*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:

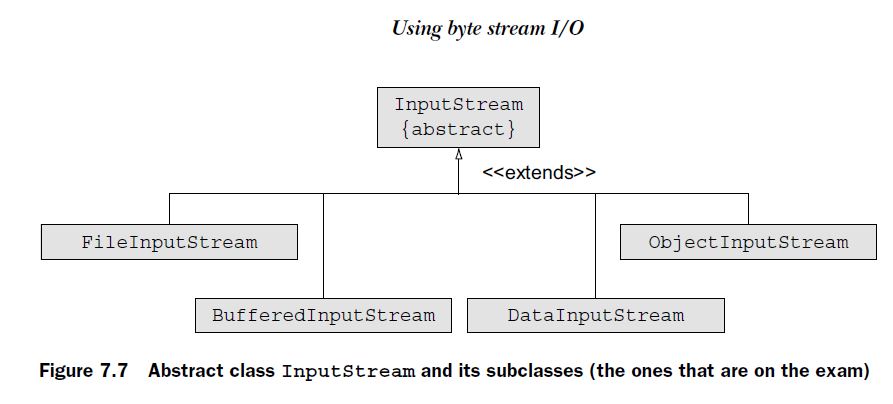
int abstract read()

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:

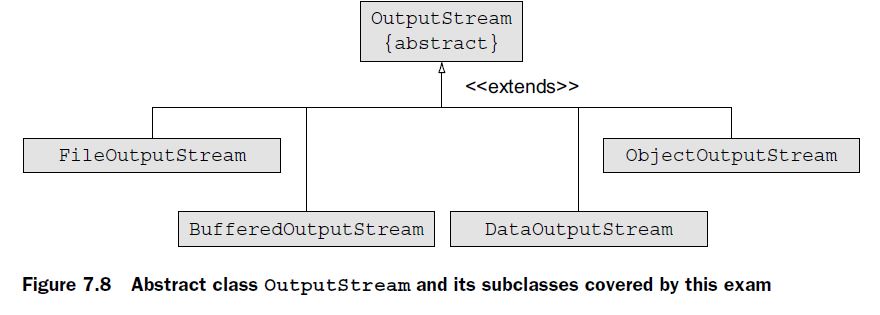
abstract void write(int b)

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.

FileInputStream(File file) throws FileNotFoundException {}

FileInputStream(String name) throws FileNotFoundException {}

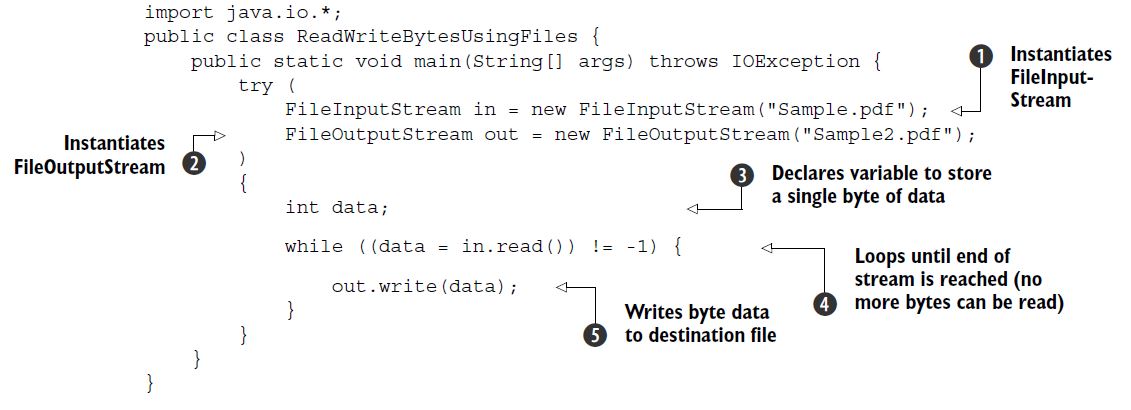
FileOutputStream(File file) throws FileNotFoundException

FileOutputStream(File file, boolean append) throws FileNotFoundException

FileOutputStream(String name) throws FileNotFoundException

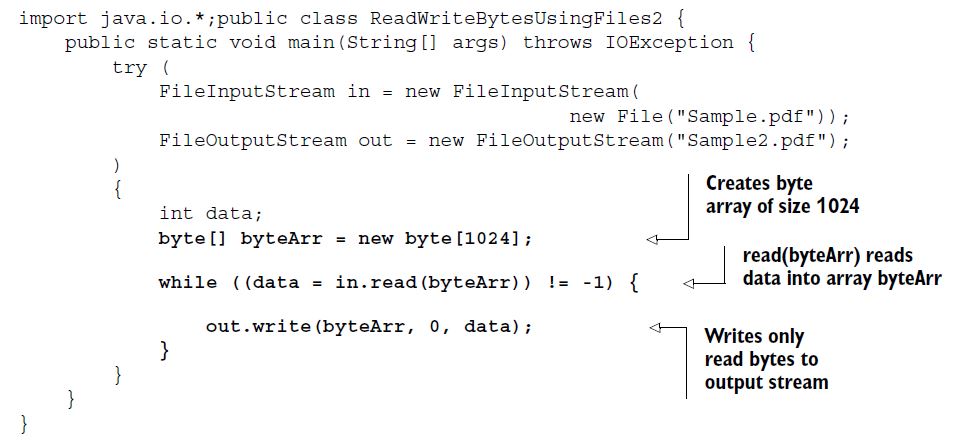
FileOutputStream(String nm, boolean append) throws FileNotFoundException

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:

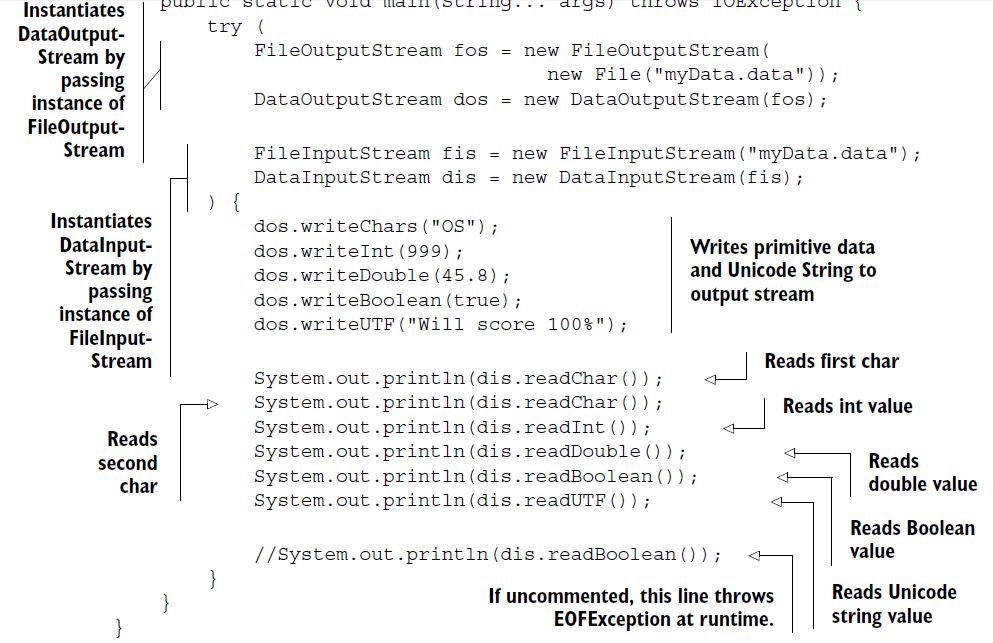
You can use FileInputStream and FileOutputStream to read and write *only* byte data they didn’t define methods to work with any other specific primitive data types or objects, which is what you might need most of the time.

***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***

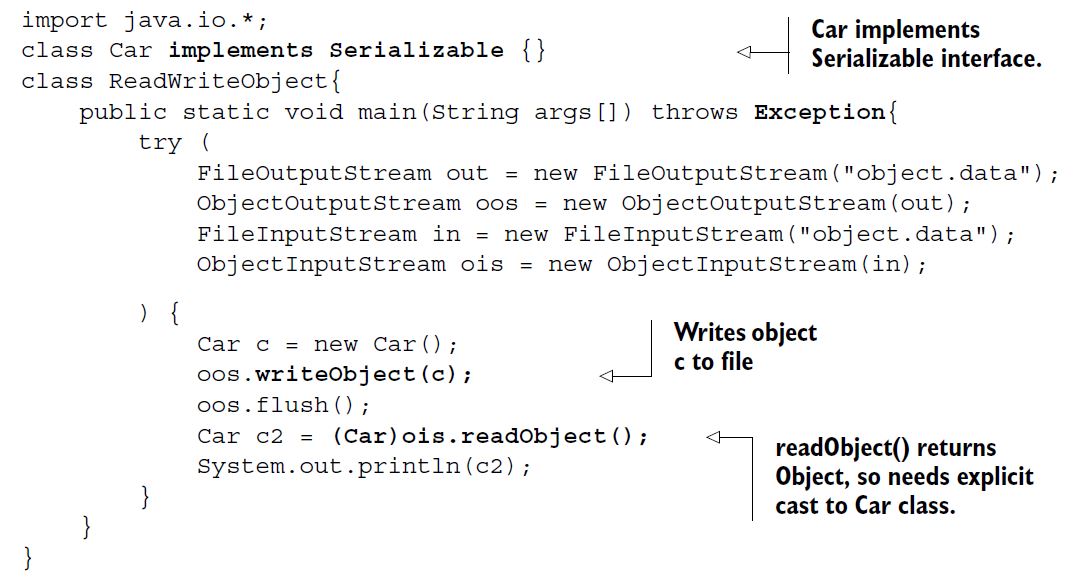
You can use classes ObjectInputStream and ObjectOutputStream to read and write objects *and* primitive values.

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**

}

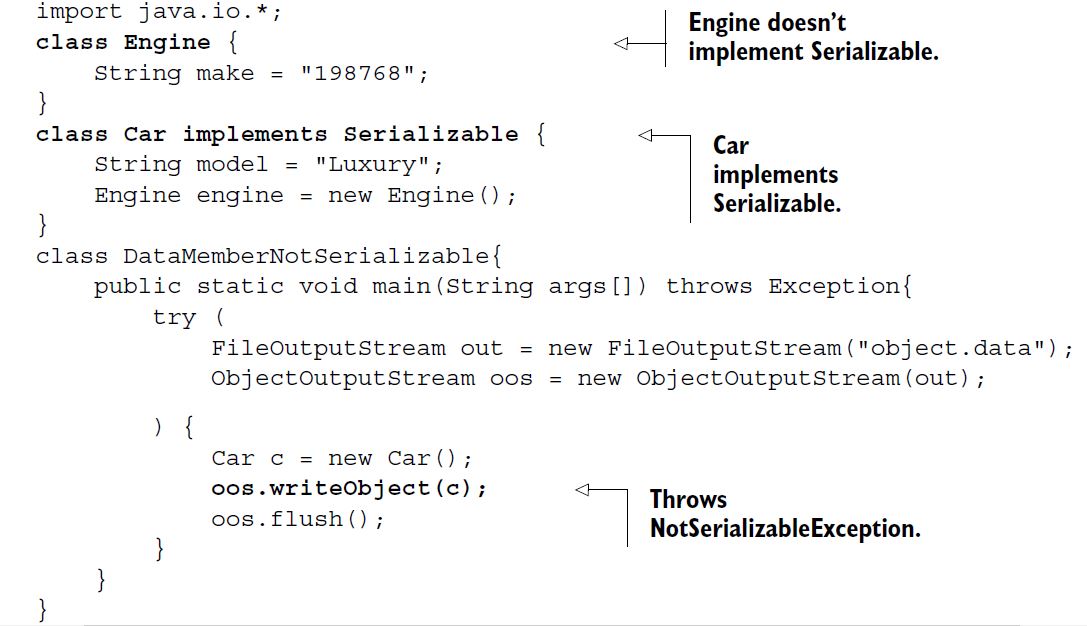
}

}

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***

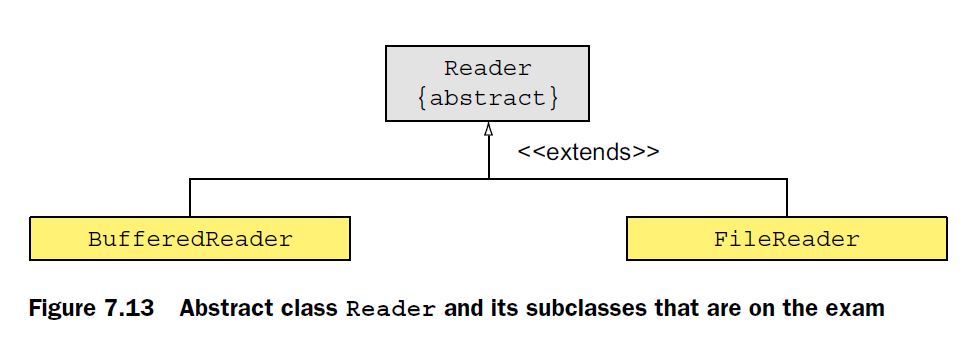
Class Reader defines overloaded read() methods to read character data from an underlying data stream:

