

Laboratorio di Reti
Lezione 5
Concurrent Collections
Java I/O, Serializzazione

12/10/2021
Federica Paganelli

Concurrent collections

CONCURRENT COLLECTIONS

- le collezioni sincronizzate garantiscono la thread-safety a discapito della scalabilità del programma
- il livello di sincronizzazione di una collezione sincronizzata può essere troppo stringente:
 - una hash table ha diversi buckets
 - perchè sincronizzare l'intera struttura, se si vuole accedere ad un solo bucket?
- Idea: accettare un **compromesso sulla semantica delle operazioni**, per mantenere un buon livello di **concorrenza** ed una buona **scalabilità**

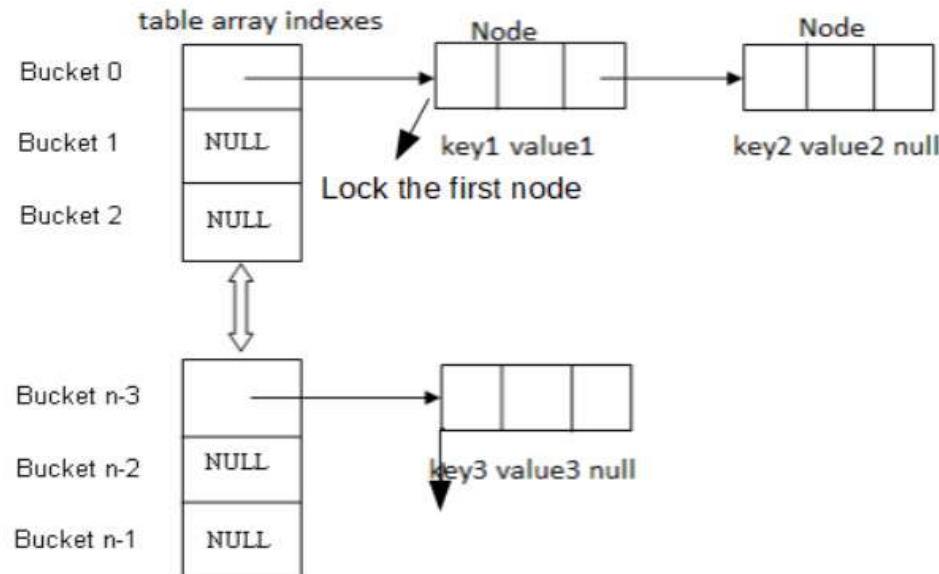
CONCURRENT COLLECTIONS

- **Concurrent Collections:**
 - implementate in `java.util.concurrent`
 - superano l'approccio “sincronizza l'intera struttura dati” tipico delle collezioni sincronizzate garantendo quindi un supporto più sofisticato per la sincronizzazione
 - approccio alternativo rispetto a `Collections.synchronizedCollections()`

CONCURRENT HASH MAP

lock striping invece di una sola lock per tutta la struttura

- Hashmap divisa in sezioni (bucket)
- utilizzate 16 locks per controllare l'accesso alla struttura
- Un numero arbitrario di reader e un numero fisso massimo di writer che lavorano simultaneamente (16 per default, può essere incrementato)
- reader e writer possono “convivere”
- possibili write simultanee se modificano sezioni diverse della tabella



CONCURRENT COLLECTIONS

```
ConcurrentHashMap <String,String> concurrentMap = new  
    ConcurrentHashMap <String,String>();
```

- Vantaggi
 - ottimizzazione degli accessi, non una singola lock
 - overlapping di operazioni di operazioni di scrittura su parti diverse della struttura
 - vantaggio: maggior livello di concorrenza e quindi miglioramento di performance e scalabilità
- Svantaggi
 - “approssimazione” della semantica di alcune operazioni: size(), isEmpty()) (i.e. restituiscono un valore approssimato)
 - non ha senso utilizzare synchronized per bloccare l’accesso all’intera struttura

Come rendere atomiche operazioni composte?

CONCURRENT HASH MAP

- per risolvere il problema della sincronizzazione delle operazioni composte da più azioni elementari, ConcurrentHashMap offre alcune nuove operazioni eseguite atomicamente, *vedi ad esempio i metodi seguenti (con semantica put if absent, remove if equal, replace if equal)*

```
public interface ConcurrentMap<K,V> extends Map<K,V> {  
    putIfAbsent(K key, V value); // If the specified key is not  
    already associated with a value, associate it with the  
    given value  
  
    boolean remove(K key, V value); // Removes the entry for a  
    key only if currently mapped to a given value  
  
    boolean replace(K key, V oldValue, V newValue); // Replaces  
    the entry for a key only if currently mapped to a  
    given value oldValue.
```

JAVA COLLECTION FRAMEWORK: Iteratori

- Iteratori:
 - oggetto di supporto usato per accedere agli elementi di una collezione, uno alla volta e in sequenza
 - associato ad un oggetto collezione (lavora su uno specifico insieme o lista, o map)
 - deve conoscere (e poter accedere) alla rappresentazione interna della classe che implementa la collezione (tabella hash, albero, array, lista puntata, ecc...)
- L'interfaccia **Collection** contiene il metodo **iterator()** che restituisce un iteratore per una collezione
 - le diverse implementazioni di **Collection** implementano il metodo **iterator()** restituendo un oggetto iteratore specifico per quel tipo di collezione
 - l'interfaccia **Iterator** prevede tutti i metodi necessari per usare un iteratore, senza conoscere alcun dettaglio implementativo

JAVA COLLECTION FRAMEWORK: Iteratori

- L'iteratore non ha alcuna funzione che lo "resetti"
 - una volta iniziata la scansione, non si può fare tornare indietro l'iteratore
 - una volta finita la scansione è necessario crearne uno nuovo
- schema generale per l'uso di un iteratore

```
// collezione di oggetti di tipo T che vogliamo scandire
Collection<T> c = ....
...

// iteratore specifico per la collezione c
Iterator<T> it = c.iterator()

// finche'non abbiamo raggiunto l'ultimo elemento
while (it.hasNext()) {
    // ottieni un riferimento all'oggetto corrente, ed avanza
    T e = it.next();
    ....    // usa l'oggetto corrente (anche rimuovendolo)
}
```

JAVA COLLECTION FRAMEWORK: Iteratori

```
HashSet<Integer> set = new HashSet<Integer>();  
  
.....  
  
Iterator<Integer> it = set.iterator()  
  
while (it.hasNext()) {  
    Integer i = it.next();  
    if (i % 2 == 0)  
        it.remove();  
    else  
        System.out.println(i);  
}
```

CONCURRENT HASH MAP:iteratore

- La struttura può essere modificata mentre viene scorsa mediante un iteratore
 - non necessario effettuare la lock sull'intera struttura
 - non viene sollevata **ConcurrentModificationException**
 - ma...le modifiche concorrenti possono non risultare visibili all'iteratore che scorre la struttura, i.e. un iteratore può/non può catturare le modifiche effettuate sulla collezione dopo la creazione di tale iteratore
-> **weakly consistent iterators**

- Da JavaDocs:

"The view's iterator is a "weakly consistent" iterator that will never throw ConcurrentModificationException, and guarantees to traverse elements as they existed upon construction of the iterator, and may (but is not guaranteed to) reflect any modifications subsequent to construction."

UN ITERATOR WEAKLY CONSISTENT

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.Iterator;
public class FailSafeItr {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> map = new
        ConcurrentHashMap<String, Integer>();
        map.put("ONE", 1);
        map.put("TWO", 2);
        map.put("THREE", 3);
        map.put("FOUR", 4);
        Iterator <String> it = map.keySet().iterator();
        while (it.hasNext()) {
            String key = (String)it.next();
            System.out.println(key + " : " + map.get(key));
            // Notice, it has not created separate copy
            // It will print 7
            map.put("SEVEN", 7); }}}
// the program prints ONE : 1 FOUR : 4 TWO : 2 THREE : 3 SEVEN :17
```



Fail-safe vs Fail-fast iterators

- L'iteratore della `ConcurrentHashMap` è fail-safe
- **Fail-safe iterator**
 - non lancia `ConcurrentModificationException` anche se la `ConcurrentHashMap` a cui si riferisce viene modificato dopo che inizia l'iterazione
- Altre Collection (es. `HashMap`) offrono iteratori fail-fast
- **Fail-fast iterator**
 - Viene lanciata una *`ConcurrentModificationException`* se un thread modifica una Collection (tipicamente da un altro thread, ma non necessariamente).

FAIL FAST: HASHMAP

```
import java.util.HashMap; import java.util.Iterator;import java.util.Map;
public class FailFastExample {
    public static void main(String[] args) {
        Map<String, String> cityCode = new HashMap<String, String>();
        cityCode.put("Delhi", "India");
        cityCode.put("Moscow", "Russia");
        cityCode.put("New York", "USA");
        Iterator<String> iterator = cityCode.keySet().iterator();
        while (iterator.hasNext()) {
            System.out.println(cityCode.get(iterator.next()));
            cityCode.put("Istanbul", "Turkey"); }}}}
```

India

Exception in thread "main"

java.util.ConcurrentModificationException

at java.util.HashMap\$HashIterator.nextNode(Unknown Source)

at java.util.HashMap\$KeyIterator.next(Unknown Source)

at FailFastExample.main(FailFastExample.java:19)



CONCURRENT HASH MAP: UN CASO D'USO

- simulazione di un magazzino rappresentato da una **ConcurrentHashMap**
 - chiave: nome della merce
 - valore: presenza o meno della merce in magazzino (**in stock, sold out**)
- simulazione effettuata mediante i seguenti threads:
 - **HashMapStock**: caricamento iniziale della merce nel magazzino
 - **HashMapEmpty**: simula uscita degli articoli dal magazzino
 - **HashMapReplace**: simula ricarico articoli in magazzino

CONCURRENT HASH MAP: UN CASO D'USO

```
import java.util.concurrent.*;

public class HashMapExample {

    public static void main(String args[]) {

        ConcurrentHashMap <String,String> concurrentMap = new
            ConcurrentHashMap <String,String>();

