

Laboratorio di Reti – A

(matricole pari)

**Autunno 2021,
instructor: Laura Ricci**

laura.ricci@unipi.it

Lezione 5

Concurrent Collections, Stream IO, Serializzazione

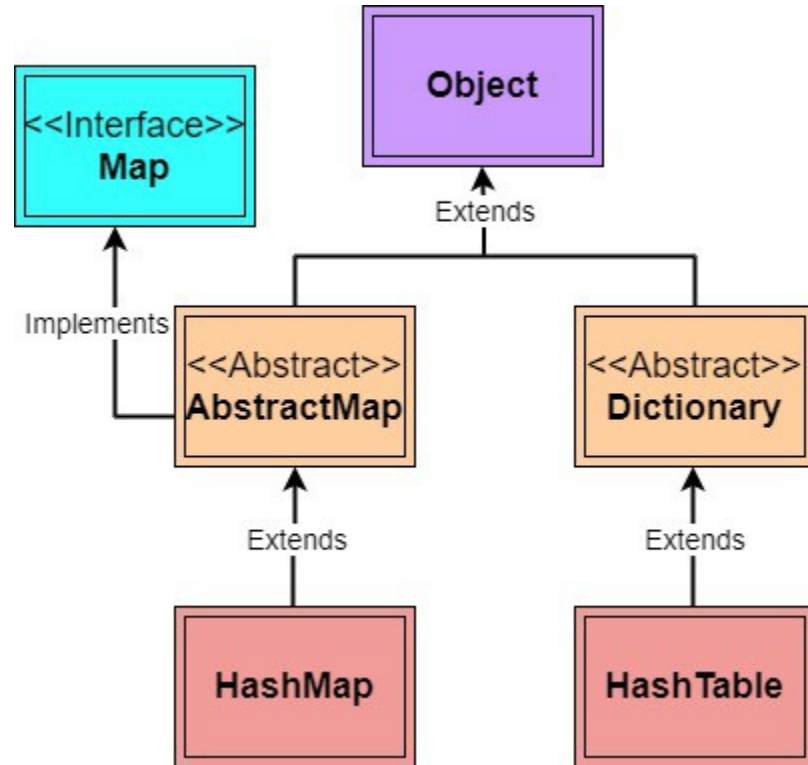
12/10/2021

CONCURRENT COLLECTIONS

- evoluzione delle precedenti librerie basata sulla esperienza nel loro utilizzo
- fine-grain locking, non bloccano l'intera collezione
 - concurrent reads, writes parzialmente concorrenti
- iteratori **fail safe/weakly consistent**
 - restituiscono tutti gli elementi che erano presenti nella collezione quando l'iteratore è stato creato
 - possono restituire o meno elementi aggiunti in concorrenza
- forniscono alcune utili operazioni atomiche composte da più operazioni elementari
- Blocking Queue è una concurrent collections
- la più utilizzata: ConcurrentHashMap $\langle K, V \rangle$, sincronizzazione ottimizzata, diverse operazioni atomiche
 - put-if-absent, remove-if-equal, replace-if-equal

HASH MAP E HASH TABLE

- ConcurrentHashMap
 - introdotta in JAVA 5, nella libreria `java.util.concurrent`
 - thread safe, è una evoluzione di `HashTable` ed `HashMap`
- collezioni che memorizzano coppie chiave/valore
 - usando la tecnica dell'hashing
- differenza principale
 - `HashTable` (da JAVA 1.0) è *threadsafe*
 - `HashMap` (da JAVA 1.2) non è *threadsafe*
 - perchè un'altra collezione thread safe?



CONCURRENT HASH MAP

```
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class TestCollections {
    public static void main(String[] args) throws Exception {
        evaluatingThePerformance(new Hashtable<String,Integer>(),5);
        evaluatingThePerformance(Collections.synchronizedMap
                                   (new HashMap<String,Integer>()),5);
        evaluatingThePerformanceWithSynchronizedBlock
            (Collections.synchronizedMap(new HashMap<String,Integer>()),5);
        evaluatingThePerformance(new ConcurrentHashMap<String,Integer>(),5);
    }
}
```

CONCURRENT HASH MAP

```
public static void performTest(Map<String,Integer> maptoEvaluateThePerformance)
{
    for(int i=0; i<500000; i++) {
        Integer randomNumber = (int) Math.ceil(Math.random() * 550000);
        Integer value =
            maptoEvaluateThePerformance.get(String.valueOf(randomNumber));
        // Put value
        maptoEvaluateThePerformance.put(String.valueOf(randomNumber),randomNumber);
    }
}
```

- test: 500000 put e get sulla tabella passata come parametre
- attiviamo un threadpool con k threads che eseguono in parallelo il codice precedente e prendiamo li tempi di esecuzione
- ripetiamo l'esperimento un certo numero di volte e calcoliamo la media dei tempi impiegati

