The Sun Certified Java Developer Exam with J2SE 1.4

MEHRAN HABIBI, JEREMY PATTERSON, AND TERRY CAMERLENGO

Apress™

The Sun Certified Java Developer Exam with J2SE 1.4
Copyright © 2002 by Mehran Habibi, Jeremy Patterson, and Terry Camerlengo

ISBN (pbk): 1-59059-030-9

Printed and bound in the United States of America 12345678910

Technical Reviewer: Brian Sam-Bodden

Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Simon Hayes, Karen Watterson, John Zukowski

Managing Editor and Production Editor: Grace Wong

Project Manager: Alexa Stuart

Copy Editor: Nicole LeClerc

Compositor: Susan Glinert

Artist: Cara Brunk, Blue Mud Productions

Indexer: Ann Rogers

Cover Designer and Manufacturing Manager: Tom Debolski

Marketing Manager: Stephanie Rodriguez

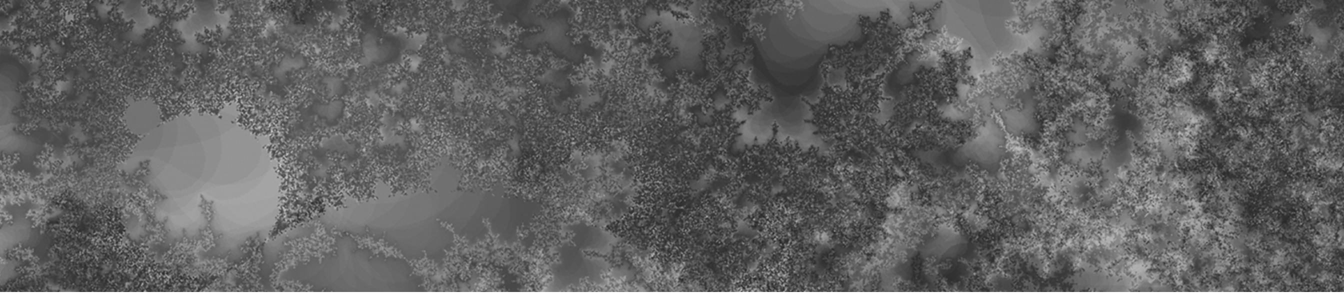
Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`.

Outside the United States, fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.



CHAPTER 7
NIO

- Learn about NIO.
- Understand buffers.
- Examine ByteChannels.
- Examine FileChannels.
- Examine SocketChannels.
- Explore alternative implementation directions.
- Learn how to use regular expressions.

THROUGHOUT MANY REVISIONS to the Java platform, Java I/O has remained relatively unchanged. Since Java 1.0, I/O has been stream-based, which allowed for the input and output of byte data. Java 1.2 introduced the concept of `Readers` and `Writers`, introduced in JDK 1.1 allowed for easy output of more complex data types, such as `Strings`. `Readers` and `Writers` also introduced the ability to use various character encodings, thus allowing internationalization.

Java's I/O implementation has grown into a very robust method of data input and output. It does, however, have many limitations. First and foremost, Java has a relatively slow I/O framework. Even though most of the JVM's I/O implementation has migrated to a native code implementation, serious speed issues still plague the framework. On average, Java's I/O capabilities are much slower than those achieved with other natively compiled languages, such as C and C++.

Traditional Java I/O has also always blocked. As you recall from Chapter 4, to *block* means to wait for an event to occur and to retain any held resources until that event does occur. For example, say you have a thread that locks an object and opens a blocking network connection. This thread will be incapable of releasing the locked object until the network operation has completed. When a Java application attempts to access network or file data, blocking will occur until the data transfer is complete. Thus, even simple I/O operations must be accounted for in order to prevent slow performance or "hanging" while they read or write data. This can be particularly important when you are striving to maintain a responsive UI for an application.

NIO: The Future of Java I/O

The release of J2SE 1.4 introduces the first fundamental change to the Java I/O framework: `NIO`. `NIO` is Java's *new I/O* package, and it is a strong supplement to traditional I/O. The changes found in `NIO` are quite extensive. This chapter offers an introduction to the new features in the `NIO` implementation as they apply to the book's sample project.

As you would expect, `NIO` is an incredibly vast subject. This chapter offers an introduction to the new features and changes, but it only scratches the surface. For a complete listing of all the changes and new additions to the Java I/O framework, please consult Sun's API documentation at `http://java.sun.com/j2se/1.4/docs/guide/nio/index.html`.

Advantages of NIO

`NIO` offers several advantages over traditional I/O, the most dramatic of which is nonblocking I/O for network connections. In addition, there is direct memory mapping, `ByteBuffer` support, platform-specific optimizations, smooth integration between the various channels, the capability to read and write to a channel at the same time (as opposed to streams), and I/O and threads that can be interrupted. Finally, `NIO` is fully compatible with previous I/O implementations. Thus, using `NIO` will not break existing code.

NOTE A channel is, logically speaking, an open, two-way connection to a resource. For example, you can open channels to files, hardware devices, remote devices, a program that performs I/O, and so forth.

Changes to Existing File I/O

There are several important changes to the existing file I/O mechanism. The changes most relevant to the approach advocated by this book follow. The `FileInputStream`, `FileOutputStream`, and `RandomAccessFile` classes have been changed to provide a `getChannel` method, which returns a `FileChannel` associated with the stream in question.

`RandomAccessFile` can now be opened in four modes. Modes `'r'` and `'rw'` work the way they traditionally did, opening the file for reading and read/write access, respectively. In addition to the older access methods, there is now an `'rws'` mode, which opens a file for read/write access and persists any changes to the file or its metadata to the local system synchronously. This is guaranteed behavior for files opened on the local file system. Mode `'rwd'` acts similarly, except that it only synchronously persists changes to the file's content, as opposed to its metadata, such as a permission or timestamp change. You should use the `'rwd'` mode if you only need to persist changes to a file's data, but you should use `'rws'` if you need to persist changes such as access time to its metadata. For example, you might obtain a channel from a `RandomAccessFile` in the following way:

```
RandomAccessFile raf = new RandomAccessFile("tmp.txt","rwd");
FileChannel fc = raf.getChannel();
```

These changes are also in effect for the `FileChannel` associated with the `RandomAccessFile`. A `FileChannel` obtained from a `RandomAccessFile` in `'rwd'` mode is guaranteed to persist content changes synchronously, so long as it is working with a file on the local system and not a networked connection.

`FileInputStream` and `FileOutputStream` also support the `getChannel` method.

It is important to note that closing a `FileInputStream`, `FileOutputStream`, or `RandomAccessFile` closes the associated `FileChannel`, thus you should be careful when modifying the underlying stream. Similarly, moving the position in a `FileInputStream`, `FileOutputStream`, and/or `RandomAccessFile` moves the position in the associated `FileChannel`. The reverse is also true: Moving the position in a `FileChannel` also moves the position in the associated `FileInputStream`, `FileOutputStream`, and/or `RandomAccessFile`.

A question naturally arises at this point: What is the difference between a stream and a channel? Excellent question. The answer is that channels are two-way streets, while streams are one-way. But channels are superior to streams in many ways outside of this. We'll detail all of that soon.

Changes to Network I/O

The most exciting change in networking I/O, as far as the SCJD exam is concerned, is the introduction of nonblocking I/O and channels. This new addition is supported by several additions to the existing I/O structure, which are detailed in the sections that follow.

InetSocketAddress

`InetSocketAddress` is a new addition to the `java.net` package. It is an extension of the `SocketAddress` class and offers the ability to bind to a host though a host name and port, or an IP address and port. If you need to connect to a remote host through a socket connection, this class can greatly simplify the process.

`InetSocketAddress` has three constructors. The first takes another `InetAddress` object and `int` port number. The second takes an `int` port number, and the last takes a `String` host name and `int` port number.

This class offers an `InetAddress getAddress` method and a `String getHostName` method. Both are self-explanatory.

Socket

The `Socket` class has been retrofitted with the methods in Table 7-1. This is by no means an exhaustive list of the changes incurred, but it should address the most common questions.

Table 7-1. New Socket Methods

Signature	**Explanation**
public void connect(SocketAddress endpoint) throws IOException	This method will connect to the specified SocketAddress object. It will also throw an IllegalBlockingModeException if the socket in question has an associated nonblocking channel.
public void connect(SocketAddress endpoint, int timeout) throws IOException	Very much like the previous method, except that this method throws an SocketTimeoutException if the timeout period expires before the method returns. This can be a powerful tool in dealing with connections that might previously have blocked indefinitely.
public void bind(SocketAddress bindpoint)	Works very much like connect, except that it does not throw the IllegalBlockingModeException.
public SocketAddress getRemoteSocketAddress()	Simply gets the SocketAddress remotely connected to or returns null if there is no connected remote SocketAddress.
public SocketAddress getLocalSocketAddress()	Simply gets the SocketAddress of the local port or returns null if not bound.
SocketChannel GetChannel()	Returns the associated SocketChannel if this socket was created using a SocketChannel.open or ServerSocketChannel.accept.
public boolean isConnected()	Is the socket still connected?
public boolean isBound()	Returns whether the socket is still bound.
public boolean isClosed()	Is the socket is closed?

The best way to gain familiarity with the `Socket` class is to look over the documentation and code some simple examples. We recommend that you take a few moments and do so.

ServerSocket

Table 7-2 succinctly details some of the more commonly used methods in the `ServerSocket` class.

