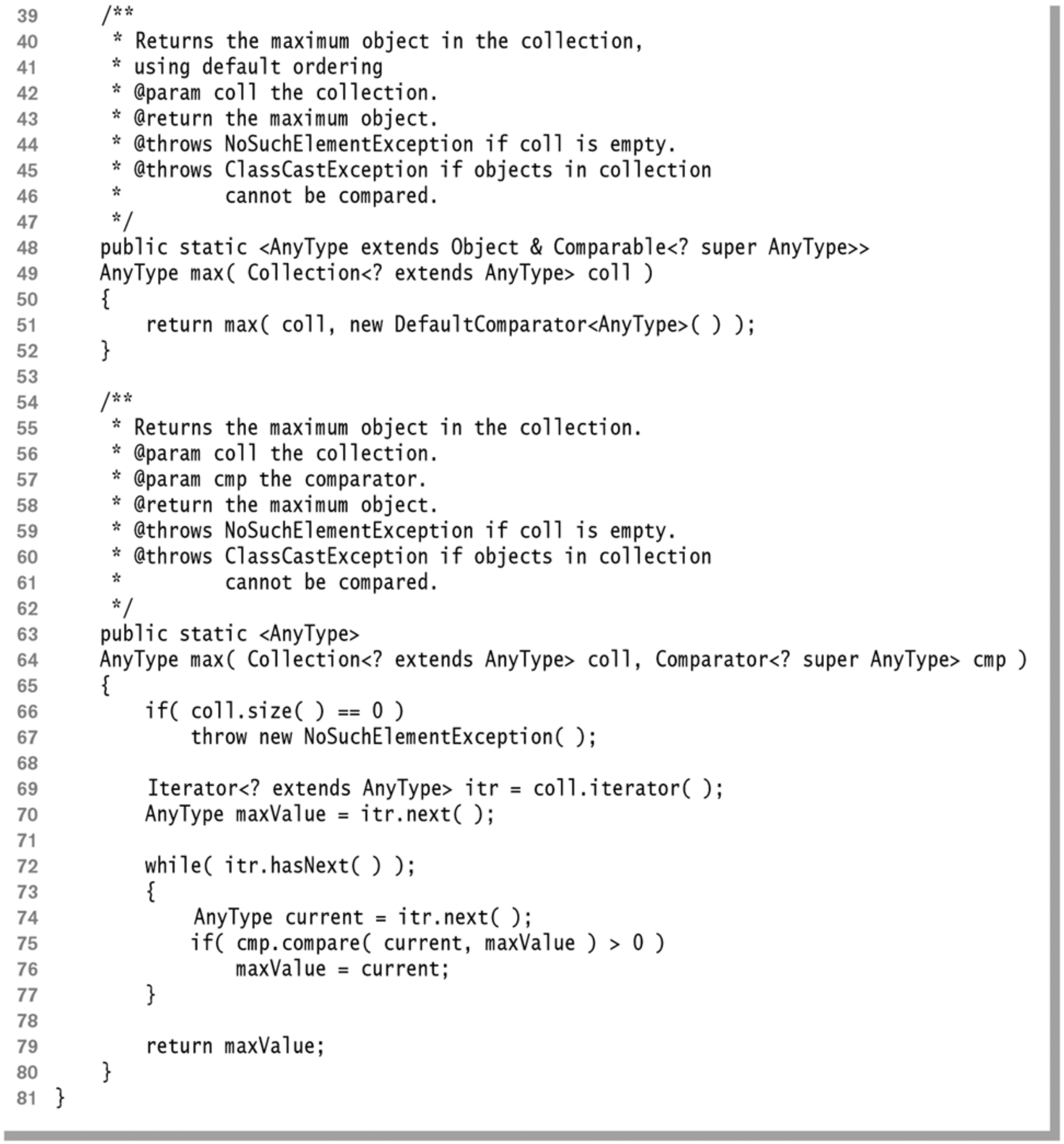
The `Comparator` interface, originally defined in `java.util` and rewritten for the `weiss.util` package



```
package weiss.util;

/**
 * Instanceless class contains static methods that operate on collections.
 */
public class Collections
{
    private Collections( )
    {
    }

    /*
     * Returns a comparator that imposes the reverse of the
     * default ordering on a collection of objects that
     * implement the Comparable interface.
     * @return the comparator.
     */
    public static <AnyType> Comparator<AnyType> reverseOrder( )
    {
        return new ReverseComparator<AnyType>( );
    }

    private static class ReverseComparator<AnyType> implements Comparator<AnyType>
    {
        public int compare( AnyType lhs, AnyType rhs )
        {
            return - ((Comparable)lhs).compareTo( rhs );
        }
    }

    static class DefaultComparator<AnyType extends Comparable<? super AnyType>>
                implements Comparator<AnyType>
    {
        public int compare( AnyType lhs, AnyType rhs )
        {
            return lhs.compareTo( rhs );
        }
    }
```

figure 6.13

The `Collections` class (part 1): private constructor and `reverseOrder`



```
    /**
     * Returns the maximum object in the collection,
     * using default ordering
     * @param coll the collection.
     * @return the maximum object.
     * @throws NoSuchElementException if coll is empty.
     * @throws ClassCastException if objects in collection
     *         cannot be compared.
     */
    public static <AnyType extends Object & Comparable<? super AnyType>>
    AnyType max( Collection<? extends AnyType> coll )
    {
        return max( coll, new DefaultComparator<AnyType>( ) );
    }

    /**
     * Returns the maximum object in the collection.
     * @param coll the collection.
     * @param cmp the comparator.
     * @return the maximum object.
     * @throws NoSuchElementException if coll is empty.
     * @throws ClassCastException if objects in collection
     *         cannot be compared.
     */
    public static <AnyType>
    AnyType max( Collection<? extends AnyType> coll, Comparator<? super AnyType> cmp )
    {
        if( coll.size( ) == 0 )
            throw new NoSuchElementException( );

        Iterator<? extends AnyType> itr = coll.iterator( );
        AnyType maxValue = itr.next( );

        while( itr.hasNext( ) );
        {
            AnyType current = itr.next( );
            if( cmp.compare( current, maxValue ) > 0 )
                maxValue = current;
        }

        return maxValue;
    }
}
```

figure 6.14

The `Collections` class (part 2): `max`

interface **SortedSet<E>** extends Set<E>

New methods:

```
E first()
E last()
SortedSet<E> headSet(E toElement)                <
SortedSet<E> tailSet(E fromElement)              >=
SortedSet<E> subSet(E fromElement, E toElement)  >=, <
Comparator<? super E> comparator()
```

