17.1 Introduction

*Java provides many classes for performing text I/O and binary I/O.*

Files can be classified as either text or binary. A file that can be processed (read, created, or

modified) using a text editor such as Notepad on Windows or vi on UNIX is called a *text file*.

All the other files are called *binary files*. You cannot read binary files using a text editor—they

are designed to be read by programs. For example, Java source programs are text files and can

be read by a text editor, but Java class files are binary files and are read by the JVM.

you can envision a text file as consisting

of a sequence of characters and a binary file as consisting of a sequence of bits. Characters

in a text file are encoded using a character encoding scheme such as ASCII or Unicode.

The advantage of binary files is that they are

more efficient to process than text files.

Java offers many classes for performing file input and output. These can be categorized as

*text I/O classes* and *binary I/O classes.*

17.2 How Is Text I/O Handled in Java?

*Text data are read using the* **Scanner** *class and written using the* **PrintWriter** *class.*

Recall that a **File** object encapsulates the properties of a file or a path but does not contain

the methods for reading/writing data from/to a file. In order to perform I/O, you need to

create objects using appropriate Java I/O classes. The objects contain the methods for reading/

writing data from/to a file.

PrintWriter output = **new** PrintWriter(**"temp.txt"**);

You can now invoke the **print** method on the object to write a string to the file.

There are many I/O classes for various purposes. In general, these can be classified as input

classes and output classes. An *input class* contains the methods to read data, and an *output*

*class* contains the methods to write data. **PrintWriter** is an example of an output class, and

**Scanner** is an example of an input class.

An input object reads a *stream* of data from

a file, and an output object writes a stream of data to a file. An input object is also called an

*input stream* and an output object an *output stream*.

17.3 Text I/O vs. Binary I/O

*Binary I/O does not involve encoding or decoding and thus is more efficient than text I/O.*

Computers do not differentiate between binary files and text files. All files are stored in binary

format, and thus all files are essentially binary files. Text I/O is built upon binary I/O to provide

a level of abstraction for character encoding and decoding, as shown in Figure 17.2a.

Encoding and decoding are automatically performed for text I/O. The JVM converts Unicode

to a file-specific encoding when writing a character, and it converts a file-specific encoding

to Unicode when reading a character.

If you write a numeric value to a file using binary

I/O, the exact value in the memory is copied into the file.

In general, you should use text input to read a file created by a text editor or a text output

program, and use binary input to read a file created by a Java binary output program.

Binary I/O is more efficient than text I/O, because binary I/O does not require encoding

and decoding. Binary files are independent of the encoding scheme on the host machine and

thus are portable. Java programs on any machine can read a binary file created by a Java program.

This is why Java class files are binary files. Java class files can run on a JVM on any

machine.

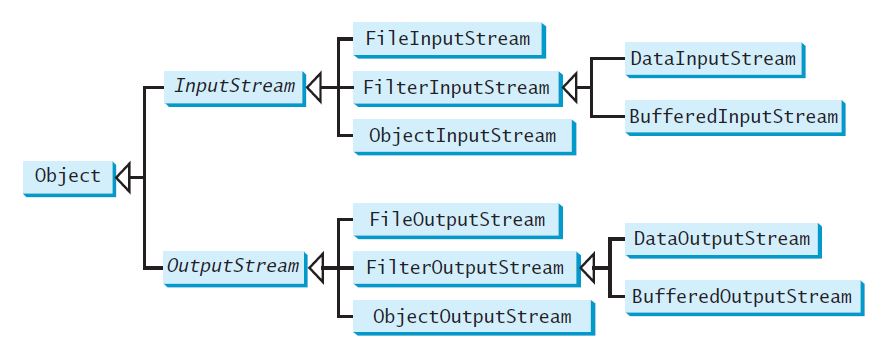
17.4 Binary I/O Classes

*The abstract* **InputStream** *is the root class for reading binary data, and the abstract*

**OutputStream** *is the root class for writing binary data.*

The design of the Java I/O classes is a good example of applying inheritance, where common

operations are generalized in superclasses, and subclasses provide specialized operations.

**InputStream** is the root for binary input classes, and **OutputStream** is the root for binary output classes.

**Note**

All the methods in the binary I/O classes are declared to throw **java.io.IOException**

or a subclass of **java.io.IOException**.

17.4.1 **FileInputStream**/**FileOutputStream**

**FileInputStream**/**FileOutputStream** is for reading/writing bytes from/to files.

All the methods in these classes are inherited from **InputStream** and **OutputStream**.

**FileInputStream**/**FileOutputStream** does not introduce new methods. To construct a

**FileInputStream**,

A **java.io.FileNotFoundException** will occur if you attempt to create a

**FileInputStream** with a nonexistent file.

To construct a **FileOutputStream**,

If the file does not exist, a new file will be created. If the file already exists, the first two

constructors will delete the current content of the file. To retain the current content and append

new data into the file, use the last two constructors and pass **true** to the **append** parameter.

Almost all the methods in the I/O classes throw **java.io.IOException**. Therefore, you

have to declare to throw **java.io.IOException** in the method or place the code in a trycatch

block,

**Tip**

When a stream is no longer needed, always close it using the **close()** method or

automatically close it using a try-with-resource statement. Not closing streams may cause

data corruption in the output file, or other programming errors.

**Note**

The root directory for the file is the classpath directory.

**Note**

An instance of **FileInputStream** can be used as an argument to construct a **Scanner**,

and an instance of **FileOutputStream** can be used as an argument to construct a

**PrintWriter**. You can create a **PrintWriter** to append text into a file using  
**new** PrintWriter(**new** FileOutputStream(**"temp.txt"**, **true**));

If **temp.txt** does not exist, it is created. If **temp.txt** already exists, new data are

appended to the file.

17.4.2 **FilterInputStream**/**FilterOutputStream**

*Filter streams* are streams that filter bytes for some purpose. The basic byte input stream

provides a **read** method that can be used only for reading bytes. If you want to read

integers, doubles, or strings, you need a filter class to wrap the byte input stream. Using a

filter class enables you to read integers, doubles, and strings instead of bytes and characters.

**FilterInputStream** and **FilterOutputStream** are the base classes for filtering

data. When you need to process primitive numeric types, use **DataInputStream** and

**DataOutputStream** to filter bytes.

17.4.3 **DataInputStream**/**DataOutputStream**

**DataInputStream** reads bytes from the stream and converts them into appropriate

primitive-type values or strings. **DataOutputStream** converts primitive-type values or

strings into bytes and outputs the bytes to the stream.

**DataInputStream** extends **FilterInputStream** and implements the **DataInput**

interface,

**DataInputStream** filters an input stream of bytes into primitive data-type values and strings.

**DataInputStream** implements the methods defined in the **DataInput** interface to read

primitive data-type values and strings. **DataOutputStream** implements the methods defined

in the **DataOutput** interface to write primitive data-type values and strings. Primitive values

are copied from memory to the output without any conversions.

Characters and Strings in Binary I/O

A Unicode character consists of two bytes. The **writeChar(char c)** method writes the

Unicode of character **c** to the output. The **writeChars(String s)** method writes the Unicode

for each character in the string **s** to the output. The **writeBytes(String s)** method

writes the lower byte of the Unicode for each character in the string **s** to the output. The high

byte of the Unicode is discarded. The **writeBytes** method is suitable for strings that consist  
of ASCII characters, since an ASCII code is stored only in the lower byte of a Unicode. If a

string consists of non-ASCII characters, you have to use the **writeChars** method to write

the string.

The **writeUTF(String s)** method writes two bytes of length information to the

output stream, followed by the modified UTF-8 representation of every character in the

string **s**. UTF-8 is a coding scheme that allows systems to operate with both ASCII and

Unicode. Most operating systems use ASCII. Java uses Unicode. The ASCII character set

is a subset of the Unicode character set. Since most applications need only the ASCII character

set, it is a waste to represent an 8-bit ASCII character as a 16-bit Unicode character.

The modified UTF-8 scheme stores a character using one, two, or three bytes. Characters

are coded in one byte if their code is less than or equal to **0x7F**, in two bytes if their code is

greater than **0x7F** and less than or equal to **0x7FF**, or in three bytes if their code is greater

than **0x7FF**.

The initial bits of a UTF-8 character indicate whether a character is stored in one byte, two

bytes, or three bytes. If the first bit is **0**, it is a one-byte character. If the first bits are **110**, it

is the first byte of a two-byte sequence.

The **writeUTF(String s)** method converts a string into a series of bytes in the UTF-8

format and writes them into an output stream. The **readUTF()** method reads a string that has

been written using the **writeUTF** method.

The UTF-8 format has the advantage of saving a byte for each ASCII character, because a

Unicode character takes up two bytes and an ASCII character in UTF-8 only one byte. If most

of the characters in a long string are regular ASCII characters, using UTF-8 is more efficient.

17.4.4 **BufferedInputStream**/**BufferedOutputStream**

**BufferedInputStream**/**BufferedOutputStream** can be used to speed up input and

output by reducing the number of disk reads and writes. Using **BufferedInputStream**,

the whole block of data on the disk is read into the buffer in the memory once. The individual

data are then delivered to your program from the buffer, as shown in Figure 17.12a.

Using **BufferedOutputStream**, the individual data are first written to the buffer in the

memory. When the buffer is full, all data in the buffer are written to the disk once,

**BufferedInputStream**/**BufferedOutputStream** does not contain new methods. All

the methods in **BufferedInputStream**/**BufferedOutputStream** are inherited from the

**InputStream**/**OutputStream** classes. **BufferedInputStream**/**BufferedOutputStream**

manages a buffer behind the scene and automatically reads/writes data from/to disk on

demand.

You

If no buffer size is specified, the default size is **512** bytes.

**Tip**

You should always use buffered I/O to speed up input and output. For small files, you

may not notice performance improvements. However, for large files—over 100 MB—

you will see substantial improvements using buffered I/O.

17.6 Object I/O

**ObjectInputStream***/***ObjectOutputStream** *classes can be used to read/write*

*serializable objects.*

**DataInputStream**/**DataOutputStream** enables you to perform I/O for primitive-type values

and strings. **ObjectInputStream**/**ObjectOutputStream** enables you to perform I/O  
for objects in addition to primitive-type values and strings. Since **ObjectInputStream**/

**ObjectOutputStream** contains all the functions of **DataInputStream**/

**DataOutputStream**, you can replace **DataInputStream**/**DataOutputStream** completely

with **ObjectInputStream**/**ObjectOutputStream**.

**ObjectInputStream** extends **InputStream** and implements **ObjectInput** and

**ObjectStreamConstants**, as shown in Figure 17.16. **ObjectInput** is a subinterface of

**DataInput** (**DataInput** is shown in Figure 17.9). **ObjectStreamConstants** contains the

constants to support **ObjectInputStream**/**ObjectOutputStream**.

**ObjectInputStream** can read objects, primitive-type values, and strings.

**ObjectOutputStream** extends **OutputStream** and implements **ObjectOutput** and

**ObjectStreamConstants**,  
**ObjectOutput** is a subinterface of

**DataOutput**

**ObjectOutputStream** can write objects, primitive-type values, and strings.

You can wrap an **ObjectInputStream**/**ObjectOutputStream** on any **InputStream**/

**OutputStream** using the following constructors:

// Create an ObjectInputStream

**public** ObjectInputStream(InputStream in)

// Create an ObjectOutputStream

**public** ObjectOutputStream(OutputStream out)

17.6.1 The **Serializable** Interface

Not every object can be written to an output stream. Objects that can be so written are said

to be *serializable*. A serializable object is an instance of the **java.io.Serializable** interface,

so the object’s class must implement **Serializable**.

The **Serializable** interface is a marker interface. Since it has no methods, you don’t

need to add additional code in your class that implements **Serializable**. Implementing this

interface enables the Java serialization mechanism to automate the process of storing objects

and arrays.

Java provides a built-in mechanism to automate the process of writing objects. This

process is referred as *object serialization*, which is implemented in **ObjectOutputStream**.

In contrast, the process of reading objects is referred as *object deserialization*, which is implemented

in **ObjectInputStream**.

Many classes in the Java API implement **Serializable**. All the wrapper classes for primitive

type values, **java.math.BigInteger**, **java.math.BigDecimal**, **java.lang.String**,

**java.lang.StringBuilder**, **java.lang.StringBuffer**, **java.util.Date**, and

**java.util.ArrayList** implement **java.io.Serializable**. Attempting to

store an object that does not support the **Serializable** interface would cause a

**NotSerializableException**.

When a serializable object is stored, the class of the object is encoded; this includes the

class name and the signature of the class, the values of the object’s instance variables, and the

closure of any other objects referenced by the object. The values of the object’s static variables

are not stored.