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:

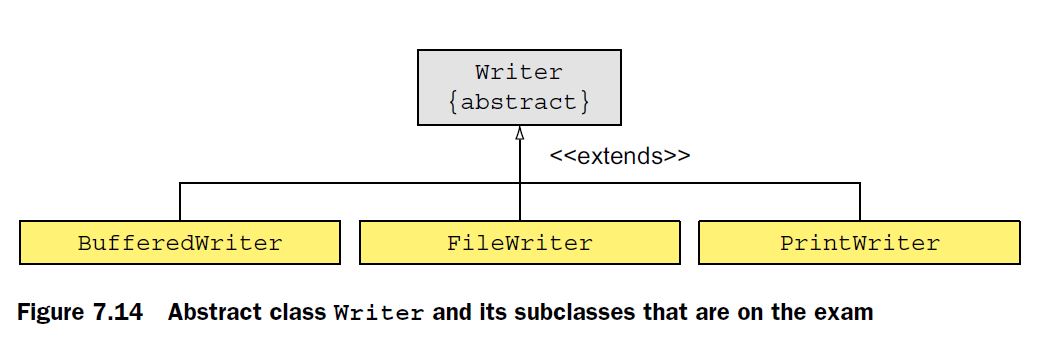
void write(char[] cbuf)

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***

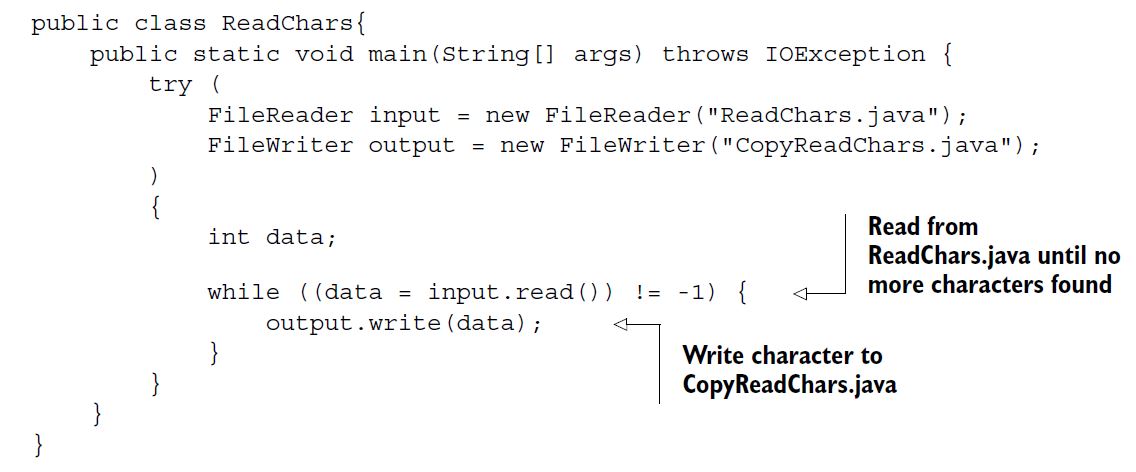
FileReader and FileWriter are convenience classes for reading and writing character data from files.

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:

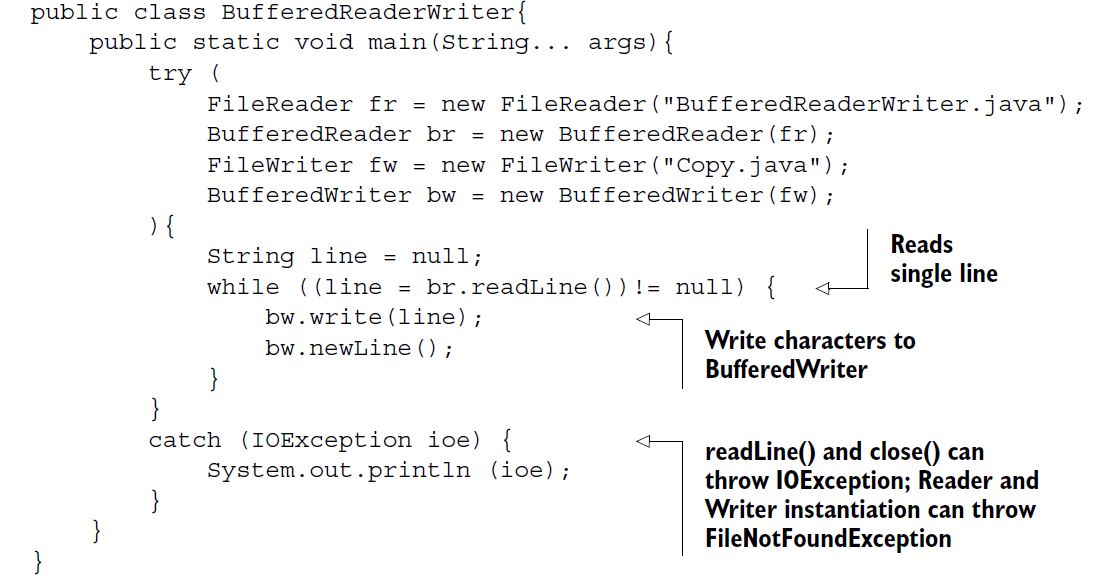
public BufferedReader(Reader in)

public BufferedReader(Reader in, int sz)

public BufferedWriter(Writer out)

public BufferedWriter(Writer out, int sz)

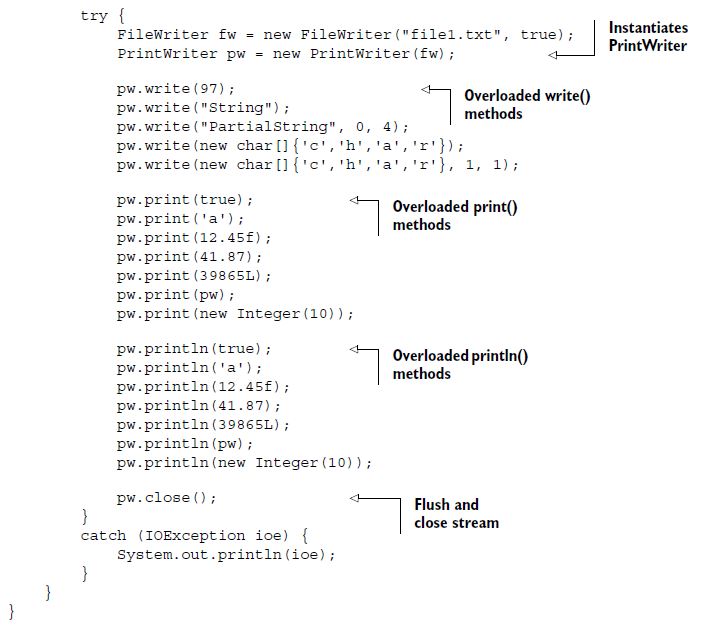
EXAM TIP The exam might test you on how to instantiate buffered character streams correctly. To instantiate BufferedReader, you must pass it an object of Reader. To instantiate BufferedWriter, you must pass it an object of Writer.



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)

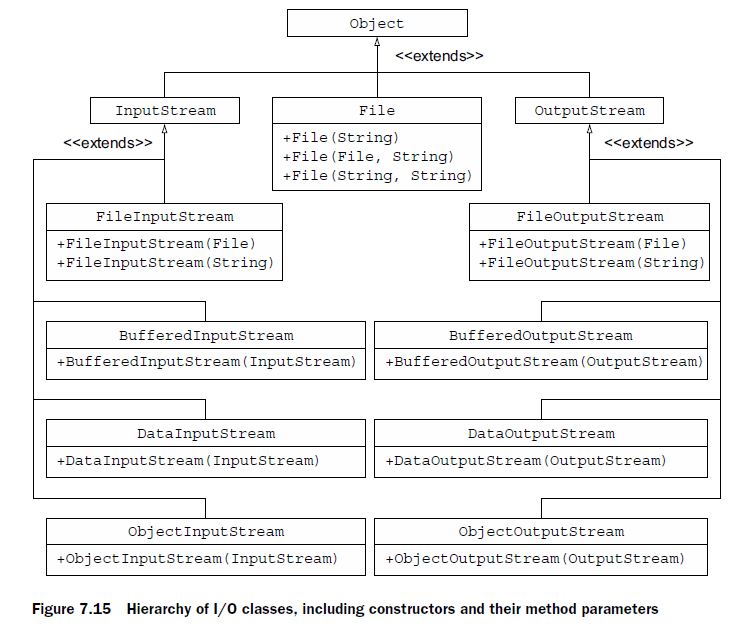
PrintWriter(File file, String charset)

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)*

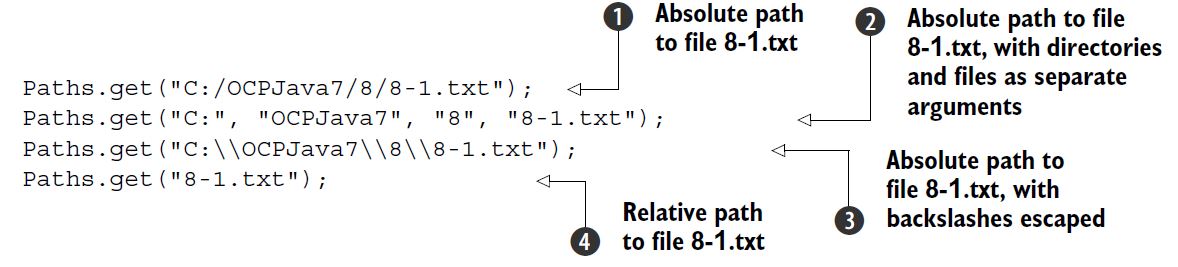
***Path objects***

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.

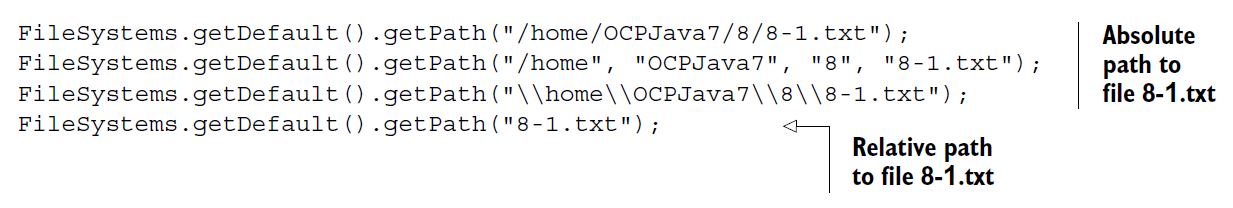
You can create Path objects by using methods from multiple classes: java.nio.file.Paths, java.nio.file.FileSystem, and java.io.File.

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

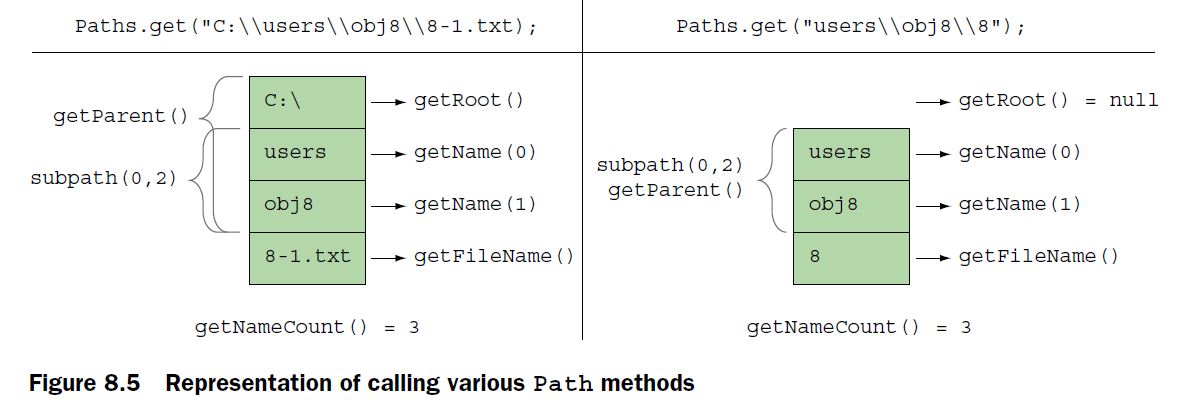
getNameCount()-> 3

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.

Path baseDir = Paths.*get*("uploads/c");

Path filename = Paths.*get*("uploads/image/test.png");

System.***out***.println(baseDir.relativize(filename));//..\image\test.png

Path baseDir = Paths.*get*("uploads/c");

Path filename = Paths.*get*("uploads/image/test.png");

System.***out***.println(filename.relativize(baseDir));//..\..\c

Path baseDir = Paths.*get*("uploads");

Path filename = Paths.*get*("uploads/image/test.png");

System.***out***.println(baseDir.relativize(filename));//image\test.png

Path baseDir = Paths.*get*("uploads");

Path filename = Paths.*get*("uploads/image/test.png");

System.***out***.println(filename.relativize(baseDir));//..\..

