The first feature improves security by not showing the password on the screen if someone happens to be sitting next to you. The second feature involves preventing passwords from entering the `String` pool.

Reviewing Console Methods

The last code sample we present asks the user a series of questions and prints results based on this information using many of various methods we learned in this section:

```
Console console = System.console();
if (console == null) {
   throw new RuntimeException("Console not available");
} else {
   String name = console.readLine("Please enter your name: ");
   console.writer().format("Hi %s", name);
   console.writer().println();

   console.format("What is your address? ");
   String address = console.readLine();

   char[] password = console.readPassword("Enter a password "
      + "between %d and %d characters: ", 5, 10);
   char[] verify = console.readPassword("Enter the password again: ");
   console.printf("Passwords "
      + (Arrays.equals(password, verify) ? "match" : "do not match"));
}
```

Assuming the `Console` is available, the output should resemble the following:

```
Please enter your name: Max
Hi Max
What is your address? Spoonerville
Enter a password between 5 and 10 characters:
Enter the password again:
Passwords match
```

Working with Advanced APIs

Files, paths, I/O streams: you've worked with a lot this chapter! In this final section, we cover some advanced features of I/O streams and NIO.2 that can be quite useful in practice—and have been known to appear on the exam from time to time!

Manipulating Input Streams

All input stream classes include the following methods to manipulate the order in which data is read from an I/O stream:

```
// InputStream and Reader
public boolean markSupported()
public void mark(int readLimit)
public void reset() throws IOException
public long skip(long n) throws IOException
```

The `mark()` and `reset()` methods return an I/O stream to an earlier position. Before calling either of these methods, you should call the `markSupported()` method, which returns `true` only if `mark()` is supported. The `skip()` method is pretty simple; it basically reads data from the I/O stream and discards the contents.

TIP Not all input stream classes support `mark()` and `reset()`. Make sure to call `markSupported()` on the I/O stream before calling these methods, or an exception will be thrown at runtime.

Marking Data

Assume that we have an `InputStream` instance whose next values are `LION`. Consider the following code snippet:

```
public void readData(InputStream is) throws IOException {
   System.out.print((char) is.read());     // L
   if (is.markSupported()) {
      is.mark(100);  // Marks up to 100 bytes
      System.out.print((char) is.read());  // I
      System.out.print((char) is.read());  // O
      is.reset();    // Resets stream to position before I
   }
   System.out.print((char) is.read());    // I
   System.out.print((char) is.read());    // O
   System.out.print((char) is.read());    // N
}
```

The code snippet will output `LIOION` if `mark()` is supported and `LION` otherwise. It's a good practice to organize your `read()` operations so that the I/O stream ends up at the same position regardless of whether `mark()` is supported.

What about the value of `100` that we passed to the `mark()` method? This value is called the `readLimit`. It instructs the I/O stream that we expect to call `reset()` after at most `100` bytes. If our program calls `reset()` after reading more than `100` bytes from calling `mark(100)`, it may throw an exception, depending on the I/O stream class.

TIP

In actuality, mark() and reset() are not putting the data back into the I/O stream but are storing the data in a temporary buffer in memory to be read again. Therefore, you should not call the mark() operation with too large a value, as this could take up a lot of memory.

Skipping Data

Assume that we have an InputStream instance whose next values are TIGERS. Consider the following code snippet:

```
System.out.print ((char)is.read()); // T
is.skip(2);  // Skips I and G
is.read();   // Reads E but doesn't output it
System.out.print((char)is.read());  // R
System.out.print((char)is.read());  // S
```

This code prints TRS at runtime. We skipped two characters, I and G. We also read E but didn't use it anywhere, so it behaved like calling skip(1).

The return parameter of skip() tells us how many values were skipped. For example, if we are near the end of the I/O stream and call skip(1000), the return value might be 20, indicating that the end of the I/O stream was reached after 20 values were skipped. Using the return value of skip() is important if you need to keep track of where you are in an I/O stream and how many bytes have been processed.

Reviewing Manipulation APIs

Table 14.11 reviews these APIs related to manipulating I/O input streams. While you may not have used these in practice, you need to know them for the exam.

TABLE 14.11 Common I/O stream methods

Method name	Description
`public boolean markSupported()`	Returns `true` if stream class supports `mark()`
`public mark(int readLimit)`	Marks current position in stream
`public void reset()`	Attempts to reset stream to `mark()` position
`public long skip(long n)`	Reads and discards specified number of characters

Discovering File Attributes

We begin our discussion by presenting the basic methods for reading file attributes. These methods are usable within any file system, although they may have limited meaning in some file systems.