        ThreadPoolExecutor executor= (ThreadPoolExecutor)
            Executors.newCachedThreadPool();

        executor.execute(new HashMapStock(concurrentMap));
        executor.execute(new HashMapEmpty(concurrentMap));
        executor.execute(new HashMapReplace(concurrentMap));

    }

}
```


CONCURRENT HASH MAP: UN CASO D'USO

```
import java.util.concurrent.ConcurrentHashMap;

public class HashMapStock implements Runnable{
    ConcurrentHashMap <String,String> cMap = new
        ConcurrentHashMap <String,String>();
    public HashMapStock(ConcurrentHashMap <String, String> cMap)
        {this.cMap= cMap;}
    public void run () {
        cMap.putIfAbsent ("Socks", "in stock");
        System.out.println("Socks in stock");
        cMap.putIfAbsent ("Shirts", "in stock");
        System.out.println("Shirts in stock");
        cMap.putIfAbsent ("Pants", "in stock");
        System.out.println("Pants in stock");
        cMap.putIfAbsent ("Shoes", "in stock");
        System.out.println("Shoes in stock"); }}}
```

CONCURRENT HASH MAP: UN CASO D'USO

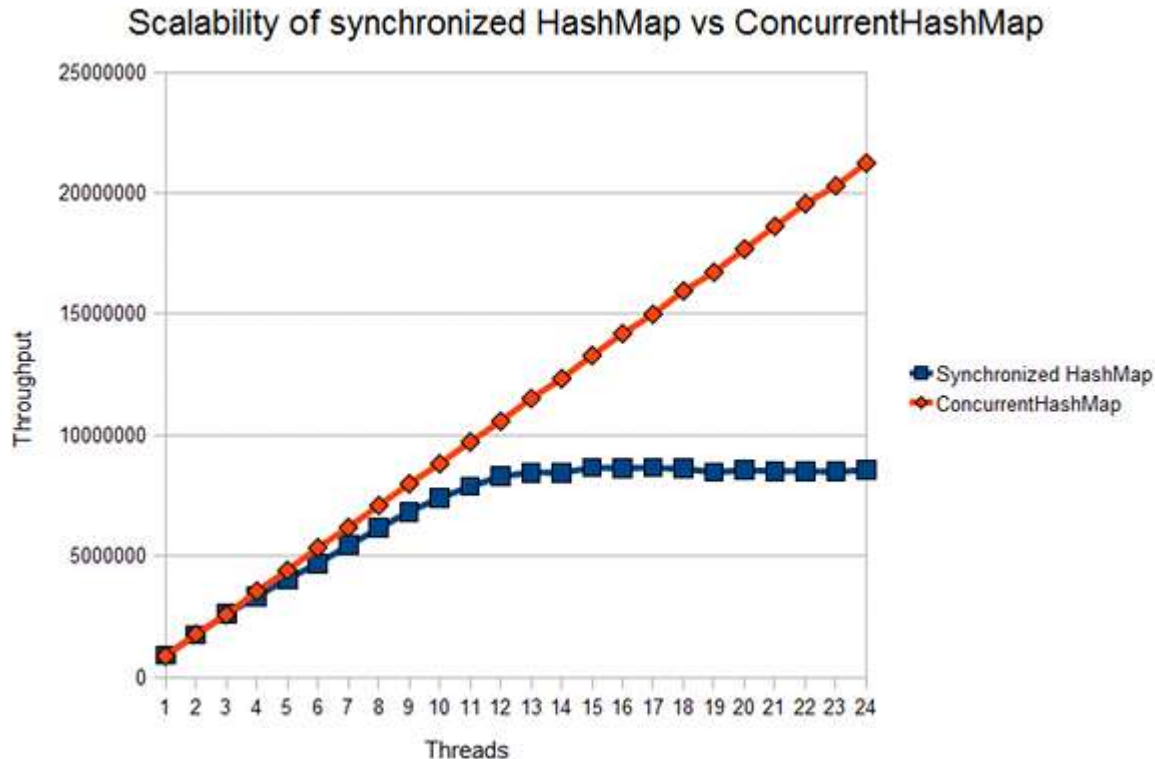
```
import java.util.Iterator; import java.util.Random;
import java.util.concurrent.*;
public class HashMapEmpty implements Runnable{
    ConcurrentHashMap <String,String> cMap = new ConcurrentHashMap
                                                <String,String>();

    public HashMapEmpty (ConcurrentHashMap <String, String> cMap)
        {this.cMap= cMap;}