**Note**

**Nonserializable fields**

If an object is an instance of **Serializable** but contains nonserializable instance

data fields, can it be serialized? The answer is no. To enable the object to be serialized,

mark these data fields with the **transient** keyword to tell the JVM to ignore them

when writing the object to an object stream.

**Note**

**Duplicate objects**

If an object is written to an object stream more than once, will it be stored in multiple

copies? No, it will not. When an object is written for the first time, a serial number is

created for it. The JVM writes the complete contents of the object along with the serial

number into the object stream. After the first time, only the serial number is stored if the  
same object is written again. When the objects are read back, their references are the

same since only one object is actually created in the memory.

17.6.2 Serializing Arrays

An array is serializable if all its elements are serializable. An entire array can be saved into a

file using **writeObject** and later can be restored using **readObject**.

17.7 Random-Access Files

*Java provides the* **RandomAccessFile** *class to allow data to be read from and*

*written to at any locations in the file.*

All of the streams you have used so far are known as *read-only* or *write-only* streams. These

streams are called *sequential streams*. A file that is opened using a sequential stream is called

a *sequential-access file*. The contents of a sequential-access file cannot be updated. However,

it is often necessary to modify files. Java provides the **RandomAccessFile** class to allow

data to be read from and written to at any locations in a file. A file that is opened using the

**RandomAccessFile** class is known as a *random-access file*.

The **RandomAccessFile** class implements the **DataInput** and **DataOutput** interfaces,  
defines the methods

for reading primitive-type values and strings (e.g., **readInt**, **readDouble**, **readChar**,

**readBoolean**, **readUTF**) and the **DataOutput** interface (see Figure 17.10) defines the

methods for writing primitive-type values and strings (e.g., **writeInt**, **writeDouble**,

**writeChar**, **writeBoolean**, **writeUTF**).

When creating a **RandomAccessFile**, you can specify one of two modes: **r** or **rw**. Mode

**r** means that the stream is read-only, and mode **rw** indicates that the stream allows both read

and write.

**Tip**

If the file is not intended to be modified, open it with the **r** mode. This prevents unintentional

modification of the file.

A random-access file consists of a sequence of bytes. A special marker called a *file pointer*

is positioned at one of these bytes. A read or write operation takes place at the location of the

file pointer. When a file is opened, the file pointer is set at the beginning of the file. When you

read or write data to the file, the file pointer moves forward to the next data item.

**CHAPTER SUMMARY**

**1.** I/O can be classified into *text I/O* and *binary I/O*. Text I/O interprets data in sequences

of characters. Binary I/O interprets data as raw binary values. How text is stored in a file

depends on the encoding scheme for the file. Java automatically performs encoding and

decoding for text I/O.

**2.** The **InputStream** and **OutputStream** classes are the roots of all binary I/O

classes. **FileInputStream/FileOutputStream** associates a file for input/output.

**BufferedInputStream**/**BufferedOutputStream** can be used to wrap any binary

I/O stream to improve performance. **DataInputStream**/**DataOutputStream** can be

used to read/write primitive values and strings.

**3. ObjectInputStream**/**ObjectOutputStream** can be used to read/write objects in

addition to primitive values and strings. To enable object *serialization*, the object’s

defining class must implement the **java.io.Serializable** marker interface.

**4.** The **RandomAccessFile** class enables you to read and write data to a file. You can

open a file with the **r** mode to indicate that it is read-only or with the **rw** mode to indicate

that it is updateable. Since the **RandomAccessFile** class implements **DataInput**

and **DataOutput** interfaces, many methods in **RandomAccessFile** are the same as

those in **DataInputStream** and **DataOutputStream**.

18.1 Introduction

*Recursion is a technique that leads to elegant solutions to problems that are difficult*

*to program using simple loops.*

An intuitive and effective solution is

to use recursion by searching the files in the subdirectories recursively.

To use recursion is to program using *recursive methods*—that is, to use methods that

invoke themselves. Recursion is a useful programming technique. In some cases, it enables

you to develop a natural, straightforward, simple solution to an otherwise difficult problem.

*A recursive method is one that invokes itself.*

**Pedagogical Note**

It is simpler and more efficient to implement the **factorial** method using a loop.

However, we use the recursive **factorial** method here to demonstrate the concept

of recursion.

18.4 Problem Solving Using Recursion

*If you think recursively, you can solve many problems using recursion.*

All recursive methods have

the following characteristics:

■ The method is implemented using an **if-else** or a **switch** statement that leads to

different cases.

■ One or more base cases (the simplest case) are used to stop recursion.

■ Every recursive call reduces the original problem, bringing it increasingly closer to a

base case until it becomes that case.

In general, to solve a problem using recursion, you break it into subproblems. Each subproblem

is the same as the original problem but smaller in size. You can apply the same

approach to each subproblem to solve it recursively.

The problem of checking

whether a string is a palindrome can be divided into two subproblems:

■ Check whether the first character and the last character of the string are equal.

■ Ignore the two end characters and check whether the rest of the substring is a

palindrome.

The second subproblem is the same as the original problem but smaller in size. There are two

base cases: (1) the two end characters are not the same, and (2) the string size is **0** or **1**. In case

1, the string is not a palindrome; in case 2, the string is a palindrome.

18.5.1 Recursive Selection Sort

Selection sort was introduced in Section 7.11. Recall that it finds the smallest element in the

list and swaps it with the first element. It then finds the smallest element remaining and swaps

it with the first element in the remaining list, and so on until the remaining list contains only

a single element. The problem can be divided into two subproblems:

■ Find the smallest element in the list and swap it with the first element.

■ Ignore the first element and sort the remaining smaller list recursively.

The base case is that the list contains only one element.

18.5.2 Recursive Binary Search  
For binary search to work, the elements in the

array must be in increasing order. The binary search first compares the key with the element

in the middle of the array. Consider the following three cases:

■ Case 1: If the key is less than the middle element, recursively search for the key in

the first half of the array.

■ Case 2: If the key is equal to the middle element, the search ends with a match.

■ Case 3: If the key is greater than the middle element, recursively search for the key

in the second half of the array.

*Recursive methods are efficient for solving problems with recursive structures.*

18.9 Recursion vs. Iteration

*Recursion is an alternative form of program control. It is essentially repetition without*

*a loop.*

When you use loops, you specify a loop body. The repetition of the loop body is controlled

by the loop control structure. In recursion, the method itself is called repeatedly. A selection

statement must be used to control whether to call the method recursively or not.

Recursion bears substantial overhead. Each time the program calls a method, the system

must allocate memory for all of the method’s local variables and parameters. This can consume

considerable memory and requires extra time to manage the memory.

Any problem that can be solved recursively can be solved nonrecursively with iterations.

Recursion has some negative aspects: it uses up too much time and too much memory. Why,

then, should you use it? In some cases, using recursion enables you to specify a clear, simple

solution for an inherently recursive problem that would otherwise be difficult to obtain.

Examples are the directory-size problem, the Tower of Hanoi problem, and the fractal problem,

which are rather difficult to solve without using recursion.

The decision whether to use recursion or iteration should be

**Note**

Recursive programs can run out of memory, causing a **StackOverflowError**.

**Tip**

If you are concerned about your program’s performance, avoid using recursion, because

it takes more time and consumes more memory than iteration. In general, recursion

can be used to solve the inherent recursive problems such as Tower of Hanoi, recursive

directories, and Sierpinski triangles.

**CHAPTER SUMMARY**

**1.** A *recursive method* is one that directly or indirectly invokes itself. For a recursive

method to terminate, there must be one or more *base cases*.

**2.** *Recursion* is an alternative form of program control. It is essentially repetition without

a loop control. It can be used to write simple, clear solutions for inherently recursive

problems that would otherwise be difficult to solve.

**3.** Sometimes the original method needs to be modified to receive additional parameters

in order to be invoked recursively. A *recursive helper method* can be defined for this

purpose.

**4.** Recursion bears substantial overhead. Each time the program calls a method, the system

must allocate memory for all of the method’s local variables and parameters. This can

consume considerable memory and requires extra time to manage the memory.

**5.** A recursive method is said to be *tail recursive* if there are no pending operations to be

performed on return from a recursive call. Some compilers can optimize tail recursion

to reduce stack size.

19.1 Introduction

*Generics enable you to detect errors at compile time rather than at runtime.*

*Generics* let you parameterize types. With this capability, you can define a

class or a method with generic types that the compiler can replace with concrete types.

The key benefit of generics is to enable errors to be detected at compile time rather than

at runtime. A generic class or method permits you to specify allowable types of objects that

the class or method can work with. If you attempt to use an incompatible object, the compiler

will detect that error.

This chapter explains

19.2 Motivations and Benefits

*The motivation for using Java generics is to detect errors at compile time.*

Here, **<T>** represents a *formal generic type*, which can be replaced later with an *actual*

*concrete type*. Replacing a generic type is called a *generic instantiation*. By convention, a

single capital letter such as **E** or **T** is used to denote a formal generic type.

*autoboxing*, Automatic Conversion between Primitive Types and Wrapper

Class Types.

Casting is not needed to retrieve a value from a list with a specified element type, because

the compiler already knows the element type.

If the elements are of wrapper types, such as **Integer**, **Double**, and **Character**, you can

directly assign an element to a primitive type variable. This is called *autounboxing*,

19.3 Defining Generic Classes and Interfaces

*A generic type can be defined for a class or interface. A concrete type must be*

*specified when using the class to create an object or using the class or interface to*

*declare a reference variable.*

Instead of using a generic type, you could simply make the type element **Object**, which can

accommodate any object type. However, using generic types can improve software reliability

and readability, because certain errors can be detected at compile time rather than at runtime.

**Caution**

To create a stack of strings, you use **new GenericStack<String>()** or **new**

**GenericStack<>()**. This could mislead you into thinking that the constructor of

**GenericStack** should be defined as

**public** GenericStack<E>()

This is wrong. It should be defined as

**public** GenericStack()

**Note**

Occasionally, a generic class may have more than one parameter. In this case, place

the parameters together inside the brackets, separated by commas—for example,

**<E1, E2, E3>**.

**Note**

You can define a class or an interface as a subtype of a generic class or interface. For

example, the **java.lang.String** class is defined to implement the **Comparable**

interface in the Java API as follows:

**public class** String **implements** Comparable<String>

19.4 Generic Methods

*A generic type can be defined for a static method.*

To declare a generic method, you place the generic type **<E>** immediately after the keyword

**static** in the method header.

To invoke a generic method, prefix the method name with the actual type in angle brackets.

A generic type can be specified as a subtype of another type. Such a generic type is

called *bounded*.

**Note**

An unbounded generic type **<E>** is the same as **<E extends Object>**.

**Note**

To define a generic type for a class, place it after the class name, such as

**GenericStack<E>**. To define a generic type for a method, place the generic type

before the method return type, such as **<E> void max(E o1, E o2)**.

19.6 Raw Types and Backward Compatibility

*A generic class or interface used without specifying a concrete type, called a raw type,*

*enables backward compatibility with earlier versions of Java.*

You can use a generic class without specifying a concrete type like this:

GenericStack stack = **new** GenericStack(); // raw type

This is roughly equivalent to

GenericStack<Object> stack = **new** GenericStack<Object>();

A generic class such as **GenericStack** and **ArrayList** used without a type parameter is

called a *raw type*. Using raw types allows for backward compatibility with earlier versions of

Java.

19.7 Wildcard Generic Types

*You can use unbounded wildcards, bounded wildcards, or lower-bound wildcards to*

*specify a range for a generic type.*

19.8 Erasure and Restrictions on Generics

*The information on generics is used by the compiler but is not available at runtime.*

*This is called type erasure.*

Generics are implemented using an approach called *type erasure*: The compiler uses the

generic type information to compile the code, but erases it afterward. Thus, the generic information

is not available at runtime. This approach enables the generic code to be backward

compatible with the legacy code that uses raw types.

The generics are present at compile time. Once the compiler confirms that a generic type

is used safely, it converts the generic type to a raw type.

When generic classes, interfaces, and methods are compiled, the compiler replaces the generic

type with the **Object** type.

If a generic type is bounded, the compiler replaces it with the bounded type.

It is important to note that a generic class is shared by all its instances regardless of its actual

concrete type.

Because generic types are erased at runtime, there are certain restrictions on how generic

types can be used.