Path baseDir = Paths.*get*("MyDir/hello.java");

Path filename = Paths.*get*("FriendDir/code");

System.***out***.println(baseDir.relativize(filename));//..\..\FriendDir\code

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:

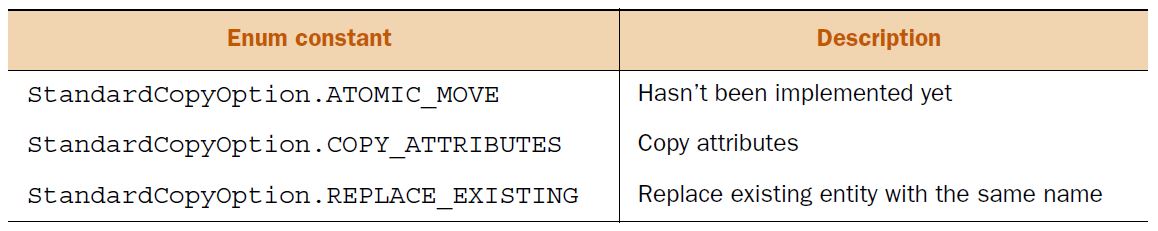
public static long copy(InputStream in, Path target, CopyOption... options)

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)**);

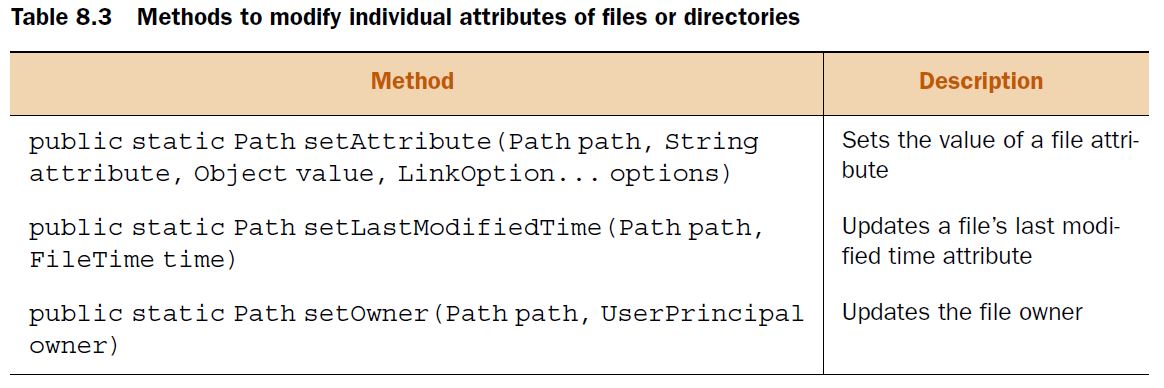
System.out.println("isWritable:" + **Files.isWritable(path)**);

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.

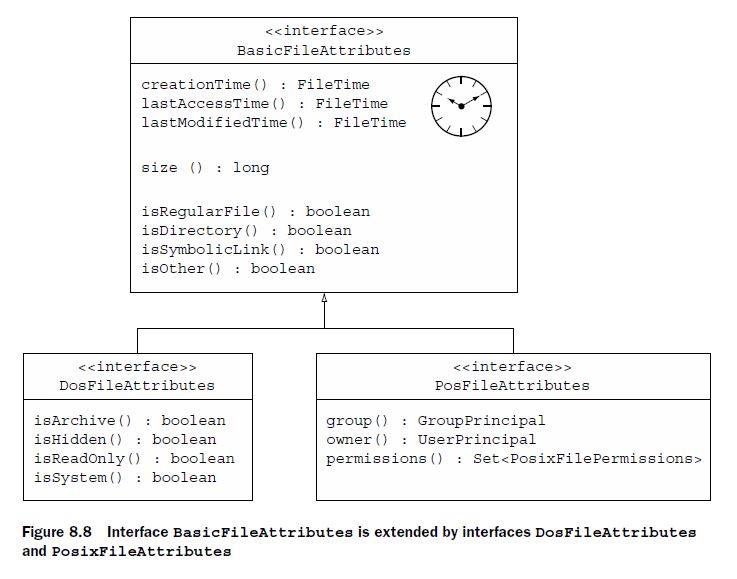
***Group of attributes***

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);

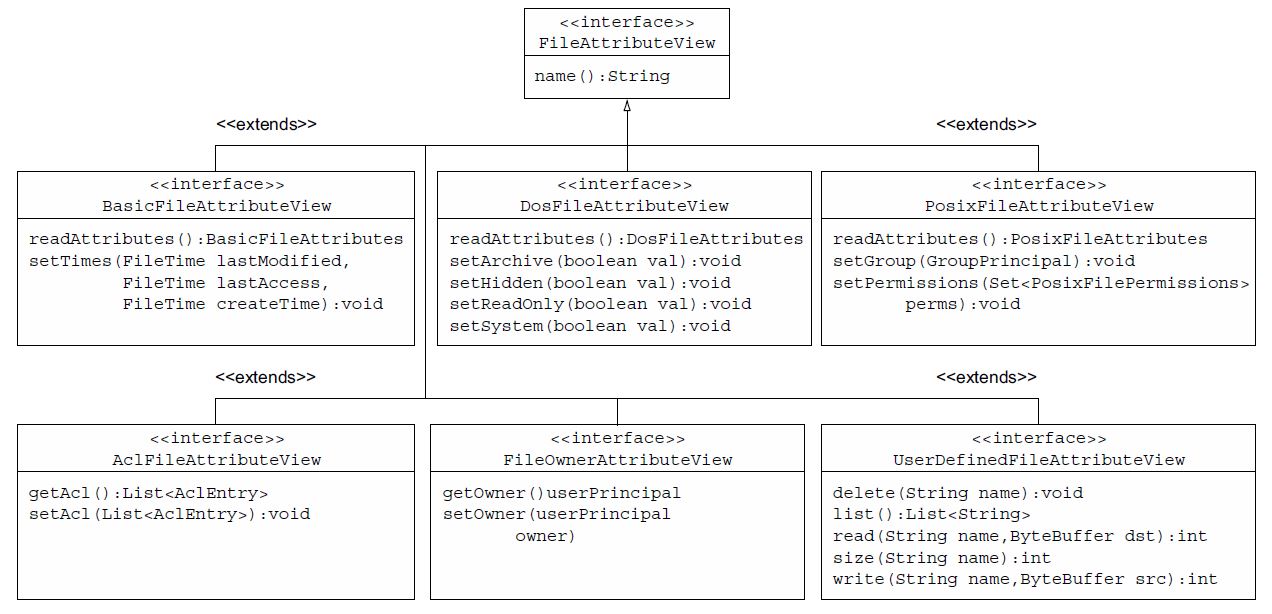
PosixFileAttributes attr = **view.readAttributes()**;

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(path, "\*");**

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)

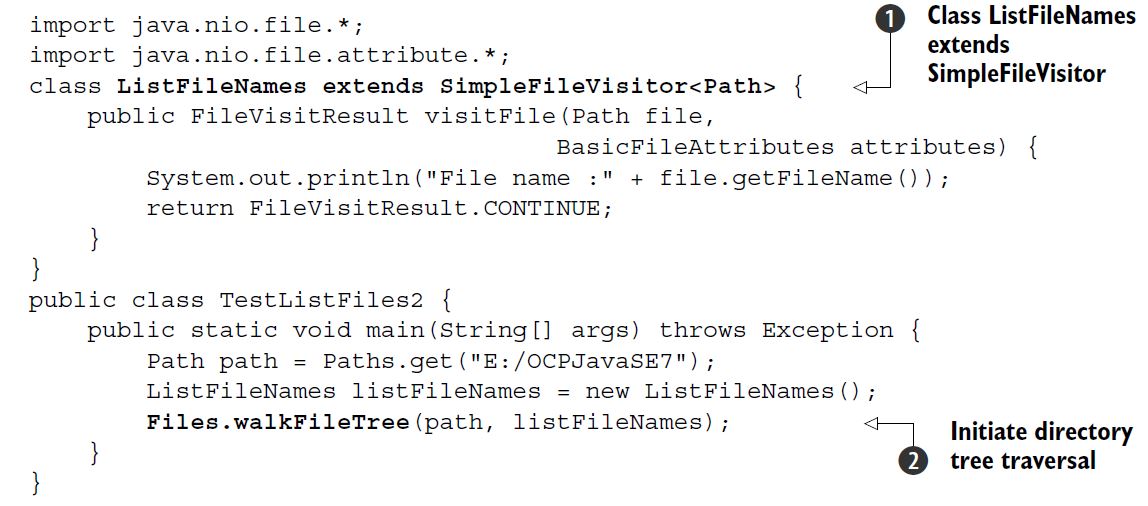
Instead of defining you own class and implement this method we can use predefined class SimpleFileVisitor.

***SimpleFileVisitor***

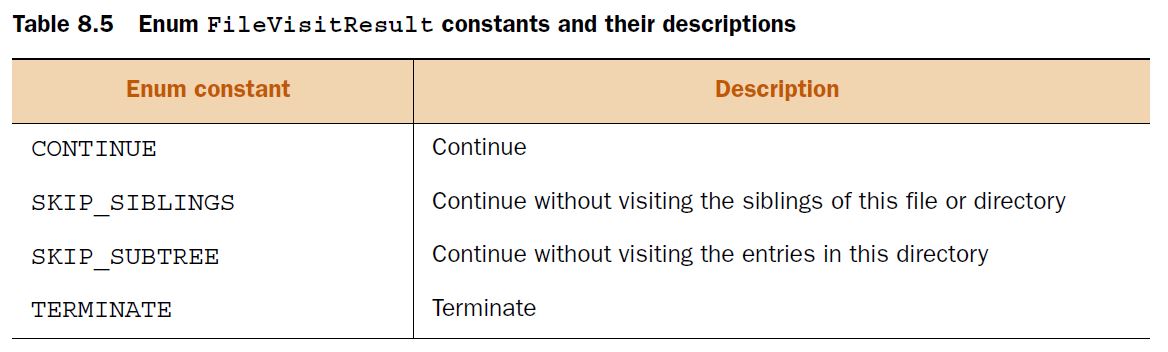
It implements the FileVisitor interface. You can extend this class to implement methods for only the required behavior.

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***

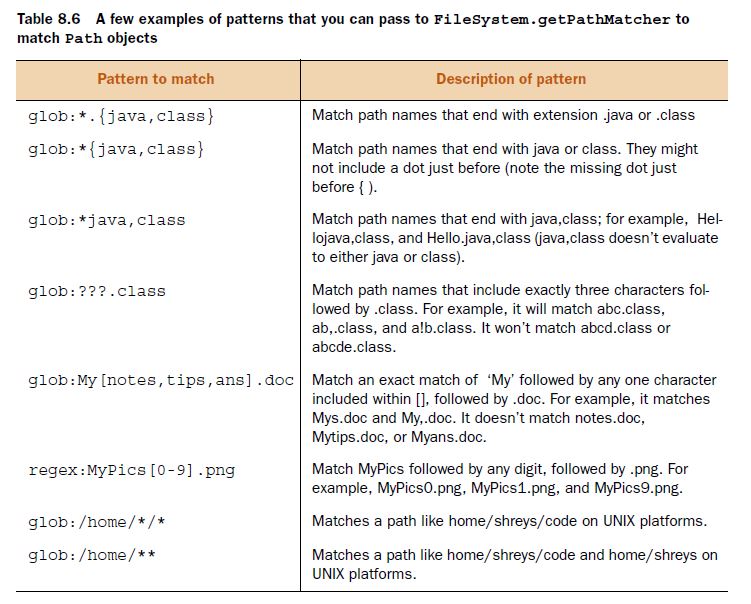
EXAM TIP In glob \* matches zero or more characters. In regex .\* matches zero or more characters.

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*

The ResultSet object can be read-only, scrollable, or updatable. By default, a ResultSet object is only read-only and can be traversed in only one (forward) direction. You can create a scrollable ResultSet (that can be traversed forward and backward) and/or an updatable ResultSet by passing the relevant parameters during the creation of a Statement object.

***Connecting to a database***

EXAM TIP For the exam, it’s important to note the difference between a JDBC driver (lowercase d) and a Driver class (uppercase D). A JDBC driver is a set of classes provided by the database vendor, or a third party, usually in a .jar or .zip file, to support the JDBC API. A Driver class is an implementation of the interface java.sql.Driver in a JDBC driver. For example, for MySQL, its platform-independent JDBC driver can be downloaded as mysql-connector-java-5.1.27.zip. The name of the class that implements java.sql.Driver in MySQL Connector/J (JDBC driver) is com.mysql.jdbc.Driver.

There are two approaches to load JDBC drivers:

* Manual (JDBC API version 3.0 and before)
* Automatic (JDBC API version 4.0 and later)

With JDBC 3.0 and its earlier versions, you need to call Class.forName(), passing it the name of the class that implements the interface java.sql.Driver. An example of loading a JDBC driver for a MySQL database is as follows:

Class.forName(“com.mysql.jdbc.Driver”);

EXAM TIP Class.forName() will throw a ClassNotFoundException (a checked exception) if the JVM is unable to locate the specified class.