Checking for Symbolic Links

Earlier, we saw that the `Files` class has methods called `isDirectory()` and `isRegularFile()`, which are similar to the `isDirectory()` and `isFile()` methods on `File`. While the `File` object can't tell you if a reference is a symbolic link, the `isSymbolicLink()` method on `Files` can.

It is possible for `isDirectory()` or `isRegularFile()` to return `true` for a symbolic link, as long as the link resolves to a directory or regular file, respectively. Let's take a look at some sample code:

```
System.out.print(Files.isDirectory(Paths.get("/canine/fur.jpg")));
System.out.print(Files.isSymbolicLink(Paths.get("/canine/coyote")));
System.out.print(Files.isRegularFile(Paths.get("/canine/types.txt")));
```

The first example prints `true` if `fur.jpg` is a directory or a symbolic link to a directory and `false` otherwise. The second example prints `true` if `/canine/coyote` is a symbolic link, regardless of whether the file or directory it points to exists. The third example prints `true` if `types.txt` points to a regular file or a symbolic link that points to a regular file.

Checking File Accessibility

In many file systems, it is possible to set a `boolean` attribute to a file that marks it hidden, readable, or executable. The `Files` class includes methods that expose this information: `isHidden()`, `isReadable()`, `isWriteable()`, and `isExecutable()`.

A hidden file can't normally be viewed when listing the contents of a directory. The readable, writable, and executable flags are important in file systems where the filename can be viewed, but the user may not have permission to open the file's contents, modify the file, or run the file as a program, respectively.

Here we present an example of each method:

```
System.out.print(Files.isHidden(Paths.get("/walrus.txt")));
System.out.print(Files.isReadable(Paths.get("/seal/baby.png")));
System.out.print(Files.isWritable(Paths.get("dolphin.txt")));
System.out.print(Files.isExecutable(Paths.get("whale.png")));
```

If the `walrus.txt` file exists and is hidden within the file system, the first example prints `true`. The second example prints `true` if the `baby.png` file exists and its contents are readable. The third example prints `true` if the `dolphin.txt` file can be modified. Finally, the last example prints `true` if the file can be executed within the operating system. Note that the file extension does not necessarily determine whether a file is executable. For example, an image file that ends in `.png` could be marked executable in some file systems.

With the exception of the `isHidden()` method, these methods do not declare any checked exceptions and return `false` if the file does not exist.

Improving Attribute Access

Up until now, we have been accessing individual file attributes with multiple method calls. While this is functionally correct, there is often a cost each time one of these methods is called. Put simply, it is far more efficient to ask the file system for all of the attributes at once rather than performing multiple round trips to the file system. Furthermore, some attributes are file system–specific and cannot be easily generalized for all file systems.

NIO.2 addresses both of these concerns by allowing you to construct views for various file systems with a single method call. A *view* is a group of related attributes for a particular file system type. That's not to say that the earlier attribute methods that we just finished discussing do not have their uses. If you need to read only one attribute of a file or directory, requesting a view is unnecessary.

Understanding Attribute and View Types

NIO.2 includes two methods for working with attributes in a single method call: a read-only attributes method and an updatable view method. For each method, you need to provide a file system type object, which tells the NIO.2 method which type of view you are requesting. By updatable view, we mean that we can both read and write attributes with the same object.

Table 14.12 lists the commonly used attributes and view types. For the exam, you only need to know about the basic file attribute types. The other views are for managing operating system–specific information.

TABLE 14.12 The attributes and view types

Attributes interface	View interface	Description
`BasicFileAttributes`	`BasicFileAttributeView`	Basic set of attributes supported by all file systems
`DosFileAttributes`	`DosFileAttributeView`	Basic set of attributes along with those supported by DOS/Windows-based systems
`PosixFileAttributes`	`PosixFileAttributeView`	Basic set of attributes along with those supported by POSIX systems, such as Unix, Linux, Mac, etc.