A concrete implementation, for instance `TreeSet`, provides at least two constructors: one without parameters, and one with a `Comparator` parameter.

`interface` **`SortedMap<K,V>`** `extends Set`

New methods:

```
K first()
K last()
SortedMap<K,V> headSet(E toElement)
SortedMap<K,V> tailSet(E fromElement)
SortedMap<K,V> subMap(E fromElement, E toElement)
Comparator<? super K> comparator()
```

A concrete implementation, for instance `TreeMap`, provides at least two constructors: one without parameters, and one with a `Comparator` parameter.

The class `Collections`

```
public class Collections {
    public static void sort(List l);
    public static void sort(List l, Comparator comp);

    public static int binarySearch(List l, Object key);
    public static int binarySearch(List l, Object key,
                                   Comparator comp);

    public static Object min(Collection c);
    public static Object min(Collection c, Comparator c);
    public static Object max(Collection c);
    public static Object max(Collection c, Comparator c);

    public static void reverse(List l);
    public static void shuffle(List l);

    public static Comparator reverseOrder();
}
```

The class `Arrays`

```
public class Arrays {
    public static<T> List<T> asList(T... a);

    public static void sort(type[] a);
    public static<T> void sort(T[] a, Comparator comp)
    public static void sort(type[] a, int from, int to);
    public static<T> void sort(T[] a, int from, int to,
                               Comparator<? super T> comp);

    public static int binarySearch(type[] a, type key);
    public static int binarySearch(Object[] a, Object key,
                                   Comparator<? super T> comp);

    public static void fill(type[] a, type value);
    public static void fill(type[] a, int from, int to, type value);

    public static boolean equals(type[] a, type[] a2);
    public static boolean deepEquals(Object[], Object[] a2);
}
```

Choosing an implementation for `Set`

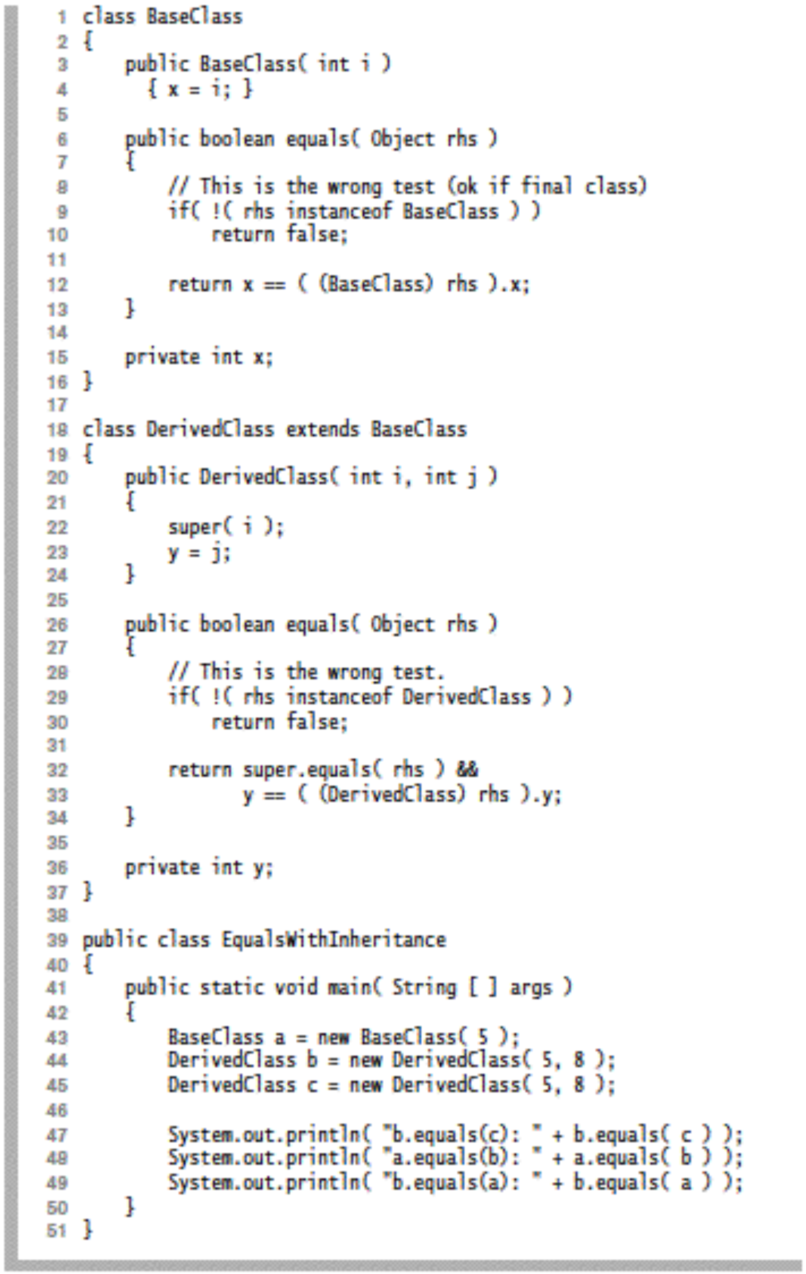
If the elements should maintain a certain order, then `TreeSet` should be used; otherwise, `HashSet`.