Table 7-2. ServerSocket Methods

Signature	**Explanation**
public void bind(SocketAddress endpoint) throws IOException	This method will connect to the specified SocketAddress socket. If the address is null, then it will connect to a short-lived port and bind to that port. A short-lived port is a port that exists only for the duration of the single operation that uses it.
public void bind(SocketAddress endpoint, int backlog) throws IOException	This method will simply connect to the specified SocketAddress object. If the address is null, then it will connect to the backlog port. If the backlog port is less than 1, then a short-lived port will be picked and bound to.
Public SocketAddress getLocalSocketAddress()	Gets the SocketAddress of the local port or returns null if not bound.
SocketChannel getChannel()	Returns the associated ServerSocketChannel if this ServerSocket was created using a ServerSocketChannel.open method.
public void connect(SocketAddress endpoint, int timeout) throws IOException	Very much like the previous method, except that this method throws a SocketTimeoutException if the timeout period expires before the method returns. This can be a powerful tool in dealing with connections that might previously have blocked indefinitely.
Public void bind(SocketAddress bindpoint)	Works very much like connect, except that it does not throw the IllegalBlockingModeException.
Public SocketAddress getRemoteSocketAddress()	Simply gets the SocketAddress remotely connected to or returns null if there is no connected remote SocketAddress.

Table 7-2. ServerSocket Methods (Continued)

Signature	**Explanation**
Public SocketAddress getLocalSocketAddress()	Simply gets the SocketAddress of the local port or returns null if not bound.
public boolean isBound()	Is the connection currently bound?.

SocketAddress

`SocketAddress` is a new abstract class put into place so that other classes that have specific protocols can subclass it. It is designed to support evolving and custom protocols. It is extended by the `InetSocketAddress` class.

Buffers

A *buffer*, in a generic sense, is a fixed container of primitive data. Buffers are enerally stored in memory, which allows for the rapid manipulation of data. For example, a `BufferedInputStream` maintains an internal buffer of byte data. Thus, a `BufferedInputStream` would be an ideal mechanism to read data from across a network.

How Buffers Work

Imagine a scenario under which a file sits on a remote server. An application needs to download the file from the remote server and copy it to the local file system. The application can be implemented one of two ways. First, the application could read a byte from the remote file and then write the corresponding byte to a local file. This method would essentially copy the file byte by byte across the network, resulting in extremely slow data transfers. This would be like unpacking groceries from your car one item at a time.

A second option is the use of a buffering mechanism, such as a `BufferedInputStream`. Under this scenario, large portions of the remote file are transferred as opposed to reading it one byte at a time. After the memory buffer is full, it is written to the local file system in one large chunk. This would be like unpacking groceries from your car by picking up as much as you can carry. The buffer, in this case, is your armful of groceries. The performance of the second option is a vast improvement over the first and a perfect demonstration of a primary effect of data buffers: speed.

Buffers are also useful when dealing with file formats. An AVI file, for instance, is stored using a custom compression scheme and must be manually decoded. The best way to decode the file is through direct access to a buffer of primitive data.

Previous to J2SE 1.4, developers did not have access to primitive data buffers, apart from items such as `BufferedInputStreams` and `BufferedOutputStreams`. In J2SE 1.4's `NIO` package, several new buffer implementations have been added that allow direct manipulation of primitive data types, except for `boolean`, in buffered memory.

Buffers in J2SE 1.4

J2SE 1.4's buffer implementation is found in the `java.nio` package. Figure 7-1 details the `Buffer` class hierarchy.

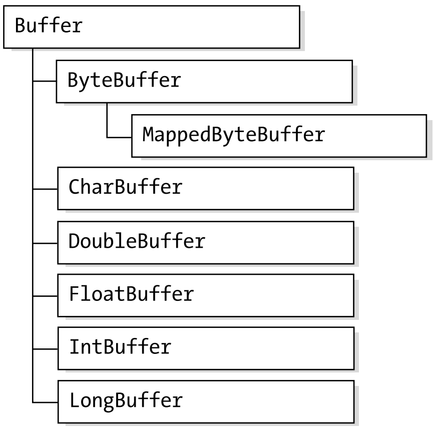


Figure 7-1. Buffer class hierarchy

There are six buffer types: `ByteBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer`, and `ShortBuffer`. All buffer implementations extend `Buffer`. All primitive buffer classes are abstract and cannot be instantiated directly. An instance can only be created by allocating a buffer, which is achieved by calling any primitive buffer class' `allocate(int capacity)` method. An example demonstrating this is shown in the `ByteChannel` section later in this chapter.

Buffer

The Buffer class offers the base functionality that all primitive buffers inherit. The buffer represents a linear block of primitive data and each buffer is of a set size. There are three essential properties of a buffer (as illustrated in Figure 7-3 later in this chapter):

- Capacity
- Limit
- Position

Capacity indicates the set size of the data buffer. All buffers have an allocation capacity that is specified when the buffer is created. Data cannot be written or read from an area greater then a buffer's capacity.

Limit denotes the end of the readable portion of a buffer. The limit is not necessarily equal to the capacity. The limit is always within a buffer's allocated range; therefore, it is never greater than the capacity or less than zero.

Position is analogous to a file pointer. Position indicates the position in the buffer at which the next read or write operation will take place. The position is always greater than zero, and it cannot move beyond the buffer limit.

NOTE A buffer may contain a programmer-defined point known as a mark. A mark is the index the position location will be placed when a buffer's reset method is called. A buffer can have, at most, one set marked index. The marked index must be greater than zero and less than the buffer's position.

The Buffer class contains methods that control and provide information about a buffer's capacity, limit, and position, as shown in Table 7-3.

Table 7-3. Buffer Methods

Return Type	**Method Name**	**Method Description**
int	capacity()	Returns this buffer's capacity
int	limit()	Returns this buffer's limit
Buffer	limit(int newLimit)	Sets this buffer's limit
int	position()	Returns this buffer's position
Buffer	position(int newPosition)	Sets this buffer's position

A few items stand out in Table 7-3. First, notice that there is no method available to set a buffer's capacity. Buffer `capacity` is set when the buffer is allocated, and a buffer cannot be dynamically resized. Repositioning the `limit` can simulate dynamic buffer sizing, but the actual buffer cannot be resized after it is created.

Also take note that methods such as `limit(int newLimit)` have the return type `Buffer`. The returned `Buffer` instance is simply a reference to the same `Buffer` on which the operation was performed. This is provided as a matter of convenience because the return type allows the programmer to chain operations on buffer instances, as illustrated in the following code segment:

```
b.position(13);
b.limit(456);
Could be condensed and rewritten as:
b.postion(13).limit(456);
```

The `Buffer` class also contains many other methods that allow the management of a data buffer, as shown in Table 7-4.

Table 7-4. Buffer Management Methods

Return Type	**Method Name**	**Method Description**
Buffer	clear()	Clears this buffer
Buffer	flip()	Removes all marks, sets the buffer's limit to position, and sets the buffer's position to zero
boolean	hasRemaining()	Indicates if there are any elements between the current position and the limit
abstract boolean	isReadOnly()	Indicates whether or not this buffer is read-only
Buffer	mark()	Sets this buffer's mark at its position
int	remaining()	Returns the number of elements between the current position and the limit
Buffer	reset()	Resets this buffer's position to the previously marked position
Buffer	rewind()	Sets the buffer's position to zero

Two methods stick out in Table 7-4: `flip` and `remaining`. The `flip` operation essentially prepares a data buffer for reading after data has been placed in it. It does this by first removing the mark from the buffer and then setting the `limit` to the current `position`. Because the `position` marks the end of the last written data, moving the `limit` to the `position` will keep all read operations in the range of newly written data. Lastly, the `flip` operation sets the `position` to zero, which moves all read operations to the beginning of the buffer.

The `remaining` method returns an `int` indicating the space left between the `position` and the `limit`. This value is relative to the `position`. As read or write operations occur, the `position` marker will move, and therefore the remaining buffer size will change relative to the new position point.

Figure 7-2 is a visual representation of a data buffer allocated with a size of 40.

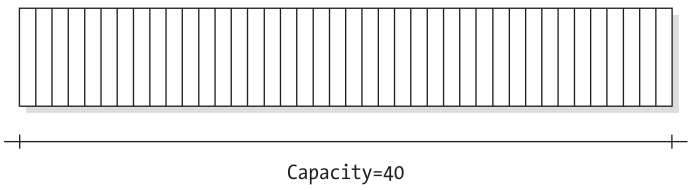


Figure 7-2. A buffer of capacity 40

After allocating the buffer, some standard buffer operations can be invoked to set the `position` and the `limit`:

```
b.postion(3).limit(15);
```

This results in a buffer that resembles Figure 7-3.

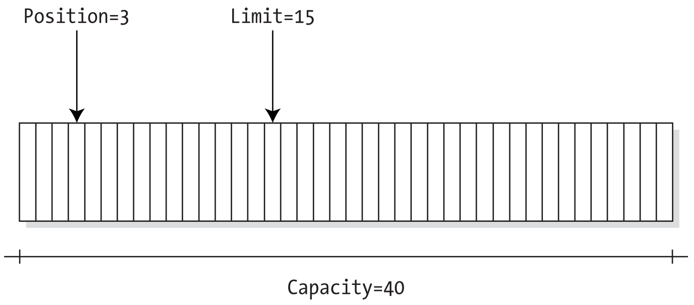


Figure 7-3. A buffer after setting the limit and position

The buffer now has a limit of 15 and a position of 3. Now, when the `remaining` method is called on this `Buffer` instance, the value returned is the distance between the `position` and the `limit`, which totals 13. Finally, the `flip` method is called. The resulting buffer is illustrated in Figure 7-4.

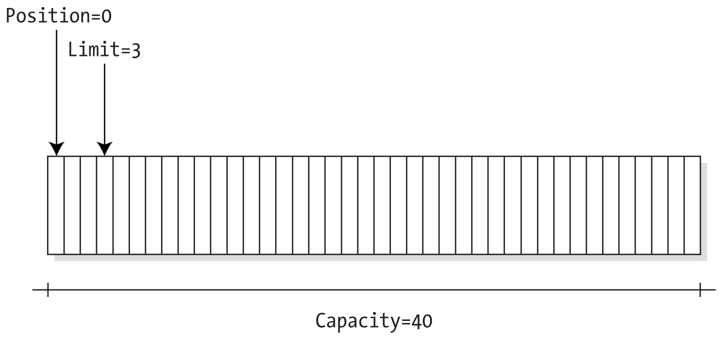


Figure 7-4. A buffer after the flip method is invoked

The `flip` method has set the `limit` equal to the `position`, and then set the `position` equal to 0.

