1. 15.1 AN OVERVIEW OF UPCOMING SRS ENHANCEMENTS - APPROACHES TO OBJECT PERSISTENCE

Enfoques de la persistencia de objetos

Siempre que ejecutamos una aplicación Java todos los objetos que instanciamos residen en la memoria asignada a la JVM. Cuando una aplicación de este tipo termina, toda la memoria de la JVM se devuelve al sistema operativo, y los estados internos de todos los objetos creados por la aplicación se olvidan, es decir, a menos que se hayan guardado de alguna manera. Mediante el uso de varias API, Java proporciona una gran cantidad de opciones con respecto a la persistencia de los estados de los objetos.

Utilizando la API JDBC, podemos guardar datos en una base de datos relacional compatible con ODBC, como Sybase, Oracle o Microsoft SQL Server, de varias maneras.

Podemos guardar información en un formato de texto bastante sencillo, legible por humanos, por ejemplo, con los datos organizados jerárquicamente mediante Extensible Markup Language (XML), intercalamos información - "contenido"- con "etiquetas" que describen cómo debe interpretarse la información.

Para las formas de persistencia de datos básicas es importante:

- Ocultar los detalles de la persistencia de un objeto encapsulándolos en una o más clases de la capa de acceso a datos, de forma que ni nuestras clases modelo ni el código cliente tengan que preocuparse por los detalles.
- Adherirse al principio de diseño de separación de capa de acceso a datos-modelo, asegurando así que nuestras clases modelo sirven como puras abstracciones de objetos de dominio del mundo real-sin saturarlas con detalles de persistencia para que sean máximamente reutilizables de una aplicación a la siguiente.
- Asegurar que cualquier enfoque que tomemos para la persistencia de un objeto sea flexible, de forma que podamos cambiar una forma de persistencia por otra.
- Proporcionar una gestión eficaz de los errores cuando algo va mal. Dado que la persistencia implica con un sistema de archivos externo, un sistema de gestión de bases de datos y/o la red, hay muchos puntos potenciales de fallo que están fuera del control inmediato de una aplicación.

2. 15.2 THE BASICS OF FILE I/O IN JAVA (CONCEPTOS BÁSICOS DE LA E/S DE ARCHIVOS EN JAVA)

La entrada/salida de archivos (File I/O) se basa en streams o en la lectura y escritura de datos desde varias fuentes.

Lectura de un archivo

El proceso de lectura de archivos en Java involucra la creación de un objeto FileReader para abrir y leer el archivo carácter por carácter, y luego pasar este objeto a un BufferedReader

para una lectura más eficiente de líneas completas. El método readLine de BufferedReader maneja internamente la recopilación y bufferización de caracteres hasta que se detecta un carácter de fin de línea, momento en el cual devuelve una línea completa al código cliente.

Ejemplo en pseudocódigo:

```
FileReader fr = new FileReader(fileName);
BufferedReader bIn = new BufferedReader(fr);

String line= bIn.readLine();
while (line != null) {
    //Process the most recently read line however we'd like ...
    line = bIn.readLine();
}

bIn.close();
```

Algunos puntos clave:

- La señal de fin de archivo es null.
- Para saltar líneas en blanco, se puede usar trim() para eliminar espacios en blanco al principio y al final de la línea.

Escritura de un archivo

Para escribir en un archivo de texto en Java, seguimos un proceso similar, pero en sentido inverso al de la lectura de archivos:

- Creamos un objeto FileOutputStream, el cual abre un archivo y es capaz de escribir datos en él carácter por carácter.
- Pasamos una referencia de ese objeto FileOutputStream como argumento al constructor de PrintWriter, un objeto más sofisticado que encapsula FileOutputStream. El método println de PrintWriter sabe cómo transferir datos de una línea completa, carácter por carácter, a su objeto FileOutputStream encapsulado, el cual los escribe en el archivo uno por uno.

Ejemplo en pseudocódigo:

```
FileOutputStream fos = new FileOutputStream(fileName);
PrintWriter pw = new PrintWriter(fos);

while (we still have more data to output) {
    pw.println(whatever data we wish to output);
}

pw.close();
```

Algunos puntos importantes a tener en cuenta:

- Es crucial cerrar explícitamente la combinación PrintWriter/FileOutputStream para asegurarse de que los datos se escriban completamente en el archivo.
- Si olvidamos cerrar la combinación PrintWriter/FileOutputStream, el archivo resultante puede estar vacío.
- Es posible especificar si se desea crear un nuevo archivo, añadir registros a un archivo existente o sobrescribir un archivo existente al crear FileOutputStream. Esto se hace utilizando un constructor con el argumento booleano append, donde true indica que se añadirán registros y false indica que se sobrescribirá el archivo.

Manejo de excepciones con E/S de archivos

Estas son algunas de las excepciones más comunes relacionadas con la lectura y escritura de archivos:

Lectura de archivos:

- FileNotFoundException: Si el archivo no existe, es un directorio en lugar de un archivo regular, o por alguna otra razón no se puede abrir para lectura.
- IOException: Si surge algún problema durante la lectura del archivo después de que se ha abierto correctamente para acceso de lectura.
- Excepciones personalizadas: Si los datos leídos están mal formateados u ocurren otros problemas específicos de la aplicación.

Escritura de archivos:

- FileNotFoundException: Si el archivo existe pero es un directorio en lugar de un archivo regular, si el archivo no existe pero no se puede crear, o si el archivo no se puede abrir para escritura por cualquier otra razón.
- IOException: Si surge algún problema durante la escritura en el archivo después de que se ha abierto correctamente para acceso de escritura.

En el texto utilizan varios bloques try/catch para manejar apropiadamente estas excepciones.