**Restriction 1: Cannot Use *new E()***

You cannot create an instance using a generic type parameter.

E object = **new** E();

The reason is that **new E()** is executed at runtime, but the generic type **E** is not available

at runtime.

**Restriction 2: Cannot Use *new E[]***

You cannot create an array using a generic type parameter. For example, the following

statement is wrong:

E[] elements = **new** E[capacity];

You can circumvent this limitation by creating an array of the **Object** type and then casting

it to **E[]**, as follows:

E[] elements = (E[])**new** Object[capacity];

However, casting to **(E[])** causes an unchecked compile warning. The warning occurs

because the compiler is not certain that casting will succeed at runtime.

Generic array creation using a generic class is not allowed, either.

**Restriction 3: A Generic Type Parameter of a Class Is Not Allowed in a Static**

**Context**

Since all instances of a generic class have the same runtime class, the static variables and

methods of a generic class are shared by all its instances. Therefore, it is illegal to refer to  
a generic type parameter for a class in a static method, field, or initializer. For example, the

following code is illegal:

**public class** Test<E> {

**public static void** m(E o1) { // Illegal

}

**public static** E o1; // Illegal

**static** {

E o2; // Illegal

}

}

**Restriction 4: Exception Classes Cannot Be Generic**

A generic class may not extend **java.lang.Throwable**, so the following class declaration

would be illegal:

**public class** MyException<T> **extends** Exception {

}

Why? If it were allowed, you would have a **catch** clause for **MyException<T>** as follows:

**try** {

...

}

**catch** (MyException<T> ex) {

...

}

The JVM has to check the exception thrown from the **try** clause to see if it matches the

type specified in a **catch** clause. This is impossible, because the type information is not present at runtime.

**CHAPTER SUMMARY**

**1.** *Generics* give you the capability to parameterize types. You can define a class or a

method with generic types, which the compiler replaces with concrete types.

**2.** The key benefit of generics is to enable errors to be detected at compile time rather than

at runtime.

**3.** A generic class or method permits you to specify allowable types of objects that the

class or method can work with. If you attempt to use a class or method with an incompatible

object, the compiler will detect the error.

**4.** A generic type defined in a class, interface, or a static method is called a *formal generic*

*type*, which can be replaced later with an *actual concrete type*. Replacing a generic type

is called a *generic instantiation*.  
**5.** A generic class such as **ArrayList** used without a type parameter is called a *raw*

*type*. Use of raw types is allowed for backward compatibility with the earlier versions

of Java.

**6.** A wildcard generic type has three forms: **?** and **? extends T**, and **? super T**, where

**T** is a generic type. The first form, **?**, called an *unbounded wildcard*, is the same as

**? extends Object**. The second form, **? extends T**, called a *bounded wildcard*,

represents **T** or a subtype of **T**. The third form, **? super T**, called a *lower-bound wildcard*,

denotes **T** or a supertype of **T**.

**7.** Generics are implemented using an approach called *type erasure*. The compiler uses

the generic type information to compile the code but erases it afterward, so the generic

information is not available at runtime. This approach enables the generic code to be

backward compatible with the legacy code that uses raw types.

20.1 Introduction

*Choosing the best data structures and algorithms for a particular task is one of the*

*keys to developing high-performance software.*

A *data structure* is a collection of data organized in some fashion. The structure not only

stores data but also supports operations for accessing and manipulating the data.

In object-oriented thinking, a data structure, also known as a *container* or *container object*,

is an object that stores other objects, referred to as data or elements. To define a data structure

is essentially to define a class. The class for a data structure should use data fields to store data

and provide methods to support such operations as search, insertion, and deletion. To create a

data structure is therefore to create an instance from the class. You can then apply the methods

on the instance to manipulate the data structure, such as inserting an element into or deleting

an element from the data structure.

Java provides several more data structures that can be used to organize and manipulate

data efficiently. These are commonly known as *Java Collections Framework*.

20.2 Collections

*The* **Collection** *interface defines the common operations for lists, vectors, stacks,*

*queues, priority queues, and sets.*

The Java Collections Framework supports two types of containers:

■ One for storing a collection of elements is simply called a *collection*.

■ The other, for storing key/value pairs, is called a *map*.

Maps are efficient data structures for quickly searching an element using a key. We will introduce

maps in the next chapter. Now we turn our attention to the following collections.

■ **Set**s store a group of nonduplicate elements.

■ **List**s store an ordered collection of elements.

■ **Stack**s store objects that are processed in a last-in, first-out fashion.

■ **Queue**s store objects that are processed in a first-in, first-out fashion.

■ **PriorityQueue**s store objects that are processed in the order of their priorities.

**Note**

All the interfaces and classes defined in the Java Collections Framework are grouped in

the **java.util** package.

**Design Guide**

The design of the Java Collections Framework is an excellent example of using interfaces,

abstract classes, and concrete classes. The interfaces define the framework. The abstract

classes provide partial implementation. The concrete classes implement the interfaces

with concrete data structures. Providing an abstract class that partially implements an

interface makes it convenient for the user to write the code. The user can simply define

a concrete class that extends the abstract class rather implements all the methods in

the interface. The abstract classes such as **AbstractCollection** are provided for

convenience. For this reason, they are called *convenience abstract classes*.

A collection is a container that stores objects.

The **Collection** interface is the root interface for manipulating a collection of objects. Its

public methods are listed in Figure 20.2. The **AbstractCollection** class provides partial

implementation for the **Collection** interface. It implements all the methods in **Collection**

except the **add**, **size**, and **iterator** methods. These are implemented in appropriate concrete

subclasses.

The **Collection** interface provides the basic operations for adding and removing elements

in a collection. The **add** method adds an element to the collection. The **addAll** method adds

all the elements in the specified collection to this collection. The **remove** method removes an

element from the collection. The **removeAll** method removes the elements from this collection

that are present in the specified collection. The **retainAll** method retains the elements

in this collection that are also present in the specified collection. All these methods return

**boolean**. The return value is **true** if the collection is changed as a result of the method

execution. The **clear()** method simply removes all the elements from the collection.

**Note**

The methods **addAll**, **removeAll**, and **retainAll** are similar to the set union,

difference, and intersection operations.

The **Collection** interface provides various query operations. The **size** method returns

the number of elements in the collection. The **contains** method checks whether the collection

contains the specified element. The **containsAll** method checks whether the collection

contains all the elements in the specified collection. The **isEmpty** method returns **true** if the

collection is empty.

The **Collection** interface provides the **toArray()** method, which returns an array representation

for the collection.

**Design Guide**

Some of the methods in the **Collection** interface cannot be implemented

in the concrete subclass. In this case, the method would throw **java.lang**

**.UnsupportedOperationException**, a subclass of **RuntimeException**.

This is a good design that you can use in your project. If a method has no meaning in

the subclass, you can implement it as follows:

**public void** someMethod() {

**throw new** UnsupportedOperationException

(**"Method not supported"**);

}

**Note**

All the concrete classes in the Java Collections Framework implement the **java.lang.**

**Cloneable** and **java.io.Serializable** interfaces except that

**java.util.PriorityQueue** does not implement the **Cloneable** interface. Thus,

all instances of **Cloneable** except priority queues can be cloned and all instances of

**Cloneable** can be serialized.

20.3 Iterators

*Each collection is* **Iterable***. You can obtain its* **Iterator** *object to traverse all the*

*elements in the collection.*

**Iterator** is a classic design pattern for walking through a data structure without having to

expose the details of how data is stored in the data structure.

The **Collection** interface extends the **Iterable** interface. The **Iterable** interface

defines the **iterator** method, which returns an iterator. The **Iterator** interface provides a

uniform way for traversing elements in various types of collections. The **iterator()** method

in the **Iterable** interface returns an instance of **Iterator**, which

provides sequential access to the elements in the collection using the **next()** method. You

can also use the **hasNext()** method to check whether there are more elements in the iterator,

and the **remove()** method to remove the last element returned by the iterator.

20.4 Lists

*The* **List** *interface extends the* **Collection** *interface and defines a collection for*

*storing elements in a sequential order.*

20.4.1 The Common Methods in the **List** Interface

**ArrayList** and **LinkedList** are defined under the **List** interface. The **List** interface

extends **Collection** to define an ordered collection with duplicates allowed. The **List**

interface adds position-oriented operations, as well as a new list iterator that enables the user

to traverse the list bidirectionally.

The **add(index, element)** method is used to insert an element at a specified index, and

the **addAll(index, collection)** method to insert a collection of elements at a specified

index. The **remove(index)** method is used to remove an element at the specified index from

the list. A new element can be set at the specified index using the **set(index, element)**

method.

The **indexOf(element)** method is used to obtain the index of the specified element’s

first occurrence in the list, and the **lastIndexOf(element)** method to obtain the index of

its last occurrence. A sublist can be obtained by using the **subList(fromIndex, toIndex)**

method.

The **listIterator()** or **listIterator(startIndex)** method returns an instance of

**ListIterator**. The **ListIterator** interface extends the **Iterator** interface to add bidirectional

traversal of the list.

The **add(element)** method inserts the specified element into the list. The element is

inserted immediately before the next element that would be returned by the **next()** method

defined in the **Iterator** interface, if any, and after the element that would be returned by

the **previous()** method, if any. If the list doesn’t contain any elements, the new element

becomes the sole element in the list. The **set(element)** method can be used to replace

the last element returned by the **next** method or the **previous** method with the specified

element.

The **hasNext()** method defined in the **Iterator** interface is used to check whether the

iterator has more elements when traversed in the forward direction, and the **hasPrevious()**

method to check whether the iterator has more elements when traversed in the backward

direction.

The **next()** method defined in the **Iterator** interface returns the next element in

the iterator, and the **previous()** method returns the previous element in the iterator.

The **nextIndex()** method returns the index of the next element in the iterator, and the

**previousIndex()** returns the index of the previous element in the iterator.

The **AbstractList** class provides a partial implementation for the **List** interface.

The **AbstractSequentialList** class extends **AbstractList** to provide support for

linked lists.

20.4.2 The **ArrayList** and **LinkedList** Classes

The **ArrayList** class and the **LinkedList** class are two concrete implementations of the

**List** interface. **ArrayList** stores elements in an array. The array is dynamically created. If

the capacity of the array is exceeded, a larger new array is created and all the elements from

the current array are copied to the new array. **LinkedList** stores elements in a *linked list*.

Which of the two classes you use depends on your specific needs. If you need to support

random access through an index without inserting or removing elements at the beginning

of the list, **ArrayList** offers the most efficient collection. If, however, your application

requires the insertion or deletion of elements at the beginning of the list, you should choose

**LinkedList**. A list can grow or shrink dynamically. Once it is created, an array is fixed. If

your application does not require the insertion or deletion of elements, an array is the most

efficient data structure.

**ArrayList** is a resizable-array implementation of the **List** interface. It also provides

methods for manipulating the size of the array used internally to store the list, as shown

in Figure 20.5. Each **ArrayList** instance has a capacity, which is the size of the array used

to store the elements in the list. It is always at least as large as the list size. As elements

are added to an **ArrayList**, its capacity grows automatically. An **ArrayList** does not

automatically shrink. You can use the **trimToSize()** method to reduce the array capacity

to the size of the list. An **ArrayList** can be constructed using its no-arg constructor,

**ArrayList(Collection)**, or **ArrayList(initialCapacity)**.

**LinkedList** is a linked list implementation of the **List** interface. In addition to implementing

the **List** interface, this class provides the methods for retrieving, inserting, and

removing elements from both ends of the list, as shown in Figure 20.6. A **LinkedList** can be

constructed using its no-arg constructor or **LinkedList(Collection)**.

20.5 The **Comparator** Interface

**Comparator** *can be used to compare the objects of a class that doesn’t implement*

**Comparable***.*

Several classes in the Java API, such as **String**, **Date**, **Calendar**,

**BigInteger**, **BigDecimal**, and all the numeric wrapper classes for the primitive types,

implement the **Comparable** interface. The **Comparable** interface defines the **compareTo**

method, which is used to compare two elements of the same class that implement the

**Comparable** interface.

You can define a *comparator* to compare the elements of different

classes. To do so, define a class that implements the **java.util.Comparator<T>** interface

and overrides its **compare** method.

**public int** compare(T element1, T element2)

Returns a negative value if **element1** is less than **element2**, a

**Note**

**Comparable** is used to compare the objects of the class that implement **Comparable**.

**Comparator** can be used to compare the objects of a class that doesn’t implement

**Comparable**.

Comparing elements using the **Comparable** interface is referred to as comparing using

*natural order*, and comparing elements using the **Comparator** interface is referred to

as comparing using *comparator*.