Primitive Buffers

All primitive buffers extend the `Buffer` class. Therefore, all basic buffer functionality remains standard and mostly consistent between buffer types. The primitive buffers follow:

- ByteBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

Each of these buffer types allows the management of the `position` and `limit` as listed previously. They also inherit the capability to flip the buffer and calculate the buffer's capacity and remaining space.

Important new functionality is included in the primitive buffer classes. It is at this level that a buffer is allocated through two different methods. The first is through the `allocate(int Capacity)` method, which is available in all primitive buffer types. This method will create a buffer of the specified capacity. As stated previously, the `allocate` method is the only way to create an instance of a primitive buffer.

NOTE Once a buffer is created, its capacity cannot be changed.

All primitive buffers also allow the creation of a buffer through various wrap methods. For example, an `IntBuffer` can be created by wrapping an array as shown here:

```
int [] iArr = {1,2,3,4,5,6};
IntBuffer iBuff = IntBuffer.wrap(iArr);
```

The preceding example will allocate an appropriately sized integer buffer and insert the integer array values into the newly created `IntBuffer` instance.

Table 7-5 presents a list of allocation methods found in the `ByteBuffer` class.

Table 7-5. ByteBuffer Allocation Methods

Method	**Method Purpose**
static ByteBuffer allocate(int capacity)	Allocates a new byte buffer
static ByteBuffer wrap(byte[] array)	Wraps a byte array into a buffer
static ByteBuffer wrap(byte[] array, int offset, int length)	Wraps a byte array into a buffer

These methods are standard for all primitive buffers—only the types change. For instance, an `IntBuffer` can wrap an `int[]`.

The `ByteBuffer` class has one additional allocation method: `allocateDirect(int capacity)`. This method creates a byte buffer known as a *direct buffer*. When a direct buffer is allocated, the underlying JVM attempts to use all native operating system routines to create and manipulate the byte buffer. There is additional overhead in the allocation of this type of buffer, but the end result is improved buffer manipulation speed. This type of buffer would be appropriate for intensive decompression algorithms, such as an application that decodes an AVI movie stored in a byte buffer.

NOTE The ByteBuffer class also allows for an endian specification, or the ordering of bits in binary data. The default endian setting in Java is big-endian, while some file formats are stored in the reverse endian, known as little-endian. Before J2SE 1.4, this fact caused many problems when decoding certain types of binary data because the byte ordering was essentially reversed.

Writing to and reading from primitive buffers is also standard across the buffer classes. Each class contains a series of "put" methods, which place data into the buffer, and "get" methods, which retrieve data from the buffer. There are many forms of get and put methods in each class that cater to different types of primitive input and output.

The following simple example first allocates a character buffer and then uses put methods to place data into the buffer. The example proceeds step by step with diagrams detailing the state of the buffer after each operation.

First, the character arrays are declared and the character buffers are allocated:

```
// Create character arrays
char [] someChars = {'c', 'h', 'a', 'r', ' ', 'b','u','f','f','e','r'};
char [] moreChars = {'y', 't', 'e'};
// Allocate a character buffer of size 40
CharBuffer cb = CharBuffer.allocate(40);
// Wrap a character array in a byte buffer
CharBuffer cb2 = CharBuffer.wrap(moreChars);
```

Notice that the first buffer is allocated by specifying a buffer size, whereas the second buffer is allocated by wrapping a byte array. The next line places the first character array into the first buffer by using the `put(char[] chars)` method:

```
cb.put(someChars);
```

After the preceding operation, the character buffer resembles Figure 7-5.

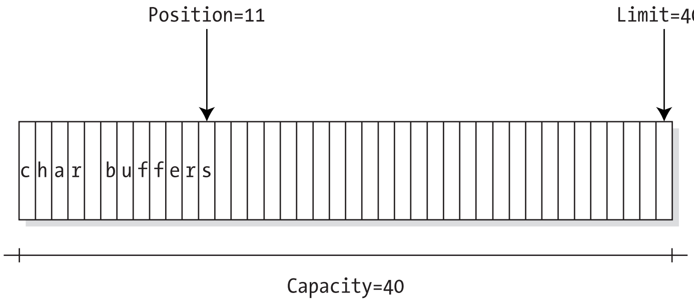


Figure 7-5. Step 1 of the code

Next, the character buffer is flipped in order to prepare it for printing:

```
cb.flip();
```

The result of the `flip` operation is shown in Figure 7-6.

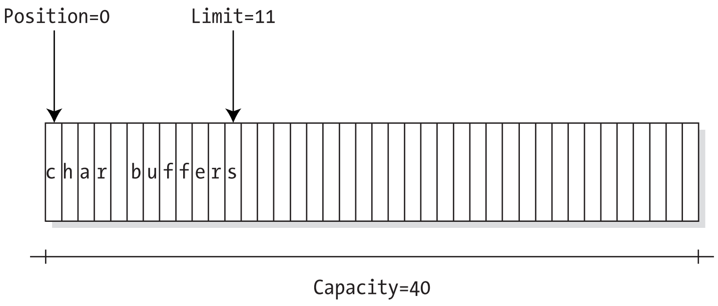


Figure 7-6. Step 2 of the code

Next, the `position` point is set to 1, placing the next insertion point at the second character in the byte buffer. The second byte buffer, which contains the characters “yte”, is the put into the first byte buffer.

```
cb.position(1);
cb.put(cb2);
```

The current state of the buffer now looks like Figure 7-7.

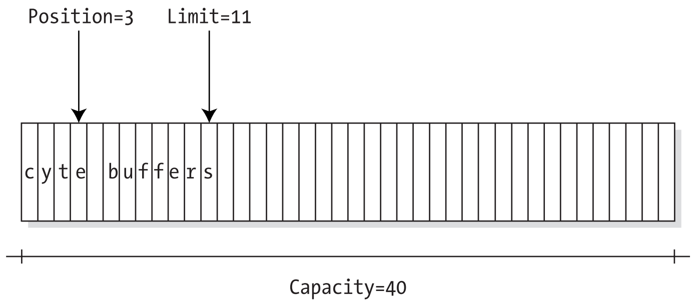


Figure 7-7. Step 3 of the code

Next, the `position` is moved to 0 and the character “b” is placed in the buffer. Then the buffer is flipped.

```
cb.position(0);
cb.put('b');
cb.flip();
```

If you were to print the buffer at this point, the output may not be what you expect. At this point, reading the character buffer would only produce one character: “b”. This is due to the behavior of the `flip` operation. Remember that `flip` places the `limit` at the `position` and then sets `position` to 0. Thus, after this operation, the buffer resembles Figure 7-8.

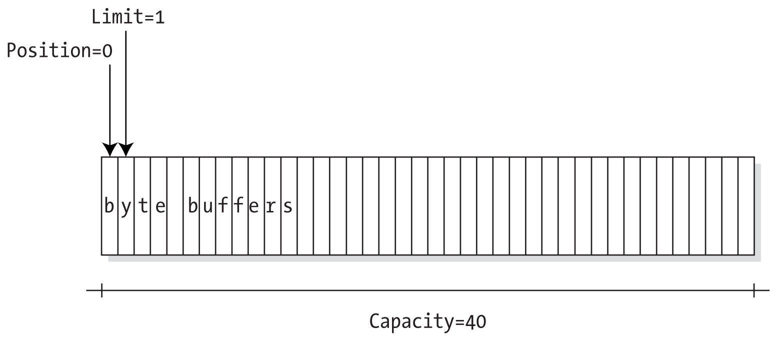


Figure 7-8. Step 4 of the code

In order to print the buffer, the `limit` must be set to the end of the character sequence:

```
cb.limit(someChars.length);
```

Following this statement, the buffer is in the state illustrated by Figure 7-9.

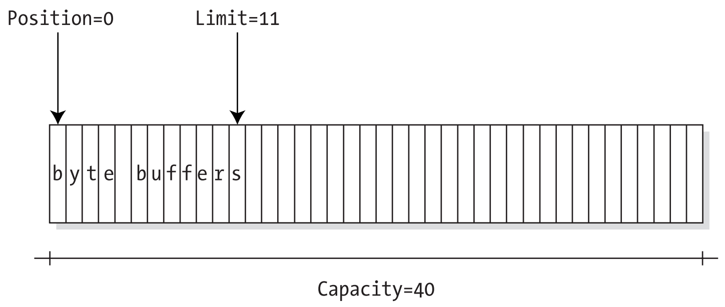


Figure 7-9. Step 5 of the code

Printing the buffer at this point will result in the string “byte buffers”, since the `limit` is now set at the end of the character sequence.

MappedByteBuffer

`MappedByteBuffer` is a subclass of `ByteBuffer`. This type of buffer maps the contents of a file to a `ByteBuffer` and allows for buffer-style manipulations of a file’s contents. A mapped byte buffer is retrieved by calling the method `map(FileChannel.MapMode mode, long position, long size)` on a `FileChannel` instance. Channels and file channels are discussed in depth later in this chapter.

The `FileChannel` class has a public static member, `MapMode`, which contains constant integer values that specify whether the file is mapped as a read-only, read/write, or private access buffer. The following constants are available in the `MapMode` class:

- MapMode.READ_ONLY
- MapMode.READ_WRITE
- MapMode.PRIVATE

After mapping a file to a buffer, the contents of the file are available for manipulation in a manner similar to a ByteBuffer. One item of importance is that once a file has been mapped into a memory buffer, there is no guarantee that another application or thread will not alter the physical file by mapping it into a byte buffer. Therefore, any application can edit and alter the contents of the physical file, while the data buffer remains unchanged. MappedByteBuffer instances should only be used when there is no need to worry about maintaining the data synchronization between the buffer and the source.