3. THE FILE CLASS: INTRODUCCIÓN - THE FILE CLASS: DIRECTORY UTILITIES

Este capítulo te dará una introducción a la variedad de clases de E/S en la biblioteca estándar de Java y cómo usarlas

La clase File

tiene un nombre engañoso. De hecho, "RutaDeArchivo" habría sido un nombre mejor para la clase. Puede representar tanto el nombre de un archivo en particular como los nombres de un conjunto de archivos en un directorio. Si son un conjunto de archivos, puedes solicitar ese conjunto usando el método list(), que devuelve un array de String.

Un listador de directorios

Supongamos que te gustaría ver una lista de directorios. El objeto File se puede usar de dos maneras.

Si llamas a list() sin argumentos, obtendrás la lista completa que contiene el objeto File.

Si deseas una lista restringida, por ejemplo, si deseas todos los archivos con una extensión .java, entonces usas un "filtro de directorio", que es una clase que indica cómo seleccionar los objetos File para mostrar. Aquí tienes un ejemplo.

```
//: io/DirList.java

// Display a directory listing using regular expressions.

// {Args: "D.*\.java"}

import java.util.regex.*;

import java.io.*;

import java.util.*;

public class DirList {

 public static void main(String[] args) {

 File path = new File(".");

 String[] list;

 if(args.length == 0)

 list = path.list();

 else

 list = path.list(new DirFilter(args[0]));

 Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);

 for(String dirItem : list)

 System.out.println(dirItem);

 }

}
```

```
class DirFilter implements FilenameFilter {
 private Pattern pattern;
 public DirFilter(String regex) {
 pattern = Pattern.compile(regex);
 }
 public boolean accept(File dir, String name) {
 return pattern.matcher(name).matches();
 }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~
```

Ten en cuenta que el resultado se ha ordenado sin esfuerzo (alfabéticamente) utilizando el método java.util.Arrays.sort() y el Comparador String.CASE_INSENSITIVE_ORDER:

La clase DirFilter implementa la interfaz FilenameFilter:

```
public interface FilenameFilter {
  boolean accept(File dir, String name);
}
```

La única razón de ser de la clase DirFilter es proporcionar el método accept() al método list() para que list() pueda "llamar" a accept() para determinar qué nombres de archivo deben incluirse en la lista. Por lo tanto, esta estructura a menudo se denomina callback.

El método accept() debe aceptar un objeto File que representa el directorio en el que se encuentra un archivo en particular, y una cadena que contiene el nombre de ese archivo. Recuerda que el método list() llama a accept() para cada uno de los nombres de archivo en

el objeto directorio para ver cuál debe incluirse; esto se indica mediante el resultado booleano devuelto por accept().

accept() utiliza un objeto de coincidencia de expresiones regulares para ver si la expresión regular regex coincide con el nombre del archivo. Usando accept(), el método list() devuelve un array.

Clases internas anónimas

En este ejemplo, se utiliza una característica de Java llamada "clases internas anónimas" para simplificar el código. En lugar de crear una clase separada, puedes definir una clase dentro del método mismo.

Esta característica permite crear una clase que solo se usará una vez, sin necesidad de nombrarla. Simplifica el código al mantenerlo más cerca del lugar donde se utiliza. En este ejemplo, la clase interna anónima se utiliza para filtrar los nombres de archivo que coinciden con un patrón específico.

```
//: io/DirList3.java
// Building the anonymous inner class "in-place."
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList3 {
  public static void main(final String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
```

```
      list = path.list(new FilenameFilter() {
        private Pattern pattern = Pattern.compile(args[0]);
        public boolean accept(File dir, String name) {
          return pattern.matcher(name).matches();
        }
      });
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);
  }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~
```

La clase interna anónima usa args[0] directamente, por lo que el argumento de main() ahora es final. Esto permite crear clases específicas para resolver problemas particulares, manteniendo el código más organizado. Sin embargo, este enfoque no siempre es fácil de leer, así que debes usarlo con cuidado.

A menudo necesitamos operar con conjuntos de archivos en nuestros programas. Para facilitar esto, creamos una clase de utilidad que puede producir una lista de archivos basada en una expresión regular. Puedes obtener una lista de archivos en el directorio local usando el método local(), o una lista de todos los archivos en el árbol de directorios comenzando desde un directorio dado con walk(). Utilizamos objetos File en lugar de nombres de archivo porque contienen más información.

```
//: io/DirList2.java
// Uses anonymous inner classes.
// {Args: "D.*\.java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList2 {
  public static FilenameFilter filter(final String regex) {
    // Creation of anonymous inner class:
    return new FilenameFilter() {
      private Pattern pattern = Pattern.compile(regex);
      public boolean accept(File dir, String name) {
        return pattern.matcher(name).matches();
      }
    }; // End of anonymous inner class
  }
  public static void main(String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(filter(args[0]));
    Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
    for(String dirItem : list)
      System.out.println(dirItem);
  }
} /* Output:
DirectoryDemo.java
DirList.java
DirList2.java
DirList3.java
*///:~
```

Utilidades de directorios

El método local() utiliza listFiles(), una variante de File.list(), para obtener un array de objetos File en el directorio local. Si necesitas una lista en lugar de un array, puedes convertir el resultado usando Arrays.asList().

El método walk() realiza un recorrido recursivo de directorios, recolectando información sobre los archivos y directorios. Devuelve un objeto TreeInfo, que es esencialmente una "tupla" de dos listas: una lista de archivos y otra de directorios.

La clase TreeInfo implementa Iterable<File>, lo que te permite iterar sobre la lista de archivos de manera predeterminada, mientras que puedes acceder a la lista de directorios usando ".dirs".