The preceding code will load class com.mysql.jdbc.Driver in memory, executing its static initializer block(s). According to the JDBC specification, a driver must register itself with the DriverManager. For example, here’s the static initializer block defined in MySQL’s class com.mysql.jdbc.Driver that **initializes and registers** itself with the DriverManager:

static {

try {

java.sql.DriverManager.registerDriver(new Driver());

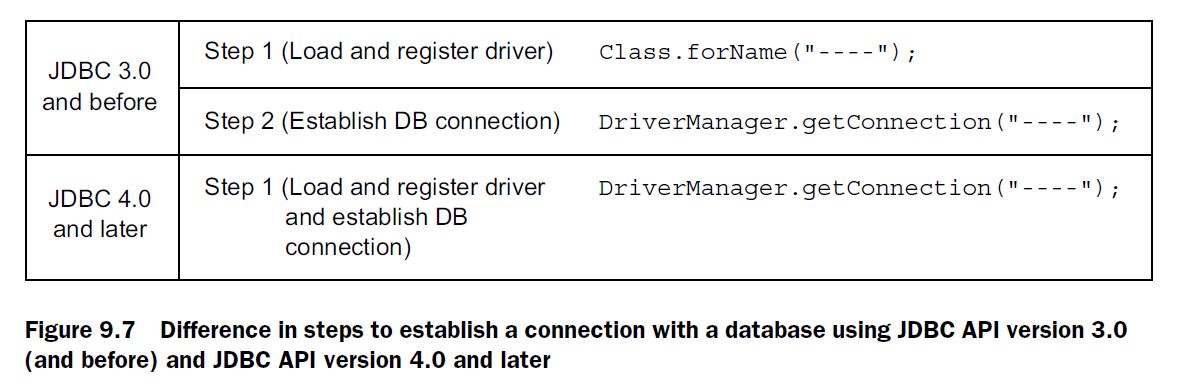
} catch (SQLException E) {

throw new RuntimeException("Can't register driver!");

}

}

JDBC 4.0 and its later versions support automatic loading and registration of all JDBC drivers accessible via an application’s class path. You no longer need to explicitly load the driver in memory using Class.forName(). The SPM automates the driver loading mechanism. Using SPM, every JDBC 4.0 driver implementation must include the configuration file with the name java.sql.Driver within the META-INF/services folder in their .jar file.



***Use DriverManager to connect to a database***

Class DriverManager manages all the instances of JDBC driver implementations and registered with a JVM when loaded into memory, its static initializer attempts to load all JDBC drivers that are referred to jdbc.drivers system property. DriverManager is a starting point to obtain database connections for Java SE. When you invoke method getConnection(), the DriverManager finds the appropriate drivers from its set of registered drivers, establishes a connection with a database, and returns a Connection object. Here are the overloaded getConnection() methods:

public static Connection getConnection(String url) throws SQLException

public static Connection getConnection(String url, Properties info) throws SQLException

public static Connection getConnection(String url, String user, String pwd) throws SQLException

jdbc:subprotocol://<host>:<port>/<database\_name>

Connection con = null;

try {

con = DriverManager.getConnection("jdbc:mysql://localhost/BookLibrary?"+ "user=test&password=test");

}

catch (SQLException e) {

}

----------------------------------------------------------------

java.util.Properties prop = new java.util.Properties();

prop.put("user", "test");

prop.put("password", "test");

DriverManager.getConnection("jdbc:mysql://localhost/BookLibrary", prop);

EXAM TIP Make note of the property names for specifying the username and password: the keys are “user” and “password”. An attempt to use any other key to specify the username and password to connect to a database will throw a SQLException.

For example, I didn’t add the JDBC driver to the application’s class-path environment variable. Here’s the exception message that I got when I tried to run this code:

java.sql.SQLException: No suitable driver found for jdbc:mysql://localhost/BookLibrary?user=test&password=test

If the class is unable to connect to a database due to invalid login credentials, you’ll get a SQLException:java.sql.SQLException: Access denied for user 'test'@'test' (using password:YES)

Note that you need to call method executeUpdate() on Statement and not executeQuery() to execute a Data Definition Language (DDL) request.

EXAM TIP Method executeUpdate() is used to execute SQL queries to insert new rows in a table, and update and delete existing rows. It’s also used to execute DDL queries, such as the creation, modification, and deletion of database objects like tables. If you use method executeQuery() for any of these operations, you’ll get a SQLException at runtime.

EXAM TIP Because the interfaces Connection, Statement, and ResultSet extend the interface AutoCloseable, you can create their instances in a try-with-resources statement.

EXAM TIP For all SQL operations on a database, the preferred programming approach is to close the Connection and Statement objects. You must either close them explicitly by calling close() on them or use them with a try-with-resources statement, which auto-closes them. You’re very likely to see a question that asks you about closing these resources and in their correct order—first Statement and then Connection object.

EXAM TIP Method executeQuery() is used for SQL SELECT statements.

ResultSet rs = statement.**executeQuery**("SELECT \* FROM book " +

"WHERE unit\_price > 47.7");){

while (rs.next()) {

System.out.print(rs.getInt("id") + "-");

System.out.print(rs.getString("title") + "-");

System.out.print(rs.getString("author") + "-");

System.out.print(rs.getInt("publication\_year") + "-");

System.out.println(rs.getDouble("unit\_price"));

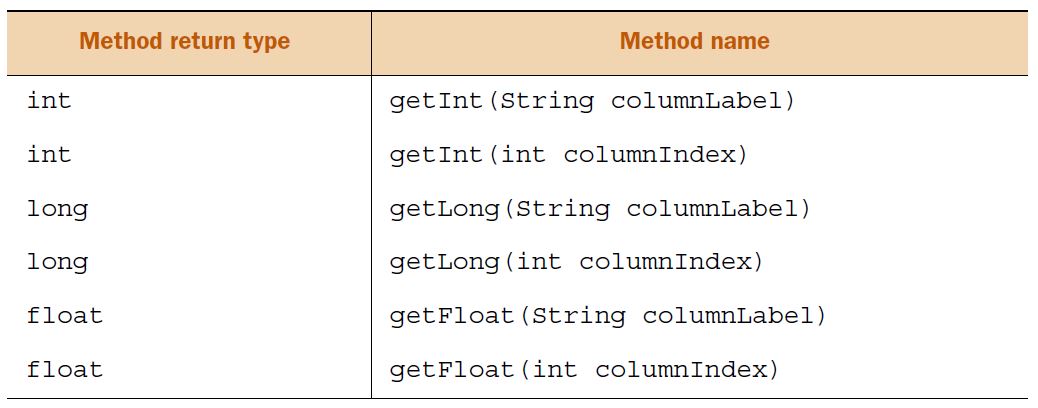
}

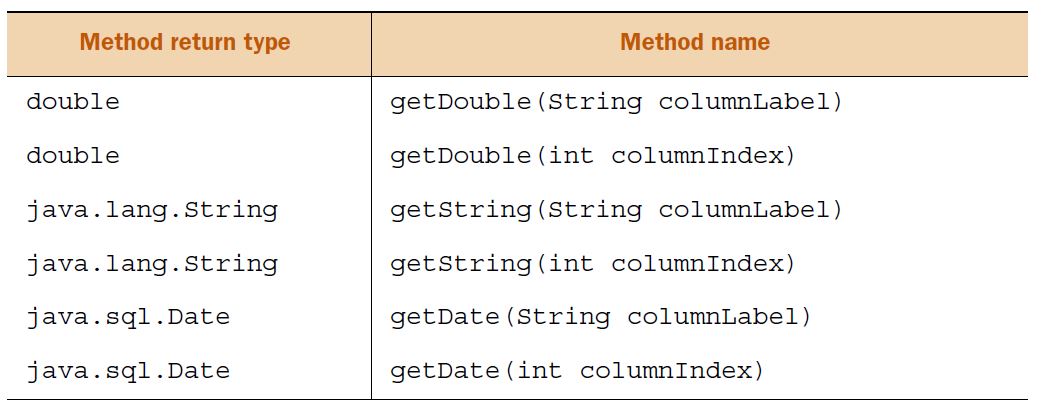
Take note of method getString() and its method arguments (column names) used to retrieve the value for a column in a row. The overloaded method getString() in the interface ResultSet accepts either the column position or column label and returns the table value as a string. It takes the following forms:

String getString(int columnLabel)

String getString(int columnIndex)

EXAM TIP Although everything in Java is 0-based, column indexes in a ResultSet are 1-based.





What happens if the target table doesn’t find any matching rows in the table for your SELECT query? In this case, the ResultSet object **isn’t** set to null.

EXAM TIP If your SQL statement doesn’t return any rows, the ResultSet object doesn’t point to a null value. In this case, you’ll get a ResultSet object that won’t include any rows.

***JDBC transactions***

EXAM TIP Method executeUpdate() returns a count of the rows that are or would be affected in the database for row insertions, modifications, and deletion. The value is returned even if the statement isn’t committed. This method returns 0 for SQL DDL statements, which create database objects and modify their structure or delete them.

To work with a transaction, you must set the auto-commit mode to false or methods rollback() and commit() will throw a SQLException.

If the commit mode of a connection is set to false and you try to set it to true during the course of a transaction, the code will throw a SQLException when you call any transaction-related method like commit() or rollback().

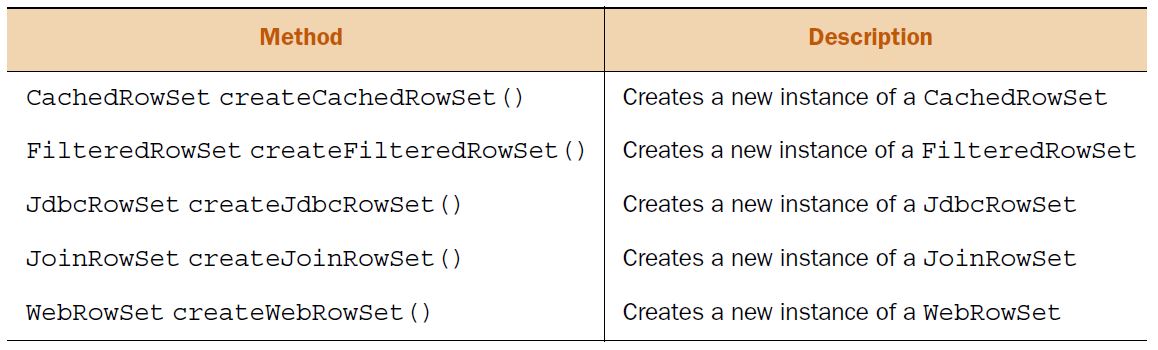
***RowSet objects***

You can configure a RowSet object by setting its properties, connecting to a JDBC data source, executing a SQL statement, and getting the results.

You can register listeners with a RowSet object so that when an event occurs on a RowSet object (like any modification to its value), the registered listeners can be notified.

RowSet objects can be *connected* or *disconnected*. A connected RowSet object, like JdbcRowSet, maintains a connection with its data source throughout its life. On the other hand, a disconnected RowSet object, like CachedRowSet, establishes a connection with the data source, gets the values, and then disconnects itself. It can still update its values and later update them in the data source by reconnecting to it.

The interface javax.sql.rowset.RowSetFactory defines the implementation of a factory that can be used to obtain different types of RowSet implementations.



To get access to an object of RowSetFactory, you can use class RowSetProvider.

Class javax.sql.rowset.RowSetProvider defines factory methods to get a RowSetFactory implementation. The RowSetFactory can then be used to create objects of different types of RowSet implementations.

You can also specify a custom factory implementation by specifying its name. For example

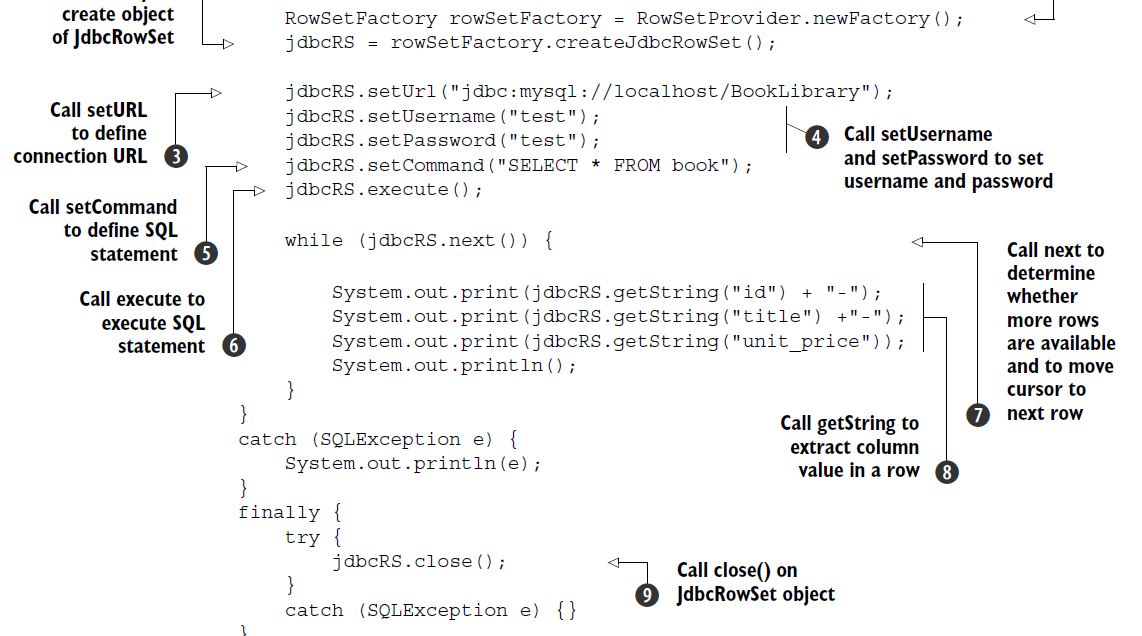
RowSetFactory rowsetFactory = RowSetProvider.newFactory(

"com.ejava.sql.rowset.CustomRowSetFactory", null);

This interface defines the following methods to create objects of RowSetFactory implementations:

static RowSetFactory newFactory()

static RowSetFactory newFactory(String factoryClassName, ClassLoader cl)



In the preceding code, note how the JdbcRowSet object manages all the database operations without any separate Connection, Statement, or ResultSet objects.

