*OCP Java SE 7 Programmer II*

*7. Java I/O fundamentals*

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

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

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

You can create File object in following 3 ways.

File(String pathname)

File(File parent, String child)

File(String parent, String child)

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

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

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

***Input streams***

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

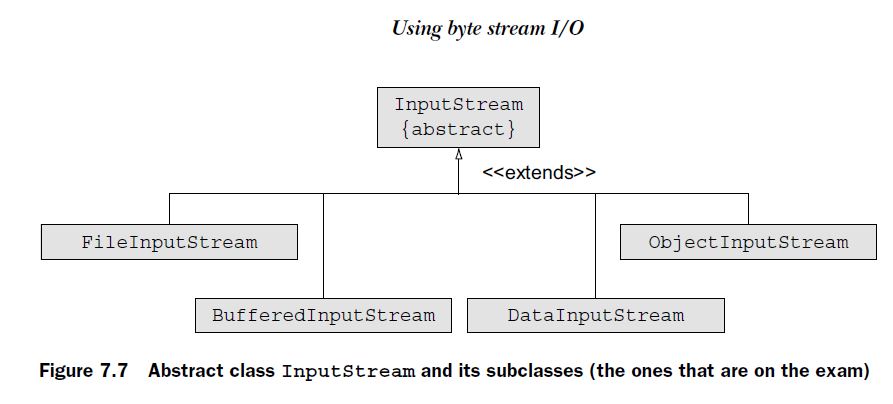
int abstract read()

int read(byte[] b)

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

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

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



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

***Output streams***

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

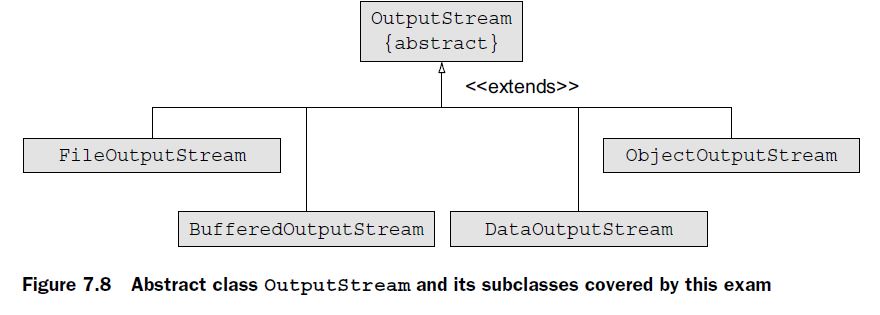
abstract void write(int b)

void write(byte[] b)

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

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

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



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

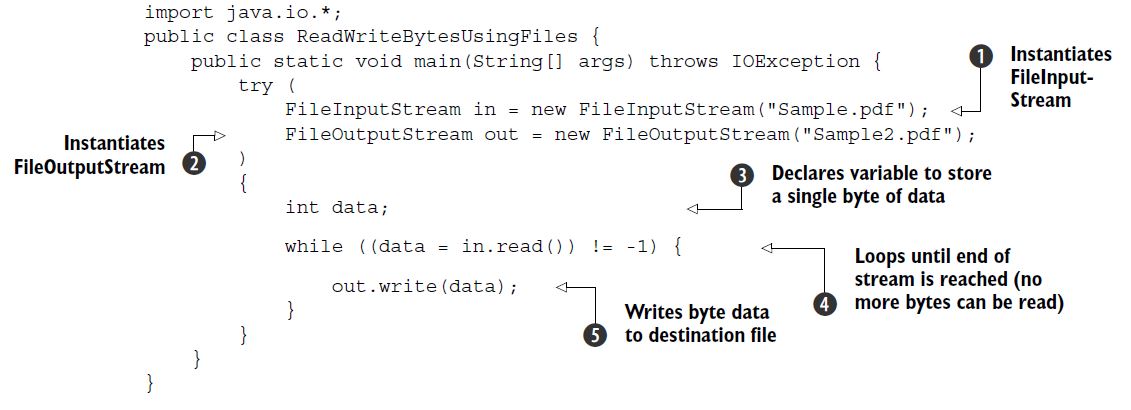
FileOutputStream(File file) throws FileNotFoundException