El método toString() de TreeInfo utiliza un "pretty printer" para formatear la salida de manera más legible, agregando nuevas líneas e indentando cada elemento.

```
//: net/mindview/util/PPrint.java
// Pretty-printer for collections
package net.mindview.util;
import java.util.*;

public class PPrint {
  public static String pformat(Collection<?> c) {
    if(c.size() == 0) return "[]";
    StringBuilder result = new StringBuilder("[");
    for(Object elem : c) {
      if(c.size() != 1)
        result.append("\n  ");
      result.append(elem);
    }
    if(c.size() != 1)
      result.append("\n");
    result.append("]");
    return result.toString();
  }
  public static void pprint(Collection<?> c) {
    System.out.println(pformat(c));
  }
  public static void pprint(Object[] c) {
    System.out.println(pformat(Arrays.asList(c)));
  }
} ///:~
```

El método pformat() produce una cadena formateada a partir de una colección, y el método pprint() utiliza pformat() para hacer su trabajo. Ten en cuenta que los casos especiales de ningún elemento y un solo elemento se manejan de manera diferente. También hay una versión de pprint() para arrays.

La utilidad Directory se encuentra en el paquete net.mindview.util para que esté fácilmente disponible. Aquí tienes un ejemplo de cómo puedes usarla:

```
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DirectoryDemo {
  public static void main(String[] args) {
    // All directories:
    PPrint.pprint(Directory.walk(".").dirs);
    // All files beginning with 'T'
    for(File file : Directory.local(".", "T.*"))
      print(file);
    print("----------------------");
    // All Java files beginning with 'T':
    for(File file : Directory.walk(".", "T.*\\.java"))
      print(file);
    print("======================");
    // Class files containing "Z" or "z":
    for(File file : Directory.walk(".",".*[Zz].*\\.class"))
      print(file);
  }
```

4. THE FILE CLASS: CHECKIN FOR AND CREATING DIRECTORIES – INPUT AND OUTPUT

La clase File en Java no solo sirve como representación de un archivo o directorio existente, sino que también permite crear nuevos directorios o rutas completas de directorios si no existen. Además, proporciona métodos para examinar las características de los archivos (tamaño, fecha de última modificación, lectura/escritura), verificar si un objeto File representa un archivo o un directorio, y eliminar archivos.

En el ejemplo proporcionado, se muestra el uso de la clase File para crear directorios y manipular archivos. Se incluyen métodos como mkdirs() para crear directorios y renameTo() para renombrar archivos. También se utiliza el método fileData() para mostrar información sobre archivos o directorios, como la ruta absoluta, la capacidad de lectura/escritura y el tipo de archivo (archivo o directorio).

El programa también incluye opciones de línea de comandos para crear, eliminar o renombrar archivos y directorios según los argumentos proporcionados. Por ejemplo, se puede ejecutar el programa con -d para eliminar un directorio o con -r para renombrar un archivo.

Además, se proporciona una descripción de los tipos de flujos de entrada (InputStream) y salida (OutputStream) en Java, que son abstracciones para manipular datos de entrada y salida. Se mencionan diferentes subclases de InputStream y OutputStream que representan diferentes fuentes y destinos de datos, como archivos, matrices de bytes, cadenas y tuberías. También se discute el concepto de decoradores (Decorator pattern) para agregar funcionalidades adicionales a los flujos de entrada y salida.

5. STANDARD I/O

El término Input/Output estándar, es utilizado como un flujo de información donde las entradas, salidas e inclusive los errores son estándares; esto implica que el input estándar de un problema, al hacer output, este puede ser el input del siguiente programa y así consecutivamente.

Las entradas System.in, System.out y System.err son entradas normales, estas no están estandarizadas, aunque out y err ya están pre-envueltas, el System.in no lo está, por lo que toca envolverla de alguna forma, esto se puede por ejemplo con la línea readLine()

```
//: io/Echo.java
// How to read from standard input.
// {RunByHand}
import java.io.*;

public class Echo {
  public static void main(String[] args)
  throws IOException {
    BufferedReader stdin = new BufferedReader(
      new InputStreamReader(System.in));
    String s;
```

/O

```
    while((s = stdin.readLine()) != null && s.length()!= 0)
      System.out.println(s);
    // An empty line or Ctrl-Z terminates the program
  }
} ///:~
```

Se puede cambiar el System.out por un PrintStream cambiando su constructor de la forma:

```
//: io/ChangeSystemOut.java
// Turn System.out into a PrintWriter.
import java.io.*;

public class ChangeSystemOut {
  public static void main(String[] args) {
    PrintWriter out = new PrintWriter(System.out, true);
    out.println("Hello, world");
  }
} /* Output:
Hello, world
*///:~
```

Además se puede redireccionar las entradas, salidas y errores de código:

Esto sirve para las salidas en el caso donde hallan demasiadas salidas al tiempo que el lector no pueda leerlas

Para la entrada por si estamos por ejemplo en una línea de commandos y queremos mandarle al programa repetidas veces el input:

setIn(InputStream)
setOut(PrintStream)
setErr(PrintStream)

```
import java.io.*;

public class Redirecting {
  public static void main(String[] args)
  throws IOException {
    PrintStream console = System.out;
    BufferedInputStream in = new BufferedInputStream(
      new FileInputStream("Redirecting.java"));
    PrintStream out = new PrintStream(
      new BufferedOutputStream(
        new FileOutputStream("test.out")));
    System.setIn(in);
    System.setOut(out);
    System.setErr(out);
    BufferedReader br = new BufferedReader(
      new InputStreamReader(System.in));
    String s;
    while((s = br.readLine()) != null)
      System.out.println(s);
    out.close(); // Remember this!
    System.setOut(console);
  }
} ///:~
```

Este código ayudara a redirigir las entradas y salidas a archivos aparte, donde estos archivos manipulan los flujos de bytes del programa.

PROCESS CONTROL

En esta sección hablan de cómo hacer una entrada y salida desde los commandos de java, donde queremos hacer las entradas de forma “fuerza bruta”, las entradas se tienen que hacer de forma línea a línea (readLine()) y los errores se mandarán al flujo de errores de la clase OSExecuteException().

NEW I/O

Actualmente en las últimas versiones de java se mejoró la velocidad del input y output, esto tanto para los archivos que sean estándares como en los de red.

