STREAMS EN JAVA

1 Serialització

La **serialització** és el procés d'escriure un objecte a un stream de bytes. És útil quan es vol fer persistent l'estat d'un programa a un fitxer per exemple. El procés invers s'anomena **deserialització**.

Només els objectes que implementen l'interfície **Serializable** poden ser desats i restaurats usant la serialització. Si una classe és serialitzable, totes les seves subclasses també ho són.

Nota: les variables ***transient*** i ***static*** no es guarden durant la serialització.

Usarem les interfícies **ObjectOutput** i **ObjectInput**, i les classes **ObjectOutputStream** i **ObjectInputStream**, que les implementen.

ObjectOutput i ObjectInput

La interfície **ObjectOutput** extén les interfícies **DataOutput** i **AutoCloseable** i suporta serialització. Defineix els mètodes ***close(), flush(), write()*** i ***writeObject()***. Aquest últim s'invoca per serialitzar un objecte. Tots els mètodes llancen **IOException** si es produeix un error.

La interfície **ObjectInput** extén les interfícies **DataInput** i **AutoCloseable** i suporta serialització. Defineix els mètodes ***close(), available(), read()*** i ***readObject()***. Aquest últim s'invoca per llegir un objecte serialitzat. Tots els mètodes llancen **IOException** si es produeix un error. El mètode **readObject()** pot llançar també l'excepció **ClassNotFoundException**.

ObjectOutputStream i ObjectInputStream

La classe **ObjectOutputStream** extén **OutputStream** i implementa la interfície **ObjectOutput**. El seu constructor és:

ObjectOutputStream(OutputStream outStream) throws IOException

La classe **ObjectInputStream** extén **InputStream** i implementa la interfície **ObjectInput**. El seu constructor és:

ObjectInputStream(InputStream inStream) throws IOException

Consulteu la documentació online de Java per a la llista de mètodes que implementen les dues classes anteriors.

Vegeu a l'Annex I un exemple d'ús de les interfícies i classes descrites.

2 La llibreria NIO

A partir de la versió 1.4, Java inclou la llibreria NIO, una aproximació basada en ***canals*** i orientada a ***buffers*** de les operacions d'entrada/sortida. A partir de la versió JDK7, el sistema NIO es va ampliar i millorar, fins el punt que sovint es referencia com a NIO.2.

Paquets i classes NIO

La següent taula mostra alguns paquets d'interès que inclou la llibreria NIO:

Paquet	Funció
java.nio	Paquet arrel del sistema NIO. Encapsula diversos tipus de buffers que contenen dades que s'usen al sistema NIO
java.nio.channels	Dóna suport als canals, base de les connexions d'E/S.
java.nio.charset	Encapsula conjunts de caràcters. També incorpora codificadors i decodificadors per a convertir caràcters a bytes i viceversa.
java.nio.file	Dóna suport a operacions amb fitxers.
java.nio.file.attribute	Dóna suport a operacions amb atributs de fitxers.

Fonaments del sistema NIO

El sistema NIO està basat en dos elements: buffers i canals. Els buffers contenen dades, mentre que els canals representen una connexió a un dispositiu d'E/S, com ara un fitxer o un socket.

Hi ha diversos tipus de canals i buffers. Els principals canals de Java NIO són:

- **FileChannel**
- **DatagramChannel**
- **SocketChannel**
- **ServerSocketChannel**

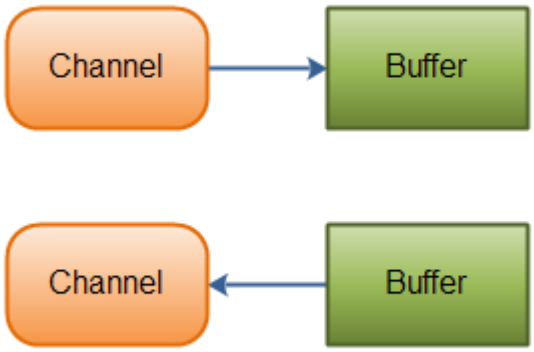
De la classe Buffer deriven classes orientades a cada tipus bàsic de Java:

- **ByteBuffer**
- **CharBuffer**
- **DoubleBuffer**
- **FloatBuffer**
- **IntBuffer**
- **LongBuffer**
- **ShortBuffer**

Java NIO Channels

El funcionament dels *channels* és similar al dels *streams*, però amb petites diferències:

- Es pot llegir i escriure a un **Channel**, mentre que un **Stream** és típicament d'un sentit (read o write).
- Es pot llegir i escriure asíncronament al un **Channel**.
- Els Channels sempre llegeixen d'un **Buffer** i escriuen a un **Buffer**.