CONCURRENT HASH MAP

```
public static void performSynchronizedTest(Map<String,Integer>
                                           mapToEvaluateThePerformance) {

    for(int i=0; i<500000; i++) {
        Integer randomNumber = (int) Math.ceil(Math.random() * 550000);

        synchronized (mapToEvaluateThePerformance) {
            Integer Value =
                mapToEvaluateThePerformance.get(String.valueOf(randomNumber));
        }

        synchronized (mapToEvaluateThePerformance) {
            mapToEvaluateThePerformance.put
                (String.valueOf(randomNumber), randomNumber);
        }
    }
}
```

CONCURRENT HASH MAP

```
public static void evaluatingThePerformance
    (Map<String,Integer> maptoEvalPerf, int size) throws Exception {
    long averageTime = 0;
    for(int j=0; j<size;j++) {
        ExecutorService executorService = Executors.newFixedThreadPool(size);
        long startTime = System.nanoTime();
        for(int i=0; i<size;i++) {
            executorService.execute(new Runnable () {public void run()
                {performTest(maptoEvalPerf);}}});
        executorService.shutdown();
        executorService.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);
        long endTime = System.nanoTime();
        long totalTime = (endTime - startTime) / 1000000L;
        averageTime += totalTime;
        System.out.println("500K entried added/retrieved by each thread in "+ totalTime+ "
                                ms");
    }
    System.out.println("For " + maptoEvalPerf.getClass() + " the average time is " +
        averageTime / 5 + "ms\n");} }
```

CONCURRENT HASH MAP

- il metodo *evaluatingThePerformanceWithSynchronizedBlock* invoca *performynsynchronizedTest*, invece di *performTest* (nel run della Runnable)
- per il resto è uguale al metodo della slide precedente

CONCURRENT HASH MAP: RISULTATI

500K entried added/retrieved by each thread in 1274 ms

500K entried added/retrieved by each thread in 1314 ms

500K entried added/retrieved by each thread in 1279 ms

.....

For class `java.util.Hashtable` the average time is 1283 ms

500K entried added/retrieved by each thread in 1288 ms

500K entried added/retrieved by each thread in 1157 ms

500K entried added/retrieved by each thread in 1249 ms

.....

For class `java.util.Collections$SynchronizedMap` the average time is 1237 ms

500K entried added/retrieved in 1238 ms

500K entried added/retrieved in 1392 ms

500K entried added/retrieved in 1375 ms

.....

For class `java.util.HashMapWith SYNCHRONIZED BLOCKS` the average time is 1423 ms

500K entried added/retrieved by each thread in 550 ms

500K entried added/retrieved by each thread in 440 ms

500K entried added/retrieved by each thread in 458 ms

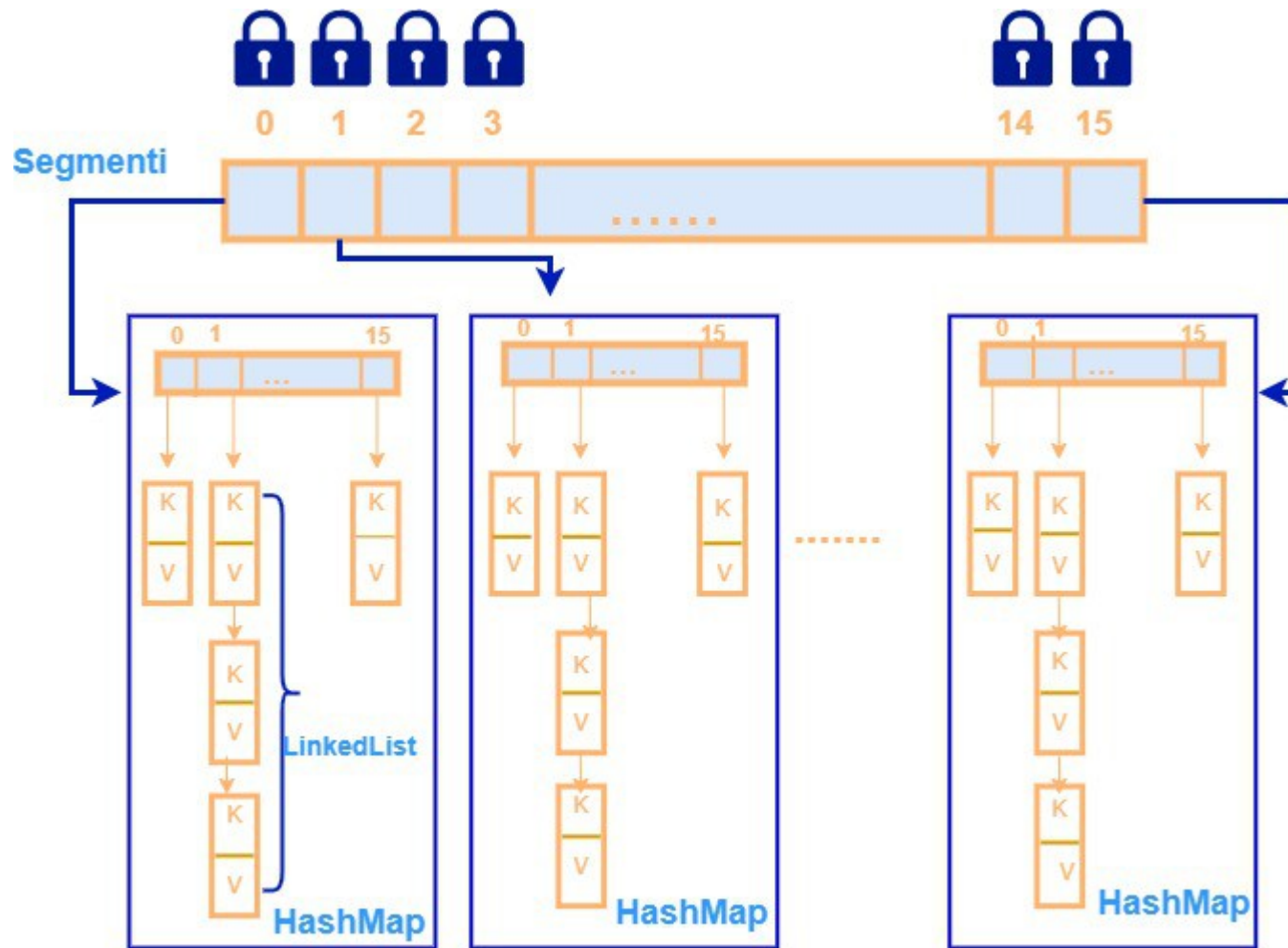
.....