La idea es usar las estructuras del Sistema más a profundidad para poder manipular la información más “sencillamente”

Por lo que ponen el ejemplo de un canal de una mina y un carro de carbón

El canal es la vía para los sistemas con los datos

Las minas de carbón son los datos

Y el usuario solo tiene que mandar el carro y el carro se llena del carbón y se lo devuelve al usuario.

Al final el usuario saca el carbón del carro hacienda referencia a que obtiene los datos y los usa a su antojo

Este tipo diferente de I/O se puede usar con la librería “nio”

Donde este usara “canales” de bytes o le pedirá al usuario que los cree para así recopilar la información y que sea de mayor facilidad para el usuario obtener la información.

```
//: io/GetChannel.java
// Getting channels from streams
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class GetChannel {
  private static final int BSIZE = 1024;
  public static void main(String[] args) throws Exception {
    // Write a file:
    FileChannel fc =
      new FileOutputStream("data.txt").getChannel();
    fc.write(ByteBuffer.wrap("Some text ".getBytes()));
```



```
    fc.close();
    // Add to the end of the file:
    fc =
      new RandomAccessFile("data.txt", "rw").getChannel();
    fc.position(fc.size()); // Move to the end
    fc.write(ByteBuffer.wrap("Some more".getBytes()));
    fc.close();
    // Read the file:
    fc = new FileInputStream("data.txt").getChannel();
    ByteBuffer buff = ByteBuffer.allocate(BSIZE);
    fc.read(buff);
    buff.flip();
    while(buff.hasRemaining())
      System.out.print((char)buff.get());
  }
} /* Output:
Some text Some more
*///:~
```

Ejemplo donde se utilizan los 2 nuevos tipos: : FileInputStream, FileOutputStream.

También hablan de los “ByteBuffer” que son los archivos de bytes, donde están guardándose toda la información; esta parte es decidida por el usuario y además se puede copiar en diferentes archivos.txt para la lectura de estos mismos.

```
//: io/ChannelCopy.java
// Copying a file using channels and buffers
// {Args: ChannelCopy.java test.txt}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class ChannelCopy {
```



```
  private static final int BSIZE = 1024;
  public static void main(String[] args) throws Exception {
    if(args.length != 2) {
      System.out.println("arguments: sourcefile destfile");
      System.exit(1);
    }
    FileChannel
      in = new FileInputStream(args[0]).getChannel(),
      out = new FileOutputStream(args[1]).getChannel();
    ByteBuffer buffer = ByteBuffer.allocate(BSIZE);
    while(in.read(buffer) != -1) {
      buffer.flip(); // Prepare for writing
      out.write(buffer);
      buffer.clear();  // Prepare for reading
    }
  }
} ///:~
```

Ejemplo de código donde se utilizan métodos de solo lectura (allocate() o allocateDirect()), la lectura del archive de bytes (read()) y extracción de bytes (flip()) y al final.

6. NEW I/O: CONVERTING DATA

En la sección "Converting Data", se aborda la forma primitiva de extraer información de un archivo byte por byte y convertir cada byte en un char en el archivo GetChannel.java.
Se menciona que esta técnica puede parecer rudimentaria y se sugiere utilizar la clase CharBuffer de java.nio como una alternativa más eficiente. Se destaca que un ByteBuffer puede ser visto como un CharBuffer a través del método asCharBuffer().
Además, se menciona que existen métodos especiales como transferTo() y transferFrom() que permiten conectar un canal directamente a otro para facilitar la transferencia de datos de manera más eficiente.

- **toString():** Devuelve una cadena que contiene los caracteres en un buffer.
- **asCharBuffer():** Permite ver un ByteBuffer como un CharBuffer, lo que facilita la manipulación de datos de bytes a caracteres.
- **transferTo():** Permite transferir datos directamente de un canal a otro, proporcionando una forma eficiente de manejar operaciones de lectura y escritura.
- **transferFrom():** Similar a transferTo(), permite transferir datos entre canales, pero en este caso, la transferencia se realiza desde el canal de salida al canal de entrada.

Fetching Primitives:
En la sección "Fetching Primitives", se explora cómo un ByteBuffer, que almacena bytes, puede ser utilizado para producir diferentes tipos de valores primitivos a partir de los bytes que contiene.

Se muestra un ejemplo que demuestra la inserción y extracción de varios valores utilizando métodos específicos para cada tipo primitivo.

```
public class GetData {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    // Allocation automatically zeroes the ByteBuffer:
    int i = 0;
    while(i++ < bb.limit())
      if(bb.get() != 0)
        print("nonzero");
    print("i = " + i);
    bb.rewind();
    // Store and read a char array:
    bb.asCharBuffer().put("Howdy!");
    char c;
    while((c = bb.getChar()) != 0)
      printnb(c + " ");
    print();
    bb.rewind();
    // Store and read a short:
    bb.asShortBuffer().put((short)471142);
    print(bb.getShort());
    bb.rewind();
    // Store and read an int:
    bb.asIntBuffer().put(99471142);
    print(bb.getInt());
    bb.rewind();
    // Store and read a long:
    bb.asLongBuffer().put(99471142);
    print(bb.getLong());
    bb.rewind();
    // Store and read a float:
    bb.asFloatBuffer().put(99471142);
    print(bb.getFloat());
    bb.rewind();
    // Store and read a double:
    bb.asDoubleBuffer().put(99471142);
    print(bb.getDouble());
    bb.rewind();
  }
```

Se destaca que la asignación de un ByteBuffer automáticamente establece en cero el contenido del buffer, lo que se verifica al comprobar que todos los valores dentro del límite del buffer son cero. Además, se menciona que la forma más sencilla de insertar valores primitivos en un ByteBuffer es obtener una "vista" apropiada del buffer y utilizar el método put() de esa vista para realizar las operaciones de inserción.