FileChannel és el canal que llegeix dades i escriu dades a fitxers.

Vegeu a l'Annex II un exemple de l'ús de **FileChannel** per a llegir dades a un **Buffer**.

Java NIO Buffers

Un buffer és essencialment un bloc de memòria en el qual es poden escriure i més tard llegir dades. El bloc de memòria s'encapsula usant la classe **Buffer**, que proveeix d'un conjunt de mètodes adients per treballar-hi.

El procediment per llegir i escriure dades en un Buffer segueix el següent patró de quatre passos:

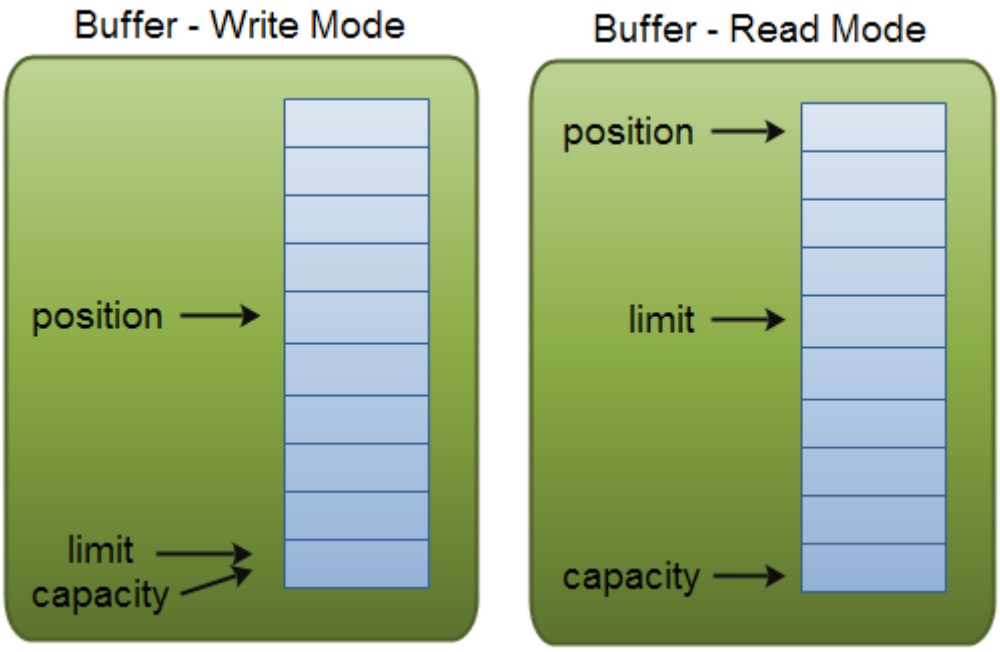
1. Escriure dades al Buffer
2. Invocar **buffer.flip()**
3. Lllegir dades del Buffer
4. Invocar **buffer.clear()** o bé **buffer.compact()**

Quan s'escriuen dades en un buffer, aquest manté un registre de la quantitat de dades que s'hi han escrit. Quan cal llegir les dades, s'ha de canviar el mode d'operació del buffer d'escriptura a lectura usant el mètode f**lip()**.

En el mode lectura es poden llegir totes les dades que s'hi han escrit. Una vegada llegides, cal netejar el buffer per poder-hi escriure de nou. Hi ha dos mètodes: **clear()**, que esborra tot el buffer, o bé **compact()**, que allibera només les dades que s'han llegit (suposant que no s'hagin llegit totes). Les dades no llegides es mouen a l'inici del buffer.

Buffer Capacity, Position i Limit

Un buffer té tres propietats fonamentals: capacitat, posició i límit.



Capacity. És la mida del Buffer en tant que bloc de memòria. Quan el buffer és ple, cal buidar-lo (llegint-lo o invocant el mètode clear()) abans d'escriure-hi de nou.

Position. Quan s'escriu en un Buffer, es fa en una posició determinada. Inicialment aquesta posició és 0 i arriba com a màxim a capacity – 1. Quan es llegeix d'un buffer, es fa en una certa posició. La posició es fixa a 0 quan es fa un 'flip' al buffer i s'incrementa en 1 cada vegada que es llegeix.

Limit. En mode escriptura el límit d'un buffer és igual a la seva capacitat, mentre que en mode lectura és l'última posició escrita.