The `HashSet` implementation requires that the `equals` and `hashCode` methods are defined properly in the class of the elements.

Think of `hashCode` as providing a hint of where the items are stored in an array. If two objects are equal according to the `equals` method, they must return the same hash code. On the other hand, it is not required that two objects that are not equal return different hash codes.

figure 6.33

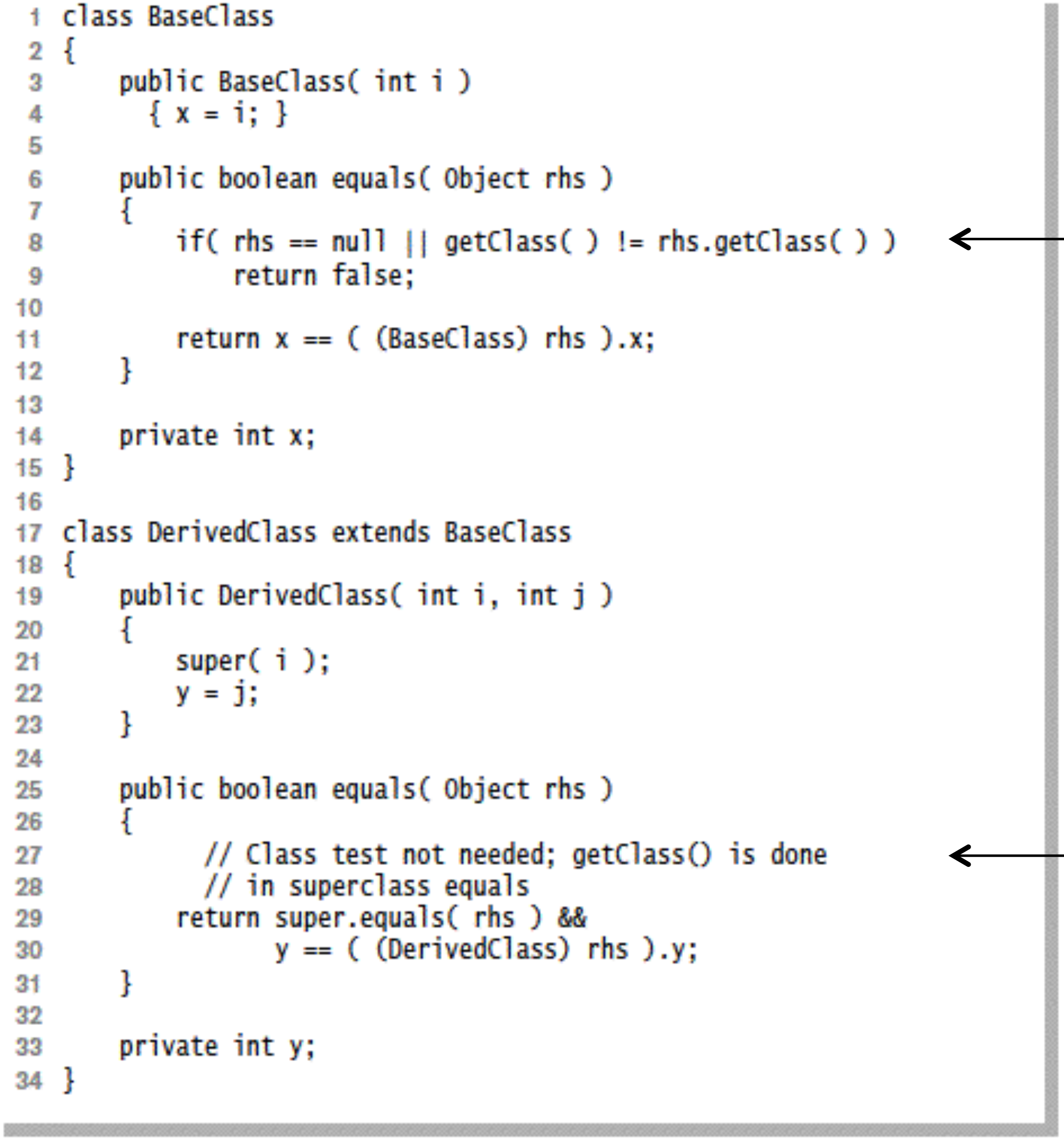
An illustration of a broken implementation of `equals`



```
class BaseClass
{
    public BaseClass( int i )
      { x = i; }

    public boolean equals( Object rhs )
    {
        // This is the wrong test (ok if final class)
        if( !( rhs instanceof BaseClass ) )
            return false;

        return x == ( (BaseClass) rhs ).x;
    }

    private int x;
}

class DerivedClass extends BaseClass
{
    public DerivedClass( int i, int j )
    {
        super( i );
        y = j;
    }

    public boolean equals( Object rhs )
    {
        // This is the wrong test.
        if( !( rhs instanceof DerivedClass ) )
            return false;

        return super.equals( rhs ) &&
               y == ( (DerivedClass) rhs ).y;
    }

    private int y;
}

public class EqualsWithInheritance
{
    public static void main( String [ ] args )
    {
        BaseClass a = new BaseClass( 5 );
        DerivedClass b = new DerivedClass( 5, 8 );
        DerivedClass c = new DerivedClass( 5, 8 );

        System.out.println( "b.equals(c): " + b.equals( c ) );
        System.out.println( "a.equals(b): " + a.equals( b ) );
        System.out.println( "b.equals(a): " + b.equals( a ) );
    }
}
```