20.6 Static Methods for Lists and Collections

*The* **Collections** *class contains static methods to perform common operations in a*

*collection and a list.*

The **Collections** class contains the **sort**, **binarySearch**, **reverse**, **shuffle**, **copy**, and

**fill** methods for lists, and **max**, **min**, **disjoint**, and **frequency** methods for collections,

You can sort the comparable elements in a list in its natural order with the **compareTo**

method in the **Comparable** interface. You may also specify a comparator to sort elements.

To sort it in descending order, you can

simply use the **Collections.reverseOrder()** method to return a **Comparator** object

that orders the elements in reverse of their natural order.

You can use the **binarySearch** method to search for a key in a list. To use this method,

the list must be sorted in increasing order. If the key is not in the list, the method returns

-(*insertion point* +1). Recall that the insertion point is where the item would fall in the list if

it were present.

You can use the **reverse** method to reverse the elements in a list.

You can use the **shuffle(List)** method to randomly reorder the elements in a list.

You can also use the **shuffle(List, Random)** method to randomly reorder the elements in

a list with a specified **Random** object. Using a specified **Random** object is useful to generate a

list with identical sequences of elements for the same original list.

You can use the **copy(det, src)** method to copy all the elements from a source list to a

destination list on the same index. The destination list must be as long as the source list. If it is

longer, the remaining elements in the source list are not affected.

You can use the **nCopies(int n, Object o)** method to create an immutable list that

consists of **n** copies of the specified object.

You can use the **fill(List list, Object o)** method to replace all the elements in the

list with the specified element.

You can use the **max** and **min** methods for finding the maximum and minimum elements

in a collection. The elements must be comparable using the **Comparable** interface or the

**Comparator** interface.

The **disjoint(collection1, collection2)** method returns **true** if the two collections

have no elements in common.

The **frequency(collection, element)** method finds the number of occurrences of the

element in the collection.

20.8 **Vector** and **Stack** Classes

**Vector** *is a subclass of* **AbstractList***, and* **Stack** *is a subclass of* **Vector** *in the*

*Java API.*

The Java Collections Framework was introduced in Java 2. Several data structures were supported

earlier, among them the **Vector** and **Stack** classes. These classes were redesigned

to fit into the Java Collections Framework, but all their old-style methods are retained for

compatibility.

**Vector** is the same as **ArrayList**, except that it contains synchronized methods for

accessing and modifying the vector. Synchronized methods can prevent data corruption

when a vector is accessed and modified by two or more threads concurrently.

For the many

applications that do not require synchronization, using **ArrayList** is more efficient than

using **Vector**.

The **Vector** class extends the **AbstractList** class. It also has the methods contained in

the original **Vector** class defined prior to Java 2

Most of the methods in the **Vector** class listed in the UML diagram

are

similar to the methods in the **List** interface. These methods were introduced before the

Java Collections Framework. For example, **addElement(Object element)** is the same

as the **add(Object element)** method, except that the **addElement** method is synchronized.

Use the **ArrayList** class if you don’t need synchronization. It works much faster

than **Vector**.

**Note**

The **elements()** method returns an **Enumeration**. The **Enumeration** interface

was introduced prior to Java 2 and was superseded by the **Iterator** interface.

**Note**

**Vector** is widely used in Java legacy code because it was the Java resizable array implementation

before Java 2.

In the Java Collections Framework, **Stack** is implemented as an extension of **Vector**

The **Stack** class was introduced prior to Java 2

The **empty()** method is the same as **isEmpty()**. The **peek()** method

looks at the element at the top of the stack without removing it. The **pop()** method removes

the top element from the stack and returns it. The **push(Object element)** method adds the

specified element to the stack. The **search(Object element)** method checks whether the

specified element is in the stack.

20.9 Queues and Priority Queues

*In a priority queue, the element with the highest priority is removed first.*

A *queue* is a first-in, first-out data structure. Elements are appended to the end of the queue

and are removed from the beginning of the queue. In a *priority queue*, elements are assigned

priorities. When accessing elements, the element with the highest priority is removed first.

20.9.1 The **Queue** Interface

The **Queue** interface extends **java.util.Collection** with additional insertion, extraction,

and inspection operations,

The **offer** method is used to add an element to the queue. This method is similar to the

**add** method in the **Collection** interface, but the **offer** method is preferred for queues.

The **poll** and **remove** methods are similar, except that **poll()** returns **null** if the queue is

empty, whereas **remove()** throws an exception. The **peek** and **element** methods are similar,

except that **peek()** returns **null** if the queue is empty, whereas **element()** throws an

exception.

20.9.2 **Deque** and **LinkedList**

The **LinkedList** class implements the **Deque** interface, which extends the **Queue** interface,

Therefore, you can use **LinkedList** to create a queue.

**LinkedList** is ideal for queue operations because it is efficient for inserting and removing

elements from both ends of a list.

**Deque** supports element insertion and removal at both ends. The name *deque* is short for

“double-ended queue” and is usually pronounced “deck.” The **Deque** interface extends **Queue**

with additional methods for inserting and removing elements from both ends of the queue. The

methods **addFirst(e)**, **removeFirst()**, **addLast(e)**, **removeLast()**, **getFirst()**,

and **getLast()** are defined in the **Deque** interface.

The **PriorityQueue** class implements a priority queue, as shown in Figure 20.14.

By default, the priority queue orders its elements according to their natural ordering using

**Comparable**. The element with the least value is assigned the highest priority and thus is

removed from the queue first. If there are several elements with the same highest priority, the

tie is broken arbitrarily. You can also specify an ordering using **Comparator** in the constructor

**PriorityQueue(initialCapacity, comparator)**.

Listing 20.8 shows an example of using a priority queue to store

**CHAPTER SUMMARY**

**1.** The Java Collections Framework supports *sets*, *lists*, *queues*, and *maps*. They are defined

in the interfaces **Set**, **List**, **Queue**, and **Map**.

**2.** A *list* stores an ordered *collection* of elements.

**3.** All the concrete classes except **PriorityQueue** in the Java Collections Framework

implement the **Cloneable** and **Serializable** interfaces. Thus, their instances can be

cloned and serialized.

**4.** To allow duplicate elements to be stored in a collection, you need to use a list. A list

not only can store duplicate elements but also allows the user to specify where they are

stored. The user can access elements by an index.

**5.** Two types of lists are supported: **ArrayList** and **LinkedList**. **ArrayList** is a

resizable-array implementation of the **List** interface. All the methods in **ArrayList**

are defined in **List**. **LinkedList** is a *linked-list* implementation of the **List** interface.

In addition to implementing the **List** interface, this class provides the methods for

retrieving, inserting, and removing elements from both ends of the list.

**6. Comparator** can be used to compare the objects of a class that doesn’t implement

**Comparable**.

**7.** The **Vector** class extends the **AbstractList** class. Starting with Java 2, **Vector** has

been the same as **ArrayList**, except that the methods for accessing and modifying

the vector are synchronized. The **Stack** class extends the **Vector** class and provides

several methods for manipulating the stack.

**8.** The **Queue** interface represents a queue. The **PriorityQueue** class implements **Queue**

for a *priority queue*.

21.1 Introduction

*A set is an efficient data structure for storing and processing nonduplicate elements.*

*A map is like a dictionary that provides a quick lookup to retrieve a value using a key.*

*You can create a set using one of its three concrete classes:* **HashSet***,*

**LinkedHashSet***, or* **TreeSet***.*

The **Set** interface extends the **Collection** interface, It does not

introduce new methods or constants, but it stipulates that an instance of **Set** contains no

duplicate elements. The concrete classes that implement **Set** must ensure that no duplicate

elements can be added to the set. That is, no two elements **e1** and **e2** can be in the set such

that **e1.equals(e2)** is **true**.

The **AbstractSet** class extends **AbstractCollection** and partially implements **Set**.

The **AbstractSet** class provides concrete implementations for the **equals** method and

the **hashCode** method. The hash code of a set is the sum of the hash codes of all the elements

in the set. Since the **size** method and **iterator** method are not implemented in the

**AbstractSet** class, **AbstractSet** is an abstract class.

21.2.1 **HashSet**

The **HashSet** class is a concrete class that implements **Set**. You can create an empty *hash*

*set* using its no-arg constructor or create a hash set from an existing collection. By default,

the initial capacity is **16** and the load factor is **0.75**. If you know the size of your set, you can

specify the initial capacity and load factor in the constructor. Otherwise, use the default setting.

The load factor is a value between **0.0** and **1.0**.

*The load factor* measures how full the set is allowed to be before its capacity is increased.

When the number of elements exceeds the product of the capacity and load factor, the capacity is

automatically doubled. For example, if the capacity is **16** and load factor is **0.75**, the capacity will

be doubled to **32** when the size reaches **12** (16\*0.75 = 12). A higher load factor decreases the

space costs but increases the search time. Generally, the default load factor **0.75** is a good tradeoff

between time and space costs.

A **HashSet** can be used to store *duplicate-free* elements. For efficiency, objects added

to a hash set need to implement the **hashCode** method in a manner that properly disperses

the hash code. Recall that **hashCode** is defined in the **Object** class. The hash codes of two

objects must be the same if the two objects are equal. Two unequal objects may have the

same hash code, but you should implement the **hashCode** method to avoid too many such

cases. Most of the classes in the Java API implement the **hashCode** method.

21.2.2 **LinkedHashSet**

**LinkedHashSet** extends **HashSet** with a linked-list implementation that supports an ordering

of the elements in the set. The elements in a **HashSet** are not ordered, but the elements

in a **LinkedHashSet** can be retrieved in the order in which they were inserted into the set. A

**LinkedHashSet** can be created by using one of its four constructors,

21.2.3 **TreeSet**

**SortedSet** is a subinterface of **Set**, which guarantees that the elements in the set are sorted.

Additionally, it provides the methods **first()** and **last()** for returning the first and last

elements in the set, and **headSet(toElement)** and **tailSet(fromElement)** for returning  
a portion of the set whose elements are less than **toElement** and greater than or equal to

**fromElement**, respectively.

**NavigableSet** extends **SortedSet** to provide navigation methods **lower(e)**,

**floor(e)**, **ceiling(e)**, and **higher(e)** that return elements respectively less than, less

than or equal, greater than or equal, and greater than a given element and return **null** if there

is no such element. The **pollFirst()** and **pollLast()** methods remove and return the first

and last element in the tree set, respectively.

**TreeSet** implements the **SortedSet** interface. To create a **TreeSet**, use a constructor,  
You can add objects into a *tree set* as long as they can be compared

with each other.

As discussed in Section 20.5, the elements can be compared in two ways: using the

**Comparable** interface or the **Comparator** interface.

**Note**

All the concrete classes in Java Collections Framework (see Figure 20.1) have at least

two constructors. One is the no-arg constructor that constructs an empty collection.

The other constructs instances from a collection. Thus the **TreeSet** class has the constructor

**TreeSet(Collection c)** for constructing a **TreeSet** from a collection **c**.

In this example, **new TreeSet<>(set)** creates an instance of **TreeSet** from the

collection **set**.

**Tip**

If you don’t need to maintain a sorted set when updating a set, you should use a hash

set, because it takes less time to insert and remove elements in a hash set. When you

need a sorted set, you can create a tree set from the hash set.

If you create a **TreeSet** using its no-arg constructor, the **compareTo** method is used to compare

the elements in the set, assuming that the class of the elements implements the **Comparable**

interface. To use a comparator, you have to use the constructor **TreeSet(Comparator**

**comparator)** to create a sorted set that uses the **compare** method in the comparator to order

the elements in the set.

21.3 Comparing the Performance of Sets and Lists

*Sets are more efficient than lists for storing nonduplicate elements. Lists are useful for*

*accessing elements through the index.*

The elements in a list can be accessed through the index. However, sets do not support indexing,

because the elements in a set are unordered. To traverse all elements in a set, use a foreach loop.

21.5 Maps

*You can create a map using one of its three concrete classes:* **HashMap***,*

**LinkedHashMap***, or* **TreeMap***.*

A *map* is a container object that stores a collection of key/value pairs. It enables fast retrieval,

deletion, and updating of the pair through the key. A map stores the values along with the keys.

The keys are like indexes. In **List**, the indexes are integers. In **Map**, the keys can be any objects.