    public void run ()
        {while(true) {
            Iterator <String> i =cMap.keySet().iterator();
            while(i.hasNext())
                {String s= i.next();
                 if (cMap.replace(s,"in stock", "sold out"))
                     {System.out.println(s+"sold out"); }
                 try { Thread.sleep(new Random().nextInt(500));
                     } catch (InterruptedException e)
                     {e.printStackTrace();}}}}}
```

Come può essere implementata HashMapReplace?

CONCURRENT HASHMAP



©https://www.javamex.com/tutorials/concurrenthashmap_scalability.shtml

throughput: the total number of accesses to the map performed by all threads together in two seconds.

Intel i7-720QM (Quad Core Hyperthreading) machine running Hotspot version 1.6.0_18 under Windows 7.

Java I/O

JAVA I/O

- I/O: programmi che recuperano informazioni da una sorgente esterna o la inviano ad una sorgente esterna. Ad es:
 - Keyboard, monitor, stampanti
 - file system: files e directories
 - connessioni di rete
- diversi tipi di device di input/output: se il linguaggio dovesse gestire ogni tipo di device come caso speciale, la complessità sarebbe enorme

JAVA I/O

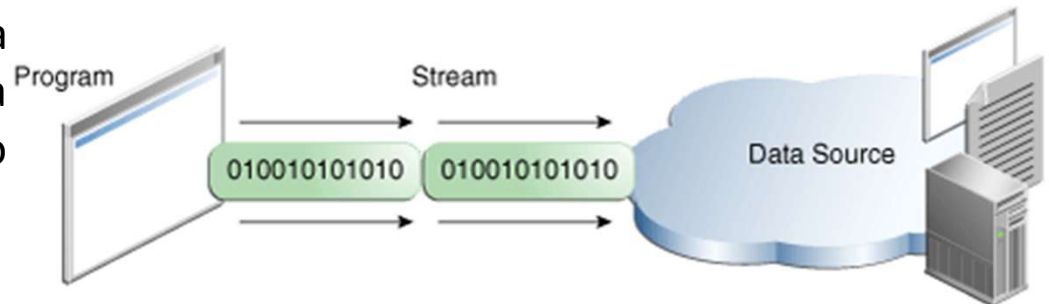
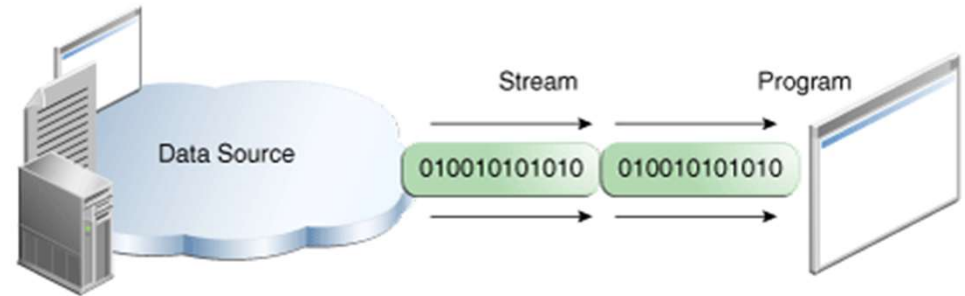
- necessità di **astrazioni opportune** per rappresentare un device di I/O
- definizione di un insieme di **astrazioni per la gestione dell'I/O**: una delle parti più complesse di un linguaggio
 - in JAVA, la principale (e la prima definita) astrazione è basata sul concetto di **stream (o flusso)**
 - Alcune ulteriori astrazioni per l'I/O
 - File: per manipolare descrittori di files
 - Channels

JAVA PACKAGES PER I/O

- Programmare semplici operazioni di I/O può risultare molto semplice. Diverso è il caso in cui si voglia programmare operazioni di I/O efficienti e portabili
- molti packages per I/O
- Classificazione I/O packages:
- **JAVA IO API**, package standard I/O contenuto in java.io, introdotto già a partire dalla JDK 1.0
 - stream based I/O
 - lavora su bytes e caratteri
- **JAVA NIO API**: (New IO) funzionalità simili a JAVA IO, ma con comportamento non bloccante: migliori performance in alcuni scenari. Introdotta in JDK 1.4
 - buffer based I/O
- **JAVA NIO.2**: alcuni miglioramenti rispetto a JAVA NIO

STREAM-BASED I/O

- stream: una sequenza di dati (un flusso di informazione di lunghezza indefinita)
 - astrazione che rappresenta una connessione tra un programma JAVA ed un dispositivo esterno (file, buffer di memoria, connessione di rete...)
 - un “canale” tra una sorgente ed una destinazione (dal programma ad un dispositivo o viceversa), da un estremo entrano dati, dall'altro escono



- l'applicazione può inserire dati ad un capo dello stream
- i dati fluiscono verso la destinazione e possono essere estratti dall'altro capo dello stream
- E viceversa

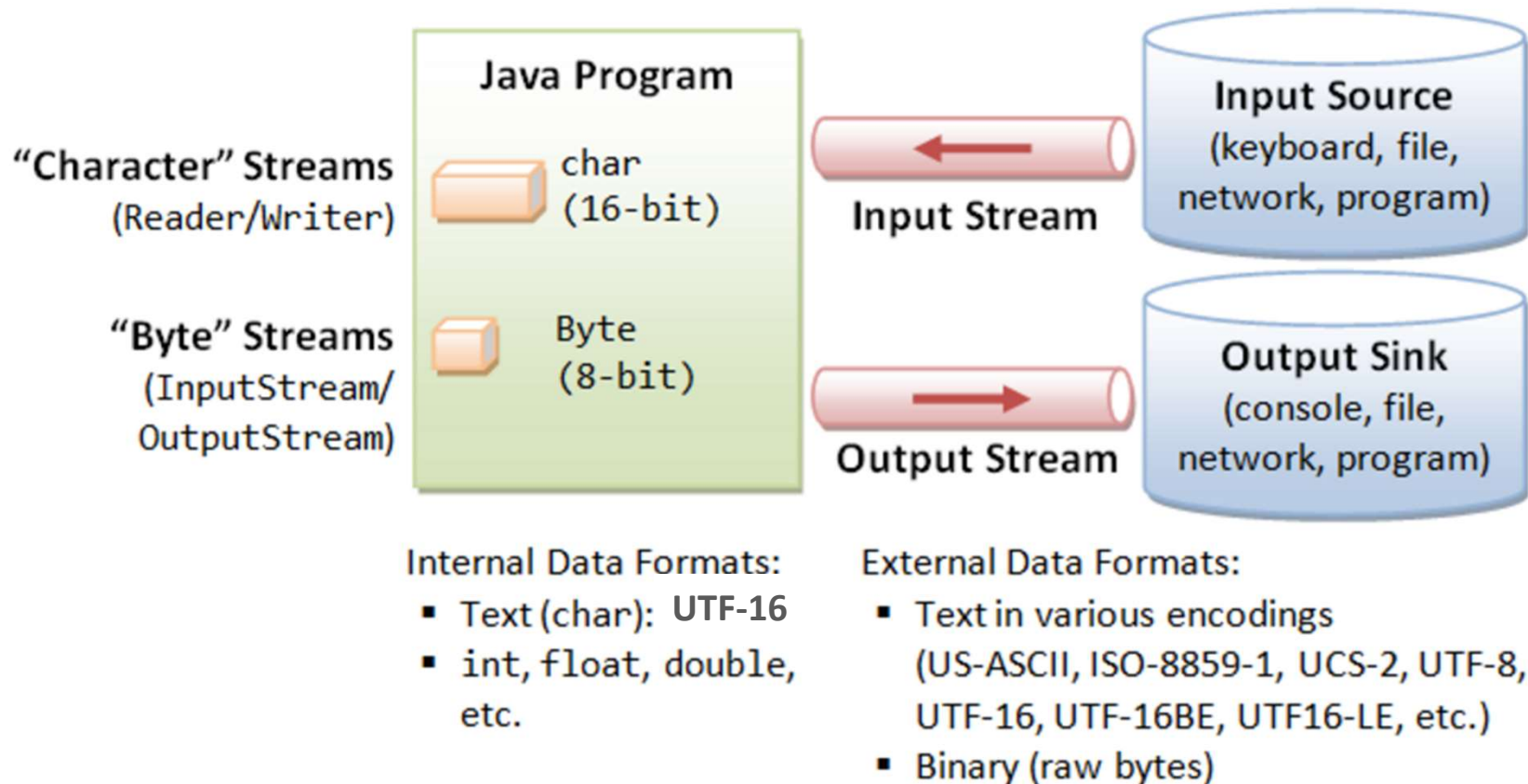
Streams in Java

- accesso **sequenziale**
 - I dati vengono letti e scritti dal dispositivo come una sequenza dove non è possibile tornare indietro
- mantengono l'ordinamento FIFO
- one way: read only o write only (a parte file ad accesso random)
 - se un programma ha bisogno di dati in input ed output, è necessario aprire due stream, uno in input e l'altro in output
- **bloccanti**: quando un'applicazione legge un dato dallo stream (o lo scrive) si blocca **finché l'operazione non è completata**
- Lettura e scrittura sono operazioni indipendenti, non è richiesta una corrispondenza stretta tra letture/scritture
 - ad esempio: una unica scrittura inietta 100 bytes sullo stream, che vengono letti con due write successive 'all'altro capo dello stream', la prima legge 20 bytes, la seconda 80 bytes

Tipi di stream

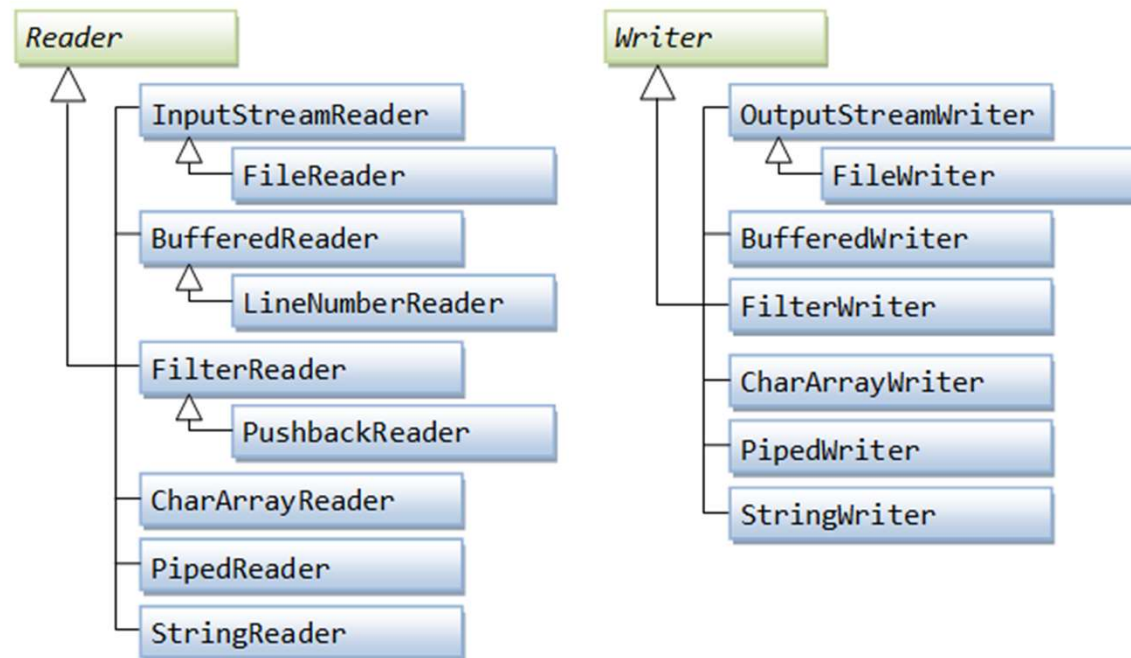
- Più di 60 tipi di diversi stream, basati su 4 classi astratte fondamentali: **InputStream, OutputStream, Reader and Writer**
- Due tipi principali di stream
 - **Stream di bytes:** servono a leggere/scrivere sequenze di byte.
 - Classi astratte: *InputStream* e *OutputStream*
 - dati copiati byte a byte senza effettuare alcuna traduzione
 - ideale per leggere/scrivere file binari, es. un'immagine, la codifica di un video...
 - **Stream di caratteri:** leggere/scrivere sequenze di caratteri UNICODE a 16 bit. L'I/O basato su character stream traduce automaticamente questo formato interno da e verso il set di caratteri locale.
 - Classi astratte: *Reader* e *Writer*.

Tipi di stream



Character streams

- Superclassi astratte Reader e Writer
 - *Reader*: contiene una parziale implementazione e le API (metodi e campi) per realizzare stream che leggono caratteri
 - *Writer*: contiene una parziale implementazione e le API (metodi e campi) per realizzare stream che scrivono caratteri
 - Sottoclassi di Reader e Writer implementano stream specializzati

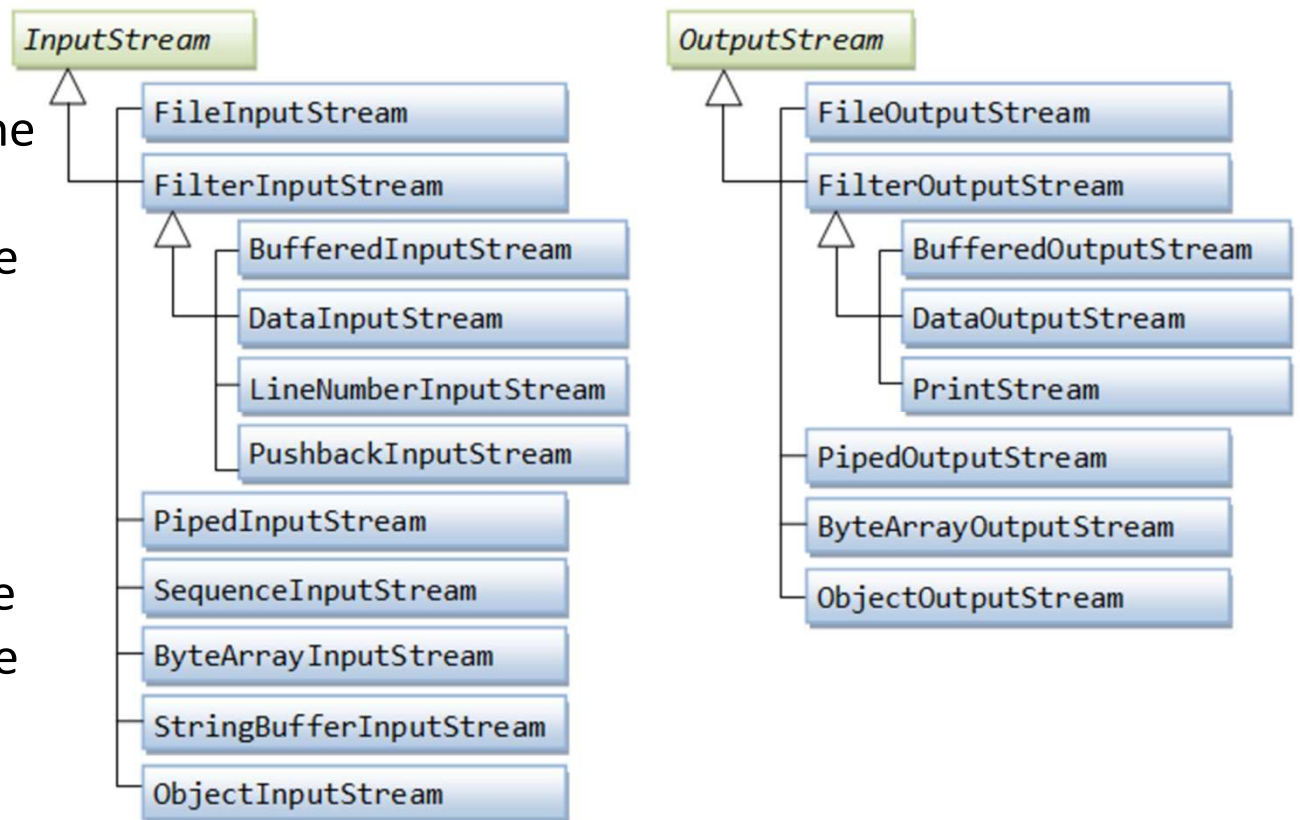


Stream di byte

- InputStream/OutputStream: possono essere “attaccati” ad ogni tipo di device di input/output:
- il programma apre uno stream per ricevere/inviare dati

- superclassi astratte

- *InputStream*: contiene una parziale implementazione e le API per realizzare stream che **leggono** byte
- *OutputStream*: contiene una parziale implementazione e le API per realizzare stream che **scrivono** byte



Input/OutputStream

- **InputStream/OutputStream**: classi astratte che forniscono operazioni base.
- Sottoclassi concrete che forniscono implementazioni per tipi diversi di I/O:
 - **console: System.in, System.out**, permette di leggere byte dalla tastiera/scrivere byte sul display
 - **files: FileInputStream/FileOutputStream** per leggere/scrivere byte a byte da un file
 - **in-memory buffers**: per trasferimento di dati da una parte all'altra di un programma JAVA.

ByteArrayInputStream, ByteArrayOutputStream

esempio: utile per generare uno stream di byte, per poi trasferirlo in un pacchetto UDP.

- **Connessioni tcp**
- **pipes**

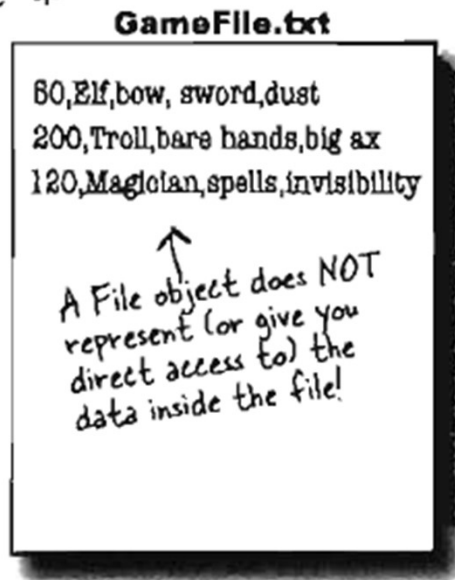
INPUT/OUTPUT STREAM

- **InputStream:** metodi
 - **int read()**
 - » un byte dallo stream letto come un intero unsigned tra 0 e 255
 - » restituisce -1 se viene individuata la “fine dello stream”
 - » solleva una IOException se c'è un errore di I/O
 - » bloccante finché sono disponibili byte, viene rilevata la fine dello stream, o viene lanciata un'eccezione
 - **int read(byte[] bytes, int offset, int length)**
 - » Tentativo di leggere "length" bytes, li memorizza nel bytes array, iniziando da offset (possono essere disponibili meno byte di length)
 - » Viene restituito il numero di byte letti
 - **public int read(byte[] bytes)**
 - riempie il vettore passato in input con tutti i dati disponibili sullo stream fino alla capacità massima dell'array e restituisce il numero di byte letti
 - equivalente a read(bytes, 0, bytes.length)
 - **available(), close()** e molti altri metodi

JAVA.IO.FILE



A File object represents the filename "GameFile.txt"



Classe File: rappresentazione astratta di file e nome del percorso

Un'istanza della classe File descrive:

- path per l'individuazione del file o di una directory
- non una semplice stringa:
 - metodi per verificare l'esistenza del path, restituire meta-informazioni sul file,...
 - quando si vuole stabilire una connessione (stream) con un file, occorre passare come parametro:
 - un oggetto di tipo File
 - una stringa...

JAVA.IO.FILE