Output:

```
b.equals(c): true
a.equals(b): true
b.equals(a): false
```



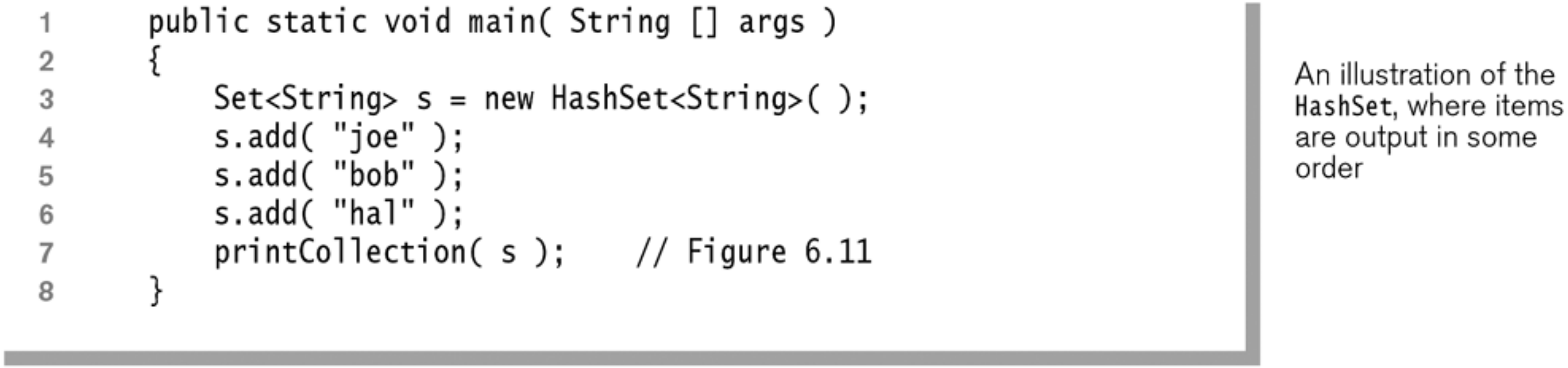
```
class BaseClass
{
    public BaseClass( int i )
      { x = i; }

    public boolean equals( Object rhs )
    {
        if( rhs == null || getClass( ) != rhs.getClass( ) )
            return false;

        return x == ( (BaseClass) rhs ).x;
    }

    private int x;
}

class DerivedClass extends BaseClass
{
    public DerivedClass( int i, int j )
    {
        super( i );
        y = j;
    }

    public boolean equals( Object rhs )
    {
          // Class test not needed; getClass() is done
          // in superclass equals
        return super.equals( rhs ) &&
               y == ( (DerivedClass) rhs ).y;
    }

    private int y;
}
```

figure 6.34

Correct implementation of `equals`

```
/**
 * Test program for HashSet.
 */
class IteratorTest
{
    public static void main( String [ ] args )
    {
        List<SimpleStudent> stud1 = new ArrayList<SimpleStudent>( );
        stud1.add( new SimpleStudent( "Bob", 0 ) );
        stud1.add( new SimpleStudent( "Joe", 1 ) );
        stud1.add( new SimpleStudent( "Bob", 2 ) ); // duplicate

           // Will only have 2 items, if hashCode is
           // implemented. Otherwise will have 3 because
           // duplicate will not be detected.
        Set<SimpleStudent>  stud2 = new HashSet<SimpleStudent>( stud1 );

        printCollection( stud1 ); // Bob Joe Bob
        printCollection( stud2 ); // Two items in unspecified order
    }
}

/**
 * Illustrates use of hashCode/equals for a user-defined class.
 * Students are ordered on basis of name only.
 */
class SimpleStudent
{
    String name;
    int id;

    public SimpleStudent( String n, int i )
      { name = n; id = i; }

    public String toString( )
      { return name + " " + id; }

    public boolean equals( Object rhs )
    {
        if( rhs == null || getClass( ) != rhs.getClass( ) )
            return false;

        SimpleStudent other = (SimpleStudent) rhs;
        return name.equals( other.name );
    }

    public int hashCode( )
      { return name.hashCode( ); }
}
```

Illustrates the `equals` and `hashCode` methods for use in `HashSet`



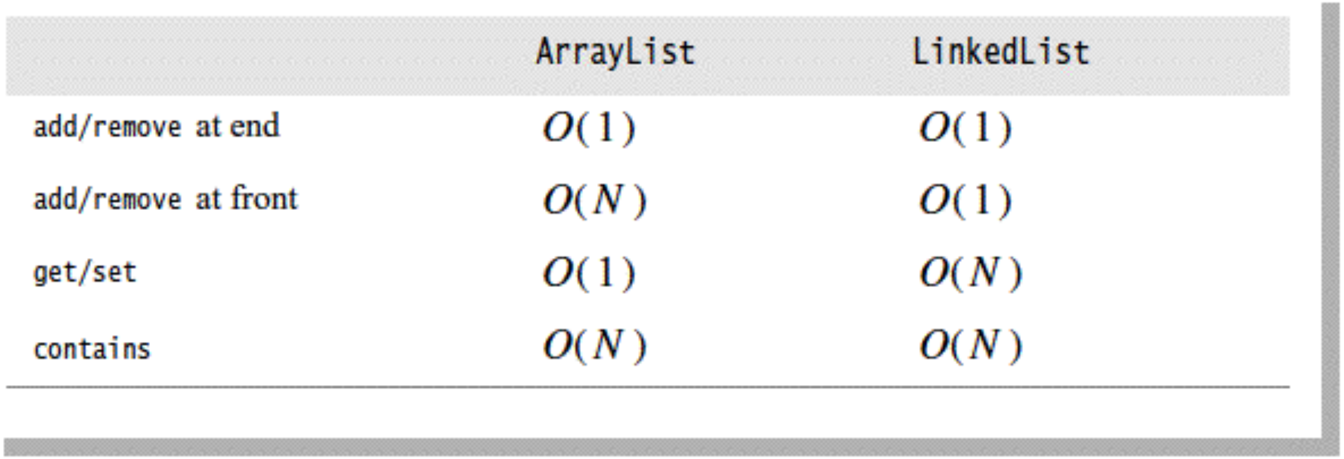
```
public static void main( String [] args )
{
    Set<String> s = new HashSet<String>( );
    s.add( "joe" );
    s.add( "bob" );
    s.add( "hal" );
    printCollection( s );    // Figure 6.11
}
```

An illustration of the `HashSet`, where items are output in some order

All items are printed, but the order that the items are printed is unknown.