A map cannot contain duplicate keys. Each key maps to one value. A key and its corresponding

value form an entry stored in a map,

There are three types of maps: **HashMap**, **LinkedHashMap**, and **TreeMap**. The common

features of these maps are defined in the **Map** interface.

The **Map** interface provides the methods for querying, updating, and obtaining a collection

of values and a set of keys,

The *update methods* include **clear**, **put**, **putAll**, and **remove**. The **clear()** method

removes all entries from the map. The **put(K key, V value)** method adds an entry for the

specified key and value in the map. If the map formerly contained an entry for this key, the

old value is replaced by the new value and the old value associated with the key is returned.

The **putAll(Map m)** method adds all entries in **m** to this map. The **remove(Object key)**

method removes the entry for the specified key from the map.

The *query methods* include **containsKey**, **containsValue**, **isEmpty**, and **size**. The

**containsKey(Object key)** method checks whether the map contains an entry for the

specified key. The **containsValue(Object value)** method checks whether the map contains

an entry for this value. The **isEmpty()** method checks whether the map contains any

entries. The **size()** method returns the number of entries in the map.

You can obtain a set of the keys in the map using the **keySet()** method, and a collection

of the values in the map using the **values()** method. The **entrySet()** method returns a set

of entries. The entries are instances of the **Map.Entry<K, V>** interface, where **Entry** is an

inner interface for the **Map** interface, as shown in Figure 21.5. Each entry in the set is a key/

value pair in the underlying map.

The **AbstractMap** class is a convenience abstract class that implements all the methods in

the **Map** interface except the **entrySet()** method.

The **HashMap**, **LinkedHashMap**, and **TreeMap** classes are three *concrete implementations*

of the **Map** interface,

The **HashMap** class is efficient for locating a value, inserting an entry, and deleting an entry.

**LinkedHashMap** extends **HashMap** with a linked-list implementation that supports an

ordering of the entries in the map. The entries in a **HashMap** are not ordered, but the entries

in a **LinkedHashMap** can be retrieved either in the order in which they were inserted into the

map (known as the *insertion order*) or in the order in which they were last accessed, from

least recently to most recently accessed (*access order*). The no-arg constructor constructs a

**LinkedHashMap** with the insertion order. To construct a **LinkedHashMap** with the access

order, use **LinkedHashMap(initialCapacity, loadFactor, true)**.

The **TreeMap** class is efficient for traversing the keys in a sorted order. The keys can

be sorted using the **Comparable** interface or the **Comparator** interface. If you create a

**TreeMap** using its no-arg constructor, the **compareTo** method in the **Comparable** interface

is used to compare the keys in the map, assuming that the class for the keys implements the

**Comparable** interface. To use a comparator, you have to use the **TreeMap(Comparator**

**comparator)** constructor to create a sorted map that uses the **compare** method in the comparator

to order the entries in the map based on the keys.

**SortedMap** is a subinterface of **Map**, which guarantees that the entries in the map are

sorted. Additionally, it provides the methods **firstKey()** and **lastKey()** for returning the

first and last keys in the map, and **headMap(toKey)** and **tailMap(fromKey)** for returning

a portion of the map whose keys are less than **toKey** and greater than or equal to **fromKey**,

respectively.

**NavigableMap** extends **SortedMap** to provide the navigation methods **lowerKey(key)**,

**floorKey(key)**, **ceilingKey(key)**, and **higherKey(key)** that return keys respectively

less than, less than or equal, greater than or equal, and greater than a given key and return

**null** if there is no such key. The **pollFirstEntry()** and **pollLastEntry()** methods

remove and return the first and last entry in the tree map, respectively.

**Tip**

If you don’t need to maintain an order in a map when updating it, use a **HashMap**.

When you need to maintain the insertion order or access order in the map, use a

**LinkedHashMap**. When you need the map to be sorted on keys, use a **TreeMap**.

21.7 Singleton and Unmodifiable Collections and Maps

*You can create singleton sets, lists, and maps and unmodifiable sets, lists, and maps*

*using the static methods in the* **Collections** *class.*

The **Collections** class contains the static methods for lists and collections. It also contains

the methods for creating immutable singleton sets, lists, and maps, and for creating read-only

sets, lists, and maps,

The **Collections** class defines three constants—**EMPTY\_SET**, **EMPTY\_LIST**, and

**EMPTY\_MAP**—for an empty set, an empty list, and an empty map. These collections are immutable.

The class also provides the **singleton(Object o)** method for creating an immutable

set containing only a single item, the **singletonList(Object o)** method for creating  
an immutable list containing only a single item, and the **singletonMap(Object key,**

**Object value)** method for creating an immutable map containing only a single entry.

The **Collections** class also provides six static methods for returning *read-only views for*

*collections*: **unmodifiableCollection(Collection c)**, **unmodifiableList(List**

**list)**, **unmodifiableMap(Map m)**, **unmodifiableSet(Set set)**,

**unmodifiableSortedMap(SortedMap m)**, and **unmodifiableSortedSet(Sorted**

**Set s)**. This type of view is like a reference to the actual collection. But you cannot modify

the collection through a read-only view. Attempting to modify a collection through a readonly

view will cause an **UnsupportedOperationException**.

**CHAPTER SUMMARY**

**1.** A set stores nonduplicate elements. To allow duplicate elements to be stored in a collection,

you need to use a list.

**2.** A *map* stores key/value pairs. It provides a quick lookup for a value using a key.

**3.** Three types of sets are supported: **HashSet**, **LinkedHashSet**, and **TreeSet**. **HashSet**

stores elements in an unpredictable order. **LinkedHashSet** stores elements in the order  
they were inserted. **TreeSet** stores elements sorted. All the methods in **HashSet**,

**LinkedHashSet**, and **TreeSet** are inherited from the **Collection** interface.

**4.** The **Map** interface maps keys to the elements. The keys are like indexes. In **List**,

the indexes are integers. In **Map**, the keys can be any objects. A map cannot contain

duplicate keys. Each key can map to at most one value. The **Map** interface provides

the methods for querying, updating, and obtaining a collection of values and a set

of keys.

**5.** Three types of maps are supported: **HashMap**, **LinkedHashMap**, and **TreeMap**.

**HashMap** is efficient for locating a value, inserting an entry, and deleting an entry.

**LinkedHashMap** supports ordering of the entries in the map. The entries in a **Hash-**

**Map** are not ordered, but the entries in a **LinkedHashMap** can be retrieved either in

the order in which they were inserted into the map (known as the *insertion order*)

or in the order in which they were last accessed, from least recently accessed to

most recently (*access order*). **TreeMap** is efficient for traversing the keys in a sorted

order. The keys can be sorted using the **Comparable** interface or the **Comparator**

interface.

22.1 Introduction

*Algorithm design is to develop a mathematical process for solving a problem.*

*Algorithm analysis is to predict the performance of an algorithm.*

22.2 Measuring Algorithm Efficiency Using

Big *O* Notation

*The Big* O *notation obtains a function for measuring algorithm time complexity based*

*on the input size. You can ignore multiplicative constants and nondominating terms in the function.*

■ First, many tasks run concurrently on a computer. The execution time of a particular

program depends on the system load.

■ Second, the execution time depends on specific input. Consider, for example, linear

search and binary search. If an element to be searched happens to be the first in the

list, linear search will find the element quicker than binary search.

It is very difficult to compare algorithms by measuring their execution time. To overcome

these problems, a theoretical approach was developed to analyze algorithms independent of

computers and specific input. This approach approximates the effect of a change on the size

of the input. In this way, you can see how fast an algorithm’s execution time increases as the

input size increases, so you can compare two algorithms by examining their *growth rates*.

Consider linear search. The linear search algorithm compares the key with the elements in

the array sequentially until the key is found or the array is exhausted. If the key is not in the

array, it requires *n* comparisons for an array of size *n*. If the key is in the array, it requires *n*/2

comparisons on average. The algorithm’s execution time is proportional to the size of the array.

If you double the size of the array, you will expect the number of comparisons to double. The

algorithm grows at a linear rate. The growth rate has an order of magnitude of *n*. Computer

scientists use the Big *O* notation to represent the “order of magnitude.” Using this notation,

the complexity of the linear search algorithm is *O*(*n*), pronounced as “*order of n*.” We call an

algorithm with a time complexity of O(n) a linear algorithm, and it exhibits a linear growth rate.

For the same input size, an algorithm’s execution time may vary, depending on the input.

An input that results in the shortest execution time is called the *best-case input*, and an input

that results in the longest execution time is the *worst-case input.* Best-case analysis and

worst-case analysis are to analyze the algorithms for their best-case input and worst-case

input. Best-case and worst-case analysis are not representative, but worst-case analysis is

very useful. You can be assured that the algorithm will never be slower than the worst case.

An *average-case analysis* attempts to determine the average amount of time among all possible

inputs of the same size. Average-case analysis is ideal, but difficult to perform, because

for many problems it is hard to determine the relative probabilities and distributions of various

input instances. Worst-case analysis is easier to perform, so the analysis is generally conducted for the worst case.

The linear search algorithm requires *n* comparisons in the worst case and *n*/2 comparisons

in the average case if you are nearly always looking for something known to be in the list.

Using the Big *O* notation, both cases require *O*(*n*) time. The multiplicative constant (1/2) can

be omitted. Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates.

Consider the algorithm for finding the maximum number in an array of *n* elements. To find

the maximum number if *n* is 2, it takes one comparison; if *n* is 3, two comparisons. In general,

it takes *n* - 1 comparisons to find the maximum number in a list of *n* elements. Algorithm

analysis is for large input size. If the input size is small, there is no significance in estimating

an algorithm’s efficiency. As *n* grows larger, the *n* part in the expression *n* - 1 dominates the

complexity. The Big *O* notation allows you to ignore the nondominating part (e.g., -1 in the

expression *n* - 1) and highlight the important part (e.g., *n* in the expression *n* - 1). Therefore,

the complexity of this algorithm is *O*(*n*).

The Big *O* notation estimates the execution time of an algorithm in relation to the input

size. If the time is not related to the input size, the algorithm is said to take *constant time* with

the notation *O*(1). For example, a method that retrieves an element at a given index in an array takes constant time, because the time does not grow as the size of the array increases.

**Note**

*Time complexity* is a measure of execution time using the Big-O notation. Similarly, you

can also measure *space complexity* using the Big-O notation. *Space complexity* measures

the amount of memory space used by an algorithm. The space complexity for most

algorithms presented in the book is *O*(*n*). i.e., they exibit linear growth rate to the input size. For example, the space complexity for linear search is *O*(*n*).

The algorithm for computing Fibonacci numbers presented here uses an approach known

as *dynamic programming*. Dynamic programming is the process of solving subproblems,

then combining the solutions of the subproblems to obtain an overall solution. This naturally

leads to a recursive solution. However, it would be inefficient to use recursion, because the  
subproblems overlap. The key idea behind dynamic programming is to solve each subproblem

only once and store the results for subproblems for later use to avoid redundant computing of the subproblems.

22.6 Finding Greatest Common Divisors Using

Euclid’s Algorithm

The greatest common divisor (GCD) of two integers is the largest number that can evenly

divide both integers. Listing 5.9, GreatestCommonDivisor.java, presented a brute-force algorithm

for finding the greatest common divisor of two integers **m** and **n**. *Brute force* refers to an

algorithmic approach that solves a problem in the simplest or most direct or obvious way. As

a result, such an algorithm can end up doing far more work to solve a given problem than a

cleverer or more sophisticated algorithm might do. On the other hand, a brute-force algorithm

is often easier to implement than a more sophisticated one and, because of this simplicity,

sometimes it can be more efficient.

**Note**

The Big *O* notation provides a good theoretical estimate of algorithm efficiency. However, two algorithms of the same time complexity are not necessarily equally efficient.

**CHAPTER SUMMARY**

**1.** The *Big O notation* is a theoretical approach for analyzing the performance of an

algorithm. It estimates how fast an algorithm’s execution time increases as the input

size increases, which enables you to compare two algorithms by examining their *growth rates*.  
**2.** An input that results in the shortest execution time is called the *best-case* input and one

that results in the longest execution time is called the *worst-case* input. Best case and

worst case are not representative, but worst-case analysis is very useful. You can be

assured that the algorithm will never be slower than the worst case.

**3.** An *average-case analysis* attempts to determine the average amount of time among all

possible input of the same size. Average-case analysis is ideal, but difficult to perform,

because for many problems it is hard to determine the relative probabilities and distributions

of various input instances.

**4.** If the time is not related to the input size, the algorithm is said to take *constant time* with

the notation *O*(1).