Retrieving Attributes

The `Files` class includes the following method to read attributes of a class in a read-only capacity:

```
public static <A extends BasicFileAttributes> A readAttributes(
   Path path,
   Class<A> type,
   LinkOption... options) throws IOException
```

Applying it requires specifying the `Path` and `BasicFileAttributes.class` parameters.

```
var path = Paths.get("/turtles/sea.txt");
BasicFileAttributes data = Files.readAttributes(path,
   BasicFileAttributes.class);

System.out.println("Is a directory? " + data.isDirectory());
System.out.println("Is a regular file? " + data.isRegularFile());
System.out.println("Is a symbolic link? " + data.isSymbolicLink());
System.out.println("Size (in bytes): " + data.size());
System.out.println("Last modified: " + data.lastModifiedTime());
```

The `BasicFileAttributes` class includes many values with the same name as the attribute methods in the `Files` class. The advantage of using this method, though, is that all of the attributes are retrieved at once for some operating systems.

Modifying Attributes

The following `Files` method returns an updatable view:

```
public static <V extends FileAttributeView> V getFileAttributeView(
   Path path,
   Class<V> type,
   LinkOption... options)
```

We can use the updatable view to increment a file's last modified date/time value by 10,000 milliseconds, or 10 seconds.

```
// Read file attributes
var path = Paths.get("/turtles/sea.txt");
BasicFileAttributeView view = Files.getFileAttributeView(path,
   BasicFileAttributeView.class);
BasicFileAttributes attributes = view.readAttributes();

// Modify file last modified time
FileTime lastModifiedTime = FileTime.fromMillis(
   attributes.lastModifiedTime().toMillis() + 10_000);
view.setTimes(lastModifiedTime, null, null);
```

After the updatable view is retrieved, we need to call `readAttributes()` on the view to obtain the file metadata. From there, we create a new `FileTime` value and set it using the `setTimes()` method:

```
// BasicFileAttributeView instance method
public void setTimes(FileTime lastModifiedTime,
   FileTime lastAccessTime, FileTime createTime)
```

This method allows us to pass `null` for any date/time value that we do not want to modify. In our sample code, only the last modified date/time is changed.

NOTE Not all file attributes can be modified with a view. For example, you cannot set a property that changes a file into a directory. Likewise, you cannot change the size of the object without modifying its contents.

Traversing a Directory Tree

While the `Files.list()` method is useful, it traverses the contents of only a single directory. What if we want to visit all of the paths within a directory tree? Before we proceed, we need to review some basic concepts about file systems. Remember that a directory is organized in a hierarchical manner. For example, a directory can contain files and other directories, which can in turn contain other files and directories. Every record in a file system has exactly one parent, with the exception of the root directory, which sits atop everything.

A file system is commonly visualized as a tree with a single root node and many branches and leaves. In this model, a directory is a branch or internal node, and a file is a leaf node.

A common task in a file system is to iterate over the descendants of a path, either recording information about them or, more commonly, filtering them for a specific set of files. For example, you may want to search a folder and print a list of all of the `.java` files. Furthermore, file systems store file records in a hierarchical manner. Generally speaking, if you want to search for a file, you have to start with a parent directory, read its child elements, then read their children, and so on.

Traversing a directory, also referred to as walking a directory tree, is the process by which you start with a parent directory and iterate over all of its descendants until some condition is met or there are no more elements over which to iterate. For example, if we're searching for a single file, we can end the search when the file is found or we've checked all files and come up empty. The starting path is usually a specific directory; after all, it would be time-consuming to search the entire file system on every request!

Don't Use *DirectoryStream* and *FileVisitor*

While browsing the NIO.2 Javadocs, you may come across methods that use the `DirectoryStream` and `FileVisitor` classes to traverse a directory. These methods predate the existence of the Stream API and were even required knowledge for older Java certification exams.

The best advice we can give you is to not use them. The newer Stream API–based methods are superior and accomplish the same thing, often with much less code.

Selecting a Search Strategy

Two common strategies are associated with walking a directory tree: a depth-first search and a breadth-first search. A *depth-first search* traverses the structure from the root to an arbitrary leaf and then navigates back up toward the root, traversing fully any paths it skipped along the way. The *search depth* is the distance from the root to current node. To prevent endless searching, Java includes a search depth that is used to limit how many levels (or hops) from the root the search is allowed to go.

Alternatively, a *breadth-first search* starts at the root and processes all elements of each particular depth before proceeding to the next depth level. The results are ordered by depth, with all nodes at depth `1` read before all nodes at depth `2`, and so on. While a breadth-first search tends to be balanced and predictable, it also requires more memory since a list of visited nodes must be maintained.