Choosing an implementation for List

If indices are used often to access the elements and the size of the list does not vary much, then `ArrayList` should be used; otherwise `LinkedList`.



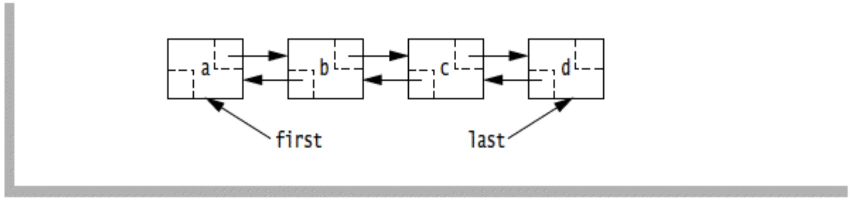
	`ArrayList`	`LinkedList`
`add/remove` at end	$O(1)$	$O(1)$
`add/remove` at front	$O(N)$	$O(1)$
`get/set`	$O(1)$	$O(N)$
`contains`	$O(N)$	$O(N)$

figure 6.21

Single-operation costs for `ArrayList` and `LinkedList`

figure 6.20

A doubly linked list

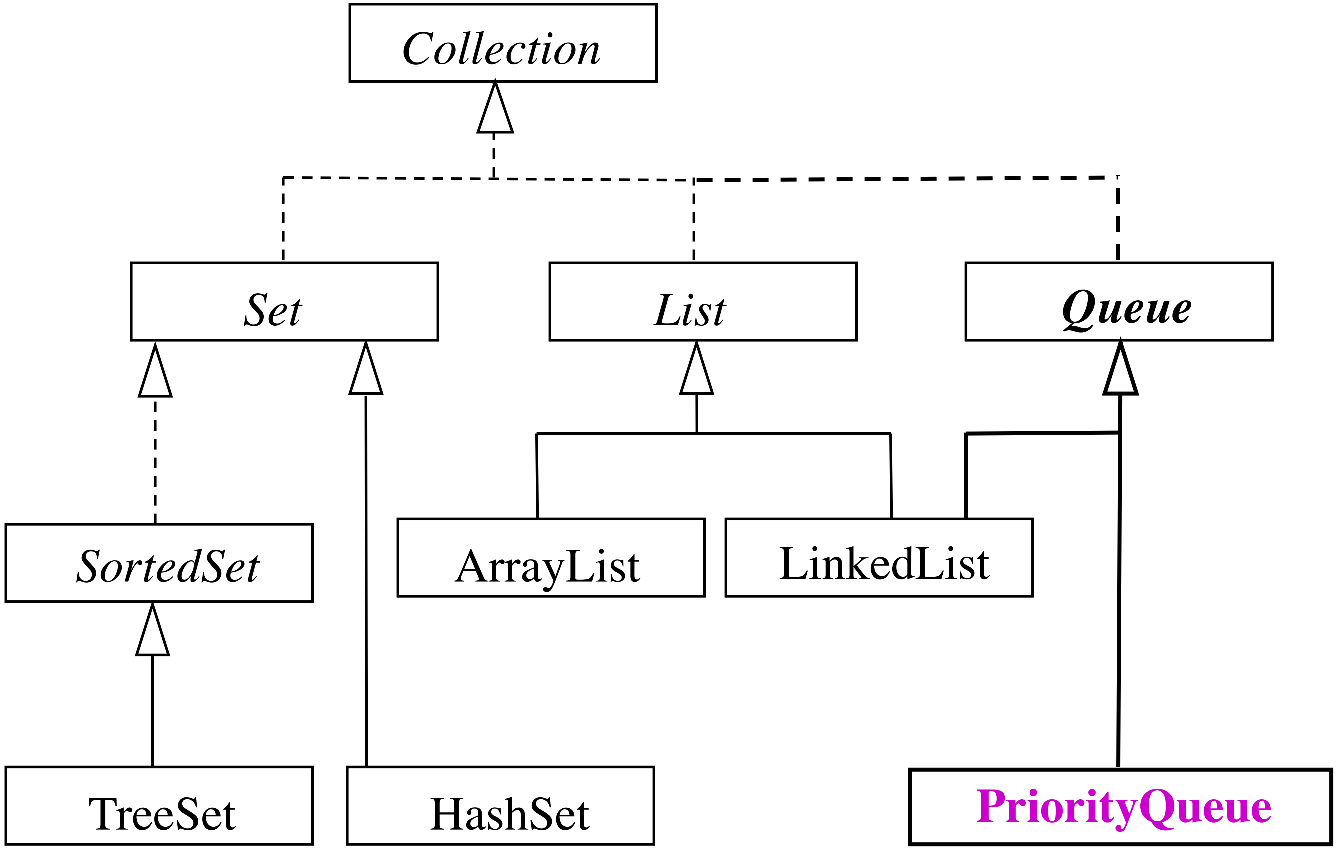


Choosing an implementation for Map

If the keys should maintain a certain order, then `TreeMap` should be used; otherwise, `HashMap`.

The `HashMap` implementation requires that the `equals` and `hashCode` methods defined properly in the class of the keys.

