*OCP Java SE 7 Programmer II*

*7. Java I/O fundamentals*

Java I/O lets you read your files, data, photos, and videos from multiple sources and write them to several destinations.

NOTE Java version 7 has introduced a new interface that offers the existing functionality of class File, addresses its existing issues, and offers additional functionality: java.nio.file.Path.

EXAM TIP The objects of class File are immutable; the pathname represented by a File object can’t be changed.

You can create File object in following 3 ways.

File(String pathname)

File(File parent, String child)

File(String parent, String child)

EXAM TIP You can create a File instance that represents a nonexistent file on your file system. And you can even invoke methods like isFile() and isDirectory() methods it will return false.

All input streams extend the base abstract class java.io.InputStream, and all output streams extend the base abstract class java.io.OutputStream. Let’s start with input streams.

***Using byte stream I/O***

***Input streams***

Class java.io.InputStream is an abstract base class for all the input streams in Java. The class InputStream defines multiple overloaded versions of method read(), which can be used to read a single byte of data as int, or multiple bytes into a byte array:

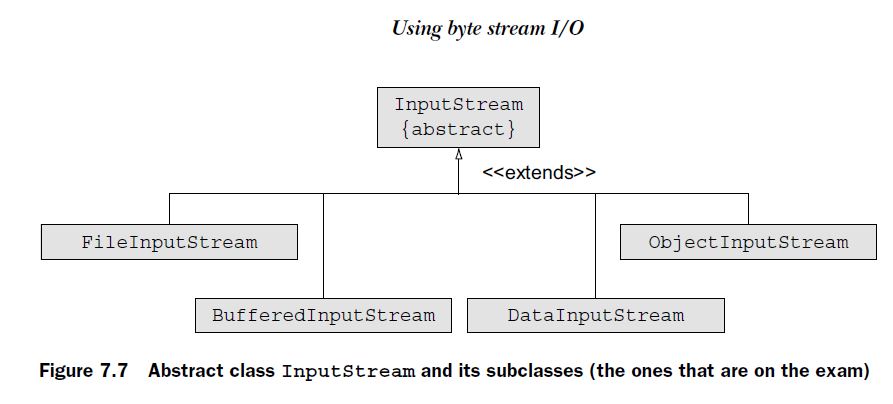
int abstract read()

int read(byte[] b)

int read(byte[] b, int off, int len)

InputStream is abstract class you can’t create object of it. You’d use method read() by more specific classes that extend the abstract class InputStream. For example, class FileInputStream extends InputStream and overrides its read() method for you to use. Method close() is another important method of class InputStream. Calling close() on a stream releases the system resources associated with it.

EXAM TIP Watch out for the use of method read() from class InputStream. It returns the next byte of data, or -1 if the end of the stream is reached. It doesn’t throw an EOFException.



Apart from image files, you can also read character data by using byte streams. But you aren’t encouraged.

***Output streams***

Class java.io.OutputStream is also an abstract class. It’s extended by all the classes that need to write bytes (for example, image data) to multiple data destinations. The most important method of this class is write(), which can be used to write a single byte of data or multiple bytes from a byte array to a data destination:

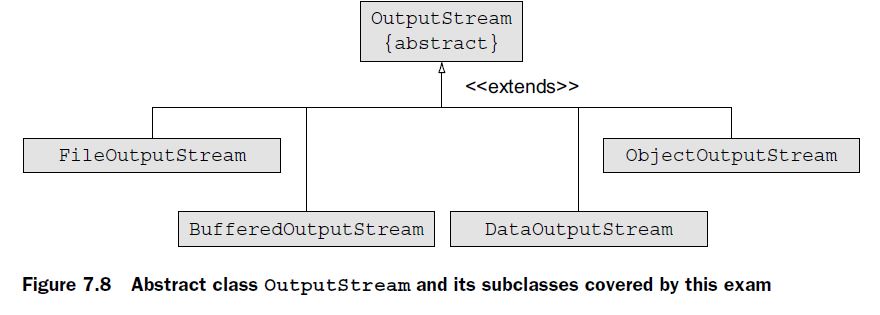
abstract void write(int b)

void write(byte[] b)

void write(byte[] b, int off, int len)

Methods close() and flush() are other important methods of class OutputStream. Often data isn’t written directly to the output stream but buffered for an efficient management of resources. If you want to write data to the output stream right away without waiting for the buffer to be full, call flush(). Method close() is used to release system resources being used by this stream.

EXAM TIP Class OutputStream defines methods write(), flush(), and close(). So these are valid methods that can be called on any objects of classes that extend class OutputStream.



EXAM TIP FileInputStream is instantiated by passing it a File or String instance. It can’t be instantiated by passing it another InputStream. The above-mentioned constructors of class FileInputStream throw a checked exception, FileNotFoundException, which must be handled accordingly. You can also pass a boolean value specifying whether to append to the existing file contents.

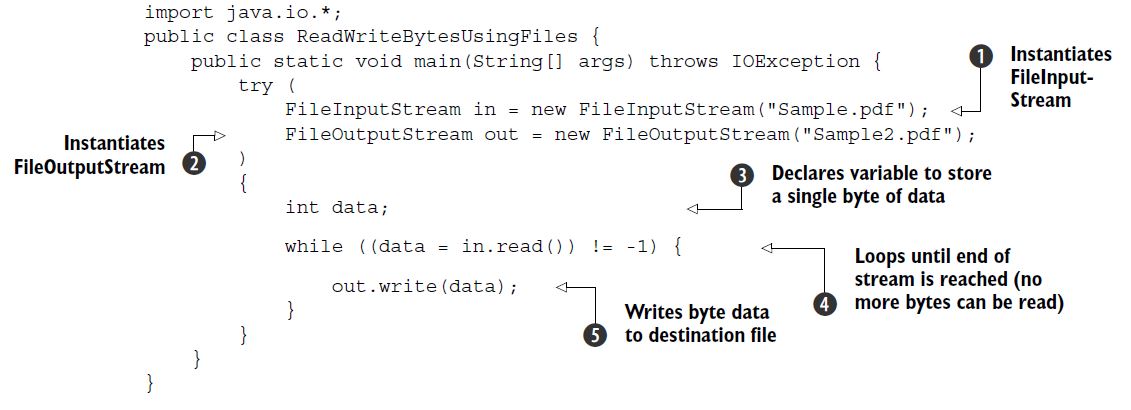
FileOutputStream(File file) throws FileNotFoundException

FileOutputStream(File file, boolean append) throws FileNotFoundException

FileOutputStream(String name) throws FileNotFoundException

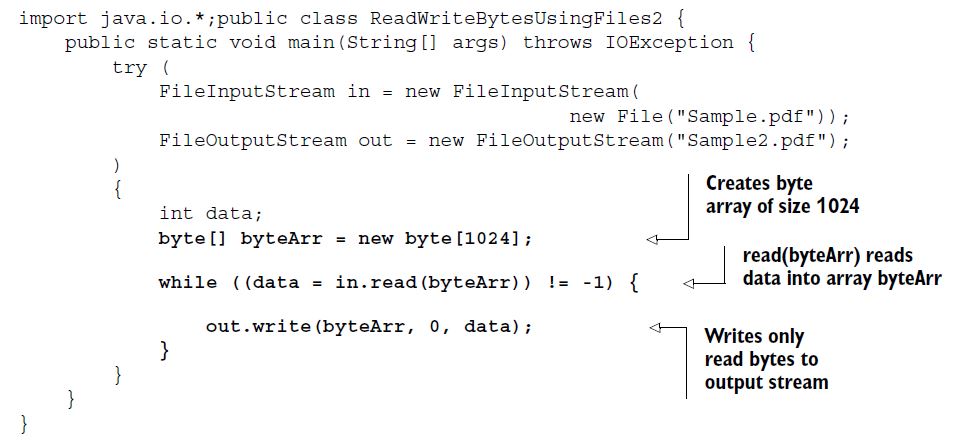
FileOutputStream(String nm, boolean append) throws FileNotFoundException

EXAM TIP The above-mentioned constructors of FileOutputStream throw a FileNotFoundException, a checked exception. Also, during its instantiation, you can specify whether to append data to an underlying file or override its contents.



EXAM TIP Are you wondering why you need to create a variable of type int to read byte data from a file in the preceding code? When a stream exhausts itself and no data can be read from it, method read() returns -1, which can’t be stored by a variable of type byte.

I/O operations that require reading and writing of a single byte from and to a file are a costly affair. To optimize these operations, you can use a byte array:



read(byte[]) method return returns the *count of bytes*, or -1.

EXAM TIP Method write(int) in class OutputStream writes a byte to the underlying output stream. If you write an int value by using this method, only the 8 low-order bits are written to the output stream; the rest are ignored.

***Buffered I/O with byte streams***

Buffering stores data in memory before sending a read or write request to the underlying I/O devices. **Buffering *drastically* reduces the time** required for performing reading and writing I/O operations.