For the exam, you don't have to understand the details of each search strategy that Java employs; you just need to be aware that the NIO.2 Stream API methods use depth-first searching with a depth limit, which can be optionally changed.

Walking a Directory

That's enough background information; let's get to more Stream API methods. The `Files` class includes two methods for walking the directory tree using a depth-first search.

```
public static Stream<Path> walk(Path start,
   FileVisitOption... options) throws IOException

public static Stream<Path> walk(Path start, int maxDepth,
   FileVisitOption... options) throws IOException
```

Like our other stream methods, `walk()` uses lazy evaluation and evaluates a `Path` only as it gets to it. This means that even if the directory tree includes hundreds or thousands of files, the memory required to process a directory tree is low. The first `walk()` method relies on a default maximum depth of `Integer.MAX_VALUE`, while the overloaded version allows the user to set a maximum depth. This is useful in cases where the file system might be large and we know the information we are looking for is near the root.

Rather than just printing the contents of a directory tree, we can again do something more interesting. The following getPathSize() method walks a directory tree and returns the total size of all the files in the directory:

```
private long getSize(Path p) {
   try {
      return  Files.size(p);
   } catch (IOException e) {
      throw new UncheckedIOException(e);
   }
}

public long getPathSize(Path source) throws IOException {
   try (var s = Files.walk(source)) {
      return s.parallel()
            .filter(p -> !Files.isDirectory(p))
            .mapToLong(this::getSize)
            .sum();
   }
}
```

The getSize() helper method is needed because Files.size() declares IOException, and we'd rather not put a try/catch block inside a lambda expression. Instead, we wrap it in the unchecked exception class UncheckedIOException. We can print the data using the format() method:

```
var size = getPathSize(Path.of("/fox/data"));
System.out.format("Total Size: %.2f megabytes", (size/1000000.0));
```

Depending on the directory you run this on, it will print something like this:

```
Total Size: 15.30 megabytes
```

Applying a Depth Limit

Let's say our directory tree is quite deep, so we apply a depth limit by changing one line of code in our getPathSize() method.

```
    try (var s = Files.walk(source, 5)) {
```

This new version checks for files only within 5 steps of the starting node. A depth value of 0 indicates the current path itself. Since the method calculates values only on files, you'd have to set a depth limit of at least 1 to get a nonzero result when this method is applied to a directory tree.

Avoiding Circular Paths

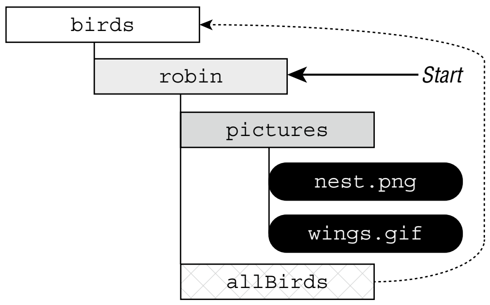
Many of our earlier NIO.2 methods traverse symbolic links by default, with a NOFOLLOW_LINKS used to disable this behavior. The walk() method is different in that it does *not* follow symbolic links by default and requires the FOLLOW_LINKS option to be enabled. We can alter our `getPathSize()` method to enable following symbolic links by adding the `FileVisitOption`:

```
try (var s = Files.walk(source,
      FileVisitOption.FOLLOW_LINKS)) {
```

When traversing a directory tree, your program needs to be careful of symbolic links, if enabled. For example, if our process comes across a symbolic link that points to the root directory of the file system, every file in the system will be searched!

Worse yet, a symbolic link could lead to a cycle in which a path is visited repeatedly. A *cycle* is an infinite circular dependency in which an entry in a directory tree points to one of its ancestor directories. Let's say we had a directory tree as shown in Figure 14.7 with the symbolic link `/birds/robin/allBirds` that points to `/birds`.