***Precompiled statements***

The objects of interfaces PreparedStatement and CallableStatement represent precompiled SQL statements. Precompiled SQL statements are compiled in the database system. The precompiled statements execute faster than their non-compiled counterparts. Another major advantage offered by PreparedStatement and CallableStatement is their ability to include placeholders in SQL statements using a ?. You can assign values to these placeholders by calling one of the appropriate setDataType(parameterIndex, value) on these objects. And the most critical advantage of using parameters with PreparedStatement and CallableStatement is that they prevent SQL injection attacks.

***Prepared statements***

The interface java.sql.PreparedStatement extends the interface java.sql.Statement. Its objects represent precompiled SQL statements. The first difference that you’d notice when you compare PreparedStatement with Statement is in its creation. Unlike Statement, you must specify the relevant SQL statement when you create an object of PreparedStatement.

PreparedStatement stmt = con.prepareStatement("SELECT \* FROM book " +

"WHERE unit\_price > ?");

Stmt.setDouble(1, 1.0);//we can use setxxx() methods for setting dynamic values.

EXAM TIP Unlike the interface Statement, where you specify the SQL query with the issue of method executeQuery() or executeUpdate(), you must specify the SQL query when you create objects of PreparedStatement.

EXAM TIP PreparedStatement defines three methods to execute its SQL statement: execute(), executeQuery(), and executeUpdate(). Method execute() can execute any type of SQL statement and returns a boolean value true when a query executes successfully, false when an error occurs. Method executeQuery() executes a SQL SELECT statement and returns a ResultSet. Method executeUpdate() executes a DDL query, like CREATE TABLE, and INSERT, UPDATE, and DELETE SQL statements. It returns 0 for DDL statements and the number of rows affected for SQL INSERT, UPDATE, and DELETE statements.

***CallableStatement***

Stored procedures are compiled and stored in the database. It’s much more efficient to execute multiple, precompiled SQL statements together in a procedure, rather than sending individual, non-compiled SQL statements from your application.

CallableStatement cs = con.prepareCall("{call book\_details()}");

ResultSet rs = cs.executeQuery();

Because the database-stored procedure book\_details doesn’t accept any parameters, it’s acceptable to drop the () following the procedure name:

CallableStatement cs = con.prepareCall("{call book\_details}");

You can define parameters of type IN, OUT, and INOUT with a database-stored procedure. The parameters of type IN can be used to pass values to a procedure, the parameters of type OUT can be used to return values from a procedure, and the parameters of type INOUT can be used to do both—pass values to a procedure and return values from it.

CallableStatement cs = con.prepareCall("{call proc\_author(?, ?)}");

cs.setString(1, "Shreya");

cs.registerOutParameter(2, Types.NUMERIC);

cs.setInt(2, rowCount);

Which SQL standard is JDBC 4.1 (Java 7) consistent with SQL: 2003

What happens if the database data type doesn’t match with Java’s getXXX()methods used to retrieve the database column value? You can use method getString() to retrieve all types of database values. But if you use other getXXX() methods like getDouble() to retrieve a database column value, the method will throw a SQLException if the column value can’t be implicitly converted to Java data type double.

*10. Threads*

A process can create multiple threads, also known as *lightweight* processes. Multiple threads share the resources of the process in which they’re created, like memory and files. They also have their own exclusive working space, like stacks and PC registers. JVM supports multiple thread execution. Multiple threads can be supported by an underlying system by using multiple hardware processors, by time-slicing a single processor, or by time-slicing multiple processors.

A thread of execution has its own set of program counter (PC) registers to store the next set of instructions to execute. It also has its own *stack* that stores *method frames* to store the state of a method invocation. The state of a method invocation includes the value of local variables, method parameters, method return values, exception handler parameters, and intermediate values.

A JVM starts execution of a Java application using a thread of execution: main.

NOTE The main thread is named ‘main’ by the JVM. Don’t confuse it with the method main().

EXAM TIP Watch out for code that overrides method start() in a class that extends Thread. If it doesn’t call method start() from its superclass Thread, it won’t start a new thread of execution.

Class Thread defines multiple overloaded constructors.

Thread()

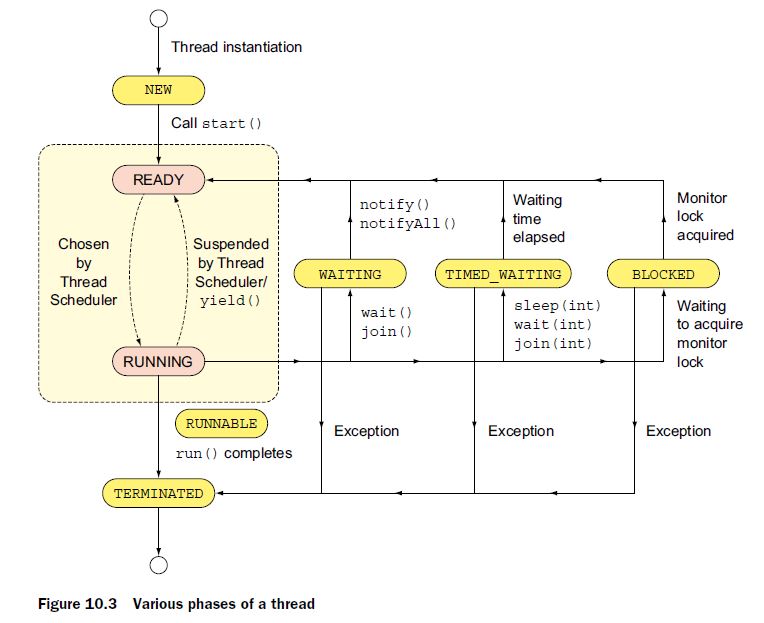
Thread(String name)

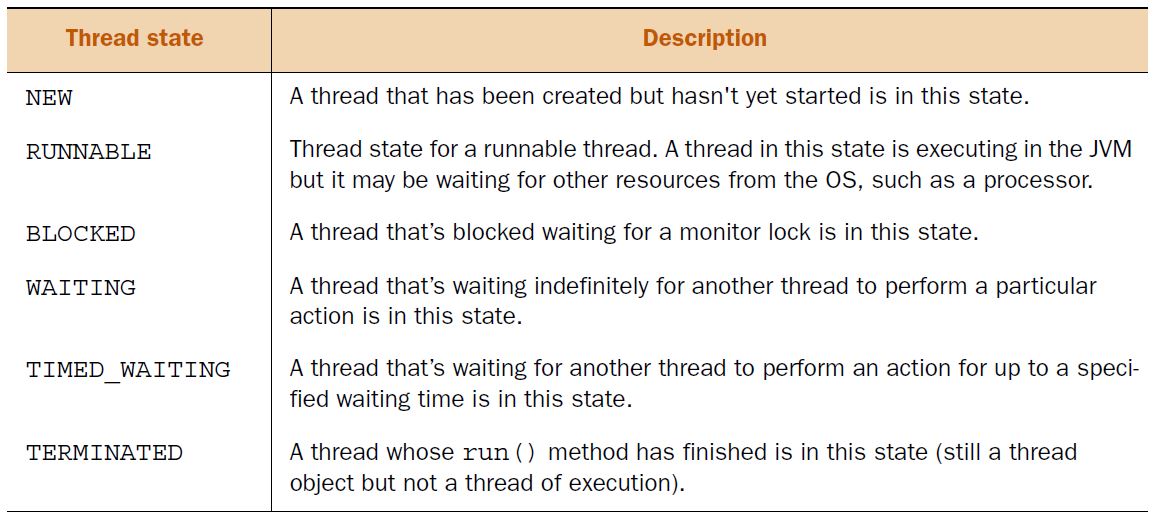
Thread(Runnable target)

Thread(Runnable target, String name)

EXAM TIP You can’t guarantee that a thread with a higher priority will always execute before a thread with a lower priority.

A thread can exist in multiple states: NEW*,* RUNNABLE*,* WAIT*,* TIMED\_WAITING*,* BLOCKED*,* or TERMINATED. A thread doesn’t begin its execution with its instantiation.





Calling start() on a new thread instance implicitly calls its run(), which transitions its state from NEW to RUNNABLE. A thread in the RUNNABLE state is all set to be executed. It’s just waiting to be chosen by the thread scheduler so that it gets the processor time. Thread scheduling is specific to the underlying OS on every system. As a programmer, you can’t control or determine when a particular thread transitions from the READY state to the RUNNING state, and when it actually gets to execute. A thread scheduler follows various scheduling mechanisms to utilize a processor efficiently, and also to give a fair share of processor time to each thread. It might suspend a running thread to give way to other READY threads and it might execute it later. The READY and the RUNNING states are together referred to as the RUNNABLE state.

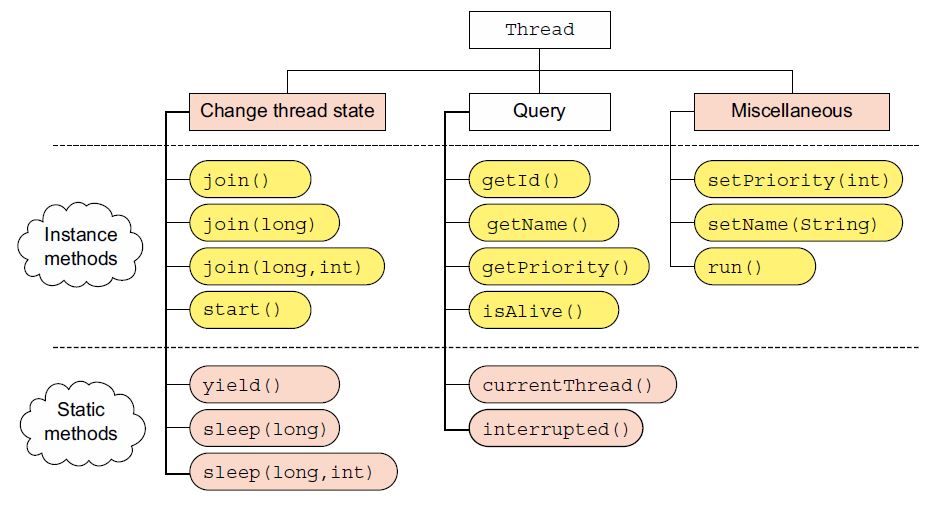
EXAM TIP The states READY and RUNNING are together referred to as the RUNNABLE state.

You can execute the following quick code to get a list of all the threads that are active on your system and the state they’re in:

Set<Thread> threadSet = Thread.getAllStackTraces().keySet();

for(Thread t : threadSet)

System.out.println(t + " --- " + t.getState());



NOTE Class Thread includes deprecated methods, like resume(), stop(), and suspend(). Because their use isn’t encouraged.

Calling start() on a Thread instance creates a new thread of execution, which executes run(). You can call start() on a thread that’s in the NEW state. Calling start() from any other thread state will throw an IllegalThreadStateException:

class Test{

public static void main(String args[]) {

Thread sing = new Sing();//**State of thread is NEW**

sing.start();//**state of the thread changed to** **RUNNABLE**

sing.start();//**throws IllegalThreadStateException**

}

}

class Sing extends Thread{

public void run() {

System.out.println("Singing");

}

}

The preceding code might print Singing and then throw an IllegalThreadStateException. It might throw an IllegalThreadStateException without printing Singing. How? The preceding code might start the thread sing, but *before* it prints Singing, method main() might call start() on the sing thread again, throwing an IllegalThreadStateException.

EXAM TIP You can call start() only once on a Thread instance when it’s in the NEW state. Calling start() on a thread in any other state will throw an IllegalThreadStateException.

Thread sing = new Sing();//**State of thread is NEW**

sing.run();//**Calls run()—State of thread remains NEW.**

sing.run();

EXAM TIP Calling run() on a Thread instance doesn’t start a *new* thread of execution. The run() continues to execute in the *same* thread. Watch out for trick questions on using start() versus run() in the exam.

EXAM TIP As a Java programmer, you can’t control or determine when the thread scheduler moves a thread from the READY state to RUNNING and vice versa. It’s specific to an OS.

The static method yield() makes the currently executing thread pause its execution and give up its current use of the processor. The static method yield() can be placed literally anywhere in your code not only in method run():

class YieldProcessorTime {

public static void main(String args[]) {

Thread sing = new Sing();

sing.start();

Thread.yield();//**Might cause thread main to yield its processor time.**

}

}

class Sing extends Thread{

public void run() {

yield();//**When executed, might cause thread sing to yield its processor time.**

System.out.println("Singing");

}

}

When called from two threads, thread 1 and thread 2, yield() might or might not *yield* its execution. So what’s guaranteed from this method call? To be precise, nothing. It might not make the currently executing thread give up its processor time. If it does, it doesn’t guarantee when it will happen and when the thread will resume its execution.