**5.** Linear search takes *O*(*n*) time. An algorithm with the *O*(*n*) time complexity is called a

*linear algorithm* and it exhibits a linear growth rate. Binary search takes *O*(log*n*) time.

An algorithm with the *O*(log *n*) time complexity is called a *logarithmic algorithm*, and

it exhibits a logarithmic growth rate.

**6.** The worst-time complexity for selection sort is *O*(*n*2). An algorithm with the *O*(*n*2) time

complexity is called a *quadratic algorithm*, and it exhibits a quadratic growth rate.

**7.** The time complexity for the Tower of Hanoi problem is *O*(2*n*). An algorithm with the

*O*(2*n*) time complexity is called an *exponential algorithm*, and it exhibits an exponential

growth rate.

**8.** A Fibonacci number at a given index can be found in *O*(*n*) time using dynamic

programming.

**9.** Dynamic programming is the process of solving subproblems, then combining the solutions

of the subproblems to obtain an overall solution. The key idea behind dynamic

programming is to solve each subproblem only once and store the results for subproblems

for later use to avoid redundant computing of the subproblems.

**10.** Euclid’s GCD algorithm takes *O*(log *n*) time.

**11.** All prime numbers less than or equal to *n* can be found in *O*￠

*n*2*n*

log *n*

≤ time.

**12.** The closest pair can be found in *O*(*n* log *n*) time using the *divide-and-conquer approach*.

**13.** The divide-and-conquer approach divides the problem into subproblems, solves the

subproblems, then combines the solutions of the subproblems to obtain the solution for

the entire problem. Unlike the dynamic programming approach, the subproblems in the

divide-and-conquer approach don’t overlap. A subproblem is like the original problem

with a smaller size, so you can apply recursion to solve the problem.

23.1 Introduction

Sorting is a classic subject in computer science. There are three reasons to study sorting

algorithms.

■ First, sorting algorithms illustrate many creative approaches to problem solving, and

these approaches can be applied to solve other problems.

■ Second, sorting algorithms are good for practicing fundamental programming techniques

using selection statements, loops, methods, and arrays.

■ Third, sorting algorithms are excellent examples to demonstrate algorithm

performance.

The data to be sorted might be integers, doubles, characters, or objects. Section 7.11, Sorting

Arrays, presented selection sort. The selection sort algorithm was extended to sort an array of

objects in Section 19.5, Case Study: Sorting an Array of Objects. The Java API contains several overloaded sort methods for sorting primitive type values and objects in the **java.util.Arrays** and **java.util.Collections** classes.

23.2 Insertion Sort

*The insertion-sort algorithm sorts a list of values by repeatedly inserting a new*

*element into a sorted sublist until the whole list is sorted.*

The insertion sort algorithm presented here sorts a list of elements by repeatedly inserting a

new element into a sorted partial array until the whole array is sorted.

23.3 Bubble Sort

*A bubble sort sorts the array in multiple phases. Each pass successively swaps the*

*neighboring elements if the elements are not in order.*

The bubble sort algorithm makes several passes through the array. On each pass, successive

neighboring pairs are compared. If a pair is in decreasing order, its values are swapped; otherwise,

the values remain unchanged. The technique is called a *bubble sort* or *sinking sort*,

because the smaller values gradually “bubble” their way to the top and the larger values sink

to the bottom. After the first pass, the last element becomes the largest in the array. After the

second pass, the second-to-last element becomes the second largest in the array. This process

is continued until all elements are sorted.

23.4 Merge Sort

*The merge sort algorithm can be described recursively as follows: The algorithm*

*divides the array into two halves and applies a*

The recursive call continues dividing the array into subarrays until each subarray contains

only one element. The algorithm then merges these small subarrays into larger sorted subarrays

until one sorted array results.

**Note**

A merge sort can be implemented efficiently using parallel processing.

The **sort** method

in the **java.util.Arrays** class is implemented using a variation of the merge sort algorithm.

23.5 Quick Sort

*A quick sort works as follows: The algorithm selects an element, called the pivot, in*

*the array. It divides the array into two parts, so that all the elements in the first part*

*are less than or equal to the pivot and all the elements in the second part are greater*

*than the pivot. The quick sort algorithm is then recursively applied to the first part and*

*then the second part.*

Each partition places the pivot in the right place. The selection of the pivot affects the performance

of the algorithm. Ideally, the algorithm should choose the pivot that divides the two

parts evenly.

On the average, the pivot will not divide the array into two parts of the same size or one

empty part each time. Statistically, the sizes of the two parts are very close. Therefore, the

average time is *O*(*n* log*n*). The exact average-case analysis is beyond the scope of this book.

Both merge sort and quick sort employ the divide-and-conquer approach. For merge sort,

the bulk of the work is to merge two sublists, which takes place *after* the sublists are sorted.

For quick sort, the bulk of the work is to partition the list into two sublists, which takes place

*before* the sublists are sorted. Merge sort is more efficient than quick sort in the worst case,

but the two are equally efficient in the average case. Merge sort requires a temporary array

for sorting two subarrays. Quick sort does not need additional array space. Thus, quick sort is

more space efficient than merge sort.

23.6 Heap Sort

*A heap sort uses a binary heap. It first adds all the elements to a heap and then*

*removes the largest elements successively to obtain a sorted list.*

*Heap sorts* use a binary heap, which is a complete binary tree. A binary tree is a hierarchical

structure. It either is empty or it consists of an element, called the *root*, and two distinct binary

trees, called the *left subtree* and *right subtree*. The *length* of a path is the number of the edges

in the path. The *depth* of a node is the length of the path from the root to the node.

A *binary heap* is a binary tree with the following properties:

■ Shape property: It is a complete binary tree.

■ Heap property: Each node is greater than or equal to any of its children.

A binary tree is *complete* if each of its levels is full, except that the last level may not be full

and all the leaves on the last level are placed leftmost.

23.6.1 Storing a Heap

A heap can be stored in an **ArrayList** or an array if the heap size is known in advance.

23.6.2 Adding a New Node

To add a new node to the heap, first add it to the end of the heap and then rebuild the tree as follows:

Let the last node be the current node;

**while** (the current node is greater than its parent) {  
Swap the current node with its parent;

Now the current node is one level up;

}

23.6.3 Removing the Root

Often you need to remove the maximum element, which is the root in a heap. After the root is

removed, the tree must be rebuilt to maintain the heap property. The algorithm for rebuilding

the tree can be described as follows:

Move the last node to replace the root;

Let the root be the current node;

**while** (the current node has children and the current node is

smaller than one of its children) {

Swap the current node with the larger of its children;

Now the current node is one level down;

}

23.6.5 Sorting Using the **Heap** Class

To sort an array using a heap, first create an object using the **Heap** class, add all the elements

to the heap using the **add** method, and remove all the elements from the heap using

the **remove** method. The elements are removed in descending order.

23.6.6 Heap Sort Time Complexity

Since the **add** method traces a path from a leaf to a root, it takes at most *h* steps to add a

new element to the heap. Thus, the total time for constructing an initial heap is *O*(*n* log*n*) for

an array of *n* elements. Since the **remove** method traces a path from a root to a leaf, it takes

at most *h* steps to rebuild a heap after removing the root from the heap. Since the **remove**

method is invoked *n* times, the total time for producing a sorted array from a heap is *O*(*n* log*n*).

Both merge sorts and heap sorts require *O*(*n* log*n*) time. A merge sort requires a temporary

array for merging two subarrays; a heap sort does not need additional array space. Therefore,

a heap sort is more space efficient than a merge sort.

23.7 Bucket Sort and Radix Sort

*Bucket sorts and radix sorts are efficient for sorting integers.*

All sort algorithms discussed so far are general sorting algorithms that work for any types of

keys (e.g., integers, strings, and any comparable objects). These algorithms sort the elements

by comparing their keys. It has been proven that no sorting algorithms based on comparisons

can perform better than *O*(*n* log*n*). However, if the keys are integers, you can use a bucket sort

without having to compare the keys.

Assume that the keys are positive integers. The idea for the *radix sort* is to divide the keys

into subgroups based on their radix positions. It applies a bucket sort repeatedly for the key

values on radix positions, starting from the least-significant position.

**CHAPTER SUMMARY**

**1.** The worst-case complexity for a selection sort, insertion sort, *bubble sort*, and *quick sort*

is *O*(*n*2).

**2.** The average-case and worst-case complexity for a *merge sort* is *O*(*n* log*n*). The average

time for a quick sort is also *O*(*n* log*n*).

**3.** *Heaps* are a useful data structure for designing efficient algorithms such as sorting. You

learned how to define and implement a heap class, and how to insert and delete elements

to/from a heap.

**4.** The time complexity for a *heap sort* is *O*(*n* log*n*).

**5.** *Bucket sorts* and *radix sorts* are specialized sorting algorithms for integer keys. These

algorithms sort keys using buckets rather than by comparing keys. They are more

efficient than general sorting algorithms.

**6.** A variation of the merge sort—called an *external sort*—can be applied to sort large

amounts of data from external files.

24.2 Common Features for Lists

*Common features of lists are defined in the* **List** *interface.*

A list is a popular data structure for storing data in sequential order—for example, a list of

students, a list of available rooms, a list of cities, a list of books. You can perform the following

operations on a list:

■ Retrieve an element from the list.

■ Insert a new element into the list.

■ Delete an element from the list.

■ Find out how many elements are in the list.

■ Determine whether an element is in the list.

■ Check whether the list is empty.

There are two ways to implement a list. One is to use an *array* to store the elements.

Array size is fixed. If the capacity of the array is exceeded, you need to create a new,

larger array and copy all the elements from the current array to the new array. The other

approach is to use a *linked structure*. A linked structure consists of nodes. Each node is

dynamically created to hold an element. All the nodes are linked together to form a list.

Thus you can define two classes for lists. For convenience, let’s name these two classes

**MyArrayList** and **MyLinkedList**. These two classes have common operations but different

implementations.

**Design Guide**

The common operations can be generalized in an interface or an abstract class. A good

strategy is to combine the virtues of interfaces and abstract classes by providing both an

interface and a convenience abstract class in the design so that the user can use either

of them, whichever is convenient. The abstract class provides a skeletal implementation

of the interface, which minimizes the effort required to implement the interface.

**Design Guide**

Protected data fields are rarely used. However, making **size** a protected data field in

the **MyAbstractList** class is a good choice. The subclass of **MyAbstractList** can

access **size**, but nonsubclasses of **MyAbstractList** in different packages cannot

access it. As a general rule, you can declare protected data fields in abstract classes.

24.3 Array Lists

*An array list is implemented using an array.*

An array is a fixed-size data structure. Once an array is created, its size cannot be changed.

Nevertheless, you can still use arrays to implement dynamic data structures. The trick is to

create a larger new array to replace the current array, if the current array cannot hold new

elements in the list.

Initially, an array, say **data** of **E[]** type, is created with a default size. When inserting a

new element into the array, first make sure that there is enough room in the array. If not, create

a new array twice as large as the current one. Copy the elements from the current array to

the new array. The new array now becomes the current array. Before inserting a new element

at a specified index, shift all the elements after the index to the right and increase the list size

by **1**,

To remove an element at a specified index, shift all the elements after the index to the left

by one position and decrease the list size by **1**,

24.4 Linked Lists

*A linked list is implemented using a linked structure.*

Since **MyArrayList** is implemented using an array, the methods **get(int index)** and

**set(int index, E e)** for accessing and modifying an element through an index and the  
**add(E e)** method for adding an element at the end of the list are efficient. However, the

methods **add(int index, E e)** and **remove(int index)** are inefficient, because they

require shifting a potentially large number of elements. You can use a linked structure to

implement a list to improve efficiency for adding and removing an element at the beginning

of a list.

24.4.1 Nodes

In a linked list, each element is contained in an object, called the *node*. When a new element

is added to the list, a node is created to contain it. Each node is linked to its next neighbor,

A linked list consists of any number of nodes chained together.

Each node contains the element and a data field named **next** that points to the next element. If

the node is the last in the list, its pointer data field **next** contains the value **null**. You can use

this property to detect the last node.

24.4.4 **MyArrayList** vs. **MyLinkedList**

Both **MyArrayList** and **MyLinkedList** can be used to store a list. **MyArrayList** is implemented

using an array and **MyLinkedList** is implemented using a linked list. The overhead

of **MyArrayList** is smaller than that of **MyLinkedList**. However, **MyLinkedList** is more

efficient if you need to insert elements into and delete elements from the beginning of the list.

