# Java I/O (entrada/salida)

## Introducción

En este capítulo te presentaremos los conceptos fundamentales de la programación i/o. Vamos a cubrir dos temas: cómo leer y escribir datos desde la consola y como usar streams para escribir y leer datos.

Programar con i/o requiere escribir algo de manejo de excepciones.

El soporte para manejo de ficheros está provisto por los paquetes java.io y java.nio. En la parte inicial de este capítulo nos enfocaremos solo en el primer paquete, después nos enfocaremos en leer y escribir datos usando streams.

Temas

* Leer y escribir datos desde la consola
* Usar streams para leer y escribir archivos

## Leer y escribir usando la consola

### Entendiendo la clase Console

Usando la clase Console (a partir de java 1.6) simplifica considerablemente leer los datos de la consola y escribirlos en la misma. Nota que la palabra “console” hace referencia al dispositivo de entrada de caracteres (típicamente el teclado) y el dispositivo para mostrar caracteres (típicamente el monitor). Puedes obtener una referencia a la consola usando el método System.console(); si la máquina virtual de java no está asociada con alguna consola este método retorna null.

El primer ejercicio es implementar un simple comando echo que imprima de regreso el texto ingresado como entrada.

class Echo {

public static void main(String... args) {

Console console = System.console();

if(console == null ) {

System.err.println(“No se creó la consola... terminando aplicación...”);

System.exit(-1);

}

console.printf(console.readLine());

}

}

Para una entrada de texto normal este programa trabaja bien. Si escribes en la entrada de texto un intento por terminar la aplicación (Ctrl + Z o Ctrl + D) el programa no recibe entrada, por lo que el método readLine() retorna null; cuando el método printf() toma un argumento null, este lanzará una excepción NullReferenceException.

Nota que ejecutaste el programa desde la línea de comandos. El método console() será exitoso si la JVM está invocada desde línea de comandos sin redireccionar las entradas o salidas ya que la JVM será asociada a la consola. Si la JVM es invocada indirectamente por un IDE, o si la JVM está invocada por un proceso en segundo plano, entonces el método System.console() fallará y retornará null.

En la siguiente tabla observaremos algunos métodos importantes de la clase Console.

|  |  |
| --- | --- |
| Método | Descripción corta |
| Reader reader() | Retorna el objeto Reader asociado con este objeto Console; puede ejecutar operaciones de lectura con esta referencia |
| PrintWriter writer() | Retorna el objeto PrintWriter asociado con este objeto Console; puede ejecutar operaciones de escritura con esta referencia |
| String readLine() | Lee una línea de texto y retorna un objeto String que no incluye ningún carácter de terminación de línea; retorna null si este falla |
| String readLine(String fmt, Object... args) | Igual que el método anterior pero primero imprime la String fmt |
| char[] readPassword() | Lee un texto password y lo retorna como un array de caracteres; echoing está deshabilitado en este método por lo que no se imprimirá nada en la consola cuando el usuario escriba el password |
| char[] readPassword(String fmt, Object... args) | Igual que el método anterior pero primero imprime la cadena que da el string de formato antes de leer la entrada del usuario |
| Console format(String fmt, Object... args) | Escribe la cadena formateada a la consola |
| Console printf(String fmt, Object... args) | Escribe la cadena formateada a la consola. Este método es el mismo eque el anterior: Este es un método conveniente ya que es más conocido para porgramadores C/C++ |
| void flush() | Vacía cualquier dato contenido que será impreso en el buffer del objeto Console |

### Entrada y salida formateada con la clase Console

La clase Console soporta entrada/salida formateada en los métodos printf() y format(), más los métodos sobrecargados readPassword() y readLine(). No cubriremos los métodos pritnf() y format() en este capítulo ya que fueron cubiertos en el capítulo 7.