Métodos Específicos para Cada Tipo Primitivo:

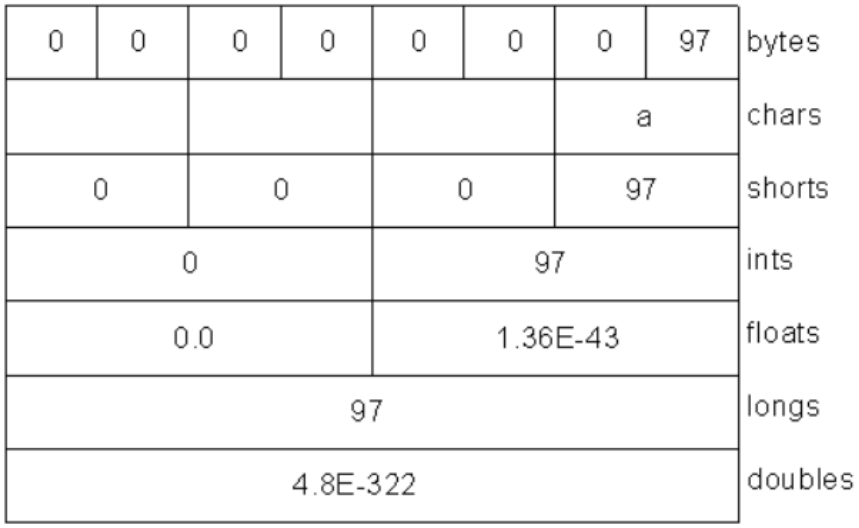
1. Métodos de Inserción:
    - **putChar(char c):** Inserta un valor de tipo char en el buffer.
    - **putShort(short s):** Inserta un valor de tipo short en el buffer.
    - **putInt(int i):** Inserta un valor de tipo int en el buffer.
    - **putLong(long l):** Inserta un valor de tipo long en el buffer.
    - **putFloat(float f):** Inserta un valor de tipo float en el buffer.
    - **putDouble(double d):** Inserta un valor de tipo double en el buffer.
2. Métodos de Extracción:

- **getChar():** Extrae un valor de tipo char del buffer.
- **getShort():** Extrae un valor de tipo short del buffer.
- **getInt():** Extrae un valor de tipo int del buffer.
- **getLong():** Extrae un valor de tipo long del buffer.
- **getFloat():** Extrae un valor de tipo float del buffer.
- **getDouble():** Extrae un valor de tipo double del buffer.

Estos métodos específicos permiten manipular y trabajar con diferentes tipos de datos primitivos de manera eficiente en un ByteBuffer, facilitando la inserción y extracción de valores según el tipo de dato requerido.

VIEW BUFFERS:

En la sección "View Buffers", se introduce el concepto de un "view buffer" que permite visualizar un ByteBuffer subyacente a través de la ventana de un tipo primitivo específico.
Se destaca que el ByteBuffer sigue siendo el almacenamiento real que respalda la vista, lo que significa que cualquier cambio realizado en la vista se refleja en modificaciones en los datos en el ByteBuffer.
Se muestra un ejemplo que manipula ints en un ByteBuffer a través de un IntBuffer, demostrando cómo insertar y leer valores primitivos en un ByteBuffer utilizando vistas.
Además, se menciona que una vez que el ByteBuffer subyacente está lleno con valores primitivos a través de un view buffer, se puede escribir directamente en un canal o leer desde un canal y convertir los datos a un tipo primitivo específico utilizando un view buffer.



This corresponds to the output from the program.

7. NEW I/O: DATA MANIPULATION WITH BUFFERS- NEW I/O: MEMORY-MAPPED FILES

DATA MANIPULATION WITH BUFFERS

Un Buffer consta de datos y cuatro índices para acceder y manipular estos datos de manera eficiente:
marca, posición, límite y capacidad. Existen métodos para configurar y restablecer estos índices y para consultar su valor.

capacity() —- retorna la capacidad del buffer

clear() —------- borra el buffer y establece la posición en cero y limita la capacidad. Se llama a este método para sobreescribir un buffer existente.

flip() —---------- Establece el límite a la posición y la posicion a cero, se utiliza para una lectura que se hará luego con datos ya escritos en él.

limit() —---------- retorna el valor del límite.

limit(int lin) —---------- establece el valor del límite.

mark() —--------- establece la marca en la posición

position() —----------- retorna la posición
position(int pos) —---------- establece la posición

remaining() —---------- retorna (limite-posición)

hasRemaining() —---------- retorna True si hay algún elemento entre la posición y el límite.

MEMORY-MAPPED FILES

Los archivos asignados en memoria permiten crear y modificar archivos que son demasiado grandes para guardarlos en memoria, cuando tenemos archivos mapeados en memoria podemos acceder a ellos tratándolos como una matriz muy grande y así podemos simplificar el código escrito, además de que si hacemos esto con los archivos, mejoramos el rendimiento del programa en vez de tener un código robusto que contiene mucha información.

8. NEW I/O: FILE LOCKING (BLOQUEO DE ARCHIVOS)

Permite sincronizar el acceso a un archivo como recurso compartido. Dos hilos que compiten por el mismo archivo pueden estar en diferentes JVMs, o uno puede ser un hilo deJava y el otro algún hilo nativo del sistema operativo.

Estos bloqueos son visibles para otros procesos del sistema operativo, ya que el bloqueo de archivos de Java se asigna directamente a la función de bloqueo nativa del sistema operativo.

Ejemplo de bloqueo de archivos:

```
//: io/FileLocking.java
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;

public class FileLocking {

  public static void main(String[] args) throws Exception {
    FileOutputStream fos= new FileOutputStream("file.txt");
    FileLock fl = fos.getChannel().tryLock();
    if(fl != null) {
      System.out.println("Locked File");
      TimeUnit.MILLISECONDS.sleep(100);
      fl.release();
      System.out.println("Released Lock");
    }
    fos.close();
  }
} /* Output:
Locked File
Released Lock
*///:~
```

Obtenemos un FileLock sobre todo el fichero llamado tryLock() o a lock() sobre un FileChannel(SocketChannel,DatagramChannel y ServerSocketChannel no necesitan bloqueo ya que son entidades inherentemente de un solo proceso, generalmente no se comparte un socket de red entre dos procesos) tryLock() no es de bloqueo.