FileOutputStream(File file, boolean append) throws FileNotFoundException

FileOutputStream(String name) throws FileNotFoundException

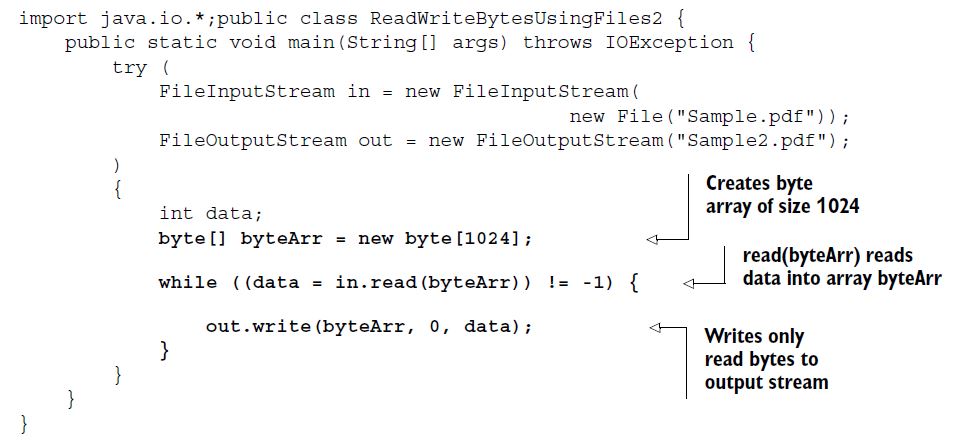
FileOutputStream(String nm, boolean append) throws FileNotFoundException

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



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

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



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

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

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

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

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

public BufferedInputStream(InputStream in)

public BufferedInputStream(InputStream in, int size)

public BufferedOutputStream(OutputStream out)

public BufferedOutputStream(OutputStream out, int size)

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

Disadvantages:

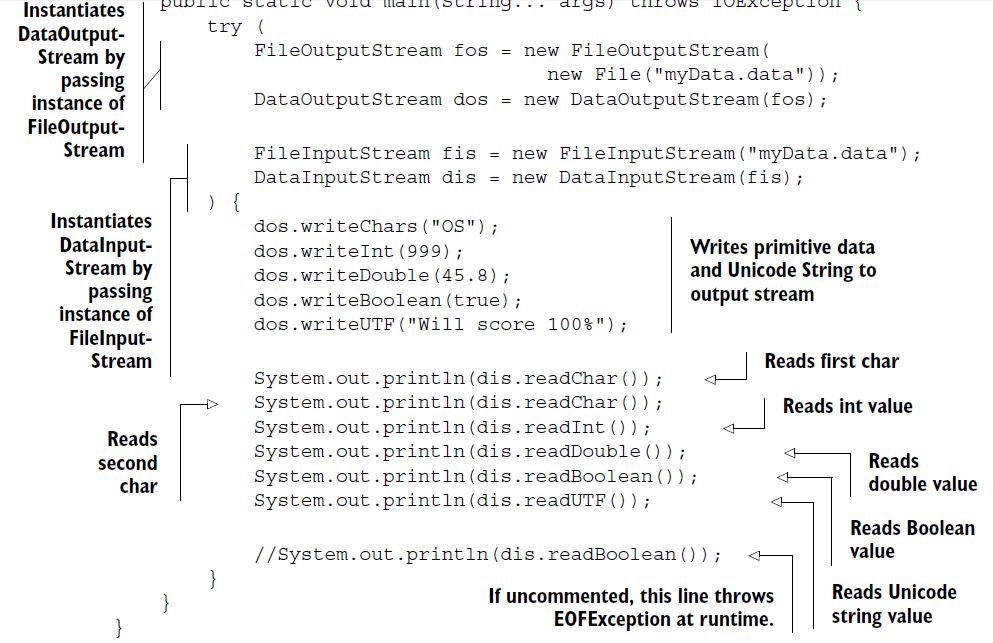
You can use FileInputStream and FileOutputStream to read and write *only* byte data they didn’t define methods to work with any other specific primitive data types or objects, which is what you might need most of the time.