New collections in Java 5



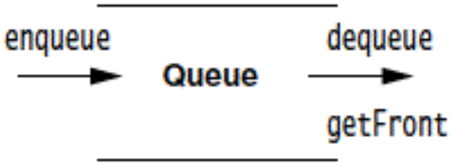
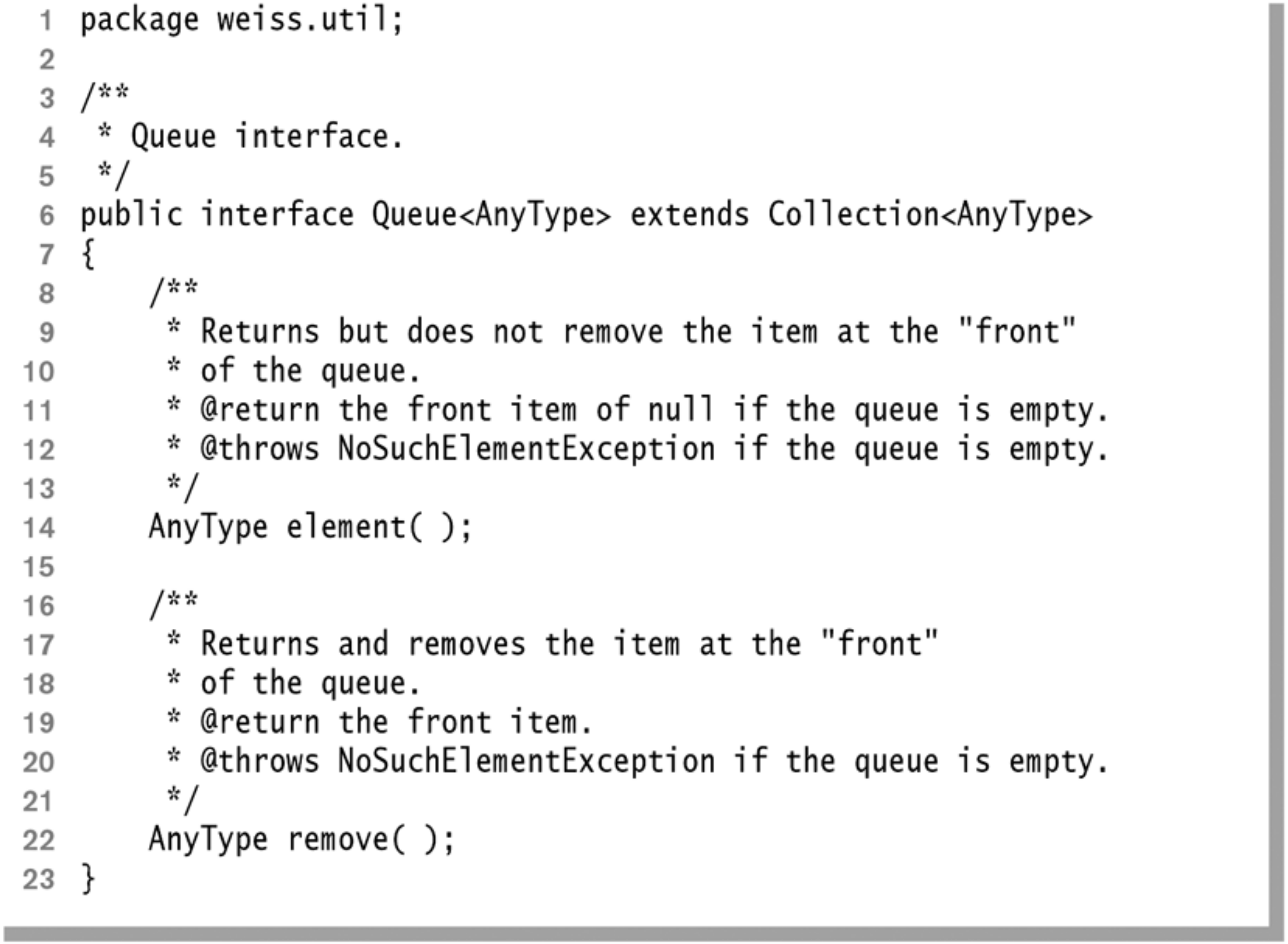


figure 6.27

The queue model: Input is by `enqueue`, output is by `getFront`, and deletion is by `dequeue`.



```
package weiss.util;

/**
 * Queue interface.
 */
public interface Queue<AnyType> extends Collection<AnyType>
{
    /**
     * Returns but does not remove the item at the "front"
     * of the queue.
     * @return the front item of null if the queue is empty.
     * @throws NoSuchElementException if the queue is empty.
     */
    AnyType element( );

    /**
     * Returns and removes the item at the "front"
     * of the queue.
     * @return the front item.
     * @throws NoSuchElementException if the queue is empty.
     */
    AnyType remove( );
}
```