En los métodos readPassword() y readLine(), el primer argumento es el especificador de formato, y los argumentos siguientes son los valores que se pasan al especificador de formato. Estos dos métodos retornan los datos de caracteres leídos de la consola. ¿Cuál es la diferencia entre estos dos métodos? La diferencia principal es que el método readPassword() no muestra la cadena escrita en la consola mientras que readLine() sí. Otra diferencia menor es que readLine() retorna un String y readPassword retorna un arreglo de caracteres. Veamos el siguiente código.

import java.io.Console;

import java.util.Arrays;

class Login {

public static void main(String... args) {

Console console = System.console();

if(console ¡= null) {

String userName = null;

char[] password = null;

userName = console.readLine(“Enter your username: ”);

password = console.readPassword(“Enter password: ”);

if(userName.equals(“scrat”) && new String(password).equals(“nuts”)) {

console.printf(“login successful!!!”);

}

else {

console.printf(“restart application an try again”);

}

Arrays.fill(password, ‘ ’);

}

}

}

Nota que no se muestra el texto escrito en la consola cuando se escribe el password. ¿Por qué se usa Arrays.fill(password, ‘ ’) en el programa? Esta es una práctica recomendada para vaciar el password leído ya que se terminó de usar; aquí usas el método fill() para este propósito. Esta es una práctica segura para evitar lecturas maliciosas del programa para descubrir las cadenas de password. De hecho, a diferencia de readLine(), que retorna un String, el método readPassword() retorna un arreglo de caracteres. Con un arreglo de caracteres, tan pronto como el password sea validado, es posible vaciarlo y remover el rastro del password de la memoria; con un objeto String, que es vaciado hasta que lo retira el garbage collector.

### Manejo de caracteres especiales en la clase Console

Escribir texto a través de los métodos printf() o format() tienen la ventaja de que estos métodos manejan caracteres especiales mejor que a través de printStream (Discutiremos Streams en la siguiente sección). Veamos un ejemplo.

import java.io.Console;

public class SpecialCharHandling {

public static void main(String... args) {

// String con tres caracteres escandinavos

Stirng scandString = “å, ä, á and ö”;

// intentar escribir con println

System.out.println(scandString);

// ahora con un objeto console

Console console = System.console();

console.printf(scandString);

}

}

Como puedes ver en la salida, la impresión desde printf() es mejor

## Usar corrientes (streams) para escribir y leer archivos

¿Qué son los Streams? Los streams son secuencias ordenadas de datos. Java trata con la entrada y la salida en términos de streams. Por ejemplo, cuando lees una secuencia de bytes de un archivo binario, estás leyendo de un stream de entrada; similarmente, cuando escribes una secuencia de bytes hacia un archivo binario, estás escribiendo a un stream de salida. Nota cómo referenciamos a leer o escribir bytes de/hacia archivos binarios, pero ¿Qué pasa con leer o escribir caracteres en archivos de texto? Java diferencia entre procesar texto y datos binarios. Antes de profundizar en los streams y leer o escribir datos en archivos, necesitas entender primero la diferencia entre streams de caracteres y de bytes, que es esencial para entender el resto del capítulo.

### Streams de caracteres y streams de bytes

Considera la diferencia entre archivos fuente de Java y archivos class generados por el compilador. Los archivos fuente tienen extensión .java y es posible que los humanos puedan leerlos como por compiladores. Sin embargo, los archivos .class no pueden ser leídos por los humanos; esto significa que serán procesados por herramientas de bajo nivel como la JVM o un desensablador (javap.exe en Windows). Nos referiremos a archivos humanamente-legibles que contienen texto (o caracteres) como archivos de texto; y nos referiremos a los archivos solo leídos por la máquina o de datos de bajo nivel como archivos binarios. Naturalmente, como tú interpretas que está dentro de archivos de texto contra archivos binarios es diferente. Por ejemplo, en los archivos de texto, puedes interpretar los datos leídos del archivo y diferenciar entre un carácter tabulador, espacio en blanco, carácter de nueva línea, etc. Sin embargo, no puedes tratar con datos en archivos binarios como esto; aquí hay solo valores de bajo nivel. Para dar otro ejemplo, considera un archivo txt que creas con un editor de texto como block de notas; este contiene contenido humanamente legible. Ahora considera guardar una foto en archivo bmp o jpg; estos archivos no son humanamente legibles. Estos archivos están diseñados para ser procesados por un editor de imágenes y los archivos contienen datos en algún predeterminado lenguaje de bajo nivel.