EXAM TIP Method yield() is static. It can be called from any method, and it doesn’t throw any exceptions.

The static method Thread.sleep() is *guaranteed* to cause the currently executing thread to temporarily give up its execution for *at least* the specified number of milliseconds (and nanoseconds) and move to the READY state.

try {

Thread.sleep(10);// **Guarantees to sleep for at least 10 milliseconds (if not interrupted).**

}

catch (InterruptedException e) {

// **If interrupted, sleeping thread might throw InterruptedException**

System.out.println(e);

}

Class Thread defines the overloaded versions of sleep() as follows:

public static native void sleep(long milli) throws InterruptedException;

public static void sleep(long milli, int nanos) throws InterruptedException

Whether a thread will sleep for the precise duration specified in nanoseconds will depend on an underlying system. Unless interrupted, the currently executing thread will sleep at least for the specified duration. On the exam, watch out for questions that state a thread will become runnable *exactly* after the expiration of the sleep duration. Unless interrupted, a thread is guaranteed to sleep for *at least* the specified duration. The exact time to resume execution depends on the thread scheduler.

EXAM TIP A thread that’s suspended due to a call to sleep doesn’t lose ownership of any monitors.

What happens if you place sleep() in MovingBall’s constructor?

MovingBall(){

try {

Thread.sleep(1000);

}

catch (InterruptedException e) {

System.out.println(e);

}

}

A thread that instantiates MovingBall will execute sleep(), and then sleep for the specified duration before it can complete MovingBall’s instantiation.

If thread A calls join() on a Thread instance B, A will wait for B to complete its execution before A can proceed to its own completion.

class ScreenDesign extends Thread {

public void run() {

for (int i = 0; i < 5; i++) System.out.println(i);

}

}

class Developer {

ScreenDesign design;

Developer(ScreenDesign design) {

this.design = design;

}

public void code() {

try {

System.out.println("Waiting for design to complete");

**design.join();**

System.out.println("Coding phase start");

}

catch(InterruptedException e) {

System.out.println(e);

}

}

}

class Project {

public static void main(String[] args) {

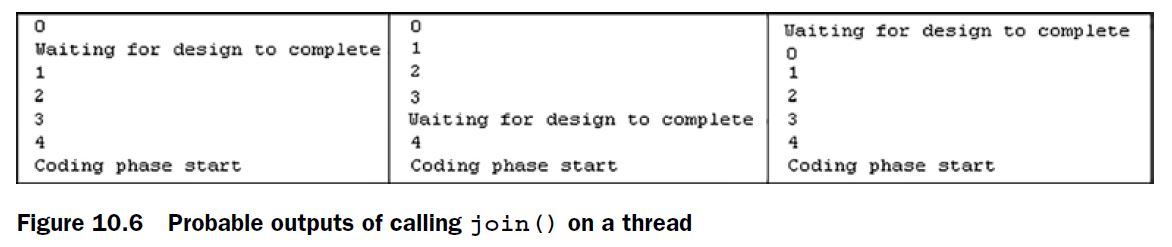
ScreenDesign design = new ScreenDesign(); design.start();

Developer dev = new Developer(design);

dev.code();

}

}



EXAM TIP Method join() guarantees that the calling thread won’t execute its remaining code until the thread on which it calls join() completes. Class Thread defines overloaded join() methods as follows:

public final synchronized void join(long milli) throws InterruptedException

public final synchronized void join(long millis, int nanos)

throws InterruptedException

public final void join() throws InterruptedException

The variations of join() that accept milliseconds and nanoseconds wait for at least the specified duration (if they’re not interrupted!). Behind the scenes, join() is implemented using methods wait(), isAlive(), and notifyAll().

A thread completes its execution when its method run() completes. You must not call the deprecated method stop() to stop execution of a thread. A thread might perform a task for a finite number of times or it might loop for an indefinite number of times. In the later case, the thread should define an exit condition to indicate its completion:

class Sing extends Thread{

boolean singStatus = true;

public void run() {

**while (singStatus)**

System.out.println("Singing");

}

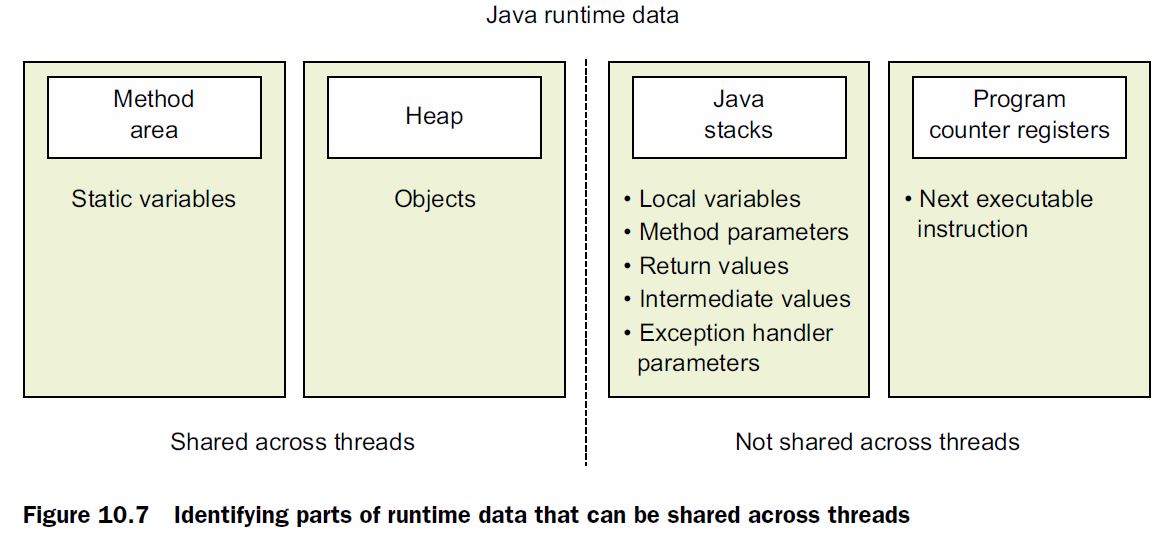
public void setSingStatus(boolean value) {

singStatus = value;

}

}

In the preceding code, the instance of thread Sing will execute until its instance variable singStatus is assigned a value of false. An application can modify the value of singStatus using setSingStatus(boolean). Once run() completes its execution, the status of a thread changes to TERMINATED.



NOTE Operations that use arithmetic and assignment operators like ++, --, +=, -=, \*=, and /= aren’t atomic. Multiple threads that manipulate variable values using these operators can interleave.

The methods, which modify the state of instance or static variables, should be defined as synchronized methods.

When a thread invokes a synchronized method, it automatically locks the monitor. If the method is an instance method, the thread locks the monitor associated with the instance on which it’s invoked (referred to as this within the method). For static methods, the thread locks the monitor associated with the Class object, thereby representing the class in which the method is defined. These locks are released once execution of the synchronized method completes, with or without an exception.

EXAM TIP For non-static instance synchronized methods, a thread locks the monitor of the object on which the synchronized method is called. To execute static synchronized methods, a thread locks the monitor associated with the Class object of its class.

EXAM TIP A thread releases the lock on an object monitor after it exits a synchronized method, whether due to successful completion or due to an exception.

Immutable objects like an instance of class String and all the wrapper classes (like Boolean, Long, Integer, etc.) are thread safe because their contents can’t be modified. So no matter how and when you access them, it doesn’t result in a *dirty read* or *dirty write*.

In the following example, once title is initialized, no matter how it’s accessed by multiple threads, it never leads to inconsistent memory or race conditions:

class Book{

private String title;

}

EXAM TIP Shared immutable objects can’t result in inconsistent object data or an incorrect object state because, once initialized, they can’t be modified.

A synchronized code block can be executed by only one thread, as an atomic operation. The volatile variables don’t support the atomicity feature. When a thread reads from or writes to a variable (both primitive and reference variables) marked with the keyword volatile, it accesses it from the main memory. This prevents multiple threads from storing a local copy of shared values that might not be consistent across threads.

So if they don’t offer features as good as synchronized code, why do we need them? Because they offer simplicity and better performance.

Threads can share the heap that includes class variables and instance variables. Each thread gets its own share of stack memory, which includes local variables, method parameters, and exception handler parameters. So a multithreaded application should safeguard the static and instance variables or attributes of its shared objects.

But what kind of operations should you care about? Should you only be concerned with the methods that change the value of a shared variable? Or, should you also synchronize the read operations?

***Operations you should care about***

To safeguard your data, you might think you only need to worry about methods that modify the value of a variable. Think again. Methods that *only* read shared variable values can also return incorrect or inconsistent data.

Imagine that agents Shreya and Harry (two threads) manage renting an exhibition ground, say, Axiom (shared resource). Harry has agreed to rent it to a customer and is in the process of signing the legal papers (one thread is updating the shared resource). Shreya has no clue about this development. Just before Harry completes signing the rental agreement, Shreya receives an enquiry about the availability of Axiom and she confirms that it’s available (another thread reads data while the shared resource is being modified). In this case, Shreya accessed *inconsistent data*.

EXAM TIP Methods wait(), notify(), and notifyAll() are defined in class Object and not in class Thread.

EXAM TIP Methods wait(), notify(), and notifyAll() *must* be called from a synchronized method or code blocks or else an IllegalMonitorStateException will be thrown by the JVM.

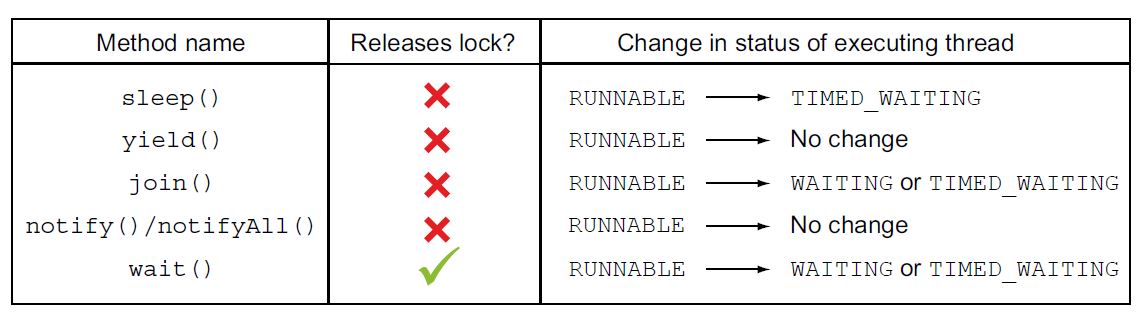
The overloaded wait() methods enable you to specify a wait timeout:

public final void wait(long timeout) throws InterruptedException

public final void wait(long timeout, int nanos) throws InterruptedException

When a thread calls the method wait() on an object specifying a timeout duration, it waits until another thread calls notify() or notifyAll on the same object, it’s interrupted by another thread, or the waiting timeout elapses. Even though the overloaded version wait(long, int) accepts a timeout duration in nanoseconds, supporting such time precision is JVM- and OS-specific.

EXAM TIP All overloaded versions of wait() throw a checked InterruptedException. Methods notify() and notifyAll() don’t throw an InterruptedException.



***Livelock***

Imagine you’re talking with your friend on your mobile phone and the call drops. You try to reconnect with her, but her phone is busy (because she’s trying to call you!). You wait for a few moments and try to reconnect, only to discover that her phone is still busy, because she waited for exactly the same duration before trying to reconnect again. If you compare yourself and your friend with threads, you both are in a livelock. You and your friend aren’t blocked—both of you are responding but aren’t being able to do what you both intend to (talk with each other).

EXAM TIP Threads in a livelock aren’t blocked; they’re responding to each other, but they aren’t able to move to completion.

* The execution of start()*happens-before* any action in a thread is started.
* When code is defined in a sequence, step 1 *happens-before* step 2.
* Unlocking of an object monitor *happens-before* any other thread acquires a lock on it.
* A write to a volatile field *happens-before* every subsequent read of that field.
* All actions in a thread *happens-before* any other thread returns from a join on that thread.

sleep(), join(), wait() methods throws InterruptedException.

*11. Concurrency*

* BlockingQueue defines a first-in-first-out data structure that blocks or times out when you attempt to add items to a full queue, or retrieve from an empty queue.
* ConcurrentMap is a sub interface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap.
* ConcurrentNavigableMap is a sub interface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is ConcurrentSkipListMap, which is a concurrent analog of TreeMap.

***BlockingQueue***

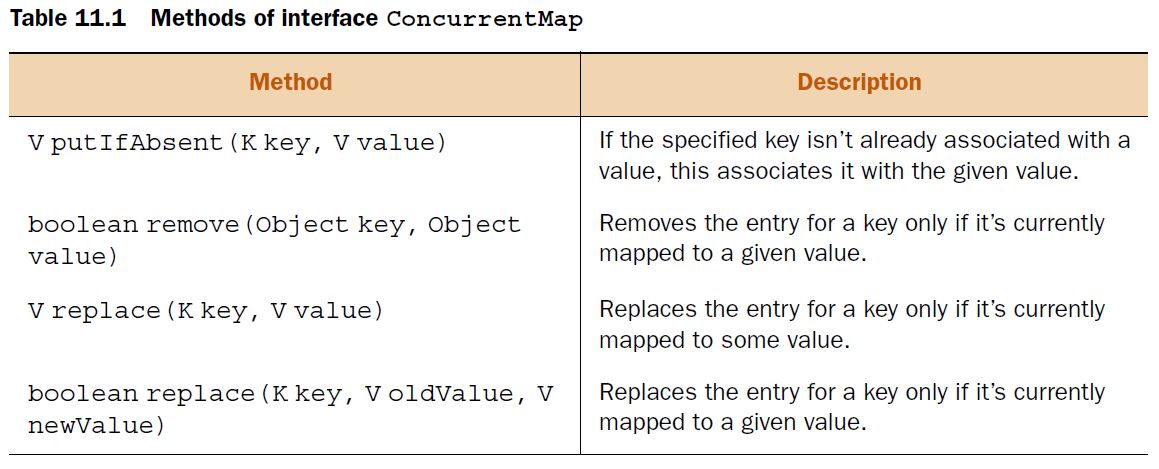
The BlockingQueue interface is a queue that’s safe to use when shared between multiple threads. The implementing classes like ArrayBlockingQueue include a constructor to define an initial capacity (which can’t be modified) from which items are added and removed. It blocks adding new elements if the queue has reached its capacity. It also blocks removing elements from an empty queue. It works on the *producer*–*consumer* pattern, which is when a single thread or multiple threads produce elements and add them to a queue to be consumed by other threads.

Imagine multiple clients (producers) that send requests to a server. The server (consumer) responds to all the requests that it receives. To manage the requests that all the clients might send to the server, the server can limit the maximum number of requests that it can accept at a given point in time. The requests can be added to a blocking queue, which will block adding new requests if it reaches its upper limit. Similarly, if no new requests are available in a queue, the server thread will block until requests are made available to it.



***ConcurrentMap***

The ConcurrentMap interface extends the java.util.Map interface. It defines methods to replace or remove a key-value pair if the key is present, or add a value if the key is absent.



***ConcurrentHashMap***

A concrete implementation of the ConcurrentMap interface, class ConcurrentHashMap is a concurrent class analogous to class HashMap. A HashMap is an unsynchronized collection. If you’re manipulating a HashMap using multiple threads, you must synchronize its access. But locking the entire HashMap object can create serious performance issues when it’s being accessed by multiple threads. If multiple threads are retrieving values, it makes sense to allow concurrent **read operations and monitor write operations**.

The ConcurrentHashMap class is the answer to improving the responsiveness of HashMap when it needs to be accessed concurrently by multiple threads. Instead of exclusively locking itself to be accessed by one thread, ConcurrentHashMap allows access by multiple threads. It concurrently allows **multiple threads to read its values** and **limited threads to modify its values**. Also, the iterators of ConcurrentHashMap **don’t throw** a ConcurrentModificationException, so you don’t need to lock the collection while iterating it. So what happens if new elements are added to ConcurrentHashMap *after* you accessed its iterator? The iterator may still traverse only the elements that existed at the time of creation of the iterator. Though not guaranteed on all platforms, the iterators might reflect the new additions.

This collection also offers some drawbacks. Because it doesn’t lock the complete collections while modifying their elements, methods like size() might not return the exact accurate size of a ConcurrentHashMap when invoked by multiple threads.

ConcurrentMap<Integer, String> map = new ConcurrentHashMap<>();

if(!map.containsKey(key))

map.put(key, value);

When you work with multiple threads, you need to synchronize access to your shared resources so that concurrent access doesn’t leave them in an inconsistent state. In the preceding example, the individual operation containsKey(key) is a read operation and put(key, value) is a write operation. Though individually these methods are thread safe, together (execute method 1, then method 2) they aren’t. A race condition can occur when method manipulateMap() is executed by multiple threads. The solution is to replace them with a single atomic method call:

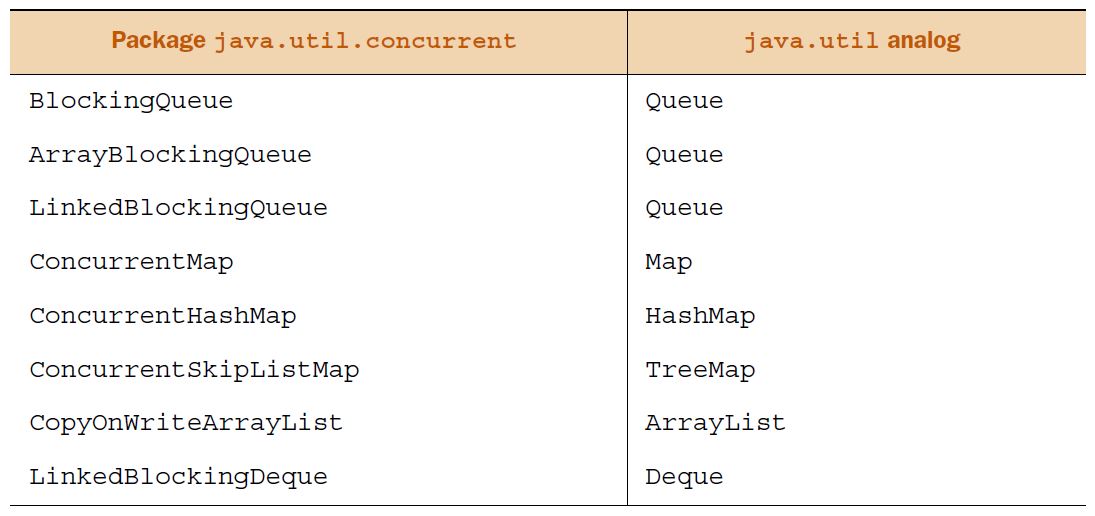
static void manipulateMap(Integer key, String value) {

// complex computations

map.replace(key, value);

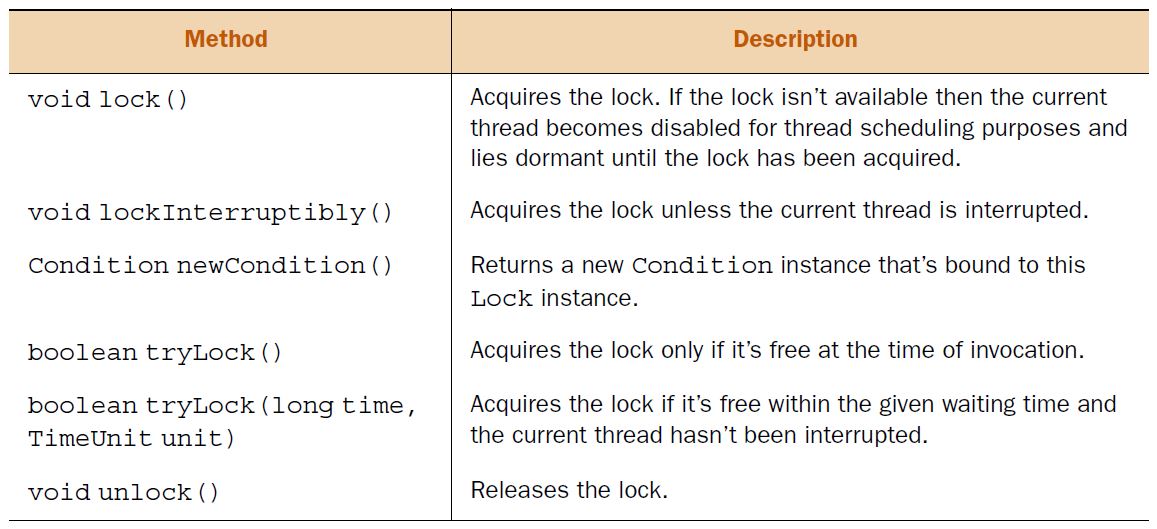
// **Atomic operation replaces value in map if corresponding key is present**

}



***Locks***

NOTE To execute synchronized code, a thread must acquire either an *implicit* or an *explicit* lock on an object’s monitor. Where no explicit Lock classes are used, I’ll refer to it as an implicit lock.



Method lock() acquires a lock on a Lock object. If the lock isn’t available, it waits until the lock can be acquired.

class Rainbow {

Lock myLock = new ReentrantLock();

static List<String> colors = new ArrayList<>();

public void addColor(String newColor) {

myLock.lock();//**Call lock to acquire lock; wait if lock not available**

try {

colors.add(newColor);

}

finally {

myLock.unlock();// **Release lock**

}

}

}

EXAM TIP Call method unlock() on a Lock object to release its lock when you no longer need it.

EXAM TIP Watch out for the use of methods acquire(), acquireLock(), release(), and releaseLock() on the exam. None of these is a valid method. Because the terms *acquire* and *release* are used to discuss methods lock(), unlock(), tryLock(), and lockInterruptibly(), these terms might be used on the exam to confuse you.

***Interruptible locks***

With intrinsic locks, you must release the lock on an object’s monitor at the end of the synchronized code blocks or methods. Because code blocks can’t span across methods, intrinsic locks can’t be acquired across methods. Extrinsic locks or a lock on Lock objects can be acquired across methods.

Here’s some example code to work with extrinsic locks:

class Bus {

ReentrantLock lock = new ReentrantLock();

boolean locked = false;

public void board(String name) {

if (**lock.tryLock()**) {

locked = true;

System.out.println(name + ": boarded");

}

}

public void deboard(String name) {

if (lock.isHeldByCurrentThread() && locked) {

System.out.println(name + ": deboarded");

**lock.unlock();**

locked = false;

}

}

}

***ReadWriteLock***

Interface ReadWriteLock maintains a pair of associated locks, one for read-only operations and another for write-only operations. The read-only lock may be held simultaneously by multiple reader threads as long as there are no writing processes in progress. The write-only lock is an exclusive lock. It can be acquired by only one thread. ReadWriteLock interface defines only two methods: readLock() and writeLock().

Lock readLock() Returns the lock used for reading.

Lock writeLock() Returns the lock used for writing.

You can use methods readLock() and writeLock() to get a reference to the read or write Lock. Let’s work with a concrete implementation of the ReadWriteLock interface, class ReentrantReadWriteLock.

EXAM TIP The ReadWriteLock interface doesn’t extend Lock or *any* other interface. It maintains a pair of associated Locks one for only reading operations and one for writing.

***ReentrantReadWriteLock***

A ReentrantReadWriteLock has a read and a write lock associated with it. You can access these locks (reference variables of type Lock) by calling its methods readLock() and writeLock(). You can acquire multiple read locks as long as no write lock has been acquired on a ReadWriteLock object. The writeLock is an exclusive lock; it can be acquired by only one thread when no read thread has been acquired.

class Rainbow {

private final ReadWriteLock myLock = new ReentrantReadWriteLock();

private static int pos;

static Map<Integer, String> colors = new HashMap<>();

public void addColor(String newColor) {

**myLock.writeLock().**lock**()**;

try {

colors.put(new Integer(++pos), newColor);

}

finally {

myLock.writeLock().unlock();

}

}

public void display() {

**myLock.readLock().**lock();

try {

for (String s : colors.values()) {

System.out.println(s);

}

}

finally {

myLock**.**readLock().unlock();

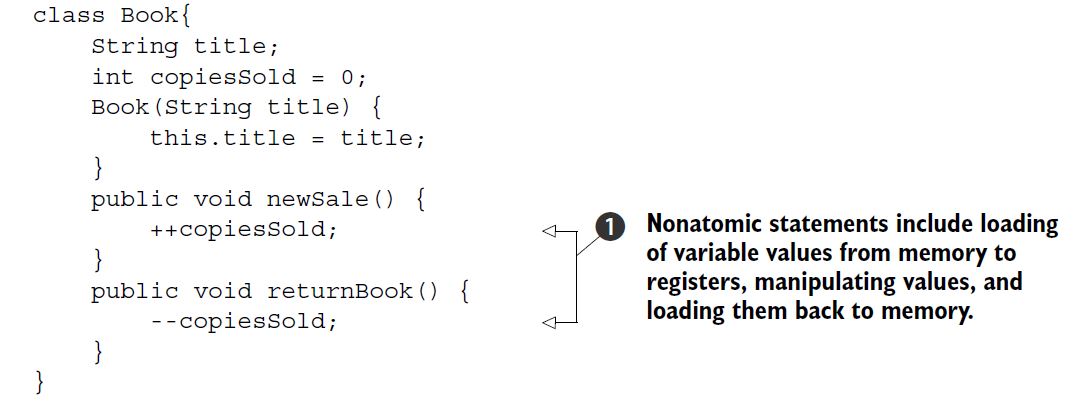
}

}

}

***Atomic variables***

The java.util.concurrent.atomic package defines multiple classes that support atomic operations of read-compare/modify-write on single variables. At the surface, these operations might seem similar to the operations with volatile variables. Though modifications to a volatile variable are guaranteed to be visible to other threads, volatile variables can’t define a sequence of operations (like read-compare/modify-write) as an atomic operation.



The code defined at 1 isn’t atomic. Incrementing or decrementing primitive values includes multiple steps. When executed by multiple concurrent threads, newSale() and returnBook() can result in thread interference. To get around this, you can define these methods as synchronized methods, but it will block thread execution. Java defines multiple convenient classes in the java.util.concurrent.atomic package that define frequently used operations like read-modify-write as atomic operations. Let’s use one of these classes, AtomicInteger, in class Book, and replace the type of its primitive int variable copiesSold:

class Book{

String title;

AtomicInteger copiesSold = new AtomicInteger(0);

Book(String title) {

this.title = title;

}

public void newSale() {

copiesSold.incrementAndGet();//**Atomic operation**

}

public void returnBook() {

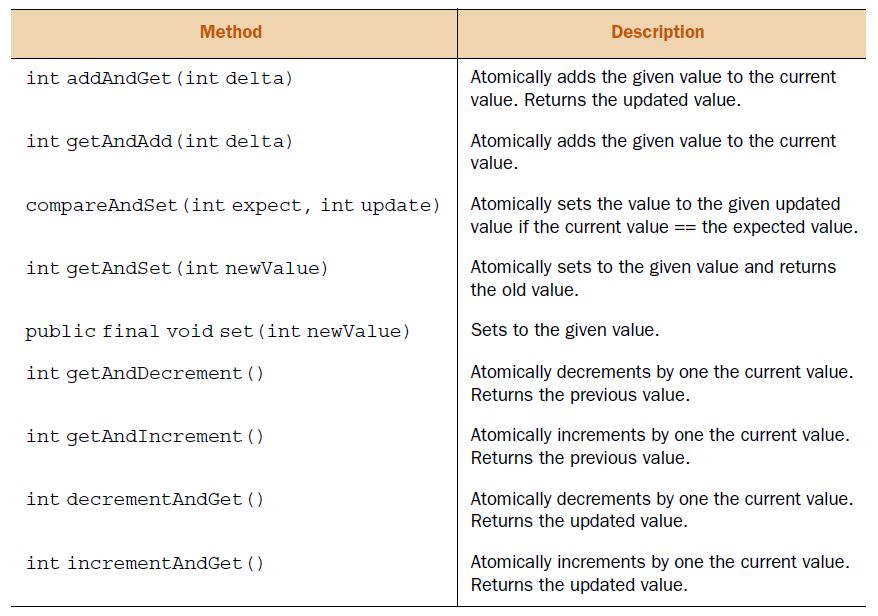
copiesSold.decrementAndGet();//**Atomic operation**

}

}

Methods incrementAndGet() and decrementAndGet() are atomic operations. Concurrent execution of these methods won’t result in interfering threads. Class AtomicInteger defines multiple methods xxxAndGet() and getAndXxx(), where Xxx refers to an operation like increment, decrement, and add. xxxAndGet() returns an updated value and getAndXxx() returns the previous value.

EXAM TIP Method incrementAndGet() returns the updated value but method AtomicInteger’s getAndIncrement() returns the previous value.



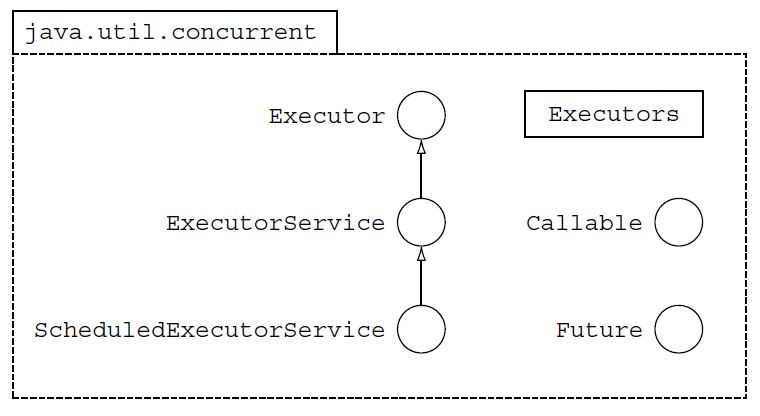
EXAM TIP Class AtomicInteger defines method compareAndSet() but not method setAndCompare().

Other commonly used classes defined in the java.util.concurrent.atomic package are AtomicBoolean, AtomicLong, AtomicIntegerArray, AtomicLongArray, and AtomicReference<V>. AtomicLong defines the same methods as class AtomicInteger. The difference is the type of method parameters and their return types (long instead of int).

EXAM TIP The java.util.concurrent.atomic package doesn’t define classes by the names AtomicShort, AtomicByte, AtomicFloat, or AtomicDouble. These invalid class names might be used on the exam.

***Executors***

The Executor framework enables decoupling of task submission with task execution. By using this framework, you can create tasks using interfaces Runnable and Callable. These tasks are submitted to Executor to launch new tasks. ExecutorService extends Executor and adds methods to manage the lifecycle of tasks and executors. ScheduledExecutorService extends ExecutorService and supports future or periodic execution of tasks. Future represents the state of asynchronous tasks and can be used to query their status or cancel them. Class Executors defines utility and factory methods for interfaces Executor, ExecutorService, and ScheduledExecutorService.



***Executor***

The Executor interface allows you to define classes that know *how* to execute Runnable tasks. It allows you to decouple task submission and its execution. By implementing its sole method, void execute(Runnable), you can determine how you want to execute the tasks:

* Which task will execute first
* The order of execution of tasks
* How many tasks can execute concurrently
* How many tasks can be queued

class Order implements Runnable {

String name;

Order(String name) {this.name = name;}

public void run() {

System.out.println(name);

}

}

class Hotel implements Executor {

final Queue<Runnable> custQueue = new ArrayDeque<>();

**public void execute(Runnable r) {**

synchronized(custQueue) {

**custQueue.offer(r);**

}

processEarliestOrder();

}

private void processEarliestOrder() {

synchronized(custQueue) {

**Runnable task = custQueue.poll();**

**new Thread(task).start();**

}

}

}

***Callable***

Comparing interfaces Runnable and Callable, method run() of the Runnable interface doesn’t return a value and can’t throw a checked exception. Both of these are taken care of by the Callable interface:

public interface Callable<V> {

V call() throws Exception;

}

In the following example, class Order implements the Callable interface. Because it isn’t interested in returning a value from call(), it uses Void as its parameterized argument:

class Order implements **Callable<Void>** {

String name;

Order(String name) {this.name = name;}

@Override

**public Void call() throws Exception {**

System.out.println(name);

if (name.equalsIgnoreCase("berry"))

**throw new Exception("Berry unavailable");**

return **null**;

}

}

EXAM TIP If you don’t want your Callable to return a value, you can create it using Callable<Void>.

***ExecutorService***

The ExecutorService interface extends the Executor interface and defines methods to manage progress and termination of tasks that are submitted to it. It defines methods to

* Submit single Runnable and Callable objects for execution, returning Future objects.
* Submit multiple Runnable objects for execution, returning Future objects.
* Shut down the ExecutorService, allowing or disallowing submitted tasks to be completed.

ExecutorService service = Executors.newFixedThreadPool(5);

***Thread pools***

* Fixed thread pool, which creates a pool with a fixed number of threads.
* Cached thread pool
* Single thread executor
* Scheduled thread pool

***ScheduledExecutorService***

Imagine you need to send out reminder emails to all the employees of your organization to submit their daily status reports. This email is sent out every day.

ScheduledExecutorService service = Executors.newScheduledThreadPool(1);

Reminder reminder = new Reminder();

service.scheduleAtFixedRate(reminder, 0, 24, HOURS);

***Parallel fork/join framework***

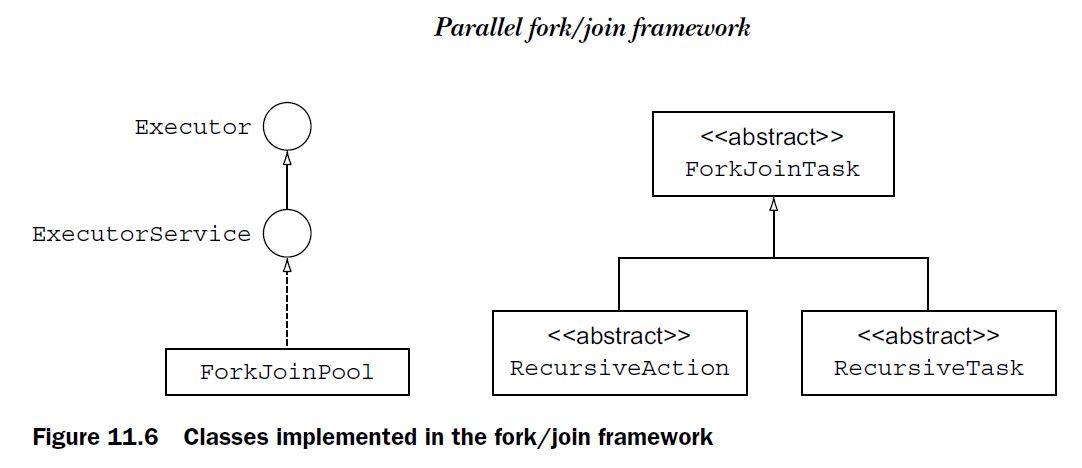
**fork-join** as name implies it divide one task into several small task as a new fork means child and join all the fork when all the sub-tasks complete.

When we compare **Executors** with old Thread it has made management of concurrent task very easy and it workon **divide and conquer** algorithm and create sub-tasks and communicate with each other to complete. But the problem with the executors framework is that a Callable is free to submit a new sub-task to its executor and wait for its result in a synchronous or asynchronous fashion. The issue is that of parallelism: When a Callable waits for the result of another Callable, it is put in a waiting state, and thus wasting an opportunity to handle another Callable queued for execution.

To solve this issue java 7 has given the concept of parallelism. New **fork-join** **framework** has been added in java.util.concurrent package. New fork-join executor framework has been created which is responsible for creating one new task object which is again responsible for creating new sub-task object and waiting for sub-task to be completed. Internally it maintains a thread pool and executor assign pending task to this thread pool to complete when one task is waiting for another task to complete. whole Idea of fork-join framework is to leverage multiple processors of advanced machine.

NOTE The fork/join framework is named so because it initiates execution of a task that *forks* or starts multiple subtasks, and waits for them to *join* back (or complete their execution).

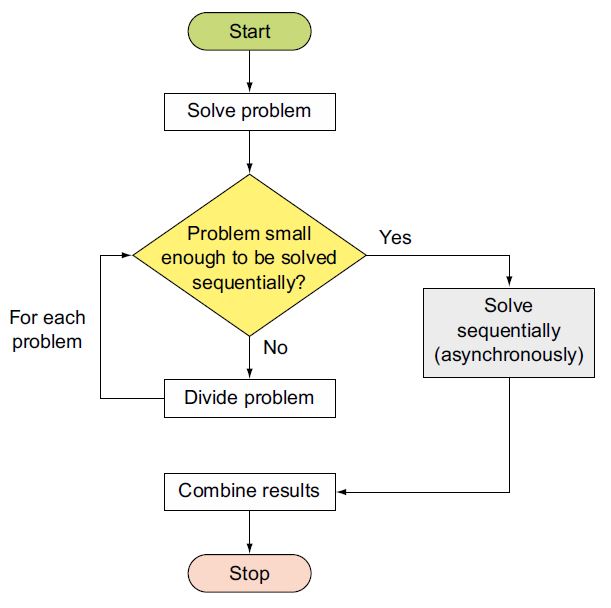
Class ForkJoinPool is a concrete implementation of the fork/join framework. It implements the ExecutorService interface. Like ExecutorService, a fork/join framework maintains a queue of tasks that are used to assigns tasks to its multiple worker threads. But it’s different from an ExecutorService because a fork/join framework implements a *work-stealing algorithm*. In this algorithm, when worker threads run out of tasks, they steal tasks from other worker threads to avoid blocking waiting threads.



Fork-join functionality is achieved by ForkjoinTask **object,** it has two method**fork() and join ()**Method.

* The **fork()** method allows  a new ForkJoinTask to be launched from an existing one.
* The **join()** method allows a ForkJoinTask to wait for the completion of another one.

Again ForkjoinTask object has been of two types: RecursiveAction and RecursiveTaskwhich is a more specialized form of this instance. While RecursiveAction represents executions that do not yield a return value, Instances of RecursiveTask yield return values.



Class ForkJoinPool provides the entry point for submissions from non-ForkJoinTask clients, as well as management and monitoring operations. Your problem-solving class should be a subclass of ForkJoinTask. ForkJoinTask is an abstract base class for tasks that run within a ForkJoinPool. A ForkJoinTask can be compared to a much lighter weight version of a thread. A large number of ForkJoinTasks can be executed by a smaller number of actual threads in a ForkJoinPool. Figure 11.6 shows class ForkJoinPool and class ForkJoinTask and its two subclasses, RecursiveTask and RecursiveAction. RecursiveAction is used for computations that don’t return a result, and RecursiveTask is used for computations that return a result.

The order of execution of calling join() and compute() on divided subtasks is important in a fork/join framework. First compute() and then join().

return(rightSum.compute() + leftSum.join());

return(leftSum.join() + rightSum.compute());//If you call like this the code won’t benefit from the fork/join framework. Code execute normally.

*12. Localization*