Intenta obtener el bloqueo, si no puede(cuando algún proceso ya tiene el mismo bloqueo y no esta compartido), simplemente devuelve la llamada al método.lock() se bloquea hasta que se obtiene el bloqueo, o el hilo que invocó a lock() se interrumpe, o el canal en el que se llama lock() se cierra. Un bloqueo se libera utilizando FileLock.release()

También es posible bloquear una parte del fichero con:

```
tryLock(long position, long size, boolean shared)
```

or

```
lock(long position, long size, boolean shared)
```

Bloqueando la región(tamaño-posición). El tercer argumento especifica si este bloque es compartido.

Aunque los métodos de bloqueo con cero argumentos se adaptan a los cambios en el tamaño de un archivo, los bloqueos con un tamaño fijo no cambian si cambia el tamaño del archivo.

Si se adquiere un bloqueo para una región desde la posición hasta la posición+tamaño y el archivo aumenta más allá de ese valor, la sección más allá de la posición+tamaño no se bloquea. Los métodos de bloqueo con cero argumentos bloquean todo el fichero, aunque crezca.

El sistema operativo subyacente debe proporcionar soporte para bloqueos exclusivos o compartidos. Si el sistema operativo no permite bloqueos compartidos y se solicita uno, se utilizará en su lugar un bloqueo exclusivo. El tipo de bloqueo se puede consultar mediante FileLock.isShared().

Bloque de partes de un archivo asignado

La asignación de archivos se suele utilizar para archivos muy grandes. Es posible que se necesite bloquear partes de un archivo muy grande para que otros procesos puedan modificar las partes que no están bloqueadas.

Esto ocurre, por ejemplo, en una base de datos, para que pueda estar disponible para muchos usuarios a la vez.

Ejemplo que tiene dos hilos, cada uno de los cuales bloquea una parte distinta del archivo

```
//: io/LockingMappedFiles.java
// Locking portions of a mapped file.
```

```
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class LockingMappedFiles {
  static final int LENGTH = 0x8FFFFFF; // 128 MB
  static FileChannel fc;
  public static void main(String[] args) throws Exception {
    fc =
      new RandomAccessFile("test.dat", "rw").getChannel();
    MappedByteBuffer out =
      fc.map(FileChannel.MapMode.READ_WRITE, 0, LENGTH);
    for(int i = 0; i < LENGTH; i++)
      out.put((byte)'x');
    new LockAndModify(out, 0, 0 + LENGTH/3);
    new LockAndModify(out, LENGTH/2, LENGTH/2 + LENGTH/4);
  }
  private static class LockAndModify extends Thread {
    private ByteBuffer buff;
    private int start, end;
    LockAndModify(ByteBuffer mbb, int start, int end) {
      this.start = start;
      this.end = end;
      mbb.limit(end);
      mbb.position(start);
      buff = mbb.slice();
      start();
    }
    public void run() {
      try {
        // Exclusive lock with no overlap:
        FileLock fl = fc.lock(start, end, false);
        System.out.println("Locked: "+ start +" to "+ end);
        // Perform modification:
        while(buff.position() < buff.limit() - 1)
          buff.put((byte)(buff.get() + 1));
        fl.release();
        System.out.println("Released: "+start+" to "+ end);
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
  }
} ///:~
```

El hilo LockAnd Modify configura la región del buffer y crea un slice() para ser modificado, y en run(), se adquiere el bloqueo en el canal del archivo (no se puede adquirir un bloqueo en el buffer solo en el canal). La llamada lock() es similar a la adquisición de un bloqueo de subprocesos en un objeto, ahora hay una “sección critica” con acceso exclusivo a esa parte del archivo.

Los bloqueos se liberan de manera automática cando la JVM sale, o el canal en el que se adquirieron se cierra, también se puede llamar explícitamente a release() en el objeto FileLock()

COMPRESSION (Compresión)

La biblioteca Java I/O contiene clases que permiten leer y escribir flujos en formato comprimido. Estas se envuelven alrededor de otras clases de E/S para proporcionar funcionalidad de compresión.

Estas clases no se derivan de las clases Reader y Writer, sino que forman parte de las jerarquías InputStream y OutputStream. Esto se debe a que la biblioteca de compresión trabaja con bytes, no con caracteres, pero a veces puede verse obligado a mezclar los dos tipos de flujos.

Nota: Recuerde que puede utilizar InputStreamReader y OutputStreamWriter para facilitar la conversión entre un tipo de flujo y otro.

Compression class	Function
CheckedInputStream	**GetCheckSum( )** produces checksum for any **InputStream** (not just decompression).
CheckedOutputStream	**GetCheckSum( )** produces checksum for any **OutputStream** (not just compression).
DeflaterOutputStream	Base class for compression classes.
ZipOutputStream	A **DeflaterOutputStream** that compresses data into the Zip file format.
GZIPOutputStream	A **DeflaterOutputStream** that compresses data into the GZIP file format.
InflaterInputStream	Base class for decompression classes.
ZipInputStream	An **InflaterInputStream** that decompresses data that has been stored in the Zip file format.
GZIPInputStream	An **InflaterInputStream** that decompresses data that has been stored in the GZIP file format.

Aunque existen muchos algoritmos de compresión, Zip Y GZIP son los más utilizados.

Compresión sencilla con GZIP

Su interfaz es sencilla, por lo que es más adecuada cuando se tiene un único flujo de datos que desea comprimir.