El paquete java.io tiene clases que soportan ambos: streams de caracteres y streams de bytes. Puedes usar streams de caracteres para operaciones i/o basadas en texto. Streams de bytes son usados para operaciones i/o basadas en datos. Los streams de caracteres para leer y escribir son llamados “readers” y “writers” respectivamente (representados por las clases abstractas Readre y Writer). Los streams de bytes para leer y escribir son llamados “input streams” y “output streams” respectivamente (representados por las clases abstractas InputStream y OutputStream). La siguiente tabla muestra las diferencias entre los streams de caracteres y los streams de bytes.

|  |  |
| --- | --- |
| Streams de caracteres | Streams de bytes |
| Significan para lectura o escritura de archivos basados en caracteres o texto, documentos de texto, XML y HTML | Significan para escribir o leer de archivos binarios como archivos ejecutables, imágenes y archivos en formatos de bajo nivel como .zip .class .obj y .exe |
| Los datos son tratados con los caracteres Unicode de 16-bit | Los datos son tratados como sus bytes (8-bit, 16-bit) |
| La entrada y la salida de caracteres son llamados readers and writers respectivamente | La entrada y la salida de bytes son llamados input streams y output streams |
| Las clases abstractas de Reader y Writer y sus clases derivadas en el paquete java.io proveen soporte para flujos de caracteres | Las clases abstractas de InputStream y OutputStream y sus clases derivadas en el paquete java.io proveen soporte para flujos de bytes |

NOTA: Si intentas usar streams de bytes cuando un stream de caracteres es necesitado o viceversa, obtendrás una sucia sorpresa en tus programas. Por ejemplo, un mapa de bits bmp debe ser procesado usando un stream de bytes; si intentas usar un stream de caracteres, tu programa no funcionará. Por lo que no mezcles los streams.

#### Streams de caracteres

En esta sección explorarás las operaciones i/o con streams de caracteres. Aprenderás como leer y escribir en filas de texto más algunas características opcionales como buffernig para acelerar las operaciones i/o. Para leer y escribir archivos de texto, puedes usar las clases derivadas de las clases abstractas Reader y Writer respectivamente. Observemos las siguientes gráficas que contienen importantes clases y sus descripciones.



|  |  |
| --- | --- |
| Clase | Descripción corta |
| StringReader | Un stream de caracteres que opera con Strings |
| InputStreamReader | Esta clase es un puente entre streams de caracteres y streams de bytes |
| FileReader | Clase derivada de InputStreamReader que provee soporte para leer archivos de caracteres |
| PipedReader | Las clases PipedReader y PipedWriter forman un par de lectoras escritoras de caracteres entubadas |
| FilterReader | Clase abstracta base para streams que soportan operaciones de filtrado aplicadas sobre los datos como caracteres que son leídos de los streams |
| PushbackReader | Clase derivada de FilterReader que permite leer caracteres para ser regresados atrás al stream |
| BufferedReader | Agrega amortiguación al flujo de caracteres subyacentes y no necesita del sistema de archivos para cada operación de lectura o escritura |
| LineNumberReader | Clase derivada de BufferedReader que mantiene la trayectoria de los números de línea como de los caracteres son leídos del stream |



|  |  |
| --- | --- |
| StringWriter | Stream de caracteres que colecta la salida en un buffer de string, el cual puede ser usado para crear un string |
| OutputStreamWriter | Esta clase es un puente entre los streams de caracteres y los de bytes |
| FileWriter | Clase derivada de OutputStreamWriter que provee soporte para escribir archivos de caracteres |
| PipedWriter | Las clases PipedReader y PipedWriter forman un par para entubar la lectura y escritura en streams de caracteres |
| FilterWriter | Clase abstracta base para streams que soporta operación de filtrado aplicado a los datos como caracteres cuando son escritos al stream |
| PrintWriter | Soporta impresión formateada de caracteres al stream de salida |
| BufferedWriter | Agrega amortiguación al stream de caracteres por lo que no es necesario leer o escribir en el sistema de archivos subyacente, por cada operación de lectura o escritura |

##### Leer archivos de texto

Las clases Reader leen los contenidos en el stream e intentan interpretarlos como caracteres, como tab, fin-de-fila, nueva línea, etc. El siguiente código implementa una versión simplificada del comando type de Windows. El comando type muestra el contenido del archivo pasado como parámetro en los argumentos de la línea de comandos.