The following code snippet maps the contents of a file to a byte buffer:

```
FileInputStream  in = new FileInputStream ("file.txt");
FileChannel fc = in.getChannel();
MappedByteBuffer mappedBuffer = fc.map(FileChannel.MapMode.READ_ONLY,
        0,fc.size());
```

Channels

A channel represents an open connection. That connection can point to a file, a printer or other device, a socket, an array of bytes, and so forth. It has one of two states, open or closed. Once closed, a channel cannot be opened. Channels are an improvement upon previous connection mechanisms in almost every way. Most of the advantages of NIO, including direct memory maps, nonblocking I/O, and well-defined threading support, occur through the use of channels.

The Relevant Channels

The following sections discuss the channels you will almost certainly want to consider using, even if you eventually decide not to do so. These are ByteChannels, FileChannels, and SocketChannels. There are others, including GatheringByteChannels and ScatteringByteChannels, that may be useful depending on your implementation decisions.

ByteChannels in NIO

The next two sections explain how NIO uses ByteChannels.

What ByteChannels Are

A ByteChannel is a subinterface of the ReadableByteChannel and the WriteableByteChannel. This means that you can use a ByteChannel to read, write, or read/write to a channel. ByteChannels become particularly important because both the SocketChannel class and the FileChannel class implement them to support reading and writing at the same time.

How to Use ByteChannels

Listing 7-1 shows a simple example of how to copy the contents of one file to another. Note that this process is so easy because a FileChannel is a ByteChannel.

Listing 7-1. ByteChannelExample.java

```java
import java.io.*;
import java.nio.channels.*;

public class ByteChannelExample{
      public static void main(String args[])
    throws IOException, FileNotFoundException{
        //open connections to the source and target files
        RandomAccessFile raf = new
        RandomAccessFile("ByteChannelExample.java","r");
        RandomAccessFile newraf =
        new RandomAccessFile("Newtmp.txt","rwd");
        FileChannel source = raf.getChannel();
        FileChannel target = newraf.getChannel();

        //since source is a FileChannel,
        //and a FileChannel is a ByteChannel
        //the transferFrom method will take
        //source as a parameter.
        target.transferFrom(source,0,source.size());

        //close the connections.
        source.close();
        target.close();
    }
}
```

FileChannels in NIO

The following sections discuss `FileChannels` as they might be relevant to the SCJD exam. Of course, this is not an exhaustive study of `FileChannels` by any means. It should, however, address the most common questions.

What FileChannels Are

A `FileChannel` is J2SE 1.4's new mechanism for interacting with files. `FileChannels` represent a two-way connection to a file, allowing you to read and write to the file using that single connection. They also offer the ability to lock files for exclusive use, the ability to map files to memory buffers, and the ability to read and write directly to `ByteBuffers`. In addition, they offer platform-specific optimizations, atomic read/writes, interruptible I/O, and smooth interoperability with other channels. As far as file I/O is concerned, `FileChannels` are Java's next and best step forward.

How to Use FileChannels

A `FileChannel` can be obtained in one of three ways. To obtain an instance of a `FileChannel`, you can use a `FileInputStream`, a `FileOutputStream`, or a `RandomAccessFile`. These classes have been retrofitted with a `getChannel` method that returns a `FileChannel` object associated with them. Depending on the read/write permissions of the underlying class, a `FileChannel` can be readable, writeable, or both. For example, if you open a `RandomAccessFile` in the following way:

```
1   import java.io.*;
2   import java.nio.channels.*;
3
4   public class FCDemo{
5       public static void main(String args[]) throws FileNotFoundException{
6       RandomAccessFile raf = new RandomAccessFile("c:\\tmp.txt","rwd");
7           FileChannel fc = raf.getChannel();
8       }
9 }
```

the `FileChannel` reference on line 7 is both readable and writeable. However, if line 6 had read

```
6 RandomAccessFile raf = new RandomAccessFile("c:\\tmp.txt","r");
```

the `FileChannel` would be read-only. This is true of `FileChannels` obtained from `FileOutputStreams`. However, a `FileInputStream` only produces read-only `FileChannel` objects.

`FileChannels` offer several helpful methods. We will not discuss those methods that act in a platform-dependent manner, as they distract from the goals of the SCJD exam. This means that we will not discuss methods having to do with memory mappings and locks.

The first method of interest is `public void force(boolean metaData)`. This method forces any changes made since the last time the `FileChannel` was invoked to be persisted to the underlying file. If the metadata `boolean` is true, then this applies for file metadata changes. If it is false, only changes made to the content of the `FileChannel` will be persisted.

The method `long position` returns the position of the current read/write point within the file, in bytes. As you might expect, the corresponding `FileChannel position(long newPosition)` method sets the position of the next read/write operation at the specified position, counting at the beginning of the file. It is important to be careful when you use this method; setting the value to a number greater than the current file size will cause the next byte to be written at the position indicated by `newPosition`. This can be dangerous because the values between the previous file size and the `newPosition` are unspecified. You could end up with garbage in your file. Interestingly, the `position(long newPosition)` method returns a `FileChannel` object.

`FileChannel` write methods are fairly straightforward. You simply wrap your data as some sort of `ByteBuffer` and then pass it as a parameter to the `write` method. There are, however, a few subtleties to be aware of.

First, if you specify a position at which to begin writing, as opposed to letting the `FileChannel` write to the next position, then you are not guaranteed that the operation will proceed atomically. This can be important if you need to be sure that all the data in the buffer was actually written to the buffer at the time you expect it to be.

A second important note is that any thread that modifies the file's position or its size blocks until it has finished, regardless of the underlying operating system. This is worth remembering, because it can be a mechanism for forcing updates to be atomic. This way, you will know that anything that you wrote to the channel was actually written to the file system when the method executes.

All read and write operations on a `FileChannel` return an `int` indicating the number of bytes affected. The only exceptions to this rule are those methods that deal with arrays. For example, `FileChannels` have the read and write methods shown in Table 7-6.

Table 7-6. FileChannel Read/Write Methods

Signature	**Explanation**
int read(ByteBuffer dst)	Read data into the ByteBuffer dst
int read(ByteBuffer dst, long position)	Read data into the ByteBuffer dst, starting at position in the channel
int write(ByteBuffer src)	Write data from src into the channel
int write(ByteBuffer src, long position)	Write data from src into the channel, starting at position in the channel
long write(ByteBuffer[] srcs)	Write data from the src array into the channel
long write(ByteBuffer[] srcs, int offset, int length)	Write data from the src array into the channel, starting from position offset in the channel and continuing until length
long read(ByteBuffer[] dsts)	Write data from the channel into the dsts array
long read(ByteBuffer[] dsts, int offset, int length)	Write data from the channel into the dsts array, starting from position in the channel and continuing until length is reached

Fortunately, the methods describe themselves fairly clearly, given their names and signatures. For example, `long write(ByteBuffer[] srcs, int offset, int length)` starts writing an array of bytes at `position offset` in the file and continues writing until `position offset + length` is reached. Thus, if `offset` were equal to 7 and `length` were equal to 8, then the invocation to the `long write(ByteBuffer[] srcs, int offset, int length)` method would not write past position 15.

There are various read methods that take `ByteBuffers` as parameters. Reads are all fairly straightforward, except for a single consideration. Once you have read data into a `ByteBuffer`, you must `remember` to rewind the `ByteBuffer` in question before extracting its value. This can be accomplished by calling the `rewind` method on the `ByteBuffer`. Otherwise, you will be reading the `ByteBuffer` at its end. This is similar to recording a television program on your VCR, and then rewinding the tape before attempting to view it: Viewing the tape without first rewinding it doesn't really make a lot of sense. A shorthand method for examining values read in from `FileChannels`, assuming the file contains `Strings`, you can always use the following mechanism:

```
//read in data from the FileChannel fc
ByteBuffer myByteBuffer = fc.size();
fc.read(myByteBuffer);
String value = new String(myByteBuffer.array());
```

The preceding code reads a sequence of bytes from this channel into the given buffer starting at the given file position.

There are three other important `FileChannel` methods that we need to mention. The method `long size` simply returns the size of the current file as measured in bytes. The other two methods to be aware are `transferFrom` and `transferTo`, as they can greatly facilitate transferring data from `FileChannels` to other channels. These can be a great help in get data from one resource to another—say, reading the data in a file and sending it across a network. Their signatures follow:

```
long transferFrom(ReadableByteChannel src, long position, long count)
and
long transferTo(long position, long count, WriteableByteChannel target).
```

Example of FileChannels

Let's revisit the code base from Chapter 4 in light of what you have learned about `FileChannels`. The `DVDDatabase` class handles the entirety of its file I/O operations though the methods `boolean persistDVD` and `DVD retrieveDVD`.