To buffer data with byte streams, you need classes BufferedInputStream and BufferedOutputStream. You can instantiate a BufferedInputStream by passing it an InputStream instance. A BufferedOutputStream can be instantiated by passing it an OutputStream instance. You can also specify a buffer size or use the default size. Here are their constructors:

public BufferedInputStream(InputStream in)

public BufferedInputStream(InputStream in, int size)

public BufferedOutputStream(OutputStream out)

public BufferedOutputStream(OutputStream out, int size)

EXAM TIP The exam might test you on how to instantiate buffered streams correctly. To instantiate BufferedInputStream, you must pass it an object of InputStream. To instantiate BufferedOutputStream, you must pass it an object of OutputStream.

Disadvantages:

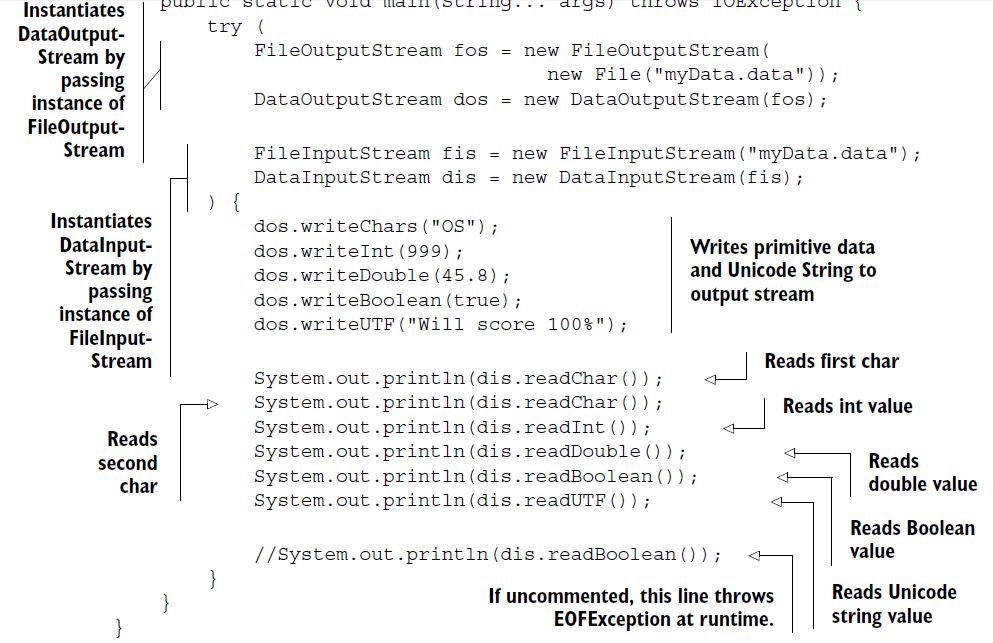
You can use FileInputStream and FileOutputStream to read and write *only* byte data they didn’t define methods to work with any other specific primitive data types or objects, which is what you might need most of the time.

***Primitive values and strings I/O with byte streams***

DataInputStream and DataOutputStream let you read and write primitive values (char, int, double, and boolean) and strings from and to an underlying I/O stream in a machine-independent way.

DataInputStream(InputStream in)

DataOutputStream(OutputStream out)



DataInputStream should read the date same order as written by DataOutputStream. If the data being read doesn’t match the data that was written, you’ll get unexpected values.

EXAM TIP If a mismatch occurs in the type of data written by DataOutputStream and the type of data read by DataInputStream, you might not get a runtime exception. Because data streams read and write bytes, the read operation constructs the requested data from the available bytes, though incorrectly.

***Object I/O with byte streams***

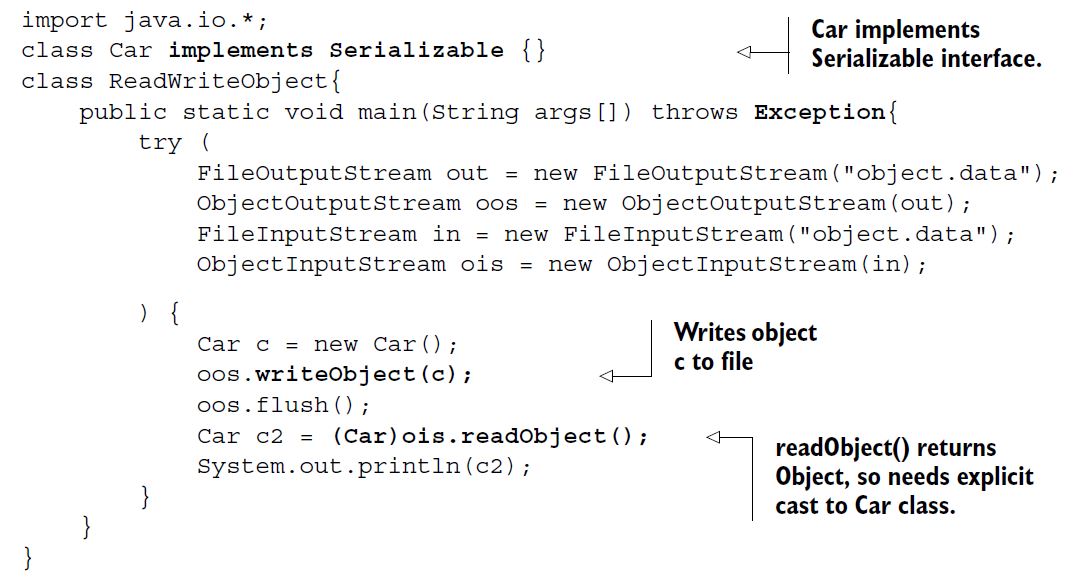
You can use classes ObjectInputStream and ObjectOutputStream to read and write objects *and* primitive values.

You can instantiate these classes by passing them objects of InputStream or OutputStream.

public ObjectInputStream(InputStream in)

public ObjectOutputStream(OutputStream out)

EXAM TIP You can use ObjectOutputStream and ObjectInputStream to read and write all serializable objects *and* primitive values.



The Car should implement the Serializable interface so that it can be written to and read from a file.

Apart from declaring to throw an IOException, method readObject() might also throw a ClassNotFoundException, if the JRE fails to retrieve the class information corresponding to the retrieved object.

EXAM TIP To write objects to a file, their classes should implement Serializable, otherwise you will get NotSerializableException.

READ AND WRITE OBJECTS WITH NONSERIALIZABLE PARENT CLASSES

**class Vehicle** {

String name = "Vehicle";

}

**class Car extends Vehicle implements Serializable** {

String model = "Luxury";

}

class ParentNotSerializable{

public static void main(String args[]) throws Exception{

try (

FileOutputStream out = new FileOutputStream("object.data");

ObjectOutputStream oos = new ObjectOutputStream(out);

FileInputStream in = new FileInputStream("object.data");

ObjectInputStream ois = new ObjectInputStream(in);

) {

Car c = new Car();

oos.writeObject(c);

oos.flush();

Car c2 = (Car)ois.readObject();

System.out.println(**c2.name** + ":" + **c2.model**);// **Prints Vehicle:Luxury**

}

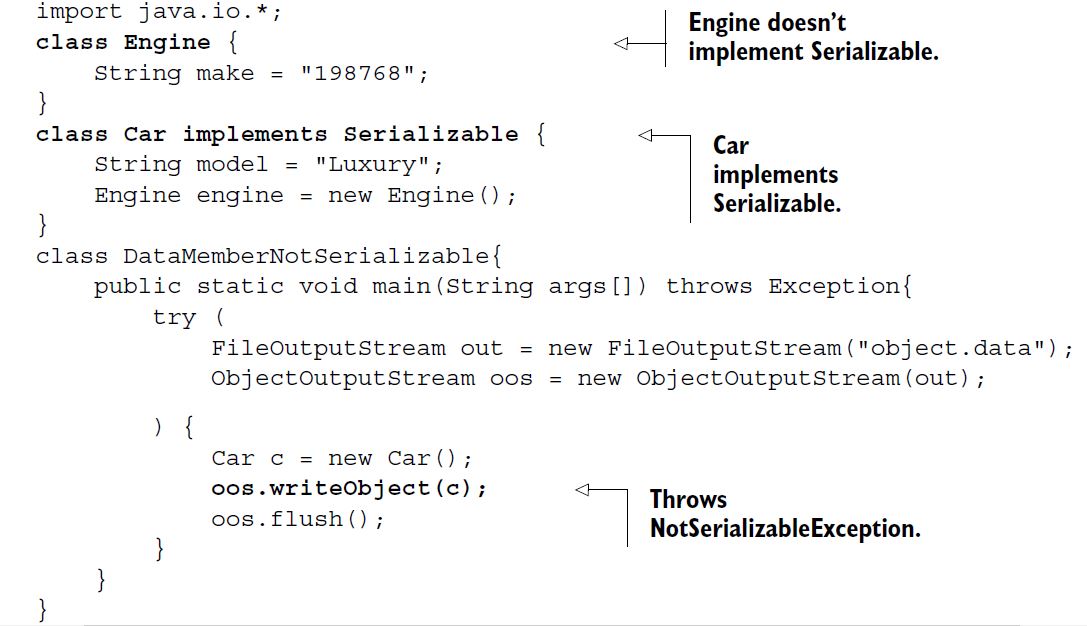
}

}

READ AND WRITE OBJECTS WITH NONSERIALIZABLE DATA MEMBERS

Would you be able to write objects to Car to a file, if any of its object fields doesn’t implement the Serializable interface? In this case, the code will throw a java .io.NotSerializableException when you attempt to *write* a Car object to a file.