Creació i ús de Buffers

S'obté un Buffer fent reserva de memòria, invocant el mètode **allocate()** que implementen les totes les seves subclases. Un exemple seria:

```
ByteBuffer buf = ByteBuffer.allocate(48);
```

Per escriure dades a un Buffer, hi ha dos mètodes:

1. Escriure dades al Buffer des d'un Channel. Exemple:

   ```
   int bytesRead = inChannel.read(buf);
   ```

2. Escriure directament dades al Buffer, usant el mètode **put().** Exemple:

   ```
   buf.put(127);
   ```

El mètode flip() canvia un Buffer del mode d'escriptura al de lectura. Quan s'invoca aquest mètode el paràmetre limit pren el valor de position i position passa a valer 0.

Anàlogament a l'escriptura, hi ha dues maneres de llegir dades d'un Buffer:

1. Llegir del Buffer i escriure a un Channel. Exemple:

```
int bytesWritten = inChannel.write(buf);
```

2. Llegir directament del Buffer, usant el mètode get(). Exemple:

```
byte aByte = buf.get();
```

El mètode **Buffer.rewind()** posa la posició novament a 0, fent possible tornar a llegir novament les dades del Buffer, mantenint el valor *limit.*

Transferència entre Channels

Java NIO permet transferir dades directament d'un canal a un altre, sempre que un d'ells sigui **FileChannel.** La classe **FileChannel** implementa els mètodes **transferTo()** i **transferFrom()** per a fer-ho:

```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel      fromChannel = fromFile.getChannel();

RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel      toChannel = toFile.getChannel();

long position = 0;
long count    = fromChannel.size();

toChannel.transferFrom(fromChannel, position, count);
```

La transferència d'un FileChannel a un Channel d'un altre tipus es faria anàlogament.

JavaNIO FileChannel

Usant la classe **FileChannel** es pot escriure i llegir a fitxer, i per això cal primer obrir-lo. Es pot obtenir un FileChannel via **InputStream**, **OutputStream** o **RandomAccessFile**. Per exemple,

```
RandomAccessFile aFile     = new RandomAccessFile("nio-data.txt", "rw");
FileChannel      inChannel = aFile.getChannel();
```

Operacions habituals

Lectura. A través d'un dels mètodes **read()**. Prèviament cal reservar memòria per a un Buffer.

```
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buf);
```

Escriptura. Mitjançant el mètode **write()**, que té com a paràmetre un Buffer. Exemple:

```
String newData = "New String to write to file..."

ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());

buf.flip();

while(buf.hasRemaining()) {
    channel.write(buf);
}
```

Tancar FileChannel. En acabar totes les operacions amb el Channel, cal tancar-lo:

```
channel.close();
```

Interfície Path

La interfície **Path** Encapsula el path a un fitxer i està ubicada al paquet **java.nio.file**.
Conté una sèrie de mètodes que permeten fer operacions amb path. Té especial importància el mètode **getName()**, que usa un índex: 0 indica la part més propera a l'arrel mentre que **getNameCount()** obté el nombre d'elements en un path.

Altres mètodes importants de la interfície són **getFileName(), getParent(), getRoot(), isAbsolute(), resolve()** i **toAbsolutePath()**. Vegeu la documentació de Javadoc per als detalls d'aquests mètodes.

La classe Files

Els mètodes de la classe Files permeten operar amb fitxers. Aquests mètodes usen un Path per especificar el fitxer sobre el que estem efectuant les operacions.

Aquestes operacions poden ser, entre altres: **copy(), createDirectory(), createFile(), delete(), exists(), move()**.

Addicionalment ofereix serveis per a consultar els atributs d'un fitxer o directori: **isDirectory(), isExecutable(), isHidden(), isReadable(), isWritable()**, entre d'altres.

La classe Paths

Com que **Path** és una interfície i no una classe, cal usar la classe **Paths** per a obtenir un path, amb el mètode **get().**

Tots els mètodes retornen excepcions en cas que l'operació no es pugui completar.