Ejemplo que comprime un único archivo:

```
//: io/GZIPcompress.java
// {Args: GZIPcompress.java}
import java.util.zip.*;
import java.io.*;

public class GZIPcompress {
  public static void main(String[] args)
  throws IOException {
    if(args.length == 0) {

      System.out.println(
        "Usage: \nGZIPcompress file\n" +
        "\tUses GZIP compression to compress " +
        "the file to test.gz");
      System.exit(1);
    }
    BufferedReader in = new BufferedReader(
      new FileReader(args[0]));
    BufferedOutputStream out = new BufferedOutputStream(
      new GZIPOutputStream(
        new FileOutputStream("test.gz")));
    System.out.println("Writing file");
    int c;
    while((c = in.read()) != -1)
      out.write(c);
    in.close();
    out.close();
    System.out.println("Reading file");
    BufferedReader in2 = new BufferedReader(
      new InputStreamReader(new GZIPInputStream(
        new FileInputStream("test.gz"))));
    String s;
    while((s = in2.readLine()) != null)
      System.out.println(s);
  }
} /* (Execute to see output) *///:~
```

Su uso es sencillo, basta con envolver el flujo de salida en un GZIPOutputStream o ZipOutputStream, y su flujo de entrada en un GZIPInputStream O ZIPInputStream, lo demás es escritura y lectura de E/S ordinaria. Este es un ejemplo de mezcla de flujos orientados a caracteres con flujos orientados a bytes.

Usando la clase Reader, mientras que el constructor de GZIPOutputStream solo puede aceptar un objeto OutputStream, no un objeto Writer. Cuando se abre el archivo, el GZIPInputStream se convierte en Reader.

Almacenamiento multifichero con Zip

La biblioteca compatible con el formato Zip es más amplia, con esta se puede almacenar fácilmente múltiples archivos, existe una clase separada para facilitar el proceso de lectura de un archivo Zip. El siguiente ejemplo tiene la misma forma que el anterior, pero maneja tantos argumentos de línea de comandos como desee. Tambien muestra el uso de las clases

Checksum para calcular y verificar la suma de comprobación del archivo. Hay dos tipos de Checksum:

- Adler32 -> más rápido
- CRC32 -> un poco más lento pero más preciso

```
//: io/ZipCompress.java
// Uses Zip compression to compress any
// number of files given on the command line.
// {Args: ZipCompress.java}
import java.util.zip.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ZipCompress {
  public static void main(String[] args)
  throws IOException {
    FileOutputStream f = new FileOutputStream("test.zip");
    CheckedOutputStream csum =
```

```
      new CheckedOutputStream(f, new Adler32());
     ZipOutputStream zos = new ZipOutputStream(csum);
     BufferedOutputStream out =
      new BufferedOutputStream(zos);
    zos.setComment("A test of Java Zipping");
    // No corresponding getComment(), though.
    for(String arg : args) {
      print("Writing file " + arg);
      BufferedReader in =
        new BufferedReader(new FileReader(arg));
      zos.putNextEntry(new ZipEntry(arg));
      int c;
      while((c = in.read()) != -1)
        out.write(c);
      in.close();
      out.flush();
    }
    out.close();
    // Checksum valid only after the file has been closed!
    print("Checksum: " + csum.getChecksum().getValue());
    // Now extract the files:
    print("Reading file");
    FileInputStream fi = new FileInputStream("test.zip");
    CheckedInputStream csumi =
      new CheckedInputStream(fi, new Adler32());
    ZipInputStream in2 = new ZipInputStream(csumi);
    BufferedInputStream bis = new BufferedInputStream(in2);
    ZipEntry ze;
    while((ze = in2.getNextEntry()) != null) {
      print("Reading file " + ze);
      int x;
      while((x = bis.read()) != -1)
        System.out.write(x);
    }
    if(args.length == 1)
    print("Checksum: " + csumi.getChecksum().getValue());
    bis.close();
    // Alternative way to open and read Zip files:
    ZipFile zf = new ZipFile("test.zip");
    Enumeration e = zf.entries();
    while(e.hasMoreElements()) {
      ZipEntry ze2 = (ZipEntry)e.nextElement();
      print("File: " + ze2);
      // ... and extract the data as before
    }
    /* if(args.length == 1) */
  }
} /* (Execute to see output) *///:~
```

Para cada fichero que se añade al archivo, debe llamar a putNextEntry() y pasarle un objeto ZipEntry().

ZipEntry() contiene una extensa interfaz que le permite obtener y establecer todos los datos disponibles en esa entrada concreta de su archivo Zip: nombre, tamaños comprimidos y sin

comprimir, fecha, suma de comprobación CRC, datos de campo extra, comentario, método de compresión y si se trata de una entrada de directorio.

Aunque el formato Zip tiene una forma de establecer una contraseña, esto no está soportado en la biblioteca Zip de Java.

Para extraer archivos, ZipInputStream tiene un método getNextEntry() que devuelve la siguiente ZipEntry si existe. Otra opción es leer el archivo utilizando un objeto ZipFile, que tiene un método entries() para devolver una enumeración a las ZipEntries.

Para leer la suma de comprobación, debe tener acceso al objeto Checksum asociado. En este caso, se mantiene una referencia a los objetos CheckedOutputStream y CheckedInputStream, pero también se podría solo mantener la referencia al objeto Checksum.

Como se muestra en ZipCompress.java, con setComment() puede establecer un comentario cuando está escribiendo un archivo, pero no hay forma de recuperar el comentario en el ZipInputStream. Los comentarios parecen ser soportados completamente en una base de entrada por entrada solo a través de ZipEntry.

El uso de las bibliotecas GZIP Y Zip no se limita a los archivos, se puede comprimir cualquier cosa, incluidos los datos que se envían por una conexión red.

Archivos Java(JAR)

Forma de reunir un grupo de archivos en un único archivo comprimido. Estos son archivos son multiplataforma, se pueden incluir archivos de audio e imagen, también archivos de clase.