For example:



EXAM TIP A class whose object fields don’t implement the Serializable interface can’t be serialized even though the class itself implements the Serializable interface. An attempt to serialize such object fields will throw a runtime exception.

READ AND WRITE OBJECTS ALONG WITH PRIMITIVE VALUES FROM AND TO A FILE

You can use ObjectInputStream and ObjectOutputStream to read and write both objects and primitive values from and to a file. The data should be retrieved in the order that it was written. In the following example, class WritePrimAndObjects writes a boolean value and then a Car instance.

try (

FileInputStream in = new FileInputStream("object.data");

ObjectInputStream ois = new ObjectInputStream(in);

) {

System.out.println(ois.readBoolean());

Car c = (Car)ois.readObject();//**readObject returns instance of Object and**

**can throw OptionalDataException**

System.out.println(c.name);

}

EXAM TIP Retrieve the data (primitive and objects) in the order it was written using object streams, or it might throw a runtime exception.

Method readObject() can throw multiple exceptions:

* ClassNotFoundException—Class of a serialized object cannot be found
* OptionalDataException—Primitive data was found in the stream instead of objects.
* IOException—Any of the usual input-/output-related exceptions

THE TRANSIENT AND STATIC VARIABLES AREN’T WRITTEN TO A FILE

class Car implements Serializable{

String name;

**transient String model;**

**transient int days;**

**static int carCount;**

Car(String value) {

name = value;

**model = "some value";**

**days = 98;**

++carCount;

}

}

class ReadWriteCarObjects{

public static void main(String args[]) throws Exception {

try (

FileOutputStream out = new FileOutputStream("object.data");

ObjectOutputStream oos = new ObjectOutputStream(out);

FileInputStream in = new FileInputStream("object.data");

ObjectInputStream ois = new ObjectInputStream(in);

) {

Car c = new Car("AAA");

oos.writeObject(c);

oos.flush();

new Car("BBB");

Car c2 = (Car)ois.readObject();

System.out.println(c2.name);

System.out.println(c2.model + ":" + c2.days);// **Prints null:0**

System.out.println(c2.carCount);

}

}

}

The value of transient variables model and days wasn’t written to the file, the deserialization process assigns default values to these variables: null for objects and 0 for int type.

***Using character I/O with readers and writers***

Reader and Writer are abstract base classes for reading and writing Unicode compliant character data. They don’t replace the byte-oriented I/O classes, but supplement them.

***Abstract class java.io.Reader***

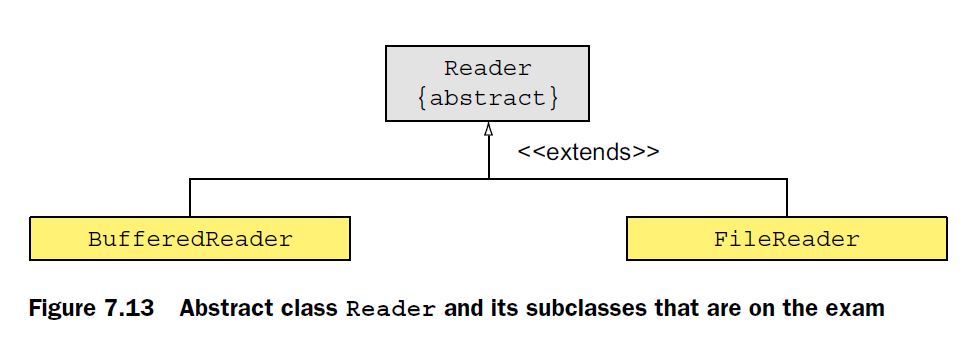
Class Reader defines overloaded read() methods to read character data from an underlying data stream:

int read()

int read(char[] cbuf)

abstract int read(char[] cbuf, int off, int len)

EXAM TIP Compare the overloaded read() methods of class InputStream with the read() methods of class Reader. The read() methods of InputStream accept an array of byte as their method parameter, and the read() methods of Reader accept an array of char as their method parameter.



***Abstract class java.io.Writer***

The abstract class Writer defines overloaded write() methods to write character data to an underlying data source:

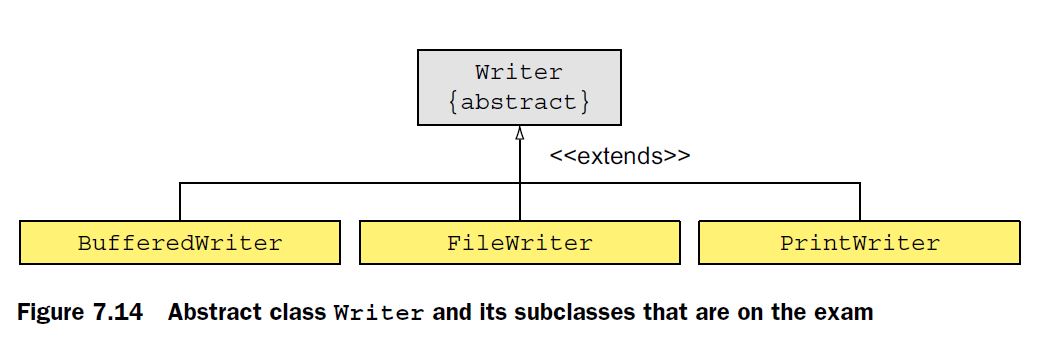
void write(char[] cbuf)

abstract void write(char[] cbuf, int off, int len)

void write(int c)

void write(String str)

void write(String str, int off, int len)



EXAM TIP With the overloaded write() methods of class Writer, you can write a single character or multiple characters, stored in char arrays or String, to a data source.

***File I/O with character streams***

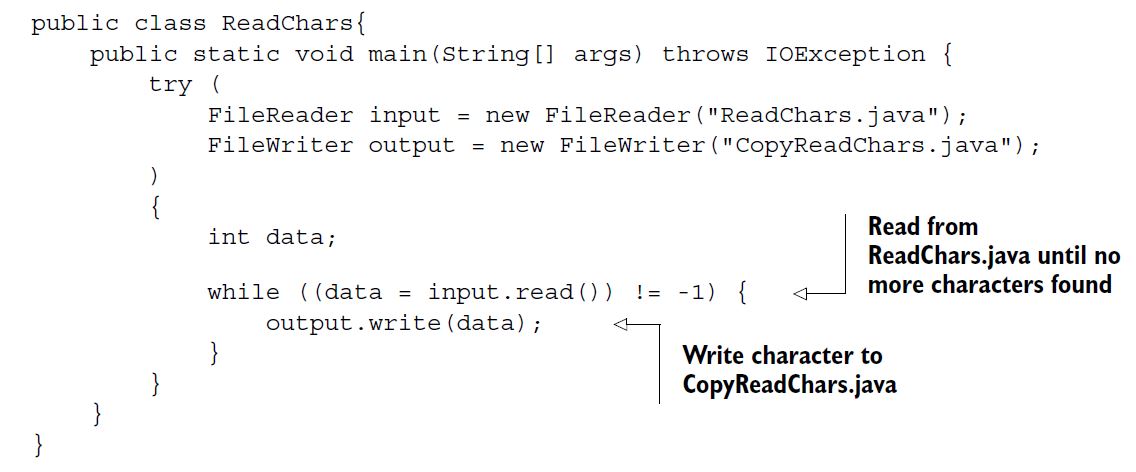
FileReader and FileWriter are convenience classes for reading and writing character data from files.

FileWriter(File file)

FileWriter(File file, boolean append)

FileWriter(String fileName)

FileWriter(String fileName, boolean append)



The preceding code is similar to the code written using FileInputStream and FileOutputStream to read and write bytes from files. But it uses FileReader to read characters from a source and FileWriter to write it to a destination. Data buffering helps produce efficient and faster I/O operations.

***Buffered I/O with character streams***

To buffer data with character streams, you need classes BufferedReader and BufferedWriter. You can instantiate a BufferedReader by passing it a Reader instance. A BufferedWriter can be instantiated by passing it a Writer instance. You can also specify a buffer size or use the default size. Here are their constructors:

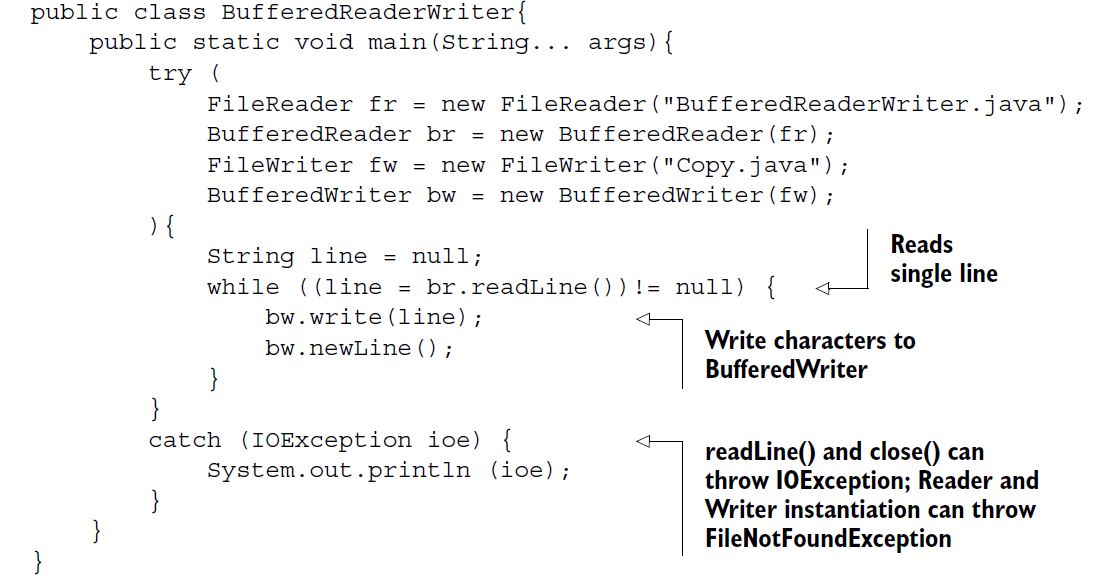
public BufferedReader(Reader in)

public BufferedReader(Reader in, int sz)

public BufferedWriter(Writer out)

public BufferedWriter(Writer out, int sz)