As you can see, the following code is fairly straightforward. Lines 185–187 open a `FileChannel` associated with the `DVD` object, lines 189–192 serialize that object, and lines 194–198 persist the `DVD` object. The remaining code simply cleans up by releasing resources and whatnot.

```
179     private boolean persistDVD(DVD dvd) throws IOException{
180         boolean retVal=false;
181
182         //open a File Channel associated with the data
183         //notice that if the DVD does not already exist,
184         //it will be created.
185         String filePath = dbName+"/"+ dvd.getUPC() + recordExtention;
186         FileOutputStream fos = new FileOutputStream(filePath);
187         FileChannel fc = fos.getChannel();
188
189         //Write the dvd's serialized state to a ByteAttayOutputStream
190         ByteArrayOutputStream baos = new ByteArrayOutputStream();
191         ObjectOutputStream oos = new ObjectOutputStream(baos);
192         oos.writeObject(dvd);
193
```

```
        //Create a ByteBuffer that is large enough to hold all the data
        ByteBuffer bf= ByteBuffer.wrap(baos.toByteArray());

        //Read in the data from the object
        fc.write(bf);

        //close all references
        oos.close();
        fos.close();
        fc.close();
        fos =null;
        oos = null;
        retVal = true;
        return retVal;
}
```

The `retrieveDVD` method is similarly constructed:

```
    private DVD retrieveDVD(String upc, String fileExtension)
        throws IOException, ClassNotFoundException{
        DVD retVal = null;
        //get the path to the object's serialized state.
        String filePath = dbName+"/"+ upc + fileExtension;
        FileInputStream fis = new FileInputStream(filePath);
        FileChannel fc = fis.getChannel();
        //Create a ByteBuffer that is large enough to hold all the data
        ByteBuffer bf= ByteBuffer.allocate((int)fc.size());
        //Read in the data from the object
        fc.read(bf);

        //deserialize the data
        ByteArrayInputStream bais = new ByteArrayInputStream(bf.array());
        ObjectInputStream ois = new ObjectInputStream(bais);
        retVal = (DVD)ois.readObject();

        ois.close();
        fis.close();
        fc.close();
        fis =null;
        fc = null;
        return retVal;
}
```

Finally, the client methods are changed to use these two private methods. For example:

```
 public synchronized boolean addDVD(DVD dvd)
  throws IOException{
        return  persistDVD(dvd);
    }
 public synchronized DVD getDVD(String upc)
  throws IOException, ClassNotFoundException{
        return retrieveDVD(upc);
}
```

You can download this code from the Downloads section of the Apress Web site (`http://www.apress.com`).

Please see Sun's Java site (`http://java.sun.com`) for a full catalog of the methods available though the `FileChannel` method.

Understanding FileLocks

As you know, every Java object has a lock associated with it (please see Chapter 4 for a review of locks, if necessary). J2SE 1.4 offers developers a particularly sophisticated and robust lock for `FileChannel` objects: the `java.nio.channels.FileLock` object. This object offers a way to control locking files, and regions of files, for exclusive use.

The `FileLock` object works across process boundaries so that it offers completely exclusive locking on the file in question. This means that no other thread or process can modify the file in question. It's a useful tool for Java developers, and you should seriously explore its advantages for your real-world applications. It would be an ideal solution for parts of the SCJD exam, except for some very serious considerations:

> Consideration 1: The FileLock's behavior is operating system–dependent. Some operating systems support shared locks and others do not. Some support file subsection locks and others do not. Some do not even offer file locking. Because the SCJD exam requires that your project work identically on every operating system, you will have to specialize your code if you choose to use FileLock. Operating system–specialized code is really against the spirit of the SCJD exam, so the use of FileLock is not recommended.

Consideration 2: FileLocks are held across the entire JVM. This means that they are not the appropriate solution for fine-grained threading. The Sun documentation is very clear on this point.

Consideration 3: Locking a file can interfere with memory mapping, depending on the underlying platform. Once again, this may hinder platform independence.

Consideration 4: Closing a channel could release every lock associated with a file, even if those locks were acquired by other FileChannels. This behavior is dependent on the underlying platform.

Consideration 5: Locking may or may not be permitted, depending on the paging system used by network file system in use.

How FileLocks Work in NIO

While the use of `FileLocks` for the SCJD exam is discouraged because of their platform-dependent behavior, they are still a useful technology to know about. It might be worthwhile to be able to explain why they are not being used

You can access the `FileLock` associated with a given `FileChannel` by calling the `FileChannel.lock` or `FileChannel.tryLock` method. For example, you might do the following to lock a region of a file:

```
//Acquire a FileChannel reference called MyFileChannel
MyFileChannel.lock();
//..write to the channel

myFileChannel.close();
         or
FileLock fl = MyFileChannel.lock();
//..write to the channel
fl.release();
```

NOTE The FileLock.release method does not seem as responsive on the Windows platform as actually closing the FileChannel object, at least for J2SE build 1.4.0-b92. If you decide to use FileLocks in your applications (we suggest that you don't use them for your SCJD exam), release the FileLock by closing the FileChannel, instead of using the FileLock.close method.

SocketChannels in NIO

This section helps to explain what SocketChannels are and how they work.

WARNING As of spring 2002, most versions of the SCJD exam explicitly require that either sockets or RMI be used as the networking protocol in your project implementation. This is a firm requirement, meaning that those adventurous souls who decide to deviate from this requirement may fail automatically upon submission. For this reason, we suggest that you do not use SocketChannels unless your exam instructions state clearly that a SocketChannel implementation is permitted. Anticipating future changes in lieu of the new J2SE release, we have decided to include a sample SocketChannel implementation. The example discussed in this section is a scaled-down version of Denny's DVDs and is not a full implementation. Rather, it is meant to point the way and introduce some of the basics steps involved in building a full-fledged SocketChannel implementation.

What SocketChannels Are

`SocketChannels` are a `NIO` feature that gives developers a high-performance, scalable alternative to `java.net.Sockets`. Some of the more interesting features of `SocketChannels` follow:

- Concurrent access is possible without blocking I/O operations.
- Servers can be written without the extensive use of Threads.
- Channels can read and write directly to Buffers in the java.nio package.

The Java API refers to `SocketChannels` as "selectable channels for stream-oriented connecting sockets." `SocketChannels` are selectable because they extend the abstract class `java.nio.channel.SelectableChannel`, which enables multiplexing via a selector. `SelectableChannels` can be multiplexed and when used in conjunction with `Selectors` form a multiplexed, nonblocking I/O facility.

Old-style sockets in `java.net` always performed I/O operations in a blocking fashion. That is, each operation on the socket would require the calling thread to wait until the method completed. The first call to block is the `accept` method. Each subsequent call on the socket also blocks. To alleviate the bottleneck associated with blocking, our socket server in Chapter 5 creates a `Thread` for each request. But this comes at a price. `Threads` are expensive and there is an upper limit to the number of threads a JVM can, and even should, create. This is often referred to as "the point of diminishing returns."

NOTE In our implementation of sockets in Chapter 5, we did not limit the number of threads that could be created. For our purposes, this is acceptable since scalability and performance were not a design consideration for this particular implementation. However, if our application needed to scale (to, say, hundreds of users), then you could provide a thread pool that would create a fixed number of threads and then allocate them as needed to the server requests.

As with traditional sockets in the `java.net` package, you have to consider both the server side and the client side of the equation with `SocketChannels`. The client side can be handled with the `java.nio.SocketChannel` class. The server side involves the `java.nio.ServerSocketChannel` class.

Example of SocketChannels

In the sample code that follows, we develop a DVD-rental client and a DVD-rental server. It is a very simple example that runs from the command line, accepts the host machine and `upc` as command-line arguments, and returns a message indicating either success or failure. Let's take a look at the client. As a pedagogical device, you will find it a useful demonstration of `Channels`, `Selectors`, `Charsets`, and `ByteBuffers`—all `NIO` features.

SocketChannel Clients

We need to perform a number of steps to create a client using `SocketChannels`. First, we need to initialize a `SocketAddress` object with the machine or host name and the port number. In the `SocketChannelClient` sample that follows, we use a `java.net.InetSocketAddress` object. This is very similar to the way we initialized sockets in Chapter 5. In that chapter we were required to supply the `socket` client with both the host and the port of the machine on which we wanted to open a socket connection. In our current example, we simply use the initialized `InetSocketAddress` object and supply it to the `SocketChannel`'s `connect` method.

Before calling `connect`, we need to get a `SocketChannel` reference. The `SocketChannel.open` method returns a new `channel`, which is created by the system-wide default `java.nio.channels.spi.SelectorProvider` object. Once we have a `SocketChannel` object, we can try to connect. There are two possible results to this attempt. We can either get a connection to the socket server or the connection can be refused and an exception thrown. Alternatively, we can combine both calls in one with the static `SocketChannel` method `open(SocketAddress remote)`, as shown here:

```
channel = SocketChannel.open();
channel.configureBlocking(false);
channel.connect(socketAddress);

selector = Selector.open();

// Register interest in when connection
channel.register(selector, SelectionKey.OP_CONNECT
| SelectionKey.OP_READ);
```

After opening the `SocketChannel`, we need to register the events that we are interested in. The `register` method in the `Channel` class is used for this purpose. In the preceding code we are interested in the CONNECT and READ events. Table 7-7 presents all of the events that we can register.

Table 7-7. Selection Key Events

Event Name	**Description**
OP_CONNECT	When Selector detects that the socket channel is ready to complete its connection sequence
OP_READ	When Selector detects that the socket channel is ready to for reading
OP_ACCEPT	When Selector detects that the socket channel is ready to accept another connection
OP_WRITE	When Selector detects that the socket channel is ready for writing

Wrapping up the UPC in a `java.nio.ByteBuffer` class and extracting the return value from the server requires a bit of `NIO`-style gymnastics. All of that is demonstrated shortly in the RentalSocketChannelClient.java listing. What's important to take out of this example is not the low-level byte handling, but rather the differences between `SocketChannels` and traditional `Sockets`. Also notice that `SocketChannels` can accomplish the same things as traditional `Sockets`, but they do so in a more scalable way because they do not necessarily require the extensive use of threads. When in non-blocking mode, `SocketChannels` perform I/O operations without requiring the calling

Thread to block. On the other hand, when a SocketChannel is in blocking mode, all of the I/O operations invoked *will* block and the SocketChannel will work in much the same way that Sockets do.

The method isBlocking can be used to determine if a SocketChannel is in blocking or nonblocking mode. By default, a SocketChannel is created in blocking mode. To create a SocketChannel in nonblocking mode, the SocketChannel.configureBlocking(boolean block) method should be used before calling connect. Here is an example:

```
channel = SocketChannel.open();
//configures channel as a non-blocking channel
 channel.configureBlocking(false);

 channel.connect(socketAddress);
```

Listing 7-2 shows the RentSocketChannelClient.java file.