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

public class Type {

public static void main(String[] files) {

if(files.length == 0) {

System.err.println(“pasa el nombre de las filas como argumento”);

System.exit(-1);

}

for(String file: files) {

try(FileReader inputFile = new FileReader(file)) {

int ch = 0;

while...

}

}

}

} // este ejemplo está incompleto, revisar los ejercicios en código del folder

Este ejemplo trabaja como se espera. En este programa estás instanciando la clase FileReader y pasando el nombre del archivo a ser leído al constructor. Si el archivo no se encuentra lanzará una excepción FileNotFoundException.

Una vez que el archivo está abierto, usas el método read() para obtener los caracteres del archivo subyacente. Estás leyendo carácter por carácter. Alternativamente puedes usar métodos como readLine() para leer línea por línea.

Nota que el método read() retorna un entero en lugar de un carácter. Esto es porque cuando el método alcanza el final del archivo (EOF), este retorna -1, que está fuera del rango de char. Entonces este método retorna un entero para indicar cuando se alcanza el final del archivo y, por lo tanto, tienes que detener los intentos por leer cualquier carácter más del stream subyacente.

En este programa tú solo lees un archivo de texto; ahora intentarás escribir también.

##### Leer y escribir archivos de texto

FileReader inputFile = new FileReader(file);

En el ejemplo previo creas un stream de caracteres como sigue:

Este usa una entrada y salida no amortiguada, lo cual es menos eficiente cuando comparas a entrada/salida amortiguada. En otras palabras, la lectura de caracteres es directamente pasada en lugar de usar un buffer temporal interno, que aumenta las operaciones I/O. Para programáticamente usar I/O amortiguada puedes pasar a la referencia FileReader a un objeto BufferedReader como sigue:

BufferedReader inputFile = new BufferedReader(new FileReader(file));

Del mismo modo puedes usar también BufferedWriter para una salida amortiguada. (En caso de streams de bytes puedes usar BufferedInputStream y BufferedOutputStream, que discutiremos después en este capítulo).

Ahora usarás I/O amortiguada para leer y escribir un archivo de texto. El siguiente código es una versión simplificada del comando copy en Windows.

[class Copy](src/java/oca2/cap08/Copy.java)

Veamos cómo funciona este programa. Quieres usar I/O amortiguado entonces pasas referencias FileReader y FileWriter a BufferReader y BufferWriter respectivamente.

try (BufferedReader inputFile = new BufferedReader(new FileReader(srcFile)); BuferedWriter outputFile = new BufferedWriter(new FileWriter(dstFile)))

Usas try-with-resources, y el método close() primero llamará al método flush() antes de cerrar el stream. Cuando usas I/O amortiguado es buena idea llamar al método flush() explícitamente en algunos lugares cuando quieras asegurar que los caracteres pendientes sean vaciados.

##### Tokenizar texto

En los últimos dos ejemplos tú solo lees o escribes en archivos de texto. Sin embargo, en programas reales puedes necesitar ejecutar algún proceso cuando lees o escribes archivos. Por ejemplo, puedes querer detectar algunos patrones, buscar cadenas específicas, reemplazar una secuencia de caracteres con otra, filtrar palabras específicas, formatear la salida de algún modo. etc. Puedes usar las APIs existentes como regex, Scanner, etc. para estos propósitos.

Para ilustrarnos, considera que quieres listar todas las palabras un archivo de texto dado y eliminar todos los espacios en blanco innecesarios, caracteres de puntuación, etc. También necesitar imprimir las palabras resultantes en orden alfabético. Para solventar este problema puedes usar un Scanner y pasar la regex que quieres coincidir o delimitar. Observa el siguiente código:

[Tokenize.java](src/java/oca2/cap08/Tokenize.java)

El programa trabaja correctamente. Ahora vamos a ver qué hace este programa. El programa primero abre el archivo usando un FileReader y lo pasa a un objeto Scanner. El programa ajusta el delimitador con el método useDelimiter(“\\W”) de la clase Scanner; el “\W” coincide para todas las no-palabras, entonces cualquier carácter que no sirva para palabras se vuelven delimitadores.