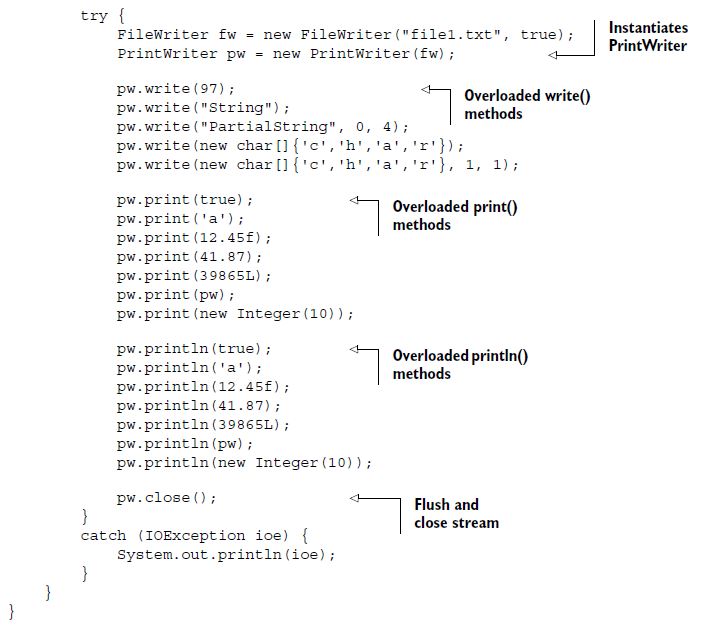
EXAM TIP The exam might test you on how to instantiate buffered character streams correctly. To instantiate BufferedReader, you must pass it an object of Reader. To instantiate BufferedWriter, you must pass it an object of Writer.



Class BufferedReader buffers data on the first read, and the subsequent request to the read() methods returns data from the buffer. But this isn’t the case with class FileReader.

***PrintWriter I/O with character streams***

Class PrintWriter can be used to print (write) formatted representations of objects to a file. This essentially means that you can use all the overloaded print methods that you’ve been using (via the class variable System.out) to write data to a file.



The overloaded versions of methods print() and println()are convenient methods to print (or write) data of primitive types and objects.

You can also instantiate PrintWriter by passing it a Writer instance and a boolean value specifying auto-flushing.

PrintWriter(File file)

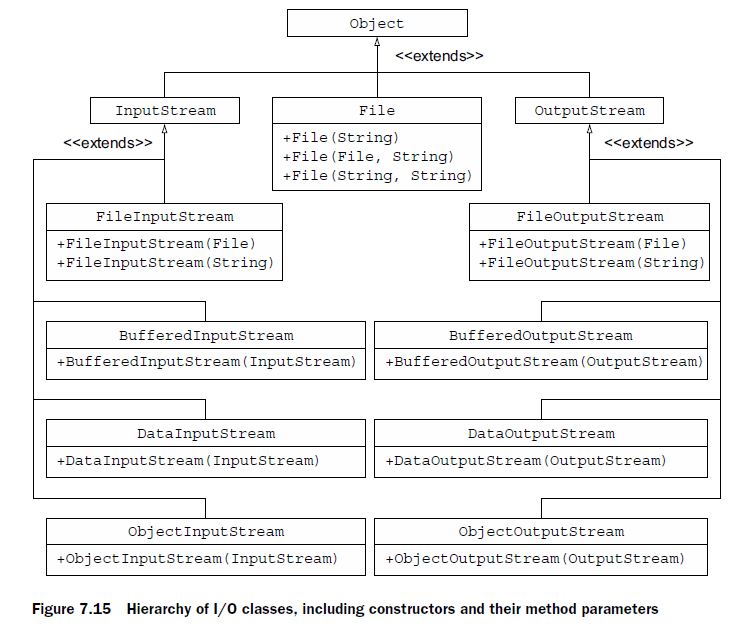
PrintWriter(File file, String charset)

PrintWriter(String fileName)

PrintWriter(String fileName, String charset)

PrintWriter(Writer out, boolean autoFlush)

PrintWriter(OutputStream out)



EXAM TIP If no console device is available, System.console() returns null. A null value signals that either the program was launched in a non interactive environment or perhaps the underlying operating system doesn’t support the console operations.

Console console = System.console();

*8. Java file I/O (NIO.2)*

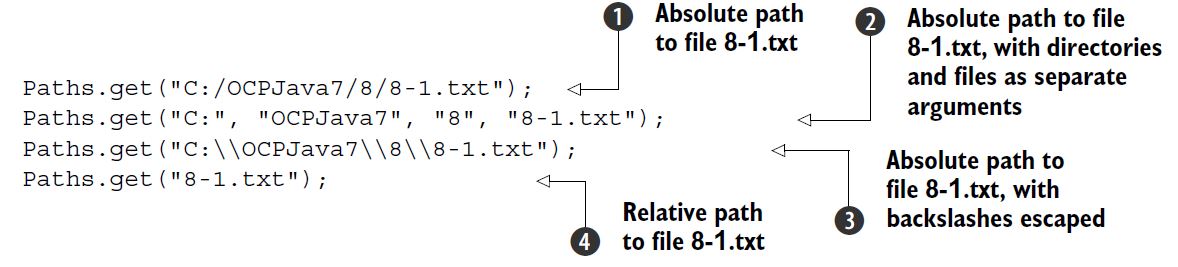
***Path objects***

NOTE Prior to Java 7, class java.io.File was used to represent the path of a file or directory in a file system. But it had several drawbacks. Its methods didn’t throw exceptions when they failed, which was essential to determine the cause of failure. Most of the methods of class File didn’t scale. For example, a request to a large directory listing could make a system hang. It didn’t support much access to the metadata. These reasons and more were responsible for the introduction of Path in Java 7.

EXAM TIP Because a Path object might not be tied to a real file or directory on a system, it can refer to a nonexistent file or directory.

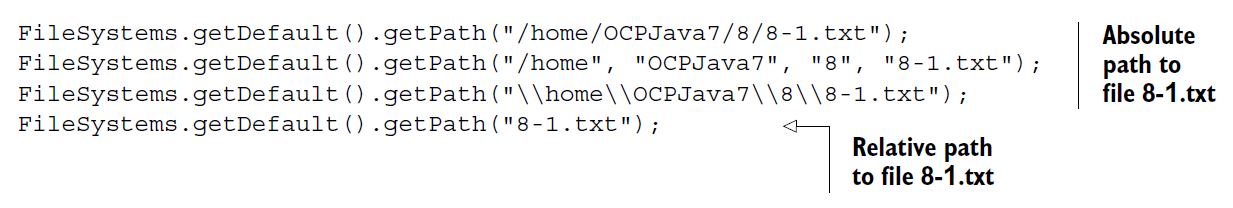
You can create Path objects by using methods from multiple classes: java.nio.file.Paths, java.nio.file.FileSystem, and java.io.File.

1. Creating Path objects by using get()method of Paths class.



When the path is provided as one string, where individual sub paths are separated by a path separator, only the last method argument is considered as a filename. All the others are assumed to be directory names.

1. Creating Path objects by using getPath()method of FileSystem class. Because class FileSystem is an abstract class, you can get a reference to the current class FileSystem object by calling getDefault() on class FileSystems



A Path object can refer to a nonexistent file or directory. Watch out for this point on the exam. Even though you didn’t create the file 8-1.txt until this point, a Path object that refers to it is valid.

EXAM TIP A Path object can refer to a nonexistent file or directory.

1. Creating Path objects by using toPath()method of File class.

Prior to Java 7, objects of class java.io.File were used to represent the file and directory paths. Starting with Java 7, a new method, toPath(), was added to class File to bridge the gap between the existing I/O classes and NIO classes. You can create a Path object by using a File instance:

File file = new File(“Hello.txt”);

Path path = file.toPath();

What happens if you create a Path object as follows?

Path path = Paths.get(“”);

Created using a zero-length string value, the preceding path variable refers to the current directory. Though path.toString() returns a zero-length string value, path.getAbsolutePath() would return its absolute path.

NOTE Behind the scenes, both Paths.get() and File.toPath() call FileSystems.getDefault().getPath().