```
public class ListFiles {  
    public static void main(String[] args) {  
        File dir = new File(".");    // current working directory  
        if (dir.isDirectory()) {  
            // List only files that meet the filtering criteria  
            String[] files = dir.list();  
            for (String file : files) {  
                if (file.endsWith(".java"))  
                    System.out.println(file);  
            }  
        }  
    }  
}
```

Crea una nuova istanza di File prendendo in input la stringa del percorso.

Costruttore File(String pathname): crea una nuova istanza di File prendendo in input la stringa del percorso.

`list()`: returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.


Copia di File

```
import java.io.*;
public class CopyFile {
    public static void main(String[] args) throws IOException{
        OutputStream fileCopy=null;
        InputStream fileIn=null;
        try {
            fileIn = new FileInputStream("./src/file.txt");
            fileCopy = new FileOutputStream("./src/copy.txt");
            int c;
            while((c = fileIn.read() )!=-1)
                fileCopy.write(c);
        }
        finally {
            if(fileIn!=null) fileIn.close();
            if(fileCopy!=null) fileIn.close();
        }
    }
}
```


e.g.
FileNotFoundException



Loop di
lettura e
scrittura
byte a byte



Dopo l'utilizzo lo
stream deve essere
chiuso (close()), per
rilasciare le risorse ad
esso associate



Copia di File - 2

```
import java.io.*;
public class CopyFile_bytearray {
    public static void main(String[] args) throws IOException{
        OutputStream fileCopy=null;
        InputStream fileIn=null;
        try {
            fileIn = new FileInputStream("./src/Pianta-Gffloor.png");
            fileCopy = new FileOutputStream("./src/copyImage.png");
            long timeIn=System.currentTimeMillis();
            byte[] buffer = new byte[1024];
            while (fileIn.read(buffer) != -1)
                fileCopy.write(buffer);
            long timeOut=System.currentTimeMillis();
            System.out.printf("time needed is %d", timeOut-timeIn);
        }
        finally {
            if(fileIn!=null) fileIn.close();
            if(fileCopy!=null) fileCopy.close();
        }
    }
}
```

N.B. più veloce
dell'esempio
precedente

TRY FINALLY: PROBLEMI

```
private static void printFile() throws IOException {  
    InputStream input = null;  
    try {  
        input = new FileInputStream("file.txt");  
        int data = input.read();  
        while(data != -1){  
            data = input.read(); }  
    } finally {  
        if(input != null){  
            input.close();  
        }  
    }  
}
```

In rosso: Codice
che può
sollevare
eccezioni

il blocco finally viene eseguito anche se una eccezione viene rilevata nel blocco try
se un'eccezione viene sollevata anche nel blocco finally, solo questa viene propagata, e
l'eccezione del blocco try ignorata, anche se è più significativa

TRY WITH RESOURCES

- Costrutto introdotto in JAVA 7
- supporto per la sistematica chiusura delle risorse e/o connessioni aperte da un programma
- **try-with-resources** si occupa di chiudere automaticamente le risorse aperte.
- si utilizza un blocco try con uno o più argomenti tra parentesi -> gli argomenti sono le risorse che JAVA garantisce di chiudere al termine del blocco

```
private static void printFilewithResources() throws IOException {  
    try (InputStream input = new FileInputStream("file.txt")){  
        int data = input.read();  
        while(data != -1){  
            data = input.read(); }  
    }  
}
```

- suppressed exceptions:
 - quando si verificano delle eccezioni sia nel blocco try-with-resources , sia durante la chiusura, la JVM sopprime l'eccezione generata nella chiusura automatica.
 - possono essere inseriti blocchi catch e finally che vengono comunque eseguiti dopo la chiusura delle risorse.

TRY WITH RESOURCES

```
import java.io.*;

public class trywithresources {

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

        try( FileInputStream input = new FileInputStream(new
                                                    File("immagine.jpg"));

            BufferedInputStream bufferedInput = new
                BufferedInputStream(input)) {

            int data = bufferedInput.read();
            while(data != -1){
                System.out.print((char) data);
                data = bufferedInput.read();
            }

        }

    }

}
```

Dalla Javadoc:

The [try-with-resources](#) statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The [try-with-resources statement ensures that each resource is closed at the end of the statement](#). Any object that implements [java.lang.AutoCloseable](#), which includes all objects which implement [java.io.Closeable](#), can be used as a resource



JAVA: FILTER STREAMS

- Funzionalità utile: concatenare gli stream di base con degli stream filtro
- classi **FilterInputStream** and **FilterOutputStream** con diverse sottoclassi
 - **BufferedInputStream** e **BufferedOutputStream** implementano filtri che bufferizzano l'input da/l'output verso lo stream sottostante
 - **DataInputStream** and **DataOutputStream** implementano filtri che permettono di “formattare” i dati presenti sullo stream
 - altre sottoclassi (compressione di dati, etc,..)

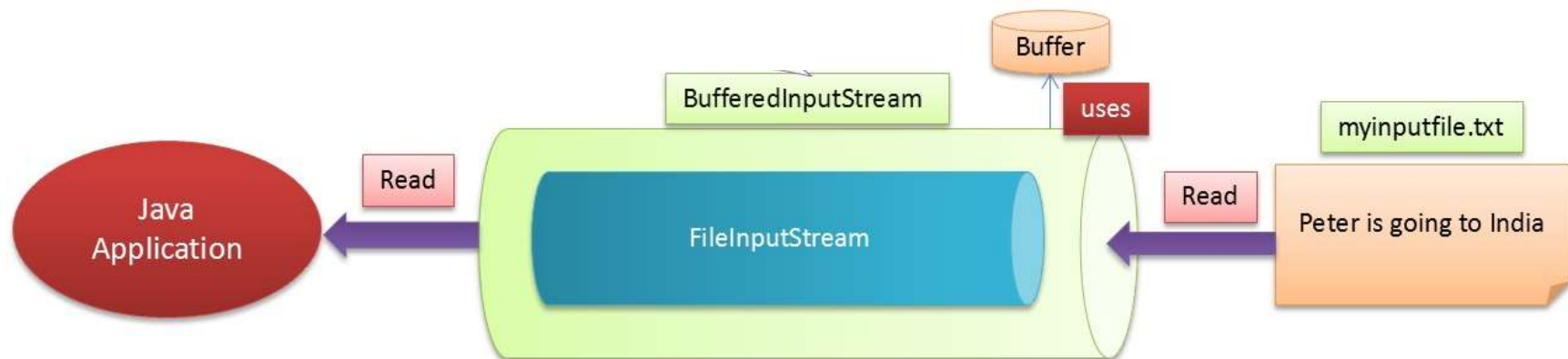
JAVA: FILTER STREAMS

- le classi filtro **BufferedInputStream** e **BufferedOutputStream** implementano una bufferizzazione per stream di input e di output, rispettivamente.
- i dati vengono scritti e letti in blocchi di bytes, invece che un solo blocco per volta
- quando si crea un **BufferedInputStream**, viene creato un buffer interno. Via via che i byte sono letti, il buffer interno viene riempito, più byte per volta. **Una read() su un BufferedInputStream è quindi una lettura dal buffer interno**
- miglioramento significativo della performance: più veloce rispetto a **InputStream**

JAVA: FILTER STREAMS