La interfície FileAttributes

Associat a cada fitxer hi ha un conjunt d'atributs. Aquests es poden obtenir usant la interfície BasicFileAttributes. Els mètodes són **isDirectory(), isOther(), isRegularFile(), isSymbolicLink(), lastAccessTime, lastModifiedTime(), size().**

3 Annex I. Serialització.

Exemple de serialització d'objectes Contact

```
package ContactObjectStream;

/**
 * Contact.java
 * @author Jose
 * @version 1.00 2011/2/4
 */
import java.io.Serializable;
public class Contact implements Serializable {
      private String name;
      private String phone;
      private int age;

    public Contact() { }

    public Contact(String name, String phone, int age)
    {
      this.name = name;
      this.phone = phone;
      this.age = age;
    }
    //accessors
    public String getName() {return name;}
    public void setName(String name) {this.name=name;}
    public String getPhone() {return phone;}
    public void setPhone(String phone) {this.phone=phone;}
    public int getAge() {return age;}
    public void setAge(int age) {this.age=age;}

    @Override
    public String toString()
    {
            return ( "{Contact [name="+getName()+"][phone="+getPhone()+"]
[age="+ getAge()+"]}" );
    }
}
```

```
package ContactObjectStream;

/**
 * @(#)ContactObjectStream.java
 *
 * @author Jose
 * @version 1.00 2011/2/15
 */
import java.util.*;
import java.io.*;

public class ContactObjectStream {

    private static final String FILE_NAME = "agenda.txt";

    public ContactObjectStream() {
    }
    public static void main(String[] args) {
      ContactObjectStream pp = new ContactObjectStream();
        List<Contact> agenda = new ArrayList<>();
        agenda.add(new Contact("Paco","931111111",25));
```

```
        agenda.add(new Contact("Montse","932222222",33));
        agenda.add(new Contact("Pedro","93333333",27));
        agenda.add(new Contact("Toni","934444444",42));
        agenda.add(new Contact("Marta","935555555",29));
        pp.showPhoneBook(agenda);
        pp.savePhoneBook(agenda);
        List<Contact> copia =pp.readPhoneBook();
        pp.showPhoneBook(copia);
    }

    /**
     * showContacts()
     * Prints the list of contacts passed as a parameter to the console
     * @param ag List of contacts
     */
    private void showPhoneBook(List<Contact> ag)
    {
      for (Contact p: ag)
      {
            System.out.println(p.toString());
      }
    }

    /**
     * savePhoneBook()
     * Stores the phonebook to a file
     * @param ag list of contacts
     */
    private void savePhoneBook(List<Contact> ag)
    {
      System.out.println("Saving phonebook ...");
      int i=0;
      try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(FILE_NAME))) {

            for (Contact p: ag)
            {
                  oos.writeObject(p);
                  i++;
            }
            System.out.println(i+" contacts have been written to file
"+FILE_NAME+".");
      }
      catch (FileNotFoundException fnfe) {
            System.out.println("File not found: "+fnfe.getMessage());
      }
      catch (IOException ioe) {
            System.out.println("Output error: "+ioe.getMessage());
      }
    }

    /**
     * readPhoneBook()
     * Reads a list of contacts from a file
     * @return a list of contacts
     */
    private List<Contact> readPhoneBook() {

        System.out.println("Reading phonebook ...");

        List<Contact> pb = new ArrayList<>();
        int i=0;
        try (ObjectInputStream ois = new ObjectInputStream(new
```

```
FileInputStream(FILE_NAME))) {
            Object obj;
            while ( ( obj=ois.readObject() ) != null)
            {
                if ( obj instanceof Contact )
                {
                    Contact p = (Contact) obj;
                    pb.add(p);
                    i++;
                }
            }
        }
        catch (EOFException eofe) {       }
        catch (FileNotFoundException fnfe) {
                System.out.println("File not found: "+fnfe.getMessage());
                pb=null;
        }
        catch (IOException ioe) {
                System.out.println("Input error: "+ioe.getMessage());
                pb=null;
        }
        finally {
                System.out.println(i+" contacts have been read from file
"+FILE_NAME+".");
                return pb;
        }
    }
}
```

4 Annex II. La llibreria NIO

Exemple bàsic de Channel

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *
 * @author Carles
 */
public class BufferChannel1 {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        try (RandomAccessFile aFile = new RandomAccessFile("nio-data.txt",
"rw")) {
            FileChannel inChannel = aFile.getChannel();

            //create buffer with capacity of 50 bytes
            ByteBuffer buf = ByteBuffer.allocate(50);

            int bytesRead = inChannel.read(buf); //read into buffer.
            while (bytesRead != -1) {

                buf.flip();  //make buffer ready for read

                while(buf.hasRemaining()){
                    System.out.print((char) buf.get()); // read 1 byte at a
time
                }

                buf.clear(); //make buffer ready for writing
                bytesRead = inChannel.read(buf);
            }
        } catch (FileNotFoundException ex) {
            Logger.getLogger(BufferChannel1.class.getName()).log(Level.SEVERE,
null, ex);
        } catch (IOException ex) {
            Logger.getLogger(BufferChannel1.class.getName()).log(Level.SEVERE,
null, ex);
        }

    }

}
```