PATH vs PATHS

Path is an *interface* and Paths is a utility *class*. The Path interface extends interfaces Comparable, Iterable, and Watchable.

EXAM TIP Most of the Path methods perform syntactic operations. They manipulate the paths to a file or directory without accessing the file systems. They’re logical operations on paths in memory.

Path path = FileSystems.getDefault().getPath("c:\\users\\obj8\\8-1.txt");

System.out.println("toString()-> " + path.toString());

System.out.println("getRoot()-> " + path.getRoot());

System.out.println("getName(0)-> " + path.getName(0));

System.out.println("getName(1)-> " + path.getName(1));

System.out.println("getFileName()-> " + path.getFileName());

System.out.println("getNameCount()-> " + path.getNameCount());

System.out.println("getParent()-> " + path.getParent());

System.out.println("subpath(0,2)-> " + path.subpath(0,2));

**Output:**

toString()-> c:\users\obj8\8-1.txt

getRoot()-> c:\

getName(0)-> users

getName(1)-> obj8

getFileName()-> 8-1.txt

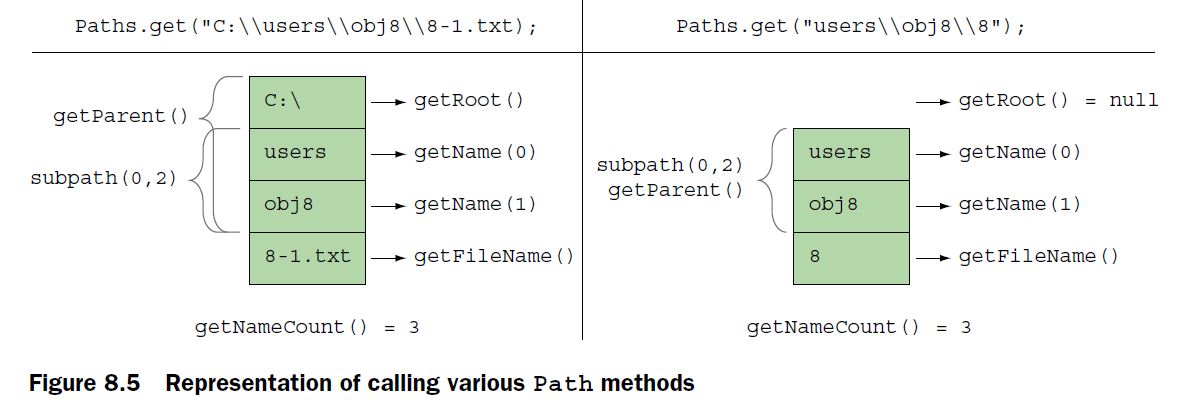
getNameCount()-> 3

getParent()-> c:\users\obj8

subpath(0,2)-> users\obj8

Note how the root of a path is not used in all the Path methods. Though it’s used by a method like getRoot(), it’s ignored by other methods like subpath() and getName().

EXAM TIP Methods getName(), getNameCount(), and subpath() don’t use the root directory of a path. Method getRoot() returns the root of an absolute path and null for relative paths. Play around with these methods—you might see them on the exam.



The Path methods that accept positions throw an IllegalArgumentException at runtime for invalid positions. For example, getName() and subpath() throw an IllegalArgumentException if you pass invalid path positions to them:

Path path = FileSystems.getDefault().getPath("c:\\users\\obj8\\8-1.txt");

System.out.println("subpath(0,4)-> " + path.subpath(0,4));

The code path.subpath(0,4) will throw an IllegalArgumentException because it refers to an invalid position in the value referred by path.

EXAM TIP Methods startsWith() and endsWith() are overloaded: startsWith(String), startsWith(Path), endsWith(String), and endsWith(Path). So if you pass null to these methods, you’ll get a compiler error.

*Converting relative paths to absolute paths*

Let’s assume that your current working directory is E:/OCPJavaSE7/FileNIO. Now assume you enter the value of the sub objective as 8-1. The following will create an absolute path to 8-1.txt in your current working directory, that is, E:\OCPJavaSE7\FileNIO\8-1.txt:

Path file = Paths.get("8-1.txt");

Path path = file.toAbsolutePath();

Imagine that you want to create a text file 8-1.txt in the parent directory of your current working directory. Knowing that you can use .. to denote your parent directory, do you think the following will help?

Path file = Paths.get("..\\8-1.txt");

Path path = file.toAbsolutePath();

System.out.println(path);//**E:\OCPJavaSE7\FileNIO\..\8-1.txt**

Yes, it will.

EXAM TIP Note that the method name to retrieve the absolute path from a Path object is toAbsolutePath() and not getAbsolutePath(). These method names are similar and might be used on the exam.

In the preceding code, path will refer to E:\OCPJavaSE7\FileNIO\..\8-1.txt. As you can see, inclusion of directories File NIO and .. is redundant in the preceding path. You can remove these redundant values by calling method normalize() on Path:

Path file = Paths.get("..\\8-1.txt");

Path path = file.toAbsolutePath();

path = path.normalize();

System.out.println(path);// **E:\OCPJavaSE7\8-1.txt**

EXAM TIP Path is immutable and calling normalize() on a Path object doesn’t change its value.

Though implicit, it’s common to use a period (.) to denote the current directory. For example, if you refer to file 8-1.txt, you refer to this path in the current directory. But it’s common for programmers to refer to this path as ./8-1.txt. Again, when you include a period in a Path object, you’re including redundant information, which can be removed too using method normalize(). The following code assumes your current working directory is E:/OCPJavaSE7/FileNIO:

Path file = Paths.get(".\\8-1.txt");

Path path = file.toAbsolutePath();

System.out.println(path);// **E:\OCPJavaSE7\FileNIO\.\8-1.txt**

path = path.normalize();

System.out.println(path);// **E:\OCPJavaSE7\FileNIO\8-1.txt**

* Do you think, when a Path object includes redundancies like . or .., that calling information retrieval methods like subpath() or getName() will also include these redundancies in the returned values?

Yes.

***Resolving paths using methods resolve and resolveSibling***

The overloaded methods resolve(String) and resolve(Path) are used to join a relative path to another path. If you pass an absolute path as a parameter, this method returns the absolute path:

Path path = Paths.get("/mydir/code");

System.out.println(path.resolve(Paths.get("world/Hello.java")));

// **/mydir/code/world/Hello.java**

System.out.println(path.resolve(Paths.get("/world/Hello.java")));

// **/world/Hello.java**

System.out.println(path.resolve("/world/Hello.java"));

// **/world/Hello.java**

System.out.println(path.resolve("world/Hello.java"));

// **/mydir/code/world/Hello.java**

Path absolutePath = Paths.get("E:/OCPJavaSE7/");

System.out.println(absolutePath.resolve(“/muni/Demo.java”));

// **E:/muni/Demo.java**

System.out.println(absolutePath.resolve(“muni/Demo.java”));

// **E:/OCPJavaSE7/muni/Demo.java**

Path path = Paths.get("/mydir/eWorld.java");

System.out.println(path.resolveSibling(Paths.get("newWorld.java")));

// **/mydir/newWorld.java**

System.out.println(path.resolveSibling("backup/eWorld.java"));

// **/mydir/backup/eWorld.java**

EXAM TIP Methods resolve() and resolveSibling() don’t check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.

EXAM TIP You can’t create a path from a relative path to an absolute path and vice versa using method relativize(). If you do so, you’ll get a runtime exception (IllegalArgumentException). Also, method relativize() doesn’t check the actual file system to verify if the file (or directory) the resulting path is referring to actually exists.

Path dir = Paths.get("/code");

Path dirC = Paths.get("C:/code/MyClass.java");

Path dirD = Paths.get("D:/notes/summary.txt");

System.out.println(dir.relativize(dirD));// **Would throw runtime exception**

System.out.println(dirC.relativize(dirD));

***Class Files***

***Create files and directories***

EXAM TIP Specifying file or directory attributes is optional with methods createFile(), createDirectory(), and createDirectories(). All these methods declare to throw an IOException, which is a checked exception.

EXAM TIP In class Files, method createDirectories() can create both the target directory and multiple nonexistent parent directories. If the directory already exists simply it can’t create a directory, no exceptions are thrown. Methods createDirectory() and createFile() create a single directory and file respectively. They throw a FileAlreadyExistsException if a directory or file with the same name already exists.

You can check for the existence of a file or directory referred by a Path object using methods exists() and notExists() in class Files:

public static boolean exists(Path path, LinkOption... options)

public static boolean notExists(Path path, LinkOption... options)

EXAM TIP Watch out for questions that state that exists() and notExists() will never return the same boolean value for the same Path object. Both methods exists() and notExists() would return false if they can’t determine the existence of the target file or directory.