figure 6.23

Possible `Queue` interface

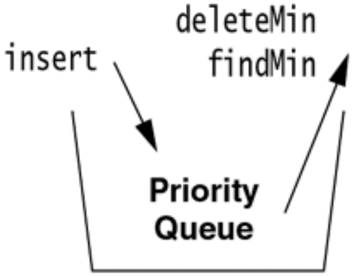
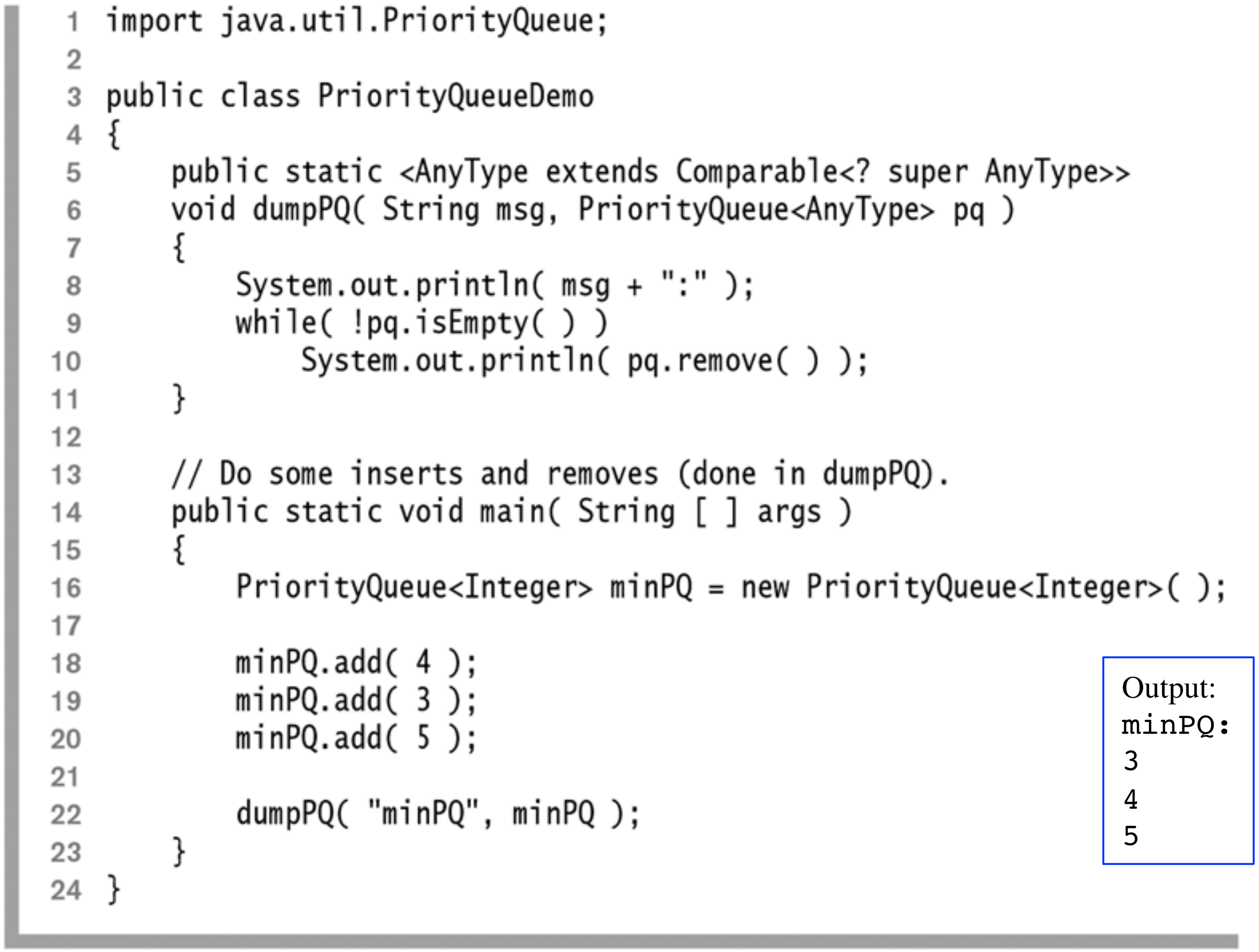


figure 6.34

The priority queue model: Only the minimum element is accessible.

figure 6.35

A routine to demonstrate the `PriorityQueue`



```
import java.util.PriorityQueue;

public class PriorityQueueDemo
{
    public static <AnyType extends Comparable<? super AnyType>>
    void dumpPQ( String msg, PriorityQueue<AnyType> pq )
    {
        System.out.println( msg + ":" );
        while( !pq.isEmpty( ) )
            System.out.println( pq.remove( ) );
    }

    // Do some inserts and removes (done in dumpPQ).
    public static void main( String [ ] args )
    {
        PriorityQueue<Integer> minPQ = new PriorityQueue<Integer>( );

        minPQ.add( 4 );
        minPQ.add( 3 );
        minPQ.add( 5 );

        dumpPQ( "minPQ", minPQ );
    }
}
```