```
BufferedInputStream in = new BufferedInputStream  
    (new FileInputStream("primitives.data"));
```

```
BufferedOutputStream out = new BufferedOutputStream(new  
    FileOutputStream("primitives.data"))
```



BUFFERIZZAZIONE “UNDER THE HOOD”

```
import java.io.*;

public class FileCopyNoBufferJDK7 {
    public static void main(String[] args) {
        String inFileStr = "blue_i1.jpg";
        String outFileStr = "blue_i1_new.jpg";
        long startTime, elapsedTime; // for speed benchmarking
        // Check file length
        File fileIn = new File(inFileStr);
        System.out.println("File size is " + fileIn.length() + " bytes");
        // "try-with-resources" automatically closes all opened resources.
        try (FileInputStream in = new FileInputStream(inFileStr);
            FileOutputStream out = new FileOutputStream(outFileStr)) {

            startTime = System.nanoTime();
            int byteRead;
```

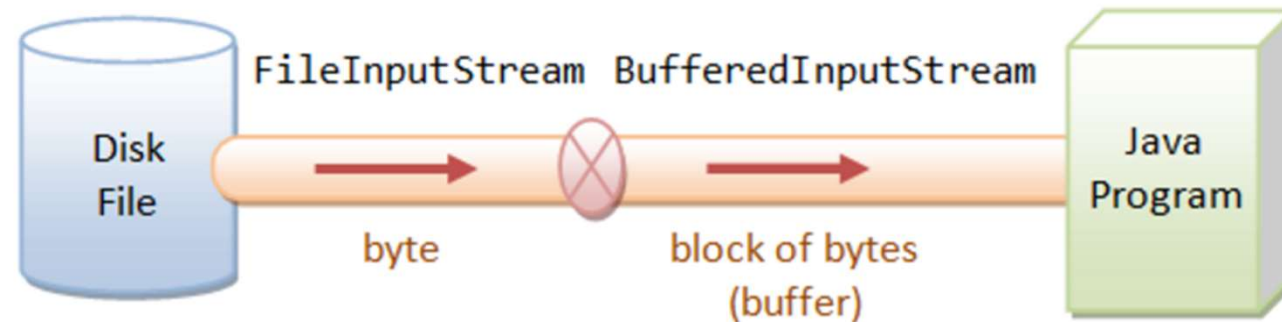
VALUTARE EFFETTI BUFFERIZZAZIONE

```
// Read a raw byte, returns an int of 0 to 255.
while ((byteRead = in.read()) != -1)
{
    out.write(byteRead);
}
elapsedTime = System.nanoTime() - startTime;
System.out.println("Elapsed Time is " + (elapsedTime /
1000000.0) + "msec");
} catch (IOException ex) { ex.printStackTrace(); }
}
}
```

File size is 955399 bytes

Elapsed Time is 4684.12669 msec

VALUTARE EFFETTI BUFFERIZZAZIONE



sostituendo:

try

```
(BufferedInputStream in = new BufferedInputStream(new
    FileInputStream(inFileStr));
    BufferedOutputStream out = new BufferedOutputStream(new
    FileOutputStream(outFileStr)))
{ ..... }
```

Si ottiene:

File size is 955399 bytes

Elapsed Time is 44.777895 msec

FORMATTED DATA STREAMS

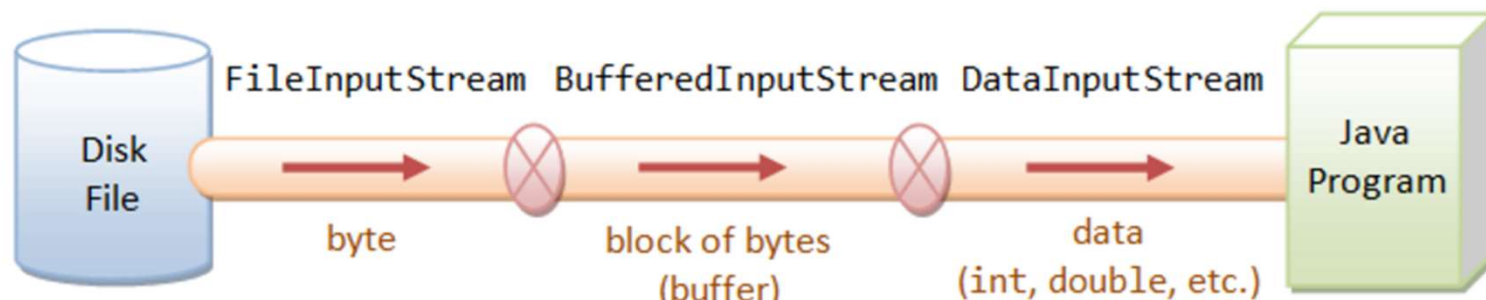
- **DataInputStream:** permette di leggere tipi di dati Java primitivi (e.g., int, double, ecc.) e String da un InputStream.
- Un'applicazione può usare DataOutputStream per scrivere dati che possono più tardi essere letti da un DataInputStream.

- Costruttore

`DataInputStream(InputStream in)`

- Alcuni metodi offerti

- `readBoolean`
- `readChar`
- `readInt`
- ...



FORMATTED DATA STREAMS

```
import java.io.*;

public class TestDataIOStream {
    public static void main(String[] args) {
        String filename = "data-out.dat"; String message =
        "Hi,$%&!";
        // Write primitives to an output file
        try (DataOutputStream out =
            new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream(filename)))) {
```

FORMATTED DATA STREAMS

```
out.writeByte(127);
out.writeShort(-1);
out.writeInt(43981);
out.writeLong(305419896);
out.writeFloat(11.22f);
out.writeDouble(55.66);
out.writeBoolean(true); out.writeBoolean(false);
for (int i = 0; i < message.length(); ++i) {
    out.writeChar(message.charAt(i)); }
out.writeChars(message);
out.writeBytes(message);
out.flush();
} catch (IOException ex) { ex.printStackTrace(); }
```

FORMATTED DATA STREAMS

```
// Read raw bytes and print in Hex
try (BufferedInputStream in =
    new BufferedInputStream(
        new FileInputStream(filename))) {
    int inByte;
    while ((inByte = in.read()) != -1) {
        System.out.printf("%02X ", inByte);
        System.out.println();// Print Hex codes
    }
    System.out.printf("%n%n");
} catch (IOException ex) {
    ex.printStackTrace();
}
```


FORMATTED DATA STREAMS

```
// Read primitives
try (DataInputStream in =
    new DataInputStream(
        new BufferedInputStream(
            new FileInputStream(filename)))) {
    System.out.println("byte:    " + in.readByte());
    System.out.println("short:   " + in.readShort());
    System.out.println("int:    " + in.readInt());
    System.out.println("long:   " + in.readLong());
    System.out.println("float:  " + in.readFloat());
    System.out.println("double: " + in.readDouble());
    System.out.println("boolean: " + in.readBoolean());
    System.out.println("boolean: " + in.readBoolean());
}
```

FORMATTED DATA STREAMS