Listing 7-2. RentSocketChannelClient.java

```
package sampleproject.socketchannels;

import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.*;


public class RentSocketChannelClient {

 public static void main(String args[]) {
    String host = args[0];
    String upc = args[1];

    SocketChannel channel = null;
        Selector selector = null;

    try {
```

```
// Setup
InetSocketAddress socketAddress =
  new InetSocketAddress(host, 3000);
Charset charset =
  Charset.forName("ISO-8859-1");
CharsetDecoder decoder =
  charset.newDecoder();
CharsetEncoder encoder =
  charset.newEncoder();

// Allocate buffers
ByteBuffer buffer =
  ByteBuffer.allocateDirect(64);
CharBuffer chBuffer = CharBuffer.allocate(64);

// Connect and configure channel for non-blocking
channel = SocketChannel.open();
channel.configureBlocking(false);
channel.connect(socketAddress);

selector = Selector.open();
// Register interest in when connection
channel.register(selector,
  SelectionKey.OP_CONNECT | SelectionKey.OP_READ);

// Wait 5 secs for something to happen from the server.
while (selector.select(5000) > 0) {

  // Get set of ready objects
  Set selKeys = selector.selectedKeys();
  Iterator selIt = selKeys.iterator();

  // Walk through set
  while (selIt.hasNext()) {

    // Get key from set
    SelectionKey key = (SelectionKey)selIt.next();

    // Remove current entry
    selIt.remove();

    // Get channel
    SocketChannel keyChannel = (SocketChannel)key.channel();
```

```
          if (key.isConnectable()) {

            // Finish connection
            if (keyChannel.isConnectionPending()) {
                  keyChannel.finishConnect();
            }

                  // Send rent request
                  keyChannel.write(encoder.encode(chBuffer.wrap(upc)));

          }
          else if (key.isReadable()) {
                  // Display the server response
                  keyChannel.read(buffer);
                  buffer.flip();

                  // Decode buffer
                  decoder.decode(buffer, chBuffer, false);
                  chBuffer.flip();
                  System.out.println(chBuffer);

                  // Clear for next request
                  buffer.clear();
                  chBuffer.clear();
          }
        }
      }
    }
    catch (Exception e) {
        System.err.println(e);
    }
    finally {
        if (channel != null) {
                try {
                        channel.close();
                }
                catch (IOException e) {
                        System.err.println(e.getMessage());
                }
      }
    }
    System.out.println("Client execution finished.");
  }
}
```

SocketChannel Servers

The server side of SocketChannels makes use of the java.nio.ServerSocketChannel class. Please notice that there is no need to create new threads or manage a thread pool with SocketChannels like we did with traditional Sockets. All we have to do is call the static method open on ServerSocketChannel to get our server object. As with java.nio.SocketChannel, the open method uses the system-wide default SelectorProvider to create the server channel. The next step is to get a java.net.ServerSocket with the ServerSocketChannel.socket method. This is the same type of object we used in Chapter 5 with traditional Sockets. We then bind the server object to our InetSocketAddress object with the bind method. Most of the remaining code is similar to our socket channel client, except that we register the OP_ACCEPT key and query isAccepetable to be notified of an accept event. This returns a ServerSocketChannel rather than a SocketChannel. Listing 7-3 shows the RentSocketChannelServer.java file.

Listing 7-3. RentSocketChannelServer.java

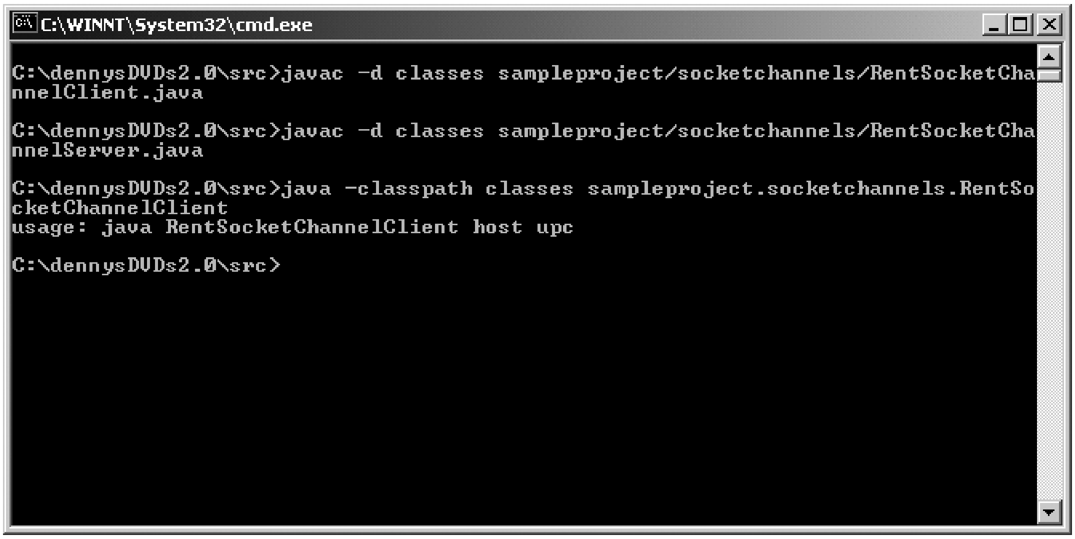
```
package sampleproject.socketchannels;

import sampleproject.db.*;
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.*;

public class RentSocketChannelServer {
   private static int port = 3000;
   public static void main(String args[])
     throws Exception {
     Selector selector = Selector.open();

     ServerSocketChannel channel =
       ServerSocketChannel.open();
     channel.configureBlocking(false);
     InetSocketAddress isa =
      new InetSocketAddress(
      InetAddress.getLocalHost(), port);
     channel.socket().bind(isa);
     System.out.println("Socket Channel is bound to " + isa);
     // register interest in Accept
     channel.register(selector, SelectionKey.OP_ACCEPT);
```

```
// Keep listening for client requests
while (selector.select() > 0) {

  Set selKeys = selector.selectedKeys();
  Iterator selKeysIt = selKeys.iterator();

  while (selKeysIt.hasNext()) {

    SelectionKey selKey =
      (SelectionKey)selKeysIt.next();
    // The current key needs to be removed
    selKeysIt.remove();

    if (selKey.isAcceptable()) {
      // Get the server socket channel
      ServerSocketChannel keyChannel =
        (ServerSocketChannel)selKey.channel();

      ServerSocket serverSocket = keyChannel.socket();

      // Accept request
      Socket aSocket = serverSocket.accept();

      // get input stream
      InputStream in = aSocket.getInputStream();
      byte[] bytes = new byte[16];

      //read the upc for the client rent request
      in.read(bytes);
      String upc = new String(bytes);
      upc = upc.trim();
      boolean result = rent(upc);

      OutputStream out = aSocket.getOutputStream();
      String status = null;
      if (result){
      status = new String("DVD with UPC of " + upc +
       " was succesfully rented");
      }
      else{
       status = new String("DVD with UPC of "
       + upc + " could not be rented");
      }
```

```
           out.write(status.getBytes());

           //clean up
           in.close();
           out.close();
         }
       }
     }
   }

   protected static boolean  rent(String upc) {
    try {
      DBClient db = new DVDDbAdapter();
      return db.rent(upc);
      }
      catch(Exception e) {
      System.err.println("exception in rent: "
      + e.getMessage());
      }

      return false;
      }
}
```

Figure 7-10 shows how to compile the class files into the destination directory "classes". Running the client without any arguments displays the proper usage for the class. The command-line arguments are host and upc, in that order:

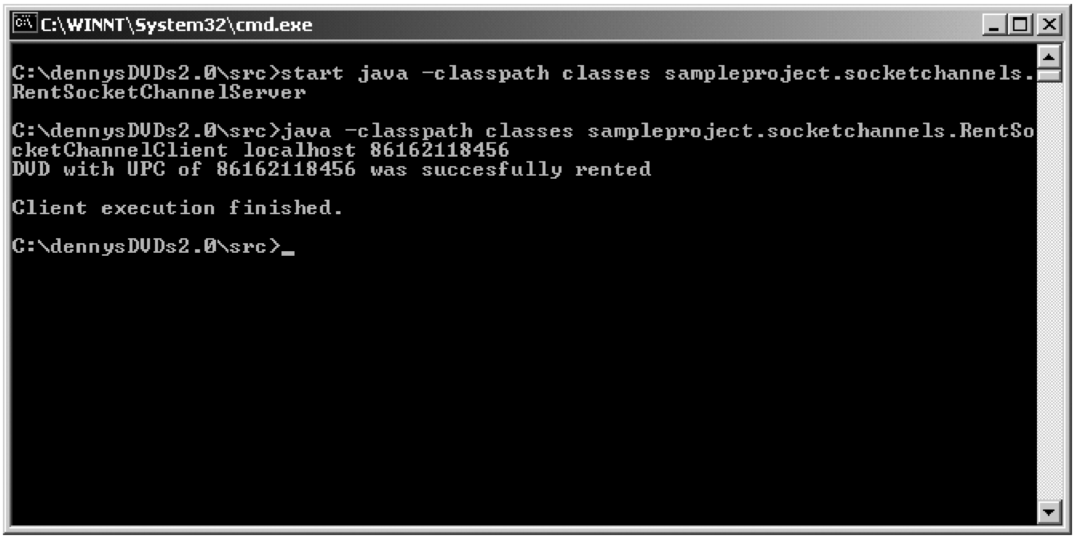
```
java sampleproject.socketchannels.RentSocketChannelClient <host> <upc to rent>
```



```
C:\dennysDVDs2.0\src>javac -d classes sampleproject/socketchannels/RentSocketCha
nnelClient.java

C:\dennysDVDs2.0\src>javac -d classes sampleproject/socketchannels/RentSocketCha
nnelServer.java

C:\dennysDVDs2.0\src>java -classpath classes sampleproject.socketchannels.RentSo
cketChannelClient
usage: java RentSocketChannelClient host upc

C:\dennysDVDs2.0\src>
```