FIGURE 14.7 File system with cycle



What happens if we try to traverse this tree and follow all symbolic links, starting with `/birds/robin`? Table 14.13 shows the paths visited after walking a depth of `3`. For simplicity, we walk the tree in a breadth-first ordering, *although a cycle occurs regardless of the search strategy used.*

TABLE 14.13 Walking a directory with a cycle using breadth-first search

Depth	Path reached
0	`/birds/robin`
1	`/birds/robin/pictures`
1	`/birds/robin/allBirds` ➢ `/birds`
2	`/birds/robin/pictures/nest.png`
2	`/birds/robin/pictures/wings.gif`

Depth	Path reached
2	**`/birds/robin/allBirds/robin`** ➢ **`/birds/robin`**
3	`/birds/robin/allBirds/robin/pictures` ➢ `/birds/robin/pictures`

3	`/birds/robin/allBirds/robin/pictures/allBirds`
	 `/birds/robin/allBirds`
	 ➢ `/birds`

After walking a distance of `1` from the start, we hit the symbolic link `/birds/robin/allBirds` and go back to the top of the directory tree `/birds`. That's okay because we haven't visited `/birds` yet, so there's no cycle yet!

Unfortunately, at depth `2`, we encounter a cycle. We've already visited the `/birds/robin` directory on our first step, and now we're encountering it again. If the process continues, we'll be doomed to visit the directory over and over again.

Be aware that when the `FOLLOW_LINKS` option is used, the `walk()` method will track all of the paths it has visited, throwing a `FileSystemLoopException` if a path is visited twice.

Searching a Directory

In the previous example, we applied a filter to the `Stream<Path>` object to filter the results, although there is a more convenient method.

```
public static Stream<Path> find(Path start,
   int maxDepth,
   BiPredicate<Path, BasicFileAttributes> matcher,
   FileVisitOption... options) throws IOException
```

The `find()` method behaves in a similar manner as the `walk()` method, except that it takes a `BiPredicate` to filter the data. It also requires a depth limit to be set. Like `walk()`, `find()` also supports the `FOLLOW_LINK` option.

The two parameters of the `BiPredicate` are a `Path` object and a `BasicFileAttributes` object, which you saw earlier in the chapter. In this manner, Java automatically retrieves the basic file information for you, allowing you to write complex lambda expressions that have direct access to this object. We illustrate this with the following example:

```
Path path = Paths.get("/bigcats");
long minSize = 1_000;
try (var s = Files.find(path, 10,
      (p, a) -> a.isRegularFile()
```

```
         && p.toString().endsWith(".java")
         && a.size() > minSize)) {
   s.forEach(System.out::println);
}
```

This example searches a directory tree and prints all `.java` files with a size of at least 1,000 bytes, using a depth limit of `10`. While we could have accomplished this using the `walk()` method along with a call to `readAttributes()`, this implementation is a lot shorter and more convenient than those would have been. We also don't have to worry about any methods within the lambda expression declaring a checked exception, as we saw in the `getPathSize()` example.

Review of Key APIs

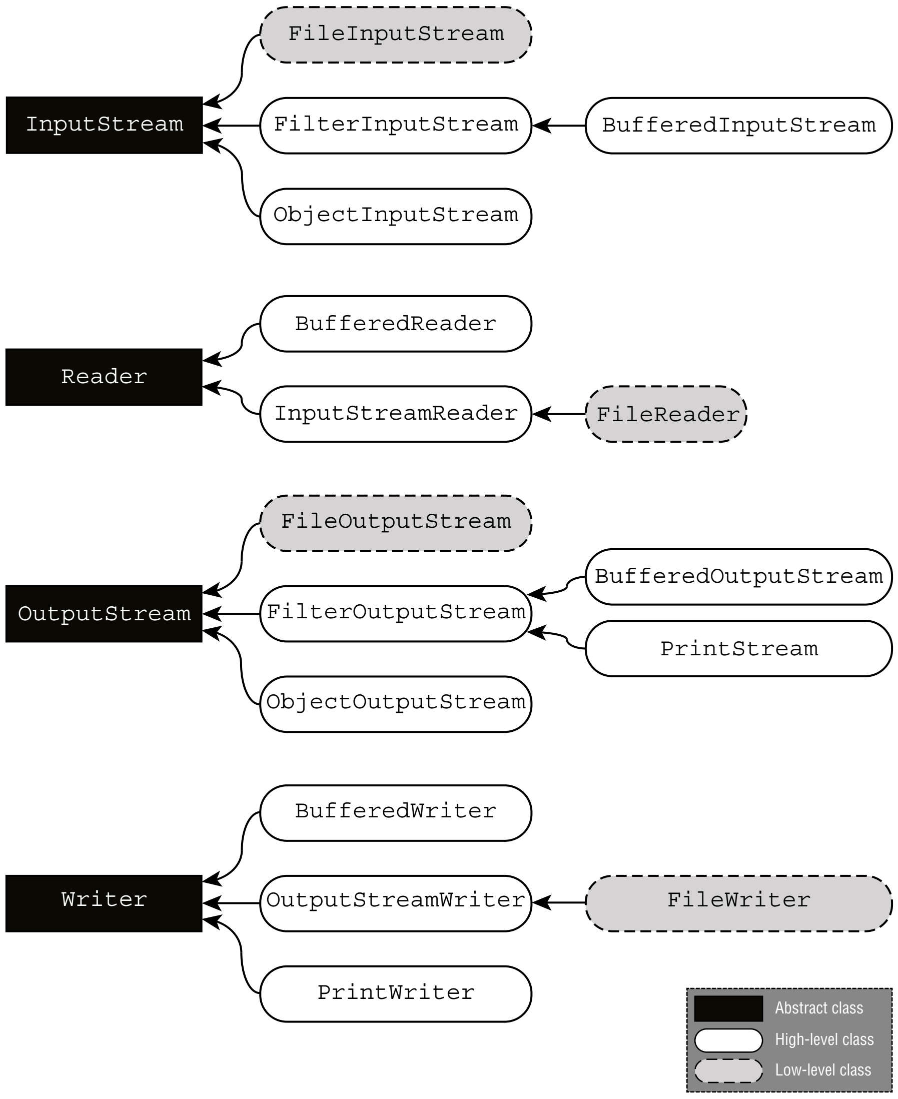
The key APIs that you need to know for the exam are listed in Table 14.14. We know some of the classes look similar. You need to know this table really well before taking the exam.