24.4.5 Variations of Linked Lists

The linked list introduced in the preceding sections is known as a *singly linked list*. It contains

a pointer to the list’s first node, and each node contains a pointer to the next node sequentially.

Several variations of the linked list are useful in certain applications.

A *circular, singly linked list* is like a singly linked list, except that the pointer of the last

node points back to the first node, as shown in Figure 24.18a. Note that **tail** is not needed

for circular linked lists. **head** points to the current node in the list. Insertion and deletion take

place at the current node. A good application of a circular linked list is in the operating system

that serves multiple users in a timesharing fashion. The system picks a user from a circular list

and grants a small amount of CPU time, then moves on to the next user in the list.

A *doubly linked list* contains nodes with two pointers. One points to the next node and the

other to the previous node, as shown in Figure 24.18b. These two pointers are conveniently

called *a forward pointer* and *a backward pointer*. Thus, a doubly linked list can be traversed forward

and backward. The **java.util.LinkedList** class is implemented using a doubly linked

list, and it supports traversing of the list forward and backward using the **ListIterator**.

A *circular*, *doubly linked list* is like a doubly linked list, except that the forward pointer of

the last node points to the first node and the backward pointer of the first pointer points to the

last node,

24.5 Stacks and Queues

*Stacks can be implemented using array lists and queues can be implemented using*

*linked lists.*

A stack can be viewed as a special type of list whose elements are accessed, inserted, and

deleted only from the end (top), as shown in Figure 10.11. A queue represents a waiting list.

It can be viewed as a special type of list whose elements are inserted into the end (tail) of the

queue, and are accessed and deleted from the beginning (head),

Since the insertion and deletion operations on a stack are made only at the end of the stack,

it is more efficient to implement a stack with an array list than a linked list. Since deletions are

made at the beginning of the list, it is more efficient to implement a queue using a linked list

than an array list.

■ Using inheritance: You can define a stack class by extending **ArrayList**, and a

queue class by extending **LinkedList**,

■ Using composition: You can define an array list as a data field in the stack class, and

a linked list as a data field in the queue class,

Both designs are fine, but using composition is better because it enables you to define a completely

new stack class and queue class without inheriting the unnecessary and inappropriate

methods from the array list and linked list. The implementation of the stack class using the composition

approach was given GenericStack.java implements the

**GenericQueue** class using the composition approach.

For a stack, the **push(e)** method adds an element to the top of the stack, and the **pop()**

method removes the top element from the stack and returns the removed element. It is easy to

see that the time complexity for the **push** and **pop** methods is *O*(1).

For a queue, the **enqueue(e)** method adds an element to the tail of the queue, and the

**dequeue()** method removes the element from the head of the queue. It is easy to see that the

time complexity for the **enqueue** and **dequeue** methods is *O*(1).

24.6 Priority Queues

*Priority queues can be implemented using heaps.*

An ordinary queue is a first-in, first-out data structure. Elements are appended to the end of

the queue and removed from the beginning. In a *priority queue*, elements are assigned with

priorities. When accessing elements, the element with the highest priority is removed first.

A priority queue can be implemented using a heap, in which the root is the object with the

highest priority in the queue.

**CHAPTER SUMMARY  
2.** To define a data structure is essentially to define a class. The class for a data structure

should use data fields to store data and provide methods to support operations such as

insertion and deletion.

**3.** To create a data structure is to create an instance from the class. You can then apply the

methods on the instance to manipulate the data structure, such as inserting an element

into the data structure or deleting an element from the data structure.

25.1 Introduction

*A tree is a classic data structure with many important applications.*

A *tree* provides a hierarchical organization in which data are stored in the nodes.

25.2 Binary Search Trees

*A binary search tree can be implemented using a linked structure.*

Recall that lists, stacks, and queues are linear structures that consist of a sequence of

elements. A *binary tree* is a hierarchical structure. It either is empty or consists of an

element, called the *root*, and two distinct binary trees, called the *left subtree* and *right*

*subtree*, either or both of which may be empty.

The *length* of a path is the number of the edges in the path. The *depth* of a node is

the length of the path from the root to the node. The set of all nodes at a given depth

is sometimes called a *level* of the tree. *Siblings* are nodes that share the same parent

node. The root of a left (right) subtree of a node is called a *left (right) child* of the node.

A node without children is called a *leaf*. The height of a nonempty tree is the length

of the path from the root node to its furthest leaf. The *height* of a tree that contains a

single node is **0**. Conventionally, the height of an empty tree is **—1**.

A special type of binary tree called a *binary search tree* (BST) is often useful. A BST (with

no duplicate elements) has the property that for every node in the tree, the value of any node in

its left subtree is less than the value of the node, and the value of any node in its right subtree

is greater than the value of the node. The binary trees in Figure 25.1 are all BSTs.

25.2.1 Representing Binary Search Trees

A binary tree can be represented using a set of linked nodes. Each node contains a value and

two links named *left* and *right* that reference the left child and right child, respectively,

25.2.2 Searching for an Element

To search for an element in the BST, you start from the root and scan down from it until a

match is found or you arrive at an empty subtree.

25.2.4 Tree Traversal

*Tree traversal* is the process of visiting each node in the tree exactly once. There are several

ways to traverse a tree. This section presents *inorder*, *postorder, preorder*, *depth-first,* and

*breadth-first* traversals.

With *inorder traversal*, the left subtree of the current node is visited first recursively, then

the current node, and finally the right subtree of the current node recursively. The inorder

traversal displays all the nodes in a BST in increasing order.

With *postorder traversal*, the left subtree of the current node is visited recursively first,

then recursively the right subtree of the current node, and finally the current node itself. An

application of postorder is to find the size of the directory in a file system.

each directory is an internal node and a file is a leaf node. You can apply postorder

to get the size of each file and subdirectory before finding the size of the root directory.

With *preorder traversal*, the current node is visited first, then recursively the left subtree

of the current node, and finally the right subtree of the current node recursively. Depth-first

traversal is the same as preorder traversal. An application of preorder is to print a structured

document.

**Note**

You can reconstruct a binary search tree by inserting the elements in their preorder.

The reconstructed tree preserves the parent and child relationship for the nodes in the

original binary search tree.

With breadth-first traversal, the nodes are visited level by level. First the root is visited,

then all the children of the root from left to right, then the grandchildren of the root from left

to right, and so on.

25.3 Deleting Elements from a BST

*To delete an element from a BST, first locate it in the tree and then consider two*

*cases—whether or not the node has a left child—before deleting the element and*

*reconnecting the tree.*

To delete an element from a binary search tree, you need to first locate the node that

contains the element and also its parent node. Let **current** point to the node that contains the

element in the binary search tree and **parent** point to the parent of the **current** node. The

**current** node may be a left child or a right child of the **parent** node. There are two cases

to consider.

***Case 1:*** The current node does not have a left child,

In

this case, simply connect the parent with the right child of the current node,

***Case 2:*** The **current** node has a left child. Let **rightMost** point to the node that contains the

largest element in the left subtree of the **current** node and **parentOfRightMost** point to the

parent node of the **rightMost** node, as shown in Figure 25.12a. Note that the **rightMost** node

cannot have a right child but may have a left child. Replace the element value in the **current**

node with the one in the **rightMost** node, connect the **parentOfRightMost** node with the

left child of the **rightMost** node, and delete the **rightMost** node,

**Note**

If the left child of **current** does not have a right child, **current.left** points to the

large element in the left subtree of **current**. In this case, **rightMost** is **current.**

**left** and **parentOfRightMost** is **current**. You have to take care of this special

case to reconnect the right child of **rightMost** with **parentOfRightMost**.

25.4 Tree Visualization and MVC

*You can use recursion to display a binary tree.*

**Pedagogical Note**

One challenge facing the data-structure course is to motivate students. Displaying a binary

tree graphically will not only help you understand the working of a binary tree but perhaps

also stimulate your interest in programming. This section introduces the techniques to

visualize binary trees. You can also apply visualization techniques to other projects.

How do you display a binary tree? It is a recursive structure, so you can display a binary tree

using recursion. You can simply display the root, then display the two subtrees recursively.

The techniques for displaying the Sierpinski triangle (Listing 18.9, SierpinskiTriangle.java)

can be applied to displaying a binary tree.

Tree visualization is an example of the model-view-controller (MVC) software architecture.

This is an important architecture for software development. The model is for storing and

handling data. The view is for visually presenting the data. The controller handles the user

interaction with the model and controls the view,

The MVC architecture separates data storage and handling from the visual representation

of the data. It has two major benefits:

■ It makes multiple views possible so that data can be shared through the same model.

For example, you can create a new view that displays the tree with the root on the left

and tree grows horizontally to the right (see Programming Exercise 25.11).

■ It simplifies the task of writing complex applications and makes the components

scalable and easy to maintain. Changes can be made to the view without affecting the

model, and vice versa.

25.5 Iterators

**BST** *is iterable because it is defined as a subtype of the* **java.lang.Iterable** *interface.*

The methods **inorder()**, **preorder()**, and **postorder()** display the elements in **inorder**,

**preorder**, and **postorder** in a binary tree. These methods are limited to displaying the elements

in a tree. If you wish to process the elements in a binary tree rather than display them,

these methods cannot be used. Recall that an iterator is provided for traversing the elements

in a set or list. You can apply the same approach in a binary tree to provide a uniform way of

traversing the elements in a binary tree.

The **java.lang.Iterable** interface defines the **iterator** method, which returns an

instance of the **java.util.Iterator** interface. The **java.util.Iterator** interface (see

Figure 25.19) defines the common features of iterators.

The **Iterator** interface defines a uniform way of traversing the elements in

a container.

The **Tree** interface extends **java.lang.Iterable**. Since **BST** is a subclass of

**AbstractTree** and **AbstractTree** implements **Tree**, **BST** is a subtype of **Iterable**.

The **Iterable** interface contains the **iterator()** method that returns an instance of

**java.util.Iterator**.

You can traverse a binary tree in inorder, preorder, or postorder. Since inorder is used

frequently, we will use inorder for traversing the elements in a binary tree. We define an

iterator class named **InorderIterator** to implement the **java.util.Iterator** interface

**Design Guide**

Iterator is an important software design pattern. It provides a uniform way of traversing

the elements in a container, while hiding the container’s structural details. By implementing

the same interface **java.util.Iterator**, you can write a program that

traverses the elements of all containers in the same way.

**Note**

**java.util.Iterator** defines a forward iterator, which traverses the elements in

the iterator in a forward direction, and each element can be traversed only once. The

Java API also provides the **java.util.ListIterator**, which supports traversing in

both forward and backward directions. If your data structure warrants flexible traversing,

you may define iterator classes as a subtype of **java.util.ListIterator**.

The implementation of the iterator is not efficient. Every time you remove an element through

the iterator, the whole list is rebuilt (line 261 in Listing 25.5 BST.java). The client should

always use the **delete** method in the **BinraryTree** class to remove an element. To prevent

the user from using the **remove** method in the iterator, implement the iterator as follows:

**public void** remove() {

**throw new** UnsupportedOperationException

(**"Removing an element from the iterator is not supported"**);

}

After making the **remove** method unsupported by the iterator class, you can implement the iterator

more efficiently without having to maintain a list for the elements in the tree. You can use a stack

to store the nodes, and the node on the top of the stack contains the element that is to be returned

from the **next()** method. If the tree is well-balanced, the maximum stack size will be *O*(log*n*).

**CHAPTER SUMMARY**

**1.** A *binary search tree* (BST) is a hierarchical data structure. You learned how to define

and implement a BST class, how to insert and delete elements into/from a BST, and

how to traverse a BST using *inorder*, *postorder*, *preorder*, depth-first, and breadth-first

searches.

**2.** An iterator is an object that provides a uniform way of traversing the elements in a container,

such as a set, a list, or a *binary tree*. You learned how to define and implement

iterator classes for traversing the elements in a binary tree.

26.4 Overriding the **insert** Method

*Inserting an element into an AVL tree is the same as inserting it to a BST, except that*

*the tree may need to be rebalanced.*

A new element is always inserted as a leaf node. As a result of adding a new node, the heights

of the new leaf node’s ancestors may increase. After inserting a new node, check the nodes

along the path from the new leaf node up to the root. If an unbalanced node is found, perform

an appropriate rotation using the algorithm

The algorithm considers each node in the path from the new leaf node to the root. Update

the height of the node on the path. If a node is balanced, no action is needed. If a node is not

balanced, perform an appropriate rotation.