Class Files’s overloaded copy() method enables you to read from InputStream and write to a Path object, read from a Path object and write to OutputStream, and read from and write to Path objects:

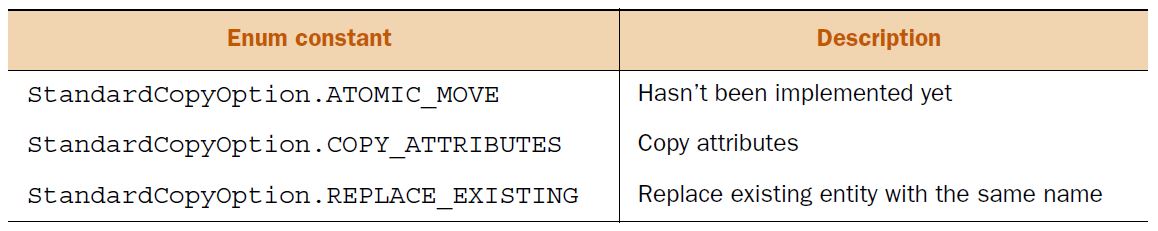
public static long copy(InputStream in, Path target, CopyOption... options)

public static long copy(Path source, OutputStream out)

public static Path copy(Path source, Path target, CopyOption... options)

EXAM TIP Files.copy() can copy only files, not directories. If the source is a directory, then in the target an empty directory is created (without copying the entries in the directory). This method returns a long or Path value, not a boolean value. Watch out for its invalid use in exam questions that use copy() to copy directories, use it in try-with-resources statements, or use its return value to test whether a file was copied or not.

Method copy() accepts objects of the CopyOption interface. You can use objects of enum StandardCopyOption, which implements this interface.



Path source = Paths.get(“c:\\test.txt”);

Path target = Paths.get(“c:\\demo.txt”);

Files.copy(source, target, StandardCopyOption.REPLACE\_EXISTING);

Method copy() in class Files doesn’t allow you to append data to an existing file. If you want to add the notes sent to you by your study group to your own text files, you need to open the target file in append mode and use I/O streams or readers/writers to do so.

EXAM TIP Method copy() in class Files doesn’t allow you to append data to an existing file; rather, it creates a new file or replaces an existing one.

To move files or directories programmatically, you can use Files.move(), which moves or renames a file to a target file:

public static Path move(Path source, Path target, CopyOption... options)

To rename a file notes.txt to copy-notes.txt, keeping the file in the same directory, you can use the following:

Path source = Paths.get("notes.txt");

Files.move(source, source.resolveSibling("copy-notes.txt"));

To move a file to a new directory, retaining the same filename and replacing any existing file of that name in the new directory, you can use the following:

Path source = Paths.get("notes.txt");

Path target = Paths.get("/home/myNotes/");

Files.move(source, target.resolve(source.getFileName()),

StandardCopyOption.REPLACE\_EXISTING);

EXAM TIP You can only move empty directories using method Files.move(). You can rename a nonempty directory by using Files.move(). But you can’t move a file or directory to a non-existing directory.

To delete a directory or a file referred to by a Path object, you can use the following methods from class Files:

public static void delete(Path path)

public static boolean deleteIfExists(Path path)

Both the preceding methods can delete a file or directory (if it’s empty). If you try to delete a directory that isn’t empty, these methods will throw a DirectoryNotEmptyException. If you try to delete a nonexistent file or directory using method delete(), it will throw a NoSuchFileException. But method deleteIfExists() won’t throw an exception if the file or directory at the specified path doesn’t exist—it will return false. The deletion operation might also fail if the target file is in use, because some operating systems don’t allow deletions of files or directories if they’re in use by an active program.

EXAM TIP Methods delete() and deleteIfExists() can be used to delete files and (nonempty) directories.

***Individual attributes***

Class Files defines static methods to access individual attributes of a file or directory referred by a Path, such as its **size**, when was it **last modified**, whether it’s **readable** or **writable**, and whether it’s a directory or a file. Following is an example to access some of the attributes of a java source file:

Path path = Paths.get("MyAttributes.java");

System.out.println("size:" + **Files.size(path)**);

System.out.println("isDirectory:" + **Files.isDirectory(path)**);

System.out.println("isExecutable:" + **Files.isExecutable(path));**

System.out.println("isHidden:" + **Files.isHidden(path)**);

System.out.println("isReadable:" + **Files.isReadable(path)**);

System.out.println("isSameFile:" + **Files.isSameFile(path, path)**);

System.out.println("isDirectory:" + **Files.isDirectory(path)**);

System.out.println("isSymbolicLink:" + **Files.isSymbolicLink(path)**);

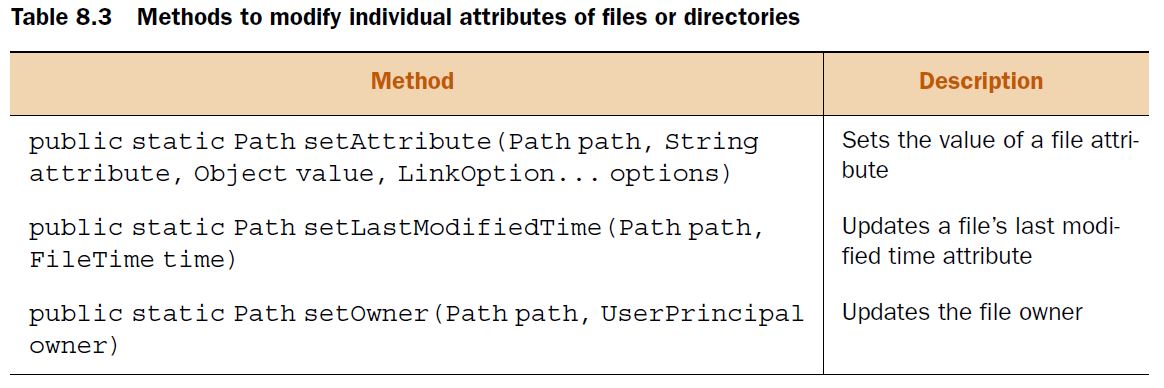
System.out.println("isWritable:" + **Files.isWritable(path)**);

System.out.println("getLastModifiedTime:" +

**Files.getLastModifiedTime(path)**);

System.out.println("getOwner:" + **Files.getOwner(path)**);

You can also access the individual attributes of a file or directory by using method Files.getAttribute(), passing to it the name of the attribute as a string value. To modify the attributes of an existing file or directory, you can use Files.setAttribute().



Path path = Paths.get("ModifyAttributes.java");

System.out.println("creationTime:" +

**Files.getAttribute(path, "creationTime"));**

FileTime newTime = FileTime.fromMillis(System.currentTimeMillis());

**Files.setAttribute(path, "creationTime", newTime);**

System.out.println("creationTime:" +

Files.getAttribute(path, "creationTime"));

EXAM TIP Methods Files.setAttribute() and Files.getAttribute() can be used to access a file or directory attribute and modify it (if allowed). The attribute name is passed to these methods as a string value.

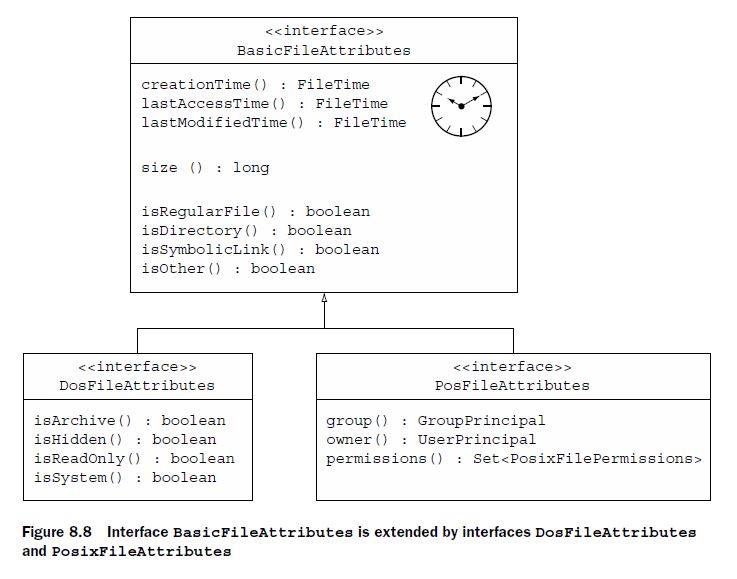
***Group of attributes***

Querying the file system multiple times to access all file or directory attributes can affect your application’s performance. To get around this, you can access a group of file attributes by calling Files.getFileAttributeView() or Files.readAttributes().

INTERFACES TO READ AND MODIFY ATTRIBUTE SETS

Different file systems might support different attribute sets. Java groups related attributes that correspond to a specific file system implementation like DOS or POSIX, or to a common functionality like file owner attributes. You can use multiple interfaces to access file and directory attributes and modify them. These groups are defined as interfaces, and the ones on the exam are as follows:

* BasicFileAttributes and BasicFileAttributeView The BasicFileAttributes interface defines methods to access the basic attributes that should be supported by all the file systems. The BasicFileAttributeView interface can be used to modify the basic attributes.
* DosFileAttributes and DosFileAttributeView The DosFileAttributes interface extends BasicFileAttributes and defines methods to access attributes specific to Windows files and directories. The DosFileAttributeView interface defines methods to modify the DOS file attributes.
* PosixFileAttributes and PosixFileAttributeView The PosixFileAttributes interface also extends BasicFileAttributes and defines methods to access attributes related to the POSIX family of standards, like Linux or UNIX. The PosixFileAttributeView interface defines methods to modify attributes related to the POSIX family.
* AclFileAttributeView Available only for Windows OS, this interface supports access and updates of a file’s access control list (ACL).
* FileOwnerAttributeView This interface supports access and updates to the owner of a file or directory. It’s supported by all systems that support the concept of file owners.
* UserDefinedFileAttributeView This interface supports the addition, modification, and deletion of user-defined metadata.