```
System.out.print("char:   ");
for (int i = 0; i < message.length(); ++i) {
    System.out.print(in.readChar()); }
System.out.println();
System.out.print("chars:  ");
for (int i = 0; i < message.length(); ++i) {
    System.out.print(in.readChar()); }
System.out.println();
System.out.println();
System.out.print("bytes:  ");
for (int i = 0; i < message.length(); ++i) {
    System.out.print((char)in.readByte()); }
System.out.println();
}
catch (IOException ex) { ex.printStackTrace(); } } }
```

Object Serialization

JAVA SERIALIZATION

- Gli oggetti esistono in memoria fino a che la JVM è in esecuzione
- **Object serialization**: scrittura e la lettura di oggetti
 - si basa sulla possibilità di scrivere lo stato di un oggetto in una forma sequenziale, sufficiente per ricostruire l'oggetto quando viene riletto
- ogni oggetto è caratterizzato da uno stato e da un comportamento
 - comportamento: specificato dai metodi della classe
 - stato: “vive” con l’istanza dell’oggetto
- flattening/salvataggio dell’oggetto riguarda il suo stato e può avvenire in diversi modi
 - accedendo ai singoli campi dell’oggetto, salvando i valori ad esempio su un text file, scegliendo un opportuno formato
 - trasformando automaticamente l’oggetto o un grafo di oggetti in uno stream di byte, mediante il meccanismo della serializzazione

JAVA SERIALIZATION

- utilizzata in diversi contesti:

1) fornire un meccanismo di **persistenza** ai programmi, consentendo l'archiviazione di un oggetto su un file

- Ad es. **storage-and-retrieve**: ad esempio, memorizzare lo stato di una sessione e renderlo disponibile per sessioni successive della stessa applicazione

2) fornire un meccanismo di interoperabilità mediante oggetti condivisi tra diverse applicazioni (es. un applicativo di magazzino produce un ordine, un altro lo elabora)

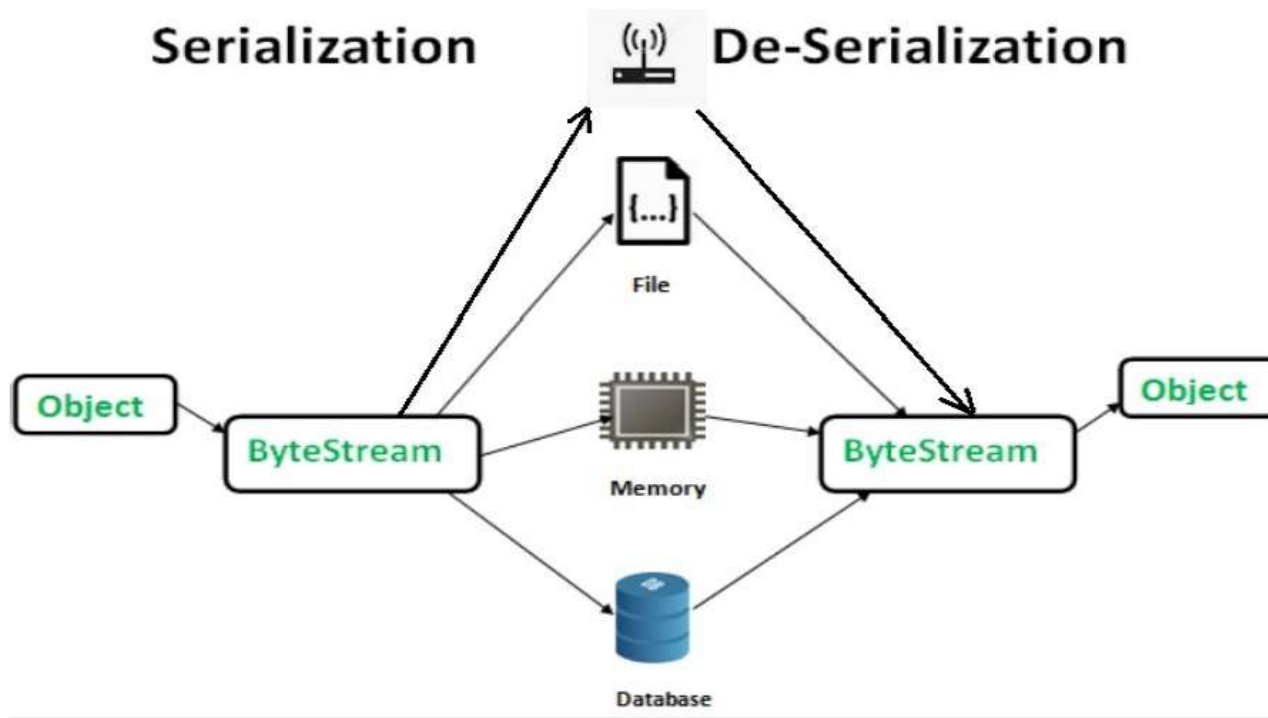
- inviare oggetti su uno stream che rappresenta una connessione TCP
- inviare oggetti che sono parametri di metodi invocati via RMI
- generare uno stream di byte da un oggetto, per poi ricavare dallo stream un pacchetto di byte da inserire in un pacchetto UDP.

1 Object on the heap

2 Object serialized



JAVA SERIALIZATION



- l'oggetto serializzato può quindi essere scritto su un qualsiasi stream di output
- come useremo la serializzazione in questo corso?
 - per inviare oggetti
 - su uno stream che rappresenta una connessione TCP
 - come parametri di metodi invocati via Remote Method Invocation
 - per generare pacchetti UDP, si scrive l'oggetto serializzato su uno stream di byte e poi si genera un pacchetto UDP

JAVA SERIALIZATION

- gli oggetti esistono nella memoria principale fino a che la JVM è in esecuzione:
 - la serializzazione consente di mantenere la persistenza degli oggetti al di fuori del ciclo di vita della JVM
 - creare una rappresentazione dell'oggetto indipendente dalla JVM che ha generato l'oggetto stesso
 - un'altra JVM, se ha accesso alla classe (class file) ed all'oggetto serializzato, può ricostruire l'oggetto
- limitata interoperabilità: utilizzabile quando sia l'applicazione che serializza l'oggetto che quella che lo deserializza sono scritte in JAVA
- per aumentare l'interoperabilità occorre utilizzare altre soluzioni:
 - il formato standard JSON (presentato in una prossima lezione...)
 - serializzazione in XML
 - ...altri formati....

JAVA STANDARD SERIALIZATION

- **Serializable Interface**

- per rendere un oggetto “persistente”, l'oggetto deve implementare l'interfaccia **Serializable**
- marker interface: è un'interfaccia completamente vuota
- marker interface: attiva il meccanismo di serializzazione che verifica se la classe ha le caratteristiche per essere serializzata
- controllo limitato sul meccanismo di linearizzazione dei dati
- tutti i tipi di dato primitivi sono serializzabili
- gli oggetti non-primitivi, se implementano Serializable, sono serializzabili (a parte alcuni oggetti - controllare documentazione)

JAVA SERIALIZATION

In **rosso** le parti relative alla serializzazione

```
import java.io.Serializable;
import java.util.Date;
import java.util.Calendar;

public class PersistentTime implements Serializable {
    private static final long serialVersionUID = 1;
    private Date time;
    public PersistentTime() {
        time = Calendar.getInstance().getTime();
    }
    public Date getTime() {
        return time;
    }
}
```

Regola #1: per serializzare un oggetto persistente la classe di cui l'oggetto è istanza deve implementare l'interfaccia Serializable oppure ereditare l'implementazione dalla sua gerarchia di classi

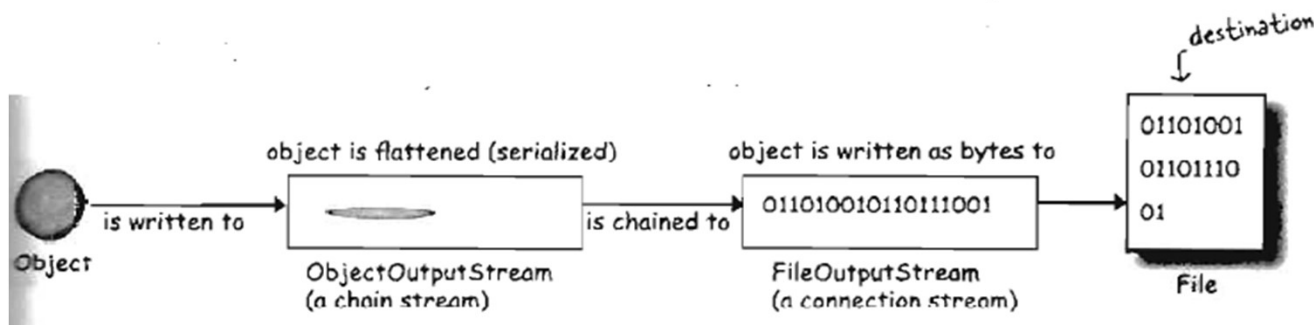
JAVA SERIALIZATION

- la serializzazione vera e propria è gestita dalla classe **ObjectOutputStream**
- Tale stream deve essere concatenato ad uno stream di bytes, che può essere:
 - **FileOutputStream**
 - uno stream di bytes associato ad un socket
 - **ByteArrayOutputStream**
 - ...

JAVA SERIALIZATION

```
import java.io.*;

public class FlattenTime {
    public static void main(String [] args) {
        String filename = "time.ser";
        if(args.length > 0)
            filename = args[0];
        PersistentTime time = new PersistentTime();
        try(FileOutputStream fos = new FileOutputStream(filename);
            ObjectOutputStream out = new ObjectOutputStream(fos);) {
            out.writeObject(time);
        }
        catch(IOException ex) {ex.printStackTrace();}}
```

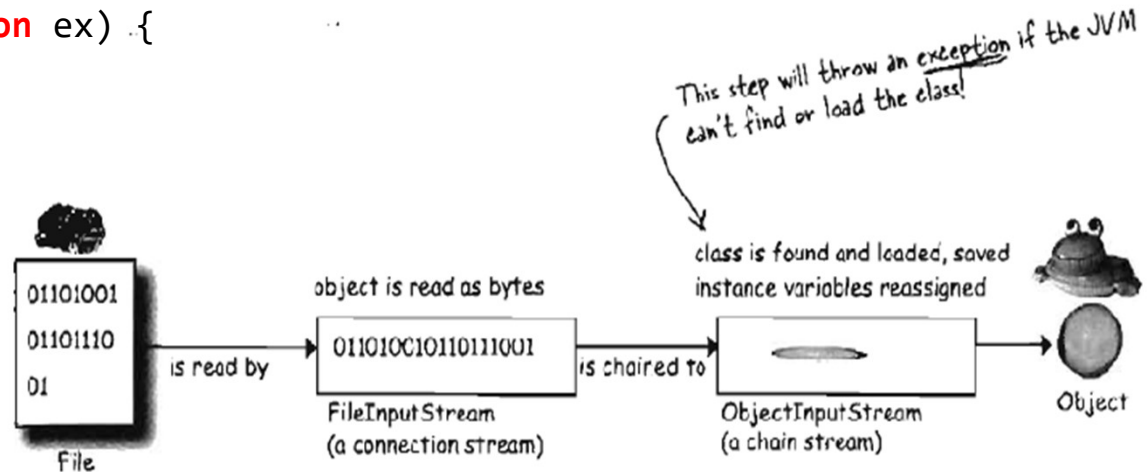


JAVA DESERIALIZATION

```
public class InflateTime {  
    public static void main(String [] args) {  
        String filename = "time.ser";  
        if(args.length > 0)  
            filename = args[0];  
        PersistentTime time = null;  
        try(FileInputStream fis = new FileInputStream(filename);  
            ObjectInputStream in = new ObjectInputStream(fis);) {  
            time = (PersistentTime)in.readObject();  
        }  
        catch(IOException ex) {  
            ex.printStackTrace();  
        }  
        catch(ClassNotFoundException ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

ClassNotFoundException:

l'applicazione tenta di caricare una classe, ma non trova nessuna definizione di classe con quel nome



JAVA DESERIALIZATION

```
// print out restored time
System.out.println("Flattened time: " + time.getTime());
System.out.println();
    // print out the current time
System.out.println("Current time: "+
                    Calendar.getInstance().getTime());}
}
```

Output ottenuto:

Flattened time: Mon Mar 12 19:11:55 CET 2012

Current time: Mon Mar 12 19:16:24 CET 2012

JAVA STANDARD SERIALIZATION

- Il metodo **readObject()** legge la sequenza di bytes memorizzati in precedenza e crea un oggetto che è l'esatta replica di quello originale
 - **readObject** può leggere qualsiasi tipo di oggetto, è necessario effettuare un **cast** al tipo corretto dell'oggetto
- la JVM determina, mediante informazione memorizzata nell'oggetto serializzato, il tipo della classe dell'oggetto e tenta di caricare quella classe
- se non la trova viene sollevata una **ClassNotFoundException** ed il processo di deserializzazione viene abortito
- viene creato un nuovo oggetto sullo heap, ma non viene invocato il costruttore dell'oggetto, in quanto i campi dell'oggetto vengono inizializzati con i valori deserializzati

COSA NON E' SERIALIZZABILE?