TABLE 14.14 Key APIs

Class	Purpose
`File`	I/O representation of location in file system
`Files`	Helper methods for working with `Path`
`Path`	NIO.2 representation of location in file system
`Paths`	Contains factory methods to get `Path`
`URI`	Uniform resource identifier for files, URLs, etc.
`FileSystem`	NIO.2 representation of file system
`FileSystems`	Contains factory methods to get `FileSystem`
`InputStream`	Superclass for reading files based on bytes
`OuputStream`	Superclass for writing files based on bytes
`Reader`	Superclass for reading files based on characters
`Writer`	Superclass for writing files based on characters

Additionally, Figure 14.8 shows all of the I/O stream classes that you should be familiar with for the exam, with the exception of the filter streams. `FilterInputStream` and `FilterOutputStream` are high-level superclasses that filter or transform data. They are rarely used directly.

FIGURE 14.8 Diagram of I/O stream classes



The InputStreamReader and OutputStreamWriter are incredibly convenient and are also unique in that they are the only I/O stream classes to have both InputStream/OutputStream and Reader/Writer in their name.

Summary

This chapter is all about reading and writing data. We started by showing you how to create File from I/O and Path from NIO.2. We then covered the functionality that works with both I/O and NIO.2 before getting into NIO.2-specific APIs. You should be familiar with how to combine or resolve Path objects with other Path objects. Additionally, NIO.2 includes Stream API methods that can be used to process files and directories. We discussed methods for listing a directory, walking a directory tree, searching a directory tree, and reading the lines of a file.

We spent time reviewing various methods available in the Files helper class. As discussed, the name of the function often tells you exactly what it does. We explained that most of these methods are capable of throwing an IOException, and many take optional varargs enum values.

We then introduced I/O streams and explained how they are used to read or write large quantities of data. While there are a lot of I/O streams, they differ on some key points:

- Byte vs. character streams
- Input vs. output streams
- Low-level vs. high-level streams

Often, the name of the I/O stream can tell you a lot about what it does. We visited many of the I/O stream classes that you will need to know for the exam in increasing order of complexity. A common practice is to start with a low-level resource or file stream and wrap it in a buffered I/O stream to improve performance. You can also apply a high-level stream to manipulate the data, such as an object or print stream. We described what it means to be serializable in Java, and we showed you how to use the object stream classes to persist objects directly to and from disk.

We explained how to read input data from the user using both the system stream objects and the Console class. The Console class has many useful features, such as built-in support for passwords and formatting.

We also discussed how NIO.2 provides methods for reading and writing file metadata. NIO.2 includes two methods for retrieving all of the file system attributes for a path in a single call without numerous round trips to the operating system. One method requires a read-only attribute type, while the second method requires an updatable view type. It also allows NIO.2 to support operating system–specific file attributes.