EXAM TIP

The BasicFileAttributes, DosFileAttributes, and PosixFileAttributes interfaces define methods to access attributes. They don’t define methods to modify (or set) the attributes. Use class Files or view interfaces to modify the attributes.

To access and update a group of attributes for a directory or file, you can access an attribute view by using method File.getFileAttributeView(). To only access (not update) an attribute group, you can use method File.readAttributes(). You can also call method readAttributes() on an attribute view to get its corresponding attribute set. Following is an example that uses these methods:

Path path = Paths.get("pathToaFile");

PosixFileAttributeView view = **Files.getFileAttributeView**(path,

PosixFileAttributeView.class);

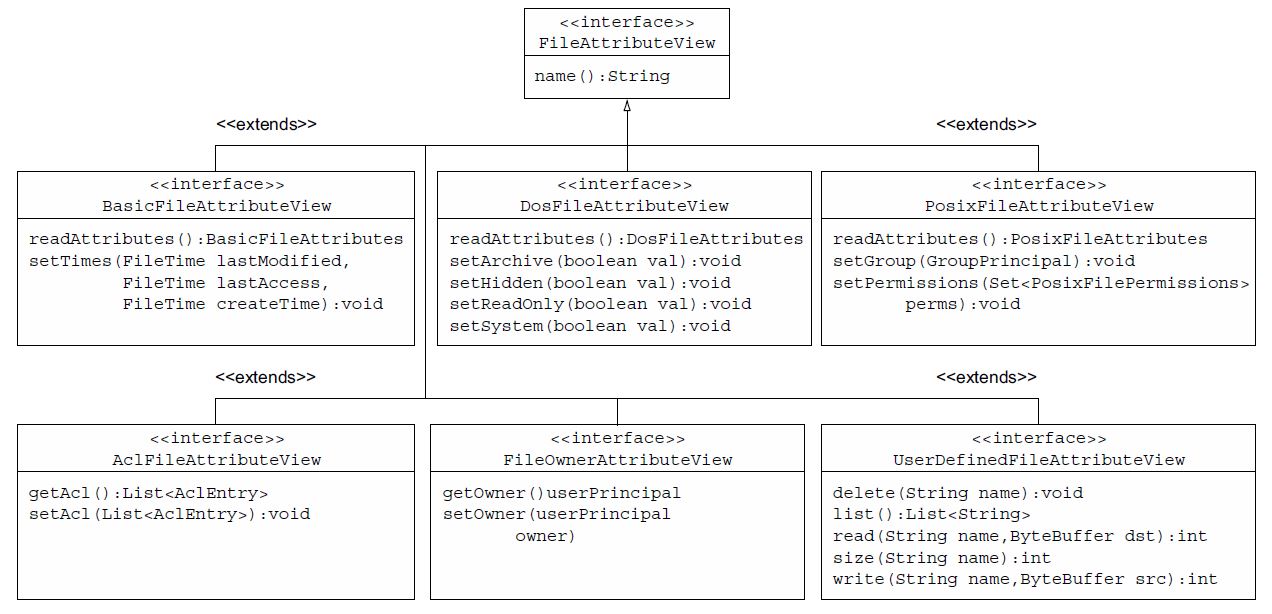
PosixFileAttributes attr = **view.readAttributes()**;

PosixFileAttributes attr2 = **Files.readAttributes**(path,

PosixFileAttributes.class);

EXAM TIP

If a file system doesn’t support an attribute view, Files.getFileAttributeView() returns null. If a file system doesn’t support an attribute set, File.readAttributes() will throw a runtime exception.



***Basic attributes***

Imagine you need to delete all files in a directory, whose creation time is older than one day, using your Java code. If you can access the creation time of a file, you can determine if the file needs to be deleted or not. Here’s an example:

Path file = Paths.get(fileName);

**BasicFileAttributes attr = Files.readAttributes(file,BasicFileAttributes.class);**

FileTime fileCreationTime = attr.creationTime();

long currentTime = System.currentTimeMillis();

FileTime dayOldFileTime = FileTime.fromMillis(currentTime - (24\*60\*60\*1000));

if (fileCreationTime.compareTo(dayOldFileTime) < 0)

Files.delete(file);

EXAM TIP If an underlying system doesn’t support all the basic timestamps that is, creationTime, lastAccessTime, and lastModifiedTime it might return system-specific information.

**Map<String,Object> values = Files.readAttributes(file, "\*");**

for (String attribute:values.keySet()) {

System.out.println(attribute + " : " + values.get(attribute));

}

FileTime newTime = FileTime.fromMillis(System.currentTimeMillis());

**Files.setAttribute(file, "lastModifiedTime", newTime);**

You can also use a comma-delimited list of values:

Map<String,Object> values = Files.readAttributes(file,

"lastModifiedTime,isDirectory");

EXAM TIP Methods Files.setAttribute() and Files.getAttribute()throw an IllegalArgumentException or UnsupportedOperationException if you pass them an invalid or unsupported attribute.

***DOS attributes***

As shown in figure 8.9, the DosFileAttributes interface makes the following attributes available:

* archive
* hidden
* readonly
* system

EXAM TIP The DOS attributes are available on a Windows system only. Trying to access them on other systems will throw a runtime exception.

DosFileAttributeView dosView = Files.getFileAttributeView(path,

DosFileAttributeView.class);

DosFileAttributes dosAttrs = dosView.readAttributes();

if (dosAttrs.isReadOnly()) {

dosView.setHidden(true);

dosView.setArchive(false);

dosView.setReadOnly(false);

dosView.setSystem(true);

}

else

System.out.println("Don't modify the attributes");

You can also access file or directory attributes by using class Files. The following code reads DOS attributes:

Map<String,Object> values = Files.readAttributes(file,

"dos:archive,hidden");

Map<String,Object> values2 = Files.readAttributes(file, "dos:\*");

DosFileAttributes attr = Files.readAttributes(file,

DosFileAttributes.class);

EXAM TIP When you read *all* DOS attributes using method Files.readAttributes(), you also read the basic attributes.

To modify a DOS attribute, you must prefix the attribute name with dos: because an attribute is implicitly prefixed with basic:

Files.setAttribute(file, "dos:hidden", true);

EXAM TIP When you read or write an invalid value to a file attribute, the code throws the runtime exception ClassCastException.

***POSIX attributes***

The POSIX attributes are as follows:

* group
* owner
* permissions

EXAM TIP The POSIX attributes are available on the POSIX family of standards, like UNIX, LINUX, etc. Trying to access them on other systems will throw a runtime exception.

PosixFileAttributeView posixView = Files.getFileAttributeView(file,

PosixFileAttributeView.class);

PosixFileAttributes posixAttrs = posixView.readAttributes();

if (posixAttrs.owner().getName().equals("admin"))

posixView.setPermissions(PosixFilePermissions.fromString("rwxrwxrwx"));

else

posixView.setPermissions(PosixFilePermissions.fromString("rwxr-x---"));

You can also use class Files to read all POSIX file attributes:

Map<String,Object> values = Files.readAttributes(file, "posix:\*");

PosixFileAttributes attr = Files.readAttributes(file,

PosixFileAttributes.class);

***AclFileAttributeView interface***

The AclFileAttributeView interface supports reading and updating a file’s ACL or file owner attributes. It defines methods getAcl() and setAcl(). This view is available only for Windows systems.

***FileOwnerAttributeView interface***

The FileOwnerAttributeView interface is supported by all file systems with a file owner concept, and this view includes methods to access and update the owner of a file or directory. If defines methods getOwner() and setOwner(UserPrincipal).

EXAM TIP To read or update the owner of a file or directory you can use the AclFileAttributeView, FileOwnerAttributeView, and PosixFileAttributeView interfaces.

***UserDefinedAttributeView interface***

The UserDefinedAttributeView interface can be used to add, delete, access, and modify additional user-defined attributes to a file or directory. It defines methods delete (String), list(), read(String, ByteBuffer), size(String), and write(String, ByteBuffer) to, respectively, delete, list, read, get the attribute’s size, and write attribute values.

***Recursively access a directory tree***

***FileVisitor interface***

You can use the FileVisitor, a generic interface to define the code that you want to execute during the traversal of a directory structure. When you traverse a directory structure, you can define what to do before or after you visit a directory, when you visit a file, or when access to a file is denied.