Output:

```
minPQ:
3
4
5
```

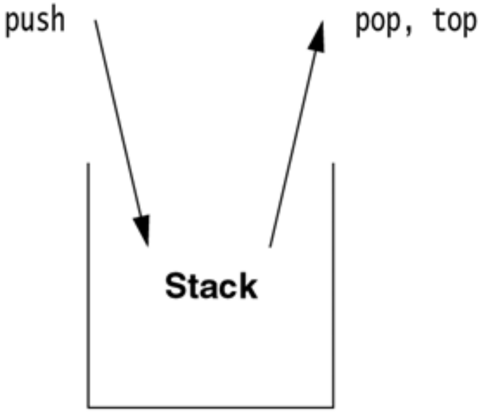


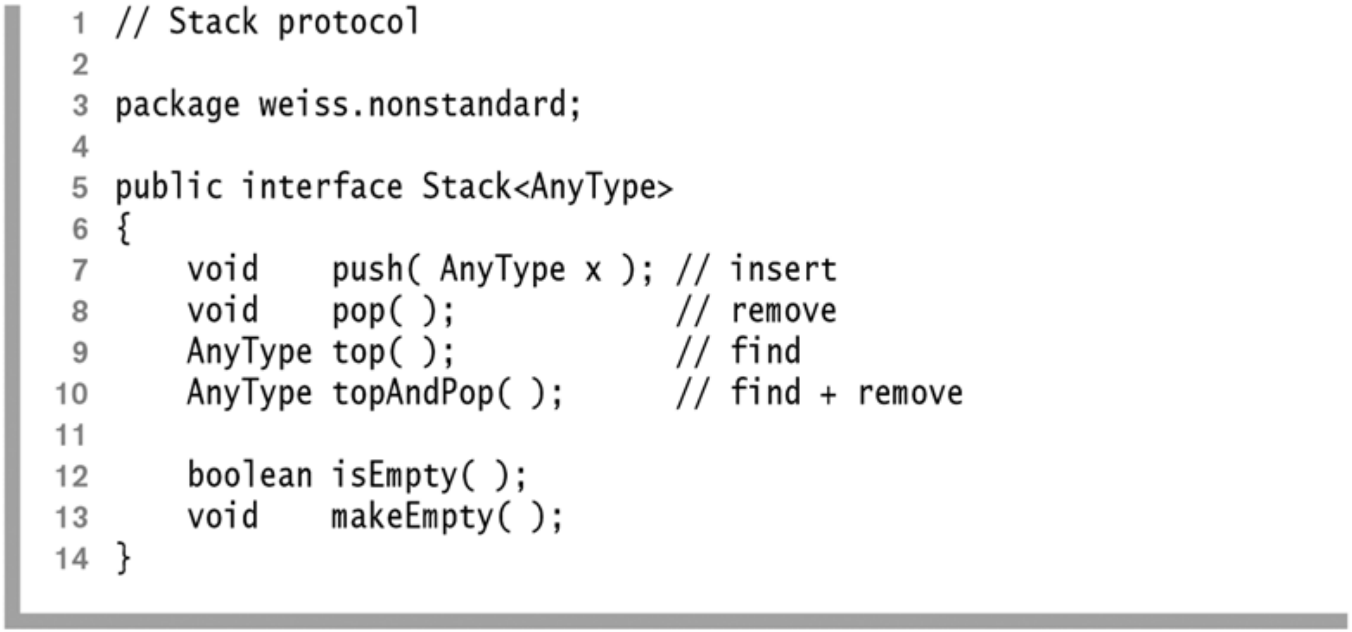
figure 6.20

The stack model: Input to a stack is by `push`, output is by `top`, and deletion is by `pop`.

Stack

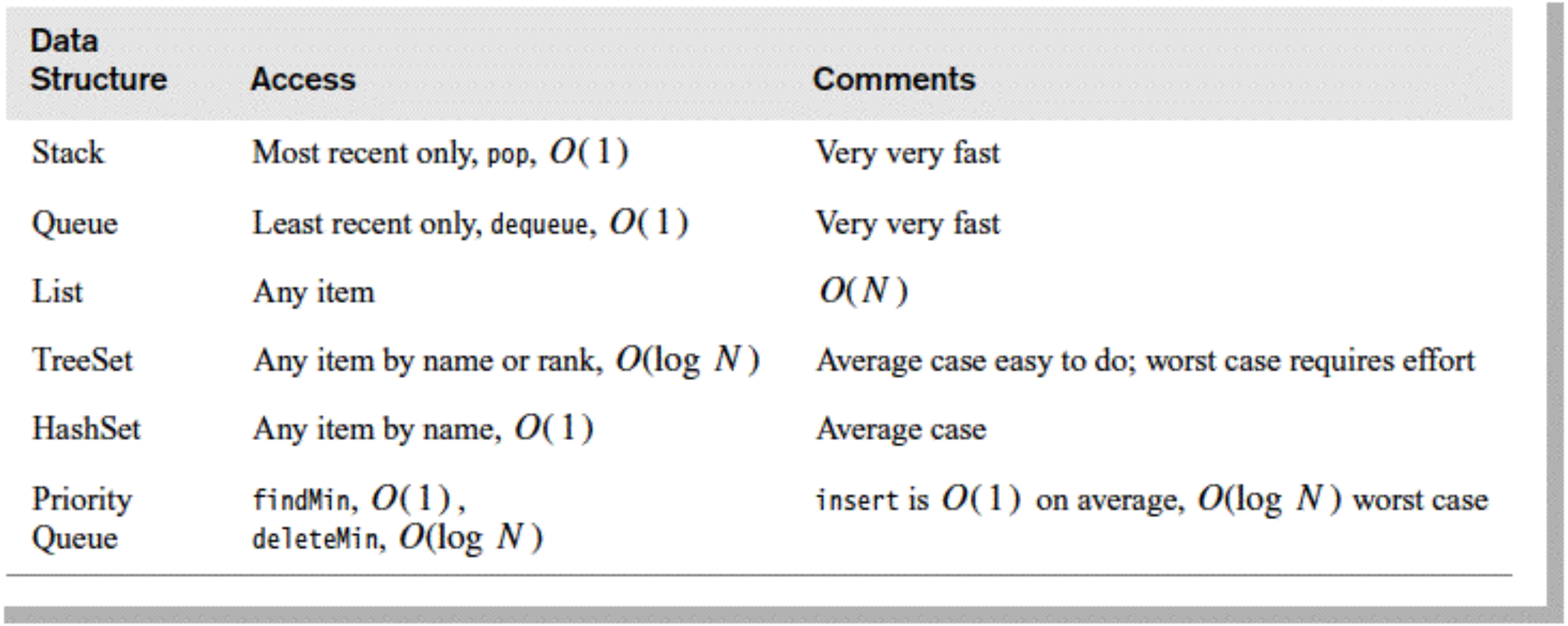
figure 6.21

Protocol for the stack



```
// Stack protocol

package weiss.nonstandard;

public interface Stack<AnyType>
{
    void    push( AnyType x ); // insert
    void    pop( );            // remove
    AnyType top( );            // find
    AnyType topAndPop( );      // find + remove

    boolean isEmpty( );
    void    makeEmpty( );
}
```

May be implemented using `ArrayList` or `LinkedList`



Data Structure	Access	Comments
Stack	Most recent only, `pop`, $O(1)$	Very very fast
Queue	Least recent only, `dequeue`, $O(1)$	Very very fast
List	Any item	$O(N)$
TreeSet	Any item by name or rank, $O(\log N)$	Average case easy to do; worst case requires effort
HashSet	Any item by name, $O(1)$	Average case
Priority Queue	`findMin`, $O(1)$, `deleteMin`, $O(\log N)$	`insert` is $O(1)$ on average, $O(\log N)$ worst case

figure 6.43

A summary of some data structures