- oggetti contenenti riferimenti specifici alla JVM o al SO (JAVA native class)
 - Esempio: **Thread, OutputStream, Socket, File**, non possono tipicamente essere ricreati, perché contengono riferimenti specifici al particolare ambiente di esecuzione
- le variabili marcate come **transient**
 - ad esempio variabili che non devono essere scritte per questioni di privacy, es. numero carta di credito
 - dichiarare **transient** i campi che non si intende serializzare
- le variabili statiche: **sono associate alla classe e non alla specifica istanza dell'oggetto che si sta serializzando** -> lette dalla classe in fase di deserializzazione
- tutti i componenti di un oggetto devono essere serializzabili: se ne esiste uno non serializzabile si solleva una **notSerializableException**

regola #2: per rendere un oggetto persistente occorre marcare tutti i campi che non sono serializzabili come transient



GRAFI DI OGGETTI: SERIALIZZAZIONE

```
import java.io.Serializable;
import java.util.*;

public class Padre implements Serializable {
    private static final long serialVersionUID = 1L;
    private String nome;
    private String cognome;
    private Collection <Figlio> figli;

    public Padre(String nome,String cognome){
        this.nome=nome;
        this.cognome=cognome;
        figli=new ArrayList<Figlio>();
    }

    public void aggiungiFiglio(Figlio f){
        figli.add(f);
    }
}
```


SERIALIZAZIONE DI GRAFI DI OGGETTI

```
public String toString(){
    StringBuilder temp=new StringBuilder("Padre");
    temp.append("\n");
    temp.append("nome: ");
    temp.append(this.nome);
    temp.append("\n");
    temp.append("cognome: ");
    temp.append(this.cognome);
    emp.append("\n");
    temp.append("Figli: ");
    temp.append(figli.toString());
    return temp.toString();
}
```

SERIALIZAZIONE DI GRAFI DI OGGETTI

```
import java.io.Serializable;

public class Figlio implements Serializable{
    private static final long serialVersionUID = 1L;

    private String nome;
    private String cognome;

    public Figlio(String nome,String cognome){
        this.nome=nome;
        this.cognome=cognome;
    }

    public String toString(){
        return "nome: "+nome+" cognome: "+cognome;
    }
}
```

SERIALIZAZIONE DI GRAFI DI OGGETTI

```
public class SerializzaPadre {  
  
    public static void main(String[] args) {  
        Figlio a=new Figlio("mario","rossi");  
        Figlio b=new Figlio("maria","rossi");  
        Padre p=new Padre("giovanni","rosso");  
        p.aggiungiFiglio(a); p.aggiungiFiglio(b);  
        try(  
            ObjectOutputStream output=new ObjectOutputStream(new  
                FileOutputStream("dati.dat"));) {  
            output.writeObject(p);  
        }  
        catch (IOException e1) {  
            System.out.println("Impossibile serializzare l'oggetto " + p);  
            e1.printStackTrace();  
            System.exit(1);  
        }  
        System.out.println("Serializzazione completata.");  
    }  
}
```

SERIALIZAZIONE DI GRAFI DI OGGETTI

```
import java.io.*;
public class DeSerializzaPadre {

    public static void main(String[] args) {
        try (ObjectInputStream ois = new ObjectInputStream(
            new FileInputStream("dati.dat"));) {
            Padre p = (Padre) ois.readObject();
            System.out.println(p.toString());
        }

        catch (IOException e1) {
            System.out.println("Errore nella creazione dello stream");
            e1.printStackTrace();
            System.exit(1);
        }

        catch (ClassNotFoundException e1) {
            System.out.println("Impossibile trovare la classe");
            e1.printStackTrace();
            System.exit(1);
        }
    }
}
```

SERIALIZZAZIONE DI GRAFI DI OGGETTI

- La sequenza di byte ottenuta contiene informazioni che identificano il tipo di oggetto (la gerarchia di classi) e il valore degli attributi
- Se l'oggetto contiene un riferimento ad un secondo oggetto, anche esso viene serializzato (un albero di oggetti)
 - L'esempio precedente mostra che si possono serializzare/deserializzare oggetti che al loro interno fanno riferimento ad altri oggetti
 - L'implementazione **serializza transitivamente** tutti gli oggetti riferiti
- Se uno degli oggetti riferiti non è serializzabile, viene sollevata una **NotSerializableException**
- E' possibile non includere il valore di alcuni attributi nell'oggetto serializzato, mediante l'utilizzo della parola chiave **transient**

USO CAMPI TRANSIENT

```
import java.io.Serializable;
import java.util.*;

public class Padre implements Serializable {
    private static final long serialVersionUID = 1L;

    private String nome;
    private String cognome;
    transient private Collection<Figlio> figli;

```

...

- l'attributo **transient** inserito nell'esempio precedente indica che non si vuole serializzare l'attributo Figli della classe Padre
- se si esegue la serializzazione e quindi la deserializzazione, il campo figli nell'oggetto deserializzato, risulta uguale a null

Write

- `void writeObject(Object o)`
- Scrive l'informazione sulla classe e i valori degli attributi da serializzare (non *transient* e non statici). Se un oggetto viene scritto due volte, la prima volta è generato un riferimento e la seconda volta scrive solo il riferimento (back-reference).
- **Attenzione:** se l'oggetto viene modificato dopo essere stato scritto la prima volta, la modifica non è salvata la seconda volta.

```
Student s= new Student("Robert","Brown","Dawson",12);  
out.writeObject(s);  
s.setName("Robbie");  
out.writeObject(s);
```

- `void writeUnshared(Object o)`: uguale a `writeObject` però oggetti ripetuti sono scritti come nuovi senza fare riferimento ai precedenti.

Read

- `Object readObject()`
 - Legge l'informazione sulla classe e i valori degli attributi da deserializzare (non *transient* e non statici).
 - Se si incontra un riferimento ad un oggetto già letto, si restituisce quell'oggetto direttamente.
- `Object readUnshared()`
 - Da usare per leggere oggetti scritti con `writeUnshared`

IL CONTROLLO DELLE VERSIONI

- per deserializzare un oggetto occorre conoscere
 - i byte che rappresentano l'oggetto serializzato
 - il codice della classe che descrive la specifica dell'oggetto
- la deserializzazione può avvenire
 - in un ambiente diverso
 - a distanza di tempo rispetto al momento in cui è stata effettuata la serializzazione
- Si può utilizzare una classe diversa per la deserializzazione, rispetto a quella usata per la serializzazione
 - cambiamenti compatibili: si può deserializzare
 - cambiamenti incompatibili: la deserializzazione non è possibile

IL CONTROLLO DELLE VERSIONI

- Una classe serializzabile può essere modificata dopo la serializzazione di un oggetto
- **modifiche compatibili:** rendono comunque possibile la deserializzazione
 - aggiunta di campi
 - aggiungere attributi (vengono inizializzati a valori di default)
 - aggiungere classi interne
 - trasformare attributi transient in non-transient
 - cambiare una classe da non-Serializable a Serializable (equivalente a aggiungere types)
 - e molti altri cambiamenti (vedere documentazione)
- in fase di deserializzazione, il meccanismo di default semplicemente imposta i valori dei campi mancanti con valori di default
- oppure....il programmatore può “fixare” i valori dei campi aggiunti in fase di deserializzazione
- **modifiche incompatibili**
 - rimuovere attributi
 - trasformare attributi non-transient in transient