For class `java.util.concurrent.ConcurrentHashMap` the average time is 524 ms

CONCURRENT HASH MAP

- le slide precedenti mostrano che la ConcurrentHashMap è molto più efficiente delle precedenti versioni di tabelle chiave-valore
- idea fondamentale: usare una diversa strategia di locking che offra migliore concorrenza e scalabilità
 - introduce un array di segmenti: ogni segmento punta ad una HashMap
 - **fine grained locking**
 - una lock per ogni segmento, **lock striping**
 - numero di segmenti determina il livello di concorrenza
 - modifiche simultanee possibili, se modificano segmenti diversi
 - 16 o più threads possono operare in parallelo su segmenti diversi
 - lettori possono accedere in parallelo a modifiche

STRUTTURA DELLA CONCURRENTHASHMAP



Struttura interna della `ConcurrentHashMap`

OPERAZIONI COMPOSTE ATOMICHE

le concurrent collections offrono un insieme di operazioni composte atomiche

- sequenze di operazioni di uso comune
- definita come una operazione unica
- la JVM traduce la singola operazione “ad alto livello” in una sequenza di operazioni a più basso livello
- garantisce inoltre la corretta sincronizzazione su tale operazione
 - ...secondo la filosofia “do not re-invent the wheel...”

OPERAZIONI COMPOSITE: ATOMICITA'

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

public class CHashMap {
    private Map<String, Object> theMap =
        new ConcurrentHashMap<>();

    public Object getOrCreate(String key) {
        Object value = theMap.get(key);
        try { Thread.sleep(5000);
            } catch (Exception e) {};
        if (value == null) {
            value = new Object();
            theMap.put(key, value); }
        return value.hashCode();}}}
```

```
public class Main {
    public static void main(String [] args)
    { CHashMap ex= new
        CHashMap();

        Thread t1 = new Thread (new Runnable()
            {public void run()
                {System.out.println
                    (ex.getOrCreate("5"))};});
        t1.start();

        Thread t2 = new Thread (new Runnable()
            {public void run()
                {System.out.println
                    (ex.getOrCreate("5"))};});
        t2.start();
    }}
```

- GetOrCreate **non è atomica**
- t1 e t2 stampano due valori diversi, entrambi associati alla stessa chiave, 5

OPERAZIONI COMPOSTE ATOMICHE

- soluzione: utilizzare istruzioni atomiche composte
- funzioni del tipo “query-then-update”, “test-and-set”
- nel nostro caso è utile la `putIfAbsent`

```
public interface ConcurrentMap<K,V> extends Map<K,V> {  
    V putIfAbsent(K key, V value);  
        // Insert into map only if no value is mapped from K  
        // returns the previous value associated to the key  
        // or null if there is no mapping for that key  
    boolean remove(K key, V value);  
        // Remove only if K is mapped to V  
    boolean replace(K key, V oldValue, V newValue);  
        // Replace value only if K is mapped to oldValue  
    V replace(K key, V newValue);  
        // Replace value only if K is mapped to some value    }
```

OPERAZIONI COMPOSITE: PUTIFABSENT

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

public class Main1 {
    static Map<String, Object> theMap = new ConcurrentHashMap<>();