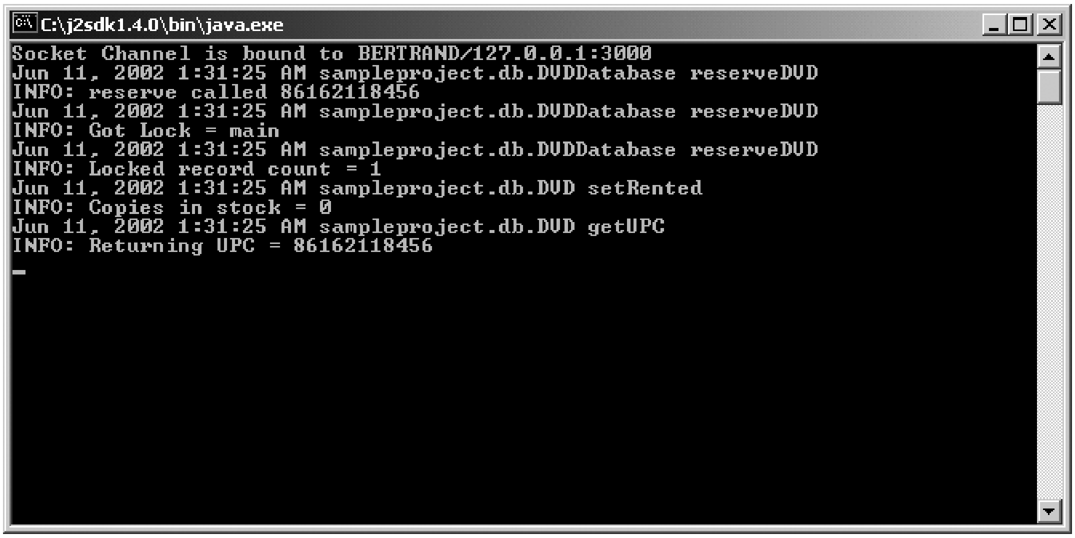
Figure 7-10. Compiling the classes

In Figure 7-11 we start the socket channel server and run the client. The `upc` of the `DVD` we decided to rent belongs to "Office Space". Figure 7-12 shows the database log messages.



```
C:\dennysDVDs2.0\src>start java -classpath classes sampleproject.socketchannels.
RentSocketChannelServer

C:\dennysDVDs2.0\src>java -classpath classes sampleproject.socketchannels.RentSo
cketChannelClient localhost 86162118456
DVD with UPC of 86162118456 was succesfully rented

Client execution finished.

C:\dennysDVDs2.0\src>_
```

Figure 7-11. Socket channel server test



```
Socket Channel is bound to BERTRAND/127.0.0.1:3000
Jun 11, 2002 1:31:25 AM sampleproject.db.DVDDatabase reserveDVD
INFO: reserve called 86162118456
Jun 11, 2002 1:31:25 AM sampleproject.db.DVDDatabase reserveDVD
INFO: Got Lock = main
Jun 11, 2002 1:31:25 AM sampleproject.db.DVDDatabase reserveDVD
INFO: Locked record count = 1
Jun 11, 2002 1:31:25 AM sampleproject.db.DVD setRented
INFO: Copies in stock = 0
Jun 11, 2002 1:31:25 AM sampleproject.db.DVD getUPC
INFO: Returning UPC = 86162118456
_
```

Figure 7-12. Output window

Regular Expressions

J2SE 1.4 adds the ability to parse and interpret regular expression statements. For those who have already heard of or used regular expressions, you know that they are a simultaneously complex and elegant tool. For those who have never used regular expressions, the following sections will introduce the new Java pattern `compiler` and `matcher` classes and cover the basics of the regular expression syntax.

What Regular Expressions Are

A *regular expression* is a pattern that can be matched against a text string. First and foremost, regular expressions are a syntax that allows programmers to set up a system of rules and then filter character sequences through the set of rules to determine if a matching pattern exists. A regular expression is a pattern that can be matched against a text string. They allow an application to perform complex pattern-matching on any type of character sequence. The real advantage of this type of pattern matching is the fact that it intelligently matches patterns, not just substring elements. Programmers are free to write their own pattern statement, essentially a shorthand heuristic for a set of rules the system will use to determine if character data is a partial or close match.

How Regular Expressions Are Useful

Regular expressions provide programmers with functionality well beyond normal string-interpretation methods. For instance, suppose you are developing a bulletin board application that will be placed on a Web site whose main audience is children. Because there is always one rotten apple in the bunch, you can pretty much count on one child becoming the designated “BBS potty mouth.” In order to prevent spoiling the rest of the bulletin board apples, the Web site must incorporate some type of monitor that checks each post for foul language and replaces the offending text before it has a chance to spoil other fragile little minds.

To end the scourge of foulmouthed children, you decide to implement an application that searches each string for a collection of predefined substrings. For example, one of the predefined offending words is “butt,” and the software you wrote will perform a substring check on each text post to see if it contains the word “butt.” If it does contain such offensive language, it will be replaced by a series of asterisk characters, so the end product will look like this: “****.”

The software you wrote seems to do its job very well. In fact, it does its job a little too well. What happens when little Timmy logs onto the bulletin board application you developed and begins an intelligent thread on the virtues of butter over margarine? The software will do its job, and Timmy will be frustrated that all of his posts about “butter” come through as “****er.” This sort of thing might actually start the potty-mouth behavior we were originally hoping to deter.

This is a prime example of a problem that begs to be solved by regular expressions. By using intelligent pattern-matching, the application can easily check to see if the word “butt” is actually used as a singular, offensive word or if it is actually part of a larger word, such as “butter.”

A common use of regular expressions is validation. Consider e-mail validation. A regular expression pattern can be authored that will check a character sequence and determine if it is formatted in a style that is similar to an e-mail address. This is a far more robust solution than simply checking to see if a string contains “@” and “.” characters. A regular expression can actually account for the general rules that govern how an e-mail address is formatted and determine if a text string is a partial match.

Regular Expressions in J2SE 1.4

Regular expressions can be an extremely powerful tool. If the preceding example did not adequately prove this, take into account that they are so useful that almost every computer language has the facility to use them built in natively, or they can easily be extended to accommodate pattern matching. Their usefulness in the realm of data validation is so great that even JavaScript has simple regular expression implementation.

For quite a few years, regular expressions have been available to Java developers but only through add-in packages. J2SE 1.4 marks a turning point: Sun has finally realized that regular expressions are important enough to include as part of the native Java API. Now every Java application can use the powerful character-parsing tools that have been the bragging right of languages such as Perl for all these years.

The regular expression package included in J2SE 1.4 is extremely simple. The utility is composed of three classes contained in the `java.util.regex` package: `Pattern`, `Matcher`, and `PatternSyntaxException`.

The first component is the `Pattern` class, which acts as the regular expression compiler. The static method `compile(String regex)` takes in a `String` representation of a regular expression statement and returns a prepared `Pattern` instance. The statement must, of course, be syntactically correct in order to compile without error. If a compilation error occurs, the `compile` method will throw an unchecked `PatternSyntaxException` exception.

Once the `Pattern` object has compiled the expression, the prepared statement is internally maintained within the returned `Pattern` instance. Thus, after compiling the regular expression, a `Pattern` instance is provided in a usable and ready state.

NOTE If a regular expression is going to be used only once, the Pattern class provides the convenient method matches(String regex, CharSequence input). This method combines the steps of compiling the expression and generating a matcher object into one step. This method does not, however, produce a reusable Pattern instance. Only use this method when the Pattern match will only be executed once.

After the `Pattern` object compiles and holds a valid regular expression, the `match(CharSequence input)` method can be called on the `Pattern` object instance. The `CharSequence` input is a sequence of character data, such as a `String`, that the `Pattern` instance will filter using its prepared regular expression. The method returns an object of type `Matcher`. The `Matcher` instance performs operations on the character sequence and then holds the results of the match operations.

NOTE The character data that the pattern matches against is specified as a CharSequence, yet the match method takes a String argument. This is because in J2SE 1.4, all character sequence types (such as String, StringBuffer, and CharBuffer) implement the CharSequence interface. This interface provides a common set of methods for manipulating and extracting parts of any type of character data. Essentially, the CharSequence interface is to String data as the Iterator is to Java collections.

The `Matcher` class provides various ways to apply a pattern to a character sequence. The `find` method searches the provided character sequence for a section that matches the compiled regular expression. If a match is located, the `find` method will return a `boolean` true. The `Matcher` class also provides the method `matches`, which matches the entire character sequence against the pattern.

If the goal is to intelligently replace portions of a character sequence, the `Matcher` class contains two methods to accomplish this very job. The first is `replaceFirst (String replacementString)`, which replaces the first instance of a pattern that is located in a character sequence. The `Matcher` class holds its own set of internal marker positions, so if the `replaceFirst()` method is called again, the `Matcher` will replace the next portion of the character sequence that matches the pattern, if it exists. The other method is `replaceAll(String replacementString)`, which replaces all matching occurrences in a character sequence.

A Simple Example of the Regular Expressions Classes

The `java.util.regex` package is deceptively easy to use. The following program demonstrates how easy it is to compile and match a regular expression against a character sequence. The difficult part, as you will soon see, is actually writing the regular expression.

The following program is quite simple. It takes in two command-line parameters: the path to a text file and a string of text representing a regular expression. The text file will contain a list of words, and the program will read the lines out of the document one by one and compare them against the specified regular expression. If the pattern matches, the program will print the matching word to the console.