Nota que ajustaste el delimitador y no el patrón que querías coincidir. El programa hace uso de un TreeSet<String> para almacenar las cadenas leídas. El programa lee palabras del stream subyacente, checa si hay cadenas no vacías y las agrega en minúscula al TreeSet. Ya que la estructura de datos es un TreeSet, este remueve los duplicados; recuerda que es un Set, el cual no permite duplicados. Además, también ordena los datos alfabéticamente. Por lo tanto el porgrama imprime correctamente las palabras del archivo.

#### Streams de Bytes

En esta sección explorarás I/O con streams de bytes. Primero aprenderás cómo leer y escribir archivos de datos y también como escribir objetos, almacenarlos en archivos y leerlos de regreso. La clase OutputStream, InputStream y sus clases derivadas se muestran en las siguientes figuras:





La siguiente tabla explica las clases más importantes de InputStream y OutputStream

|  |  |
| --- | --- |
| Nombre de la clase | Descripción corta |
| PipedInputStream, PipedOutputStream | Crean un canal de comunicación donde los datos pueden ser enviados y recibidos. |
| FileInputStream, FileOutputStream | Recibe un stream de bytes de un archivo; Envía un stream de bytes a un archivo. |
| FilterInputStream, FilterOutputStream | Estos streams filtrados son usados para agregar funcionalidades a streams planos. La salida de un InputStream puede ser filtrada usando FilterInputStream. La salida de un OutputStream puede ser filtrada usando un FilterOutputStream |
| BufferedInputStream, BufferedOutputStream | Agregan amortiguación a la entrada y salida de streams |
| PushbackInputStream | Subclase de FilterInputStream agrega funcionalidad para hacer retroceder algún flujo de entrada |
| DataInputStream, DataOutputStream | Para leer y escribir tipos de datos primitivos a un stream |

##### Leyendo streams de bytes

Los streams de bytes son usados para procesar archivos que no contienen texto humanamente legible. Por ejemplo, un archivo fuente de Java tiene contenido legible para los humanos, pero un archivo .class no. Un archivo .class está diseñando para ser procesado por la JVM, por lo tanto necesitas usar streams de bytes para procesar este tipo de archivos.

El contenido de un archivo .class está escrito en un formato específico, descrito en las especificaciones de la JVM. No te preocupes; tú no tienes que entender este complejo formato de archivo, pero sí checarás el “número mágico”. Cada formato de archivo tiene un número mágico usado para checar rápidamente el formato del archivo. Por ejemplo “.MZ” es el número mágico (o más adecuado, cadena mágica) para archivos .exe en Windows. Similarmente, los archivos .class tienen un número mágico “0xCAFEBABE”, escrito en valor hexadecimal. Estos números mágicos son escritos típicamente como los primeros bytes de la variable de tamaño en el formato de fila.

Para entender cómo los streams de bytes trabajan, ahora checarás los números mágicos de las filas dadas. Observa el siguiente código.

[MagicNumberCheck.java](src/java/oca2/cap08/MagicNumberCheck.java)

Sí, funciona. Las clases InputStream y OutputStream forman la base de la jerarquía de los streams de bytes. Puedes ejecutar operaciones I/O en archivos, para lo cual abres el archivo dado como un FileInputStream. Necesitas checar los primeros cuatro bytes, por lo que lees cuatro bytes en un buffer temporal. Necesitas comparar el contenido de este buffer con la secuencia de caracteres de bytes 0xCA, 0xFE, 0xBA, 0xBe. Si el contenido de estos dos arrays no son iguales, entonces el archivo pasado no es un archivo .class.

En este programa manipulas directamente el stream de bytes subyacente usando un FileInputStream. En el caso que necesitaras acelerar el programa cuando lees un gran número de bytes, puedes usar un stream de salida amortiguado como sigue:

BufferedInputStream bis = new BufferedInputStream(new FileInputStream(fileName));

Similarmente a estos streams de entrada, puedes usar streams de salida para escribir una secuencia de bytes a un archivo. Puedes usar FileOutputStream y BufferedOutputStream.