Let’s create a class, say MyFileVisitor, which implements the FileVisitor interface and override the below methods and define logic.

preVisitDirectory(Path dir, BasicFileAttributes attrs)

postVisitDirectory(Path dir, IOException exc)

visitFile(Path file, BasicFileAttributes attrs)

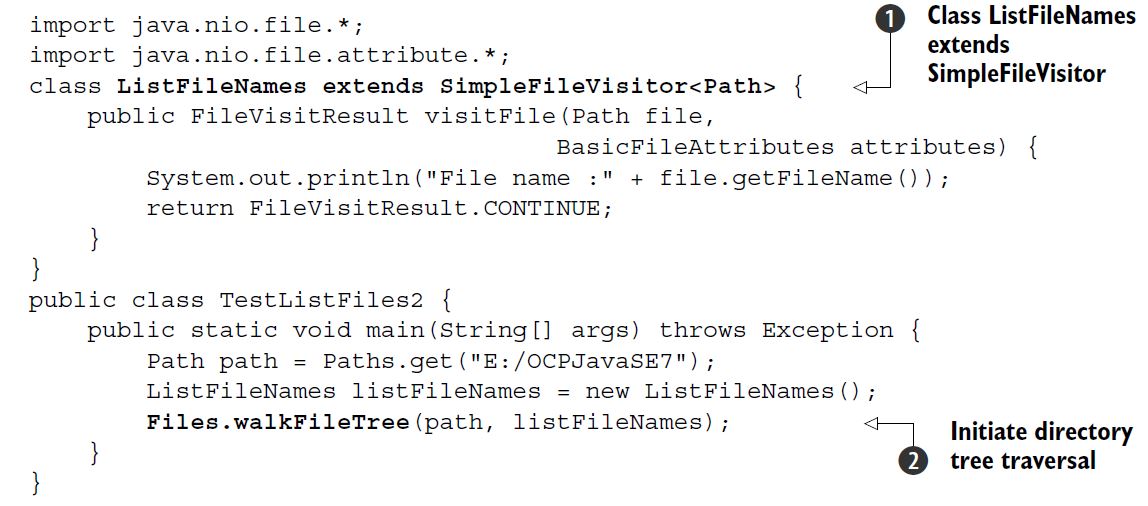
Instead of defining you own class and implement this method we can use predefined class SimpleFileVisitor.

***SimpleFileVisitor***

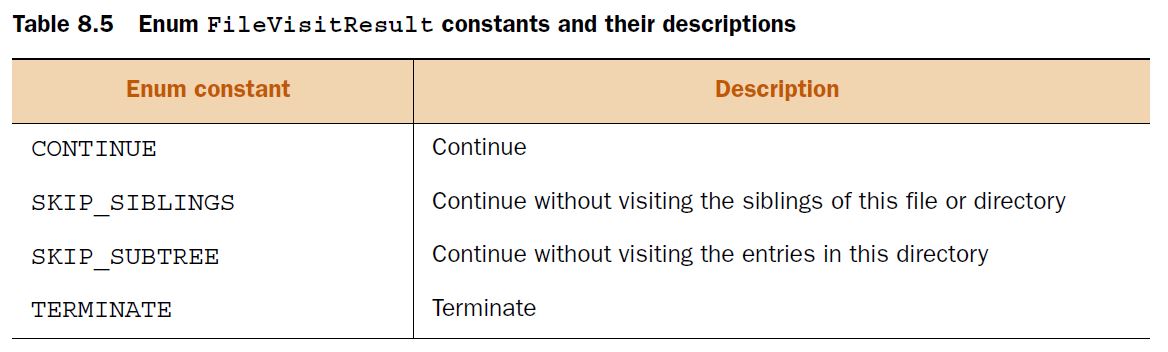
It implements the FileVisitor interface. You can extend this class to implement methods for only the required behavior.

Imagine you need to recursively traverse a directory structure and print only the names of all the files. In this case, you can extend the SimpleFileVisitor class and override only the visitFile() method.

You can initiate traversal of a directory by calling the overloaded method walkFileTree() from class Files:



EXAM TIP A directory tree is traversed depth-first. But the order in which the subdirectories are traversed is unpredictable.



***DirectoryStream interface***

The DirectoryStream interface can be used to iterate over all the files and directories in a directory. You can use an Iterator or for-each construct to iterate over a directory. The order in which the directory contents are iterated is unpredictable.

Path dir = Paths.get("E:/OCPJavaSE7");

try (**DirectoryStream<Path> stream=Files.newDirectoryStream(dir)**) {

for (Path value : stream) {

System.out.println(value + ":" + Files.isDirectory(value));

}

}

What happens if you try to iterate a file (and not a directory) using DirectoryStream? In this case you’ll get a runtime exception (NotDirectoryException).

EXAM TIP

If you pass Path to a file (and not a directory) to Files.newDirectoryStream(), it will throw a runtime exception. The order of iteration of files and directories in a specified directory using DirectoryStream is unpredictable.

The next example uses an Iterator to iterate over the files and directories of a directory using method Files.DirectoryStream(Path dir, String glob).

Path dir = Paths.get("E:/OCPJavaSE7/FileNIO");

try (**DirectoryStream<Path> stream = Files.newDirectoryStream(**

dir, "\*.{txt,java}")) {

Iterator iterator = stream.iterator();

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

}

***Using PathMatcher***

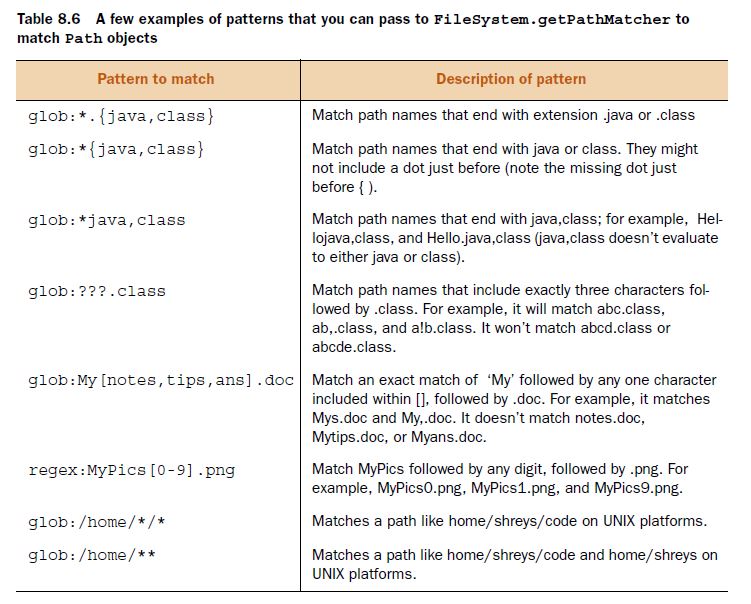
EXAM TIP In glob \* matches zero or more characters. In regex .\* matches zero or more characters.

To match a Path object with a pattern, you should create an object of java.nio .file.PathMatcher. PathMatcher is an interface with just one method: matches(). It returns true if a given path matches this matcher’s pattern:

boolean matches(Path path)

You can create a PathMatcher by calling FileSystem.getPathMatcher() and passing it the pattern to be matched:

public abstract PathMatcher getPathMatcher(String syntaxAndPattern)



PathMatcher matcher = FileSystems.getDefault().getPathMatcher

("regex:[1-9]\*[0-9]?-[1-9]?.txt");

Path file = Paths.get("12-1.txt");

if (**matcher.matches(file)**) {

System.out.println(file);

}

***Watch a directory for changes***

The first step to create a WatchService object, you can use the FileSystem class, which provides method newWatchService() to create a WatchService object:

WatchService watchService = FileSystems.getDefault().newWatchService();

The next step is to register directories with the WatchService object by using method register() of Path. Multiple events can be registered in the same method call. Each registration process returns a WatchKey.

The directories that need to be watched for additions, modifications, or deletions must be registered with a WatchService object. A WatchService object watches a directory for the following events:

* StandardWatchEventKinds.ENTRY\_CREATE—This event occurs when a new file or directory is created, moved, or renamed in the directory being watched.
* StandardWatchEventKinds.ENTRY\_DELETE—This event occurs when an existing file or directory is deleted, moved, or renamed in the directory being watched.
* StandardWatchEventKinds.ENTRY\_MODIFY—This event is platform-dependent. It usually occurs when contents of an existing file are modified. It can also occur if the attributes of a file or directory (in the directory being watched) are modified.
* StandardWatchEventKinds.OVERFLOW—This indicates that an event has been lost.

WatchService watchService = FileSystems.getDefault().newWatchService();

Path dir1 = Paths.get("E:/OCPJavaSE7");

Path dir2 = Paths.get("E:/OCPJavaSE7/8");

WatchKey regWatchKey = dir1.register(watchService,

StandardWatchEventKinds.ENTRY\_MODIFY,

StandardWatchEventKinds.ENTRY\_DELETE,

StandardWatchEventKinds.ENTRY\_CREATE);

dir2.register(watchService,

StandardWatchEventKinds.ENTRY\_MODIFY,

StandardWatchEventKinds.ENTRY\_DELETE,

StandardWatchEventKinds.ENTRY\_CREATE);

EXAM TIP You can watch a directory for changes. If you try to register a file for changes, you’ll get a runtime exception (NotDirectoryException). Registering a directory for any event (create, modify, or delete) doesn’t implicitly register its subdirectories.

getParent() returns null if a Path object doesn’t have a parent.

Files.copy(Path source, Path destination);

Files.copy(InputStream source, Path destination);

Files.copy(Path source, OutputStream destination);

The glob pattern /mydir/\*/\* evaluates to root (/), followed by dir mydir, followed by any two subdirectories. Only option (mydir/notes/java) matches this pattern.

*9. Building database applications with JDBC*

*10. Threads*

*11. Concurrency*

*12. Localization*