Útiles cuando se trabaja con internet, cada entrada de un archivo Jar puede firmarse digitalmente para mayor seguridad

Es un único archivo que contiene una colección de archivos comprimidos junto con un manifiesto que los describe.

Para comprimir los archivos que elijas:

```
jar [options] destination [manifest] inputfile(s)
```

c	Creates a new or empty archive.
t	Lists the table of contents.
x	Extracts all files.
x file	Extracts the named file.
f	Says, "I'm going to give you the name of the file." If you don't use this, **jar** assumes that its input will come from standard input, or, if it is creating a file, its output will go to standard output.
m	Says that the first argument will be the name of the user-created manifest file.
v	Generates verbose output describing what **jar** is doing.

o	Only stores the files; doesn't compress the files (use to create a JAR file that you can put in your classpath).
M	Doesn't automatically create a manifest file.

Si se incluye un subdirectorio en los archivos del archivo JAR, se añade automáticamente.

Formas de invocar a Jar:

Crear el archivo que contiene todos los archivos del directorio actual y el manifiesto generado automáticamente

```
| jar cf myJarFile.jar *.class
```

Añade un nuevo manifiesto creado por el usuario

```
| jar cmf myJarFile.jar myManifestFile.mf *.class
```

Producir una tabla de contenidos de los archivos de myJarFile.ja

```
| jar tf myJarFile.jar
```

Añadir la bandera "verbose" para información detallada de myJarFile.jar

```
| jar tvf myJarFile.jar
```

Para audios y e imágenes

```
| jar cvf myApp.jar audio classes image
```

Otra opción para crear un archivo jar con la opción cero

```
| CLASSPATH="libl.jar;lib2.jar;"
```

No se puede añadir o actualizar un archivo jar existente, no se pueden mover archivos a un archivo jar

9. OBJECT SERIALIZATION (SERIALIZACIÓN DE OBJETOS)

La serialización de objetos en Java permite tomar cualquier objeto que implemente la interfaz serializable y convertirlo en una secuencia de bytes que luego se puede restaurar por completo para regenerar el objeto original. El mecanismo de serialización compensa automáticamente las diferencias de los sistemas operativos.

La serialización de objetos permite implementar una persistencia ligera. (Persistencia significa que la vida útil de un objeto no está determinada por si un programa se está ejecutando; el objeto vive entre invocaciones del programa).

No es posible definir un objeto usando una palabra clave, se debe serializar y deserializar explícitamente los objetos en el programa.

La serialización de objetos se agregó para admitir dos características principales:

La invocación de método remoto (RMI) permite que los objetos que viven en otras máquinas se comporten como si vivieran en su máquina, la serialización de objetos en este caso es necesaria para transportar los argumentos y los valores de retorno.

La serialización de objetos también es necesaria para JavaBeans, cuando su utiliza un Bean su información de estado generalmente se configura en tiempo de diseño, la serialización realiza la tarea de almacenar esta información y luego recuperarla cuando se inicial el programa.

Para serializar un objeto es necesario implementar la interfaz **Serializable** (Esta es una interfaz de etiquetado y no tiene métodos). Incluso los objetos de clase se pueden serializar.

> Para serializar un objeto, crea algún tipo de objeto OutputStream y luego lo envuelve dentro de un objeto ObjectOutputStream . En este punto, solo necesita llamar a writeObject( ), y su objeto se serializa y se envía a OutputStream (la serialización de objetos está orientada a bytes y, por lo tanto, utiliza las jerarquías InputStream y OutputStream). Para revertir el proceso, envuelve un InputStream dentro de un ObjectInputStream y llama a readObject( ). Lo que regresa es, como siempre, una referencia a un Objeto abatido, por lo que debes abatir para aclarar las cosas.

Red de objetos: La serialización de objetos no solo guarda una imagen de su objeto, sino que también sigue todas las referencias contenidas del mismo y guarda estos objetos.

Finding the class

Para que un objeto se recupere de su estado serializado la JVM debe poder encontrar el archivo .class asociado.

Controlling serialization

Es posible controlar el proceso de serialización implementando la interfaz **Externalizable** en lugar de la interfaz **Serializable.** La interfaz Externalizable extiendes la interfaz Serializable

y agrega dos métodos, writeExternal() y readExternal(), que se llaman automáticamente para el objeto durante la serialización y deserialización.

The transient keyword

Cuando se controla la serialización es posible que haya un subobjeto que no se desea que el mecanismo de serialización de Java guarde y restaure automáticamente. Una forma de realizar esto es implementar la clase como Externalizable, con esto nada se serializa automáticamente y se puede serializar explícitamente las partes deseadas dentro de writeExternal().

Sin embargo, si se está trabajando con un objeto serializable, toda la serialización ocurre automáticamente. Para controlar esto, se puede desactivar la serialización campo por campo usando la palabra clave transitoria, que dice :"No se moleste en guardar o restaurar esto; yo me encargaré de ello".

An alternative to Externalizable

Otro enfoque es implementar la interfaz Serializable y agregar métodos llamados writeObject() y readObject() que se llamarán automáticamente cuando el objeto se serialice y desserialice, respectivamente.

Using persistence

Es posible utilizar la serialización de objetos hacia y desde una matriz de bytes como una forma de hacer una "copia profunda" de cualquier objeto que sea serializable. (Una copia profunda significa que se está duplicando toda la red de objetos, en lugar de solo el objeto básico y sus referencias).

Lo más seguro si se desea guardar el estado de un sistema es serializarlo como una operación "atómica". Es decir, colocar todos los objetos que componen el estado de sistema en un solo contenedor y escribir ese contenedor en una sola operación. Luego también se puede restaurar con una sola llamada a un método.

Otro tema para considerar es la seguridad, pues la serialización también guarda datos privados. Estos datos deben marcarse como transitorios.