Después de leer este programa, ¿puedes pensar que leer un array de cuatro bytes y comparar los contenidos de los arrays es torpe; en lugar de comparar directamente los valores enteros? En otras palabras, 0xCAFEBABE es un valor entero y podrías leer este valor directamente como un valor entero y compararlo con el valor leído. Para esto, necesitas usar streams de datos, que proveen métodos como readInt() que discutiremos ahora.

##### Streams de datos

Para entender cómo escribir o leer con streams de datos, vamos a escribir un programa simple que escribe y lee valores constantes de un archivo de datos. Para mantener el problema simple, escribirás solo los valores del 0 al 9 en la forma de los siguientes tipos primitivos: byte, short, int, long y double.

[DataStreamExample.java](src/java/oca2/cap08/DataStreamExample.java)

Funciona. Hora, como mencioné anteriormente, el contenido de los archivos de datos no son humanamente legibles. En este caso estás escribiendo valores como los enteros 0 o 9 y estos son almacenados en términos de bytes. Sin embargo, el comando type de Windows intenta convertir esos bytes en caracteres legibles, sin embargo la salida no tiene sentido. Los datos tendrán sentido solo si conocemos el formato de los datos almacenados en el archivo y los leemos de acuerdo a ese formato. El comando type mostrará los valores como basura por lo mismo.

Ahora regresemos al programa y veamos cómo funciona. El programa escribe al archivo hard – codeado llamado temp.data en el directorio donde el programa está corriendo. Este programa primero escribe los datos, para lo que abre el archivo como un OutputStream. ¿Qué significa la siguiente sentencia en el primer bloque try?

DataOutputStream dos = new DataOutputStream(new FileOutputStream(“temp.data”));

Puedes ejecutar directamente escritura y lectura binaria con OutputStream y sus clases derivadas FileOutputStream, pero para procesar tipos de dato formateados como primitivos necesitas usar la clase DataOutputStream, que funciona como un envoltorio sobre el FileOutputStream subyacente. Por lo tanto usas DataOutputStream en el try con recursos, por lo tanto el método close() será invocado automáticamente cuando el Stream sea desocupado. El método close() también vacía el stream; este método también cierra la referencia a FileOutputStream.

Ya que el archivo fue escrito, lees los datos del stream del mismo modo. Abres un FileInputStream y lo envuelves con un DataInputStream. Lees los datos del stream y los imprimes en la consola, con el método printf(). En este programa escribiste y leíste datos primitivos. ¿Qué pasa con las referencias a objetos tales como Objects, Maps, etc.? Leer y escribir objetos se logra a través de streams de objetos, que discutiremos ahora.

##### Escribir y leer de streams de objetos: Serialization

Las clases ObjectInputStream y ObjectOutputStream soportan lectura y escritura de objetos java que usas en el programa. Por ejemplo, si estás creando un sitio web de comercio electrónico para hacer compras, puedes escoger escribir los objetos como los proveedores, órdenes de compra hechas, etc. a una Base de Datos (que se discutirá en el capítulo 10), o alternativamente almacenar los objetos directamente en un archivo plano. En este caso, debes saber cómo escribir y leer objetos en streams.

Vamos a presentarte algunos términos relacionados a este tema antes de continuar. El proceso de convertir objetos en memoria en una secuencia de bytes es conocido como serialization. Ek mecanismo para almacenar objetos en memoria en archivos es conocido como persistencia. Aunque estos dos términos son conocidos como serialization únicamente.

El siguiente código contiene un ejemplo simple de escribir los contenidos de una estructura de datos Map a un archivo y leerlos de regreso para ilustrar el uso de las clases ObjectOutputStream y ObjectInputStream para leer y escribir objetos. Almacenarás los datos de los últimos tres presidentes de EU en este Map.