Documentazione <https://docs.oracle.com/javase/8/docs/platform/serialization/spec/version.html>



serialVersionUID

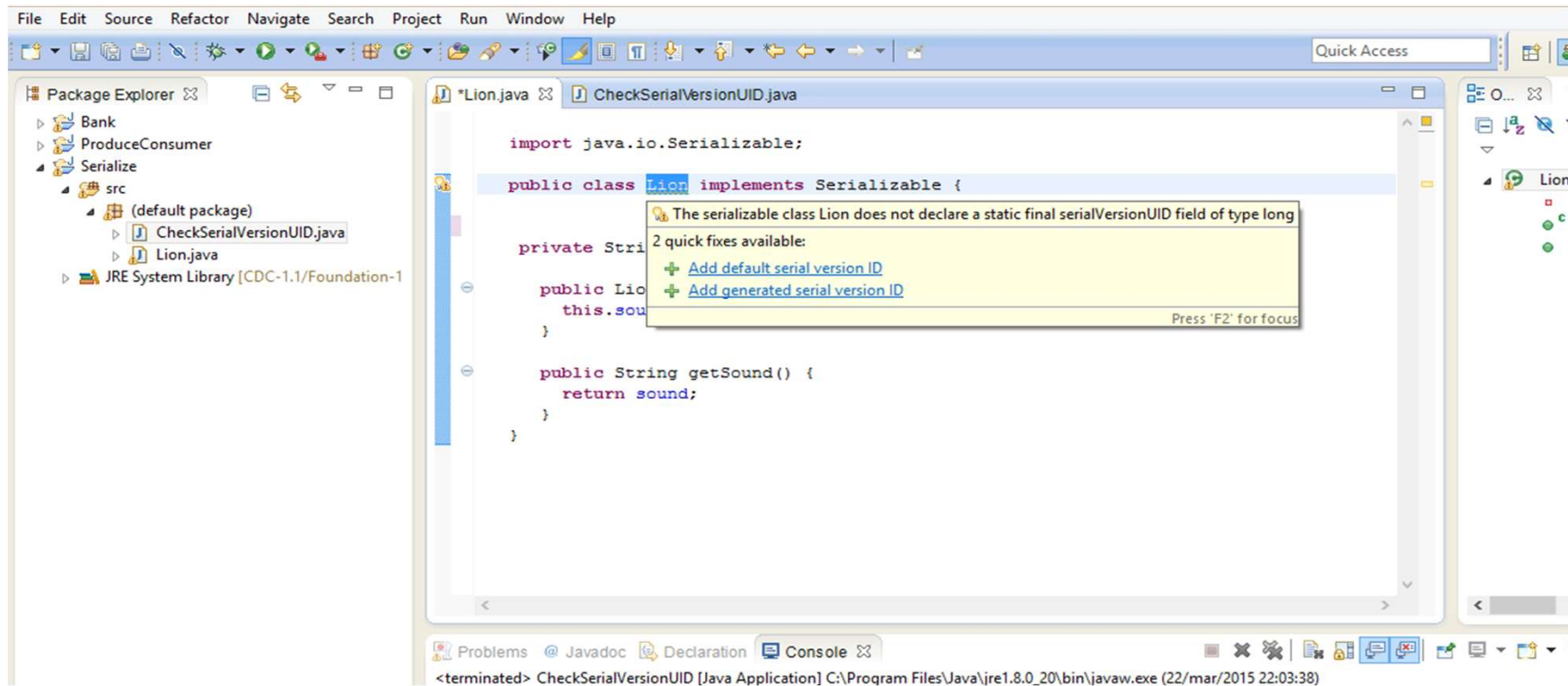
- Per garantire che i processi di serializzazione e deserializzazione usino versioni compatibili di una classe, Java definisce un UID (*unique ID*) per la definizione di una classe
 - classi compatibili: stesso serialVersionUID in entrambe le classi
 - classi incompatibili: diverso serialVersionUID per la classe modificata
- il supporto:
 - controlla se l'utente ha dichiarato esplicitamente il serialVersionUID ed, in questo caso, usa questo valore.
 - altrimenti genera un identificatore implicito
- generazione automatica di identificatori
 - mediante tool automatici di JAVA, data la definizione della classe restituiscono un identificatore unico della classe
- dichiarazione identificatori espliciti

```
private static final long serialVersionUID = 42L;
```

 - se la classe evolve però è sempre backward-compatible, allora si deve usare lo stesso serialVersionUID anche nelle versioni nuove
 - se la nuova versione non è compatibile con quella precedente, il serialVersionUID deve essere cambiato

GENERAZIONE DI SUID

- generazione di serialVersionUID unico mediante l'algoritmo utilizzato da JAVA:
- in Eclipse puntare il mouse sul nome di una classe Serializzabile (addGeneratedSerialVersionUID)
- in altri ambienti: usare tool di generazione specifici (serialver)



GENERAZIONE DI SUID implicita

```
import java.io.Serializable;  
public class Lion implements Serializable {
```



The serializable class Lion does not declare a static final serialVersionUID field of type long

```
    public Lion(String sound) {  
        this.sound = sound;    }  
    public String getSound() {  
        return sound;          }  
}
```

- se il programmatore ignora il warning e non aggiunge esplicitamente un serialVersionUID ?
- JAVA genera automaticamente un serialVersionUID e lo associa alla classe

CONTROLLO DELLE VERSIONI: UN ESEMPIO

```
import java.io.Serializable;

public class Lion implements Serializable {
    private static final long serialVersionUID = 1L;
    private String sound;
    public Lion(String sound) {
        this.sound = sound;
    }
    public String getSound() {
        return sound;
    }
}
```

CONTROLLO DELLE VERSIONI: UN ESEMPIO

```
public class CheckSerialVersionUID {  
    public static void main(String args[]) throws IOException, ClassNotFoundException {  
        Lion leo = new Lion("roar");  
        FileOutputStream fos = new FileOutputStream("serial.out");  
        ObjectOutputStream oos = new ObjectOutputStream(fos);  
        oos.writeObject(leo);  
        System.out.println("Serialization done.");  
        oos.close();  
        FileInputStream fis = new FileInputStream("serial.out");  
        ObjectInputStream ois = new ObjectInputStream(fis);  
        Lion deserializedObj = (Lion) ois.readObject();  
        System.out.println("DeSerialization done. Lion: " +  
                           deserializedObj.getSound());  
        ois.close();  
    }  
}
```

Serialization done. DeSerialization done. Lion: roar

CONTROLLO DELLE VERSIONI: UN ESEMPIO

```
import java.io.Serializable;

public class Lion implements Serializable {
    private static final long serialVersionUID = 2L;
    private String sound;
    public Lion(String sound) {
        this.sound = sound;
    }
    public String getSound() {
        return sound;
    }
}
```

- Ipotezziamo di modificare la classe, ad esempio elimino un campo (modifica non rappresentata), di conseguenza modifico il suo serialVersionUID.
- non serializzo nuovamente l'oggetto, ma utilizzo la versione precedentemente serializzata

CONTROLLO DELLE VERSIONI: UN ESEMPIO

```
public class CheckSerialVersionUID {  
    public static void main(String args[]) throws IOException,  
                                                ClassNotFoundException {  
  
        Lion leo= new Lion("roar");  
        //FileOutputStream fos = new FileOutputStream("serial.out");  
        //ObjectOutputStream oos = new ObjectOutputStream(fos);  
        //oos.writeObject(leo);  
        //System.out.println("Serialization done.");  
        // oos.close();  
        FileInputStream fis = new FileInputStream("serial.out");  
        ObjectInputStream ois = new ObjectInputStream(fis);  
        Lion deserializedObj = (Lion) ois.readObject();  
        System.out.println("DeSerialization done. Lion:" +  
                           deserializedObj.getSound());  
        ois.close();  
    }  
}
```

CONTROLLO DELLE VERSIONI: UN ESEMPIO

Risultato dell'esecuzione:

Exception in thread "main" java.io.InvalidClassException:

Lion; local class incompatible: stream classdesc serialVersionUID = 1, local class serialVersionUID = 2

```
at java.io.ObjectStreamClass.initNonProxy(Unknown Source)
at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
at java.io.ObjectInputStream.readClassDesc(Unknown Source)
at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
at java.io.ObjectInputStream.readObject0(Unknown Source)
at java.io.ObjectInputStream.readObject(Unknown Source)
at CheckSerialVersionUID.main(CheckSerialVersionUID.java:24)
```

CONTROLLO VERSIONI

- infine JAVA suggerisce di indicare esplicitamente un `SerialVersionUID`
- Da JAVAdoc: “the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization”
- spesso si ottiene una eccezione anche se le classe utilizzate in fase di serializzazione e deserializzazione sono in realtà le stesse.
- per questo è consigliato di specificare comunque esplicitamente un `serialVersionUID`

LA SERIALIZZAZIONE: “UNDER THE WOOD”

- Cosa avviene quando un oggetto viene serializzato? Si registrano sullo stream:
 - i “magic data”
 - `STREAM_MAGIC` = “acde” - indica che si tratta di un protocollo di serializzazione
 - `STREAM_VERSION` = versione della serializzazione
 - i metadati che descrivono la classe associata all'istanza dell'oggetto serializzato (la classe Padre nell'esempio precedente)
 - la descrizione include il nome della classe, il `serialVersionUID` della classe, il numero di campi, altri flag.
 - i metadati di eventuali superclassi, fino a raggiungere **`java.lang.Object`**
 - i valori associati all'oggetto istanza della classe, partendo dalla super classe a più alto livello
 - i dati degli oggetti eventualmente riferiti dall'oggetto istanza della classe, iniziando dai metadati e poi registrando i valori. (Le istanze della classe Figlio, nell'esempio precedente).
 - non si registrano i metodi della classe

LA SERIALIZZAZIONE: “UNDER THE WOOD”

```
Length: 220
Magic: ACED
Version: 5
OBJECT
  CLASSDESC
    Class Name: "SimpleClass"
    Class UID: -D56EDC726B866EBL
    Class Desc Flags: SERIALIZABLE;
    Field Count: 4
    Field type: object
    Field name: "firstName"
    Class name: "Ljava/lang/String;"
    Field type: object
    Field name: "lastName"
    Class name: "Ljava/lang/String;"
    Field type: float
    Field name: "weight"
    Field type: object
    Field name: "location"
    Class name: "Ljava/awt/Point;"
    Annotation: ENDBLOCKDATA
    Superclass description: NULL
  STRING: "Brad"
  STRING: "Pitt"
  float: 180.5
  OBJECT
    CLASSDESC
      Class Name: "java.awt.Point"
      Class UID: -654B758DCB8137DAL
      Class Desc Flags: SERIALIZABLE;
      Field Count: 2
      Field type: integer
      Field name: "x"
      Field type: integer
      Field name: "y"
      Annotation: ENDBLOCKDATA
      Superclass description: NULL
    integer: 49.345
    integer: 67.567
```

- Consideriamo la classe **SimpleClass** con campi
 - firstName,
 - lastName,
 - weight
 - Location
- L'oggetto istanza della classe contiene i campi
{"Brad", "Pitt", 180.5, {49.345, 67.567}}
- A fianco: risultato della serializzazione
 - Notare: per la memorizzazione di un oggetto di 20 bytes utilizzati circa 220 bytes
 - Questo può costare molto, ad esempio in termini di banda consumata, se si deve spedire l'oggetto