26.5 Implementing Rotations

*An unbalanced tree becomes balanced by performing an appropriate rotation operation.*

26.6 Implementing the **delete** Method

*Deleting an element from an AVL tree is the same as deleing it from a BST, except that*

*the tree may need to be rebalanced.*

to delete an element from a

binary tree, the algorithm first locates the node that contains the element. Let **current** point

to the node that contains the element in the binary tree and **parent** point to the parent of the

**current** node. The **current** node may be a left child or a right child of the **parent** node.

Two cases arise when deleting an element.

***Case 1:*** The **current** node does not have a left child, To delete

the **current** node, simply connect the **parent** node with the right child of the **current**

node

The height of the nodes along the path from the **parent** node up to the **root** may have

decreased. To ensure that the tree is balanced, invoke

balancePath(parent.element); // Defined in Listing 26.1

***Case 2:*** The **current** node has a left child. Let **rightMost** point to the node that contains

the largest element in the left subtree of the **current** node and **parentOfRightMost** point

to the parent node of the **rightMost** node, The **rightMost** node

cannot have a right child but it may have a left child. Replace the element value in the **current**

node with the one in the **rightMost** node, connect the **parentOfRightMost** node with the

left child of the **rightMost** node, and delete the **rightMost** node

The height of the nodes along the path from **parentOfRightMost** up to the root may have

decreased. To ensure that the tree is balanced, invoke

balancePath(parentOfRightMost);

26.7 The **AVLTree** Class

*The* **AVLTree** *class extends the* **BST** *class to override the* **insert** *and* **delete**

*methods to rebalance the tree if necessary.*

26.9 AVL Tree Time Complexity Analysis

*Since the height of an AVL tree is O(log n), the time complexity of the* **search***,*

**insert***, and* **delete** *methods in* **AVLTree** *is O(log n).*

The time complexity of the **search**, **insert**, and **delete** methods in **AVLTree** depends on

the height of the tree. We can prove that the height of the tree is *O*(log *n*).

The **search**, **insert**, and **delete** methods involve only the nodes along a path in the tree.

The **updateHeight** and **balanceFactor** methods are executed in a constant time for each

node in the path. The **balancePath** method is executed in a constant time for a node in the

path. Thus, the time complexity for the **search**, **insert**, and **delete** methods is *O*(log *n*).

**CHAPTER SUMMARY**

**1.** An *AVL tree* is a *well-balanced* binary tree. In an AVL tree, the difference between the

heights of two subtrees for every node is **0** or **1**.

**2.** The process for inserting or deleting an element in an AVL tree is the same as in a regular

binary search tree. The difference is that you may have to rebalance the tree after an

insertion or deletion operation.

**3.** Imbalances in the tree caused by insertions and deletions are rebalanced through subtree

rotations at the node of the imbalance.

**4.** The process of rebalancing a node is called a *rotation*. There are four possible rotations:

*LL rotation*, *LR rotation*, *RR rotation*, and *RL rotation*.

**5.** The height of an AVL tree is *O*(log *n*). Therefore, the time complexities for the **search**,

**insert**, and **delete** methods are *O*(log *n*).

27.1 Introduction

*Hashing is superefficient. It takes O(1) time to search, insert, and delete an element*

*using hashing.*

27.2 What Is Hashing?

*Hashing uses a hashing function to map a key to an index.*

map, which is a data structure that is implemented

using hashing. Recall that a *map* is a container object that stores

entries. Each entry contains two parts: a *key* and a *value*. The key, also called a *search key*, is

used to search for the corresponding value. For example, a dictionary can be stored in a map,

in which the words are the keys and the definitions of the words are the values.

**Note**

A map is also called a *dictionary*, a *hash table*, or an *associative array*.

The Java Collections Framework defines the **java.util.Map** interface for modeling maps.

Three concrete implementations are **java.util.HashMap**, **java.util.LinkedHashMap**,

and **java.util.TreeMap**. **java.util.HashMap** is implemented using hashing, **java.**

**util.LinkedHashMap** using **LinkedList**, and **java.util.TreeMap** using red-black

trees. You will learn the concept of hashing

and use it to implement a hash map in this chapter.

If you know the index of an element in the array, you can retrieve the element using the

index in *O*(1) time. So does that mean we can store the values in an array and use the key as

the index to find the value? The answer is yes—if you can map a key to an index. The array

that stores the values is called a *hash table*. The function that maps a key to an index in the

hash table is called a *hash function*. A hash function obtains an index

from a key and uses the index to retrieve the value for the key. *Hashing* is a technique that

retrieves the value using the index obtained from the key without performing a search.

A hash function maps a key to an index in the hash table.

we would like to design a function that maps each search key to a different index in the hash table. Such  
a function is called a *perfect hash function*. However, it is difficult to find a perfect hashfunction. When two or more keys are mapped to the same hash value, we say that a *collision*

has occurred. Although there are ways to deal with collisions, which are discussed later in this

chapter, it is better to avoid collisions in the first place. Thus, you should design a fast and

easy-to-compute hash function that minimizes collisions.

27.3 Hash Functions and Hash Codes

*A typical hash function first converts a search key to an integer value called a hash*

*code, then compresses the hash code into an index to the hash table.*

Java’s root class **Object** has the **hashCode** method, which returns an integer hash code. By

default, the method returns the memory address for the object. The general contract for the

**hashCode** method is as follows:

1. You should override the **hashCode** method whenever the **equals** method is overridden

to ensure that two equal objects return the same hash code.

2. During the execution of a program, invoking the **hashCode** method multiple times

returns the same integer, provided that the object’s data are not changed.

3. Two unequal objects may have the same hash code, but you should implement the

**hashCode** method to avoid too many such cases.

27.3.1 Hash Codes for Primitive Types

For search keys of the type **byte**, **short**, **int**, and **char**, simply cast them to **int**. Therefore,

two different search keys of any one of these types will have different hash codes.

For a search key of the type **float**, use **Float.floatToIntBits(key)** as the hash

code. Note that **floatToIntBits(float f)** returns an **int** value whose bit representation

is the same as the bit representation for the floating number **f**. Thus, two different search keys

of the **float** type will have different hash codes.

For a search key of the type **long**, simply casting it to **int** would not be a good choice,

because all keys that differ in only the first 32 bits will have the same hash code. To take the

first 32 bits into consideration, divide the 64 bits into two halves and perform the exclusiveor

operation to combine the two halves. This process is called *folding*. The hash code for a

**long** key is

**int** hashCode = (**int**)(key ^ (key >> **32**));

Note that **>>** is the right-shift operator that shifts the bits 32 positions to the right.

For a search key of the type **double**, first convert it to a **long** value using the

**Double.doubleToLongBits** method, and then perform a folding as follows:

**long** bits = Double.doubleToLongBits(key);

**int** hashCode = (**int**)(bits ^ (bits >> **32**));

27.3.2 Hash Codes for Strings

Search keys are often strings, so it is important to design a good hash function for strings. An

intuitive approach is to sum the Unicode of all characters as the hash code for the string. This

approach may work if two search keys in an application don’t contain the same letters, but  
it will produce a lot of collisions if the search keys contain the same letters, such as **tod** and

**dot**.

A better approach is to generate a hash code that takes the position of characters into consideration.

27.3.3 Compressing Hash Codes

The hash code for a key can be a large integer that is out of the range for the hash-table index,

so you need to scale it down to fit in the index’s range.

27.4 Handling Collisions Using Open Addressing

*A collision occurs when two keys are mapped to the same index in a hash table.*

*Generally, there are two ways for handling collisions: open addressing and separate*

*chaining.*

*Open addressing* is the process of finding an open location in the hash table in the event of

a collision. Open addressing has several variations: *linear probing*, *quadratic probing*, and

*double hashing*.

27.4.1 Linear Probing

When a collision occurs during the insertion of an entry to a hash table, *linear probing* finds

the next available location sequentially.

To remove an entry from the hash table, search the entry that matches the key. If the entry

is found, place a special marker to denote that the entry is available. Each cell in the hash table

has three possible states: occupied, marked, or empty. Note that a marked cell is also available

for insertion.

Linear probing tends to cause groups of consecutive cells in the hash table to be occupied.

Each group is called a *cluster*. Each cluster is actually a probe sequence that you must search

when retrieving, adding, or removing an entry. As clusters grow in size, they may merge into even

larger clusters, further slowing down the search time. This is a big disadvantage of linear probing.

27.4.2 Quadratic Probing

*Quadratic probing* can avoid the clustering problem that can occur in linear probing. Linear

probing looks at the consecutive cells beginning at index *k*.

Quadratic probing works in the same way as linear probing except for a change in the

search sequence. Quadratic probing avoids linear probing’s clustering problem, but it has its

own clustering problem, called *secondary clustering*; that is, the entries that collide with an

occupied entry use the same probe sequence.

Linear probing guarantees that an available cell can be found for insertion as long as the

table is not full. However, there is no such guarantee for quadratic probing.

27.4.3 Double Hashing

Another open addressing scheme that avoids the clustering problem is known as *double hashing*.

27.5 Handling Collisions Using Separate Chaining

*The separate chaining scheme places all entries with the same hash index in the same*

*location, rather than finding new locations. Each location in the separate chaining*

*scheme uses a bucket to hold multiple entries.*

You can implement a bucket using an array, **ArrayList**, or **LinkedList**. We will use

**LinkedList** for demonstration. You can view each cell in the hash table as the reference to

the head of a linked list, and elements in the linked list are chained starting from the head,

27.6 Load Factor and Rehashing

*The load factor measures how full a hash table is. If the load factor is exceeded,*

*increase the hash-table size and reload the entries into a new larger hash table. This*

*is called rehashing.*

*Load factor* l (*lambda*) measures how full a hash table is. It is the ratio of the number of

elements to the size of the hash table, that is, l =

*n*

*N*

, where *n* denotes the number of elements

and *N* the number of locations in the hash table.

Note that l is zero if the hash table is empty. For the open addressing scheme, l is between

**0** and **1**; l is **1** if the hash table is full. For the separate chaining scheme, l can be any value.

As l increases, the probability of a collision increases. Studies show that you should maintain

the load factor under **0.5** for the open addressing scheme and under **0.9** for the separate

chaining scheme.

Keeping the load factor under a certain threshold is important for the performance of hashing.

In the implementation of the **java.util.HashMap** class in the Java API, the threshold

**0.75** is used. Whenever the load factor exceeds the threshold, you need to increase the hashtable

size and *rehash* all the entries in the map into a new larger hash table. Notice that you

need to change the hash functions, since the hash-table size has been changed. To reduce the

likelihood of rehashing, since it is costly, you should at least double the hash-table size. Even

with periodic rehashing, hashing is an efficient implementation for map.

27.8 Implementing Set Using Hashing

*A hash set can be implemented using a hash map.*

A*set* (introduced in Chapter 21) is a data structure that stores distinct values. The Java Collections

Framework defines the **java.util.Set** interface for modeling sets. Three concrete implementations

are**java.util.HashSet**,**java.util.LinkedHashSet**, and **java.util.TreeSet**.

**java.util.HashSet** is implemented using hashing, **java.util.LinkedHashSet** using

**LinkedList**, and **java.util.TreeSet** using red-black trees.

You can implement **MyHashSet** using the same approach as for implementing **MyHashMap**.

The only difference is that key/value pairs are stored in the map, while elements are stored in

the set.

**CHAPTER SUMMARY**

**1.** A *map* is a data structure that stores entries. Each entry contains two parts: a *key* and a

*value*. The key is also called a *search key*, which is used to search for the corresponding

value. You can implement a map to obtain *O*(1) time complexity on searching, retrieval,

insertion, and deletion using the hashing technique.

**2.** A *set* is a data structure that stores elements. You can use the hashing technique to implement

a set to achieve *O*(1) time complexity on searching, insertion, and deletion for a set.

**3.** *Hashing* is a technique that retrieves the value using the index obtained from a key

without performing a search. A typical *hash function* first converts a search key to  
an integer value called a *hash code*, then compresses the hash code into an index to the

*hash table*.

**4.** A *collision* occurs when two keys are mapped to the same index in a hash table. Generally,

there are two ways for handling collisions: *open addressing* and *separate chaining*.

**5.** Open addressing is the process of finding an open location in the hash table in the event

of collision. Open addressing has several variations: *linear probing*, *quadratic probing*,

and *double hashing*.

**6.** The *separate chaining* scheme places all entries with the same hash index into the

same location, rather than finding new locations. Each location in the separate chaining

scheme is called a *bucket*. A bucket is a container that holds multiple entries.

+