[ObjectStreamExample.java](src/java/oca2/cap08/ObjectStreamExample.java)

El proceso de serialización convierte los contenidos de los objetos en memoria con la descripción del contenido (conocido como metadatos). Cuando el objeto tiene una referencia a otros objetos, el mecanismo de serialización también los incluye como parte de los bytes serializados. Si intentas abrir el archivo donde el objeto es almacenado, no puedes leer estos objetos serializados. Por ejemplo, si intentas leer el archivo object.data, observarás caracteres ilegibles.

Ahora regresemos al programa y veamos cómo funciona. En este programa llenas el HashMap con detalles de los últimos tres presidentes de EU. Entonces abres un output stream como sigue:

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(“object.data”));

El FileOutputStream abre un archivo temporal llamado object.data en el directorio actual. El ObjectOutputStream es un envoltorio sobre el subyacente FileOutputStream. Dentro del try-con-recursos solo tienes una sola sentencia oos.writeObject(pMap), que escribe el objeto al archivo object.data.

Leer el objeto requiere un poco más de trabajo que escribirlo. El método readObject() en ObjectInputStream retorna un tipo Object. Necesitas convertirlo de regreso a Map<String, String>. Antes del downcast a Map<String, String> primero checas si el objeto es de tipo Map. Nota que tú no tienes que checar si es Map<String, String> porque estos tipos genéricos se pierden con el proceso “type erasure”. Ya que el downcast es exitoso, lees los valores de contenidos en este objeto.

##### Serialization: más detalles

Esto es relevante para nosotros para elaborar más en el tema de serialization. Como se ilustró en la sección anterior, serialization es un proceso para convertir un objeto en una secuencia de bytes. Puedes escribir un objeto serializado a un archivo como lo hiciste en el ejemplo anterior o lo puedes colocar en un socket para enviarlo a través de la red.

El último ejemplo ilustra cómo escribir objetos a streams; que no es más que serialización con persistencia. En el último ejemplo creaste una instancia de HashMap y después la serializaste y deserializaste. ¿Qué pasa si tú quieres serializar un objeto de una clase que tú creaste? Bueno, puedes serializar objetos de todas las clases que implementen la interface Serializable. En otras palabras, una clase no es serializable por default; tienes que implementar la interface Serializable para hacerla serializable. En el último ejemplo, el HashMap implementa esta interface.

Ahora asume que quieres serializar un objeto que contiene un no-serializable miembro de clase (digamos un Thread o un Socket). O imagina una situación donde no quieres serializar una variable miembro. Para estas situaciones, Java ofrece la palabra clave “transient”. Puedes declarar una variable miembro como transient y esta variable no será serializada por la JVM. Vamos a ver un ejemplo para entender esto mejor. Asume que tienes una clase USPresident que almacena el nombre, periodo, y el término de oficio del presidente. Quieres serializar el objeto de esta clase, por lo que implementa la interface Serializable. Sin embargo, no quieres serializar un campo, digamos termino de oficio term. El siguiente código muestra cómo se logra esto:

[TransientSerialization.java](src/java/oca2/cap08/TransientSerialization.java)

Este programa es muy simple. Primero creas una instancia de USPresident con todos los campos requeridos. Después imprimes el contenido del objeto. Después de esto serializas el objeto y después lo des-serializas. Imprimes el contenido del objeto nuevamente. Observas de la salida es que el valor del term no está almacenado por el programa ¿Por qué? Porque lo declaraste como transient. Todas los miembros de la clase declarados como transient no serán serializados, por lo que sus valores son perdidos en el proceso de des-serialización.

Una cosa más requiere atención aquí: serialVersionUID. En este ejemplo este es ajustado a 1. Si estás implementando Serializable y no defines serialVersionUID, obtendrás un mensaje de advertencia. De hecho, si no lo defines la JVM lo define por ti; JVM lo calcula en base del comportamiento de la clase. Pero ¿Por qué este es requerido? Bueno, esto es para prevenir la carga de una versión errónea de la clase cuando se des-serializa. También, definir serialVersionUID habilita al programa serializado a través de diferentes JVM de la misma manera (lo que no es el caso cuando no lo defines explícitamente). En la última línea: siempre que hagas un cambio en una clase serializada, no olvides cambiar el serialVersionUID.