The first part of the application simply reads in the command-line parameters and reacts accordingly:

```
import java.io.*;
import java.util.regex.*;

public class FileMatcher {
  public static void main(String[] args) {
    if (args.length != 2){
   String msg = "Usage: Please list a file name and";
   msg +="  specify a regular expression.";
      System.out.println(msg);
    }
    else {
      FileMatcher fmatcher = new FileMatcher (args[0], args[1]);
    }
  }
```

If all is well with the command-line parameters, the application creates a new instance of the `FileMatcher` class. The constructor of this class is where the real magic happens. First, the regular expression is compiled. If the compilation fails, the error is caught and the user is notified.

```
public FileMatcher(String filePath, String expression) {
    // Compile the pattern
    Pattern p = null;
    try{
      p = Pattern.compile(expression);
    }
    catch (PatternSyntaxException pex){
      System.err.println("Error compiling expression " + expression + ".");
      System.exit(0);
    }
```

Next, the application opens the text file and reads it line by line. Each resulting `String` is matched against the compiled pattern. If the line contains a pattern match, the listing is printed to the console.

```
    // Open the input file
    BufferedReader input = null;
    try{
      input = new BufferedReader (new FileReader(filePath) );

      String listing = null;

      listing = input.readLine();
      while(listing != null){
        Matcher m = p.matcher(listing);

        if (m.find()){
          System.out.println(listing);
        }
        listing = input.readLine();
       }

      // Close the stream
        input.close();
      }
    catch (Exception ex){
      String msg = "Error reading from file "+ filePath;
      System.err.println(msg);
      System.exit(0);
    }
  }
```

The next section makes extensive use of this example in order to demonstrate the principals of regular expression syntax. It is highly recommended that you compile this class and use it in order to experiment with the syntax as the section goes along.

Introduction to Regular Expression Syntax

Regular expression syntax is by far the most difficult part of the new regular expression capabilities found in J2SE 1.4. The syntax is cryptic and somewhat difficult to learn. It is not uncommon, in fact, to write a very large regular expression that validates some type of text sequence and then forget how the expression even does what it does after leaving it on a server for a few months. The following section provides the basics for writing massive pattern-matching statements. It should have you writing regular expressions in no time.

Let's begin with a simple example of a regular expression. First, we create a text file named list.txt that contains the following words:

- glad
- sad
- mad
- dad
- glee
- dabble
- add
- jeep
- sleep
- flap
- madd

This text file will serve as the input for the regular expression application that was detailed in the last section.

Now let's run the programming using our first regular expression:

```
C:\>java FileMatcher c:\list.txt "ad"
```

NOTE You may have to delimit the regular expression with quotes to run it from the command line, depending on your operating system. Thus, on Windows, your command line might look like C:\>java FileMatcher c:\list.txt "ad".

In the preceding command, the application is told to read in the file list.txt line by line and match each resulting `String` against the pattern “ad”. The results of this command are really not all that surprising. The application simply produces a list of all the words that contain the substring “ad”. The resulting list looks like this:

- glad
- sad
- mad
- dad
- add
- madd

The preceding match produces the expected results, but it does not really offer better character sequence interpretation than the facilities already provided in the `String` class. In order to see the power of regular expressions, let’s change the match slightly:

```
C:\>java FileMatcher c:\list.txt "ad$"
```

This command produces much different results. The returned word list is as follows:

- glad
- sad
- mad
- dad

Take note of the character pattern all of these words have in common: All the preceding words end in “ad”. Adding only one new character to the end of the specified pattern changed the search. Instead of searching the character sequence for a substring composed of “ad”, the pattern is now told to only match the “ad” substring if it appears at the end of the string. Thus, the pattern “ad$” introduces the first part of the regular expression syntax: boundary matchers.

Boundary Matchers

Boundary matchers are also very simple to understand. They specify a location, or locations, that a pattern may occur in a character sequence. Table 7-8 presents the full list of boundary matchers.

Table 7-8. List of Boundary Matchers

Boundary Matcher	**Description**
^	Match the pattern at the beginning of the line.
$	Match the pattern at the end of the line.
\b	Match the pattern at the beginning or end of a word.
\B	Match the pattern at the beginning or end of a nonword.
\A	Match the pattern at the beginning of input.
\G	Match the pattern at the end of the previous match.
\z	Match the pattern at the end of input.

Let's revisit the previous example. The pattern "ad$" can be further refined by using ^. This symbol specifies that the match must occur at the beginning of the character sequence. Thus, the command

```
C:\>java FileMatcher c:\list.txt "^ad"
```

produces one lonely pattern match: add.

Escape Characters

By now you should have noticed that most special characters and commands in the regular expression language begin with an escape character. Table 7-9 presents a list of escaped sequences that may be used in the construction of regular expressions.

Table 7-9. List of Escape Characters

Escape Character	Description
\\	The backslash character
\0n	The character with octal value 0n (0 <= n <= 7)
\0nn	The character with octal value 0nn (0 <= n <= 7)
\0mnn	The character with octal value 0mnn (0 <= m <= 3, 0 <= n <= 7)
\xhh	The character with hexadecimal value 0xhh
\uhhhh	The character with hexadecimal value 0xhhhh
\t	The tab character ('\u0009')
\n	The newline (line feed) character ('\u000A')
\r	The carriage return character ('\u000D')
\f	The form feed character ('\u000C')
\a	The alert (bell) character ('\u0007')
\e	The escape character ('\u001B')
\cx	The control character corresponding to x

In the regular expression syntax, a backslash followed by a character is considered reserved, just as `new` is considered reserved in the Java syntax. By reserving this sequence of characters, pattern syntax has the capability to accommodate future reserved characters.

Character Classes

Regular expressions treat each character as a grouping of similar items. For example, the character “c” is a class and any character that is “c” falls into the domain of the class. Therefore, only “c” would be a valid match for this class.

Essentially, character classes allow for the definition of a custom range of characters that are considered valid matches. Suppose we needed to match the character “f”, regardless of whether or not it was uppercase. I could create a custom character class “[fF]” that would encompass the realm of all “f” characters and produce a valid match for the upper- and lowercase versions of the character.

Table 7-10 presents some sample character classes. Notice that the classes can contain some basic logic and can include a range of characters or even exclude a range of characters.

Table 7-10. Character Classes

Character Class	**Description**
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z] (subtraction)

Let’s revisit the `FileMatcher` application and test out a character class. Recall the first example, in which the regular expression was simply “ad”. In this case, any `String` that contained “ad” *in that order* was interpreted as a match. Note that this pattern did not match the word “dabble,” since the “d” comes before the “a.” In order to change this, “ad” can be converted to the character class “[ad]”. Now, when this command is run:

```
C:\>java FileMatcher c:\list.txt "[ad]"
```

any word that contains “a” or “d”, in any order, will be considered a match. The resulting matching reflects this capability:

- glad
- sad
- mad
- dad

- dabble
- add
- flap
- madd

Next, run the command again, this time with the negation character class "`[^ad]`". Now the resulting list will contain words that have a character that is *not* "a" or "d". Thus, words such as "add" will not match this pattern, since all the characters in the sequence are either "a" or "d".

- glad
- sad
- mad
- glee
- dabble
- jeep
- sleep
- flap
- madd

Notice that all of the resulting words contain a character other than "a" or "d". For example, the token "glad" matches, because despite the fact that it has an "ad" character sequence in it, it also has a "gl" character sequence in it. The character sequence "gl" meets the criterion of not being "ad", so "glad," as a whole, is considered a match. "sad" similarly passes inspection, because it contains the character "s". For the same reason, the token "add" does *not* match. It does not include any characters that are not an "a" or a "d".

Greedy Quantifiers

Greedy quantifiers allow a pattern to match a character that is repeated *n* times. Table 7-11 presents a list of some quantifiers.

Table 7-11. Greedy Quantifiers

Greedy Quantifier	**Description**
X?	X, once or not at all
X*	X, zero or more times
X+	X, one or more times
X{n}	X, exactly n times
X(n,}	X, at least n times
X{n,m}	X, at least n but not more than m times

Let's modify our trusty example regular expression once again. This time, the pattern must match items that contain exactly two "e" characters:

```
C:\>java FileMatcher c:\list.txt "e{2}"
```

The preceding pattern "e{2}" will only match words that contain exactly two "e" characters. Thus, the printed lines are as follows:

- glee
- jeep
- sleep

Logical Operators and Capturing Groups

The last important regular expression elements we discuss are logical operators and capturing groups. The most prevalent logical operator is |, also known as "logical or." This allows for patterns such as "[q-s|y-z]", which will only match items that contain characters of range q–s or y–z. In the list.txt file, only two words match this pattern: "sad" and "sleep."

Next, pattern expressions can define capturing groups. Capturing groups are the equivalent of nested pattern matches contained between a set of parentheses.

For example, "e{2}(p)" will only match character sequences that contain two "e" characters followed by the character "p". Therefore, running the command

```
C:\>java FileMatcher c:\list.txt "e{2}(p)"
```

results in a list of two words that match the heuristic: "jeep" and "sleep."

Summary

The introduction of `NIO` marks a turning point for the Java platform. No longer are Java developers forced to live with a slow, cumbersome I/O framework. `NIO` offers I/O channels to improve speed and buffers to improve the platform's data processing capabilities. Regular expressions, though not part of the `NIO` package, offer a new level of character-sequence processing that is both fast and robust. One thing is certain: The improvements laid out in `NIO` are the groundwork for many great new possibilities for the Java platform.

FAQs

Q: Are there any regular expression packages for previous Java releases?

A: There are two regular expression packages that are free. They are written and maintained by The Apache Software Foundation. You can find the distributions at `http://jakarta.apache.org` by searching under "regexp" and "oro".

Q: Can I use third-party packages in the SCJD exam?

A: No, the use of third-party packages is expressly prohibited by the SCJD exam.

Q: Can I use nonblocking file I/O?

A: No, the `FileChannel` class does not support nonblocking file I/O.

Q: What kind of innovations can be expected from the NIO's buffer implementation?

A: Applications that must compress or decompress large data buffers have traditionally run slowly in Java, due to the lack of data buffer manipulation. This is not as much of an issue with `NIO`. Expect to see applications such as image compression/decompression utilities or even MP3 players.

Q: Where can I find help on regular expressions?

A: The Regular Expression Library (`http://www.regexplib.com`) offers an excellent and extensive resource for common (and uncommon) regular expressions. If you need a nontrivial regular expression, you will probably save a great deal of time by checking here first.