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

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

DataInputStream(InputStream in)

DataOutputStream(OutputStream out)



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

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

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

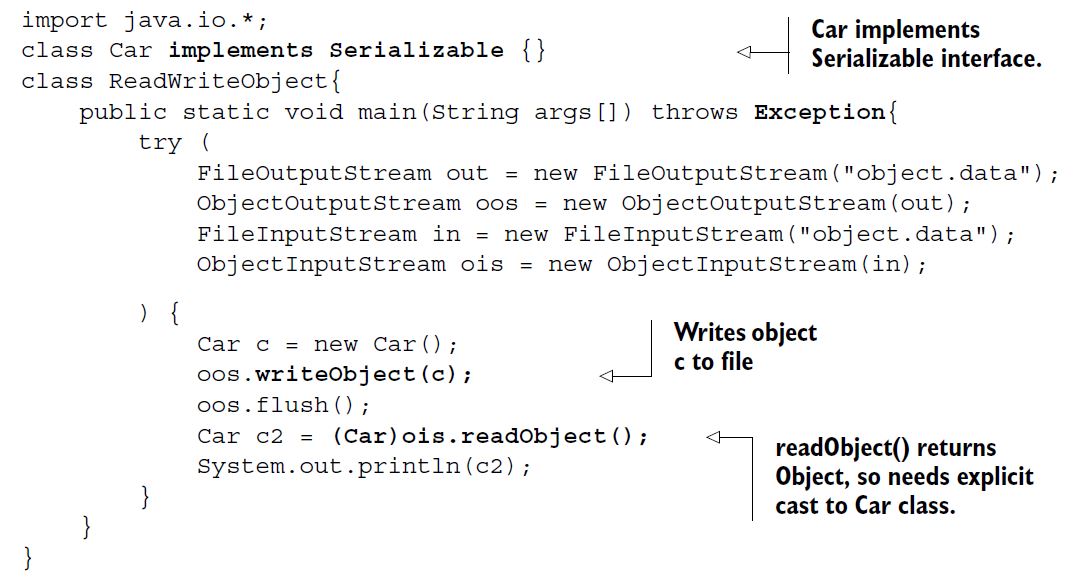
You can use classes ObjectInputStream and ObjectOutputStream to read and write objects *and* primitive values.

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

public ObjectInputStream(InputStream in)

public ObjectOutputStream(OutputStream out)

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



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

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

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

READ AND WRITE OBJECTS WITH NONSERIALIZABLE PARENT CLASSES

**class Vehicle** {

String name = "Vehicle";

}

**class Car extends Vehicle implements Serializable** {

String model = "Luxury";

}

class ParentNotSerializable{

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

try (

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

ObjectOutputStream oos = new ObjectOutputStream(out);

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

ObjectInputStream ois = new ObjectInputStream(in);

) {

Car c = new Car();

oos.writeObject(c);

oos.flush();

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

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

}

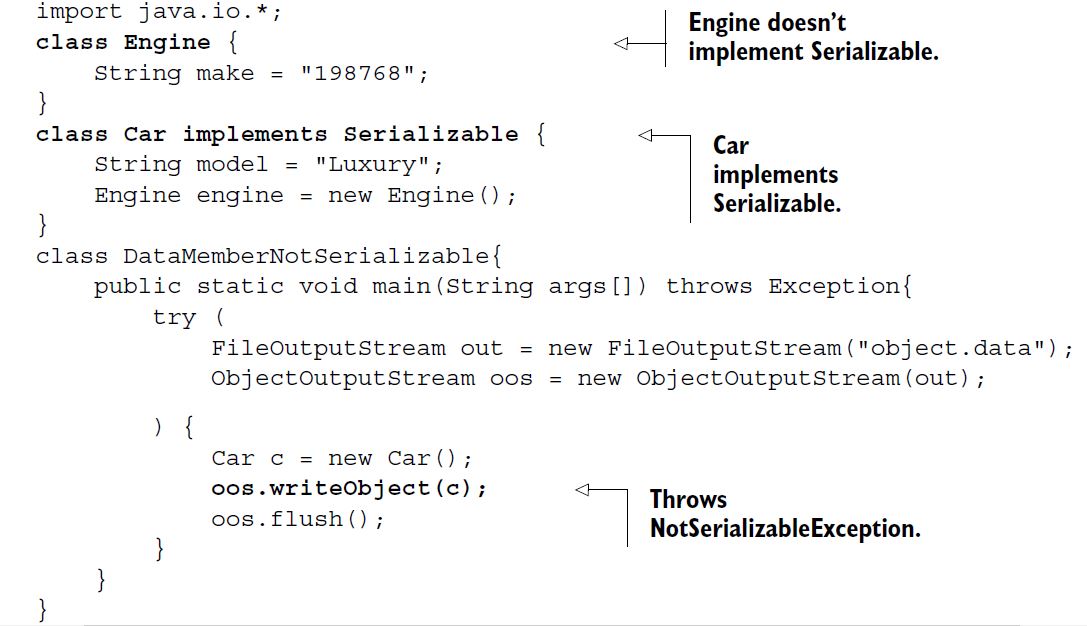
}

}

READ AND WRITE OBJECTS WITH NONSERIALIZABLE DATA MEMBERS

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

For example:



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

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

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

try (

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

ObjectInputStream ois = new ObjectInputStream(in);

) {

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

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

**can throw OptionalDataException**

System.out.println(c.name);

}

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

Method readObject() can throw multiple exceptions:

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

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

class Car implements Serializable{

String name;

**transient String model;**

**transient int days;**

**static int carCount;**

Car(String value) {

name = value;

**model = "some value";**

**days = 98;**

++carCount;

}

}

class ReadWriteCarObjects{

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

try (

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

ObjectOutputStream oos = new ObjectOutputStream(out);

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

ObjectInputStream ois = new ObjectInputStream(in);

) {

Car c = new Car("AAA");

oos.writeObject(c);

oos.flush();

new Car("BBB");

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

System.out.println(c2.name);

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

System.out.println(c2.carCount);

}

}

}

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

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

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

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

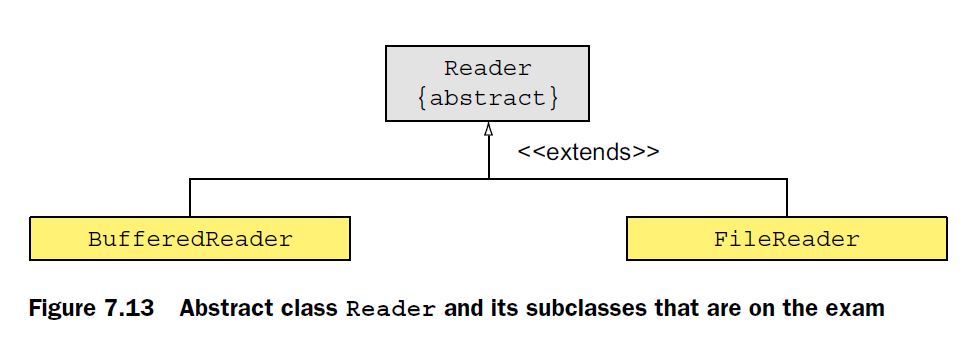
Class Reader defines overloaded read() methods to read character data from an underlying data stream:

int read()

int read(char[] cbuf)

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

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



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

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

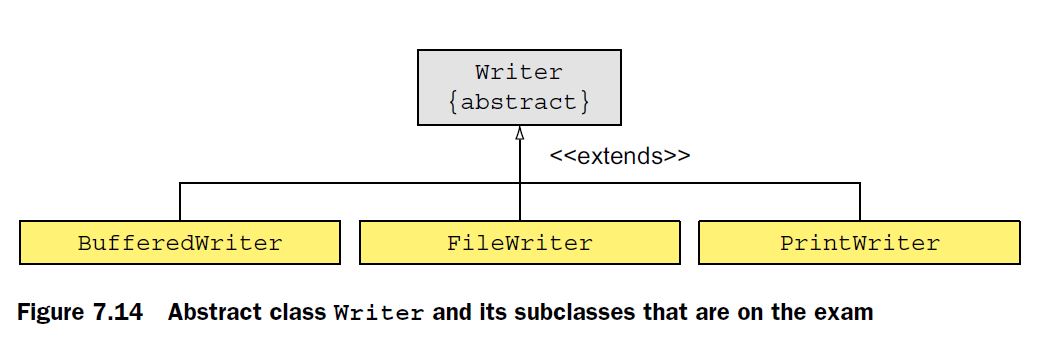
void write(char[] cbuf)

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

void write(int c)

void write(String str)

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



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

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

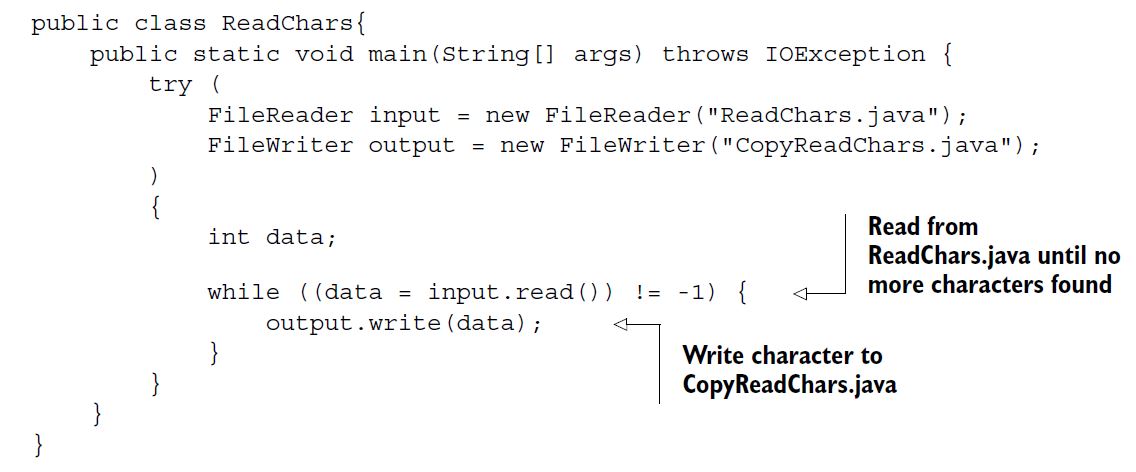
FileReader and FileWriter are convenience classes for reading and writing character data from files.

FileWriter(File file)

FileWriter(File file, boolean append)

FileWriter(String fileName)

FileWriter(String fileName, boolean append)



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

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

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

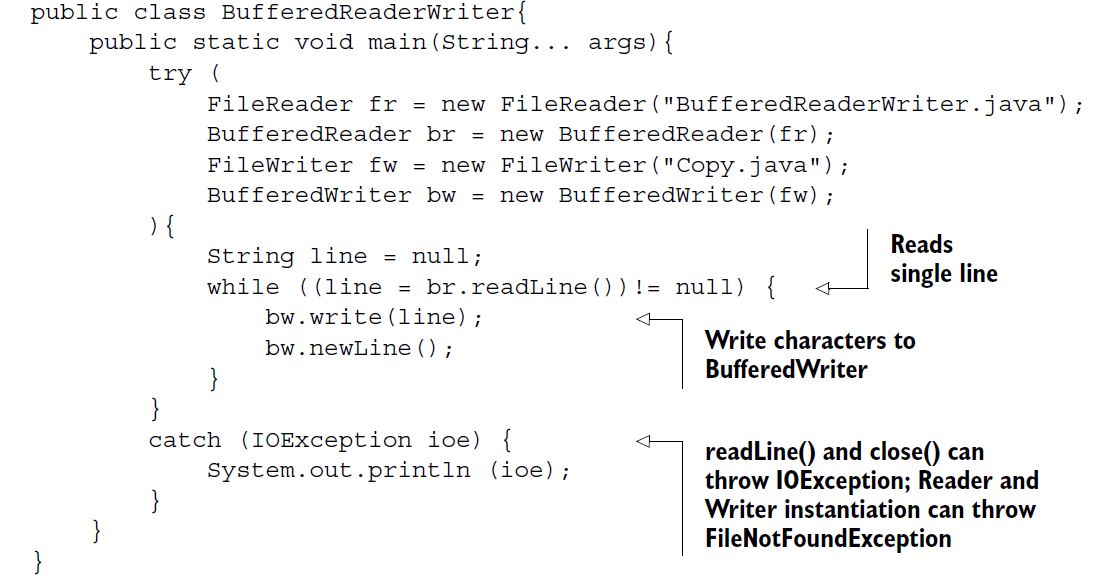
public BufferedReader(Reader in)

public BufferedReader(Reader in, int sz)

public BufferedWriter(Writer out)

public BufferedWriter(Writer out, int sz)

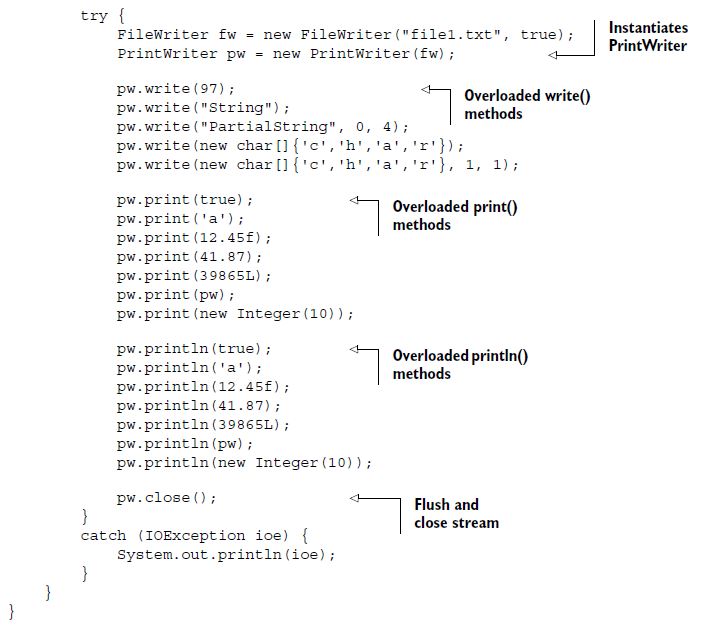
EXAM TIP The exam might test you on how to instantiate buffered character streams correctly. To instantiate BufferedReader, you must pass it an object of Reader. To instantiate BufferedWriter, you must pass it an object of Writer.



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

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

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



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

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

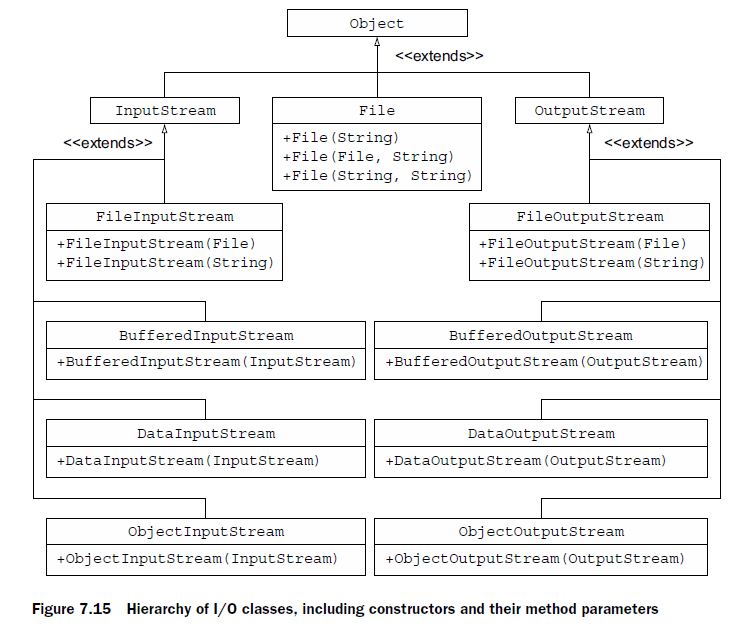
PrintWriter(File file)

PrintWriter(File file, String charset)

PrintWriter(String fileName)

PrintWriter(String fileName, String charset)

PrintWriter(Writer out, boolean autoFlush)



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)*

*9. Building database applications with JDBC*

*10. Threads*

*11. Concurrency*

*12. Localization*