Puntos a recordar

* Cuando usas streams amortiguados, debes de llamar a flush() siempre que quieras una buena transmisión de datos. El buffer interno es posible que mantenga algo de datos, la cual será limpiada y enviada al destino siempre que llames a flush(). Sin embargo, el método close() en el stream automáticamente llamará al método flush().
* Es posible que ya hayas observado que combinas objetos stream. Puedes crear un objeto de BufferedInputStream que toma un FileInputStream en el constructor. De este modo, la salida de un stream es cambiado al stream filtrado. Esto es lo importante, útil y buen camino para customizar el stream en un camino específico.
* La interface Serializable es una interface de marca. Esto significa que la interface Serializable no declara ningún método dentro de ella.
* Si quieres personalizar el proceso de serialización, puedes implementar readObject() y writeObject(). Nota que estos dos métodos son métodos privados, lo que significa que no puedes sobre-escribir o sobrecargar estos métodos. JVM checa la implementación de estos métodos y los llama en lugar de los métodos usuales. Si suena raro pero este es el camino de personalizar la serialización en la JVM.
* Como se discutió en las secciones anteriores, un objeto serializado puede comunicarse sobre la red y des-serializarlo en otra máquina. Sin embargo, el archivo de clase del objeto debe de estar en el path de la máquina destino, de otra manera solo el estado del objeto será restaurado, no todo el objeto (no puedes invocar métodos del objeto, por ejemplo).
* Puedes crear tu propio protocolo de serialización. Para esto necesitas implementar la interface Externalizable en lugar de Serializable.
* Cuando no especificas el serialVersionUID en una clase serializada, la JVM calcula uno por ti. Sin embargo, cada implementación de la JVM tiene diferentes mecanismos para calcularlo; por lo tanto, no está garantizado que tu clase serializada funcione en dos JVM diferentes cuando no especificas el serialVersionUID de manera explícita. Por lo tanto, es muy recomendable que lo especifiques explícitamente.

### Resumen

#### Leer y escribir datos a la consola

* Puedes obtener una referencia de la consola usando el método System.console(); si la JVM no está asociada con ninguna consola, este método falla y retornará null.
* Muchos métodos provistos en la clase Console soportan operaciones I/O formateadas. Puedes usar el método printf() y format() disponibles en la clase Console para imprimir texto formateado; los métodos sobrecargados readLine() y readPassword toman cadenas formateadas como argumentos.
* Usa el método readPassword() para leer de manera segura cadenas como passwords. Es recomendable usar el método fill de la clase Arrays para vaciar el password almacenado en el array de caracteres y con esto evitar acceso malicioso a los passwords escritos.
* Los métodos en la clase Console tienen mejor soporte para caracteres especiales comparados con los métodos de PrintStreams.

#### Leer y escribir archivos con streams

* El paquete java.io tiene clases que soportan dos streams: carácter y bytes.
* Puedes usar streams de caracteres para operaciones I/O basadas en texto. Streams de bytes para operaciones I/O basadas en datos.
* Los streams de caracteres para leer y escribir son llamados readers y writers respectivamente (Representados por las clases Reader y Writer)
* Los streams de bytes para leer y escribir son llamados input streams y output streams respectivamente (representados por las clases InputStream y OutputStream)
* Deberás utilizar los streams de caracteres únicamente para procesar archivos de texto y streams de bytes para archivos de datos. Si intentas usar un tipo de stream en lugar del otro, tu programa no funcionará como lo esperas; incluso si trabaja, obtendrás resultados raros y bugs. NO mezcles streams, usa el stream correcto para la tarea correspondiente.
* Para ambos, byte y carácter streams, puedes usar amortiguación (buffering). Las clases buffer son provistas como envoltorios para los streams correspondientes. Usar amortiguación acelera el proceso de I/O masivo.
* Para procesar datos con tipos primitivos y Strings puedes usar data streams.
* Serialization: El proceso de convertir los objetos en memoria, en una serie de bytes.
* Persistence: El mecanismo para almacenar objetos de la memoria en archivos.
* Puedes usar streams de objetos para persistencia de objetos. Leer y escribir objetos en memoria hacia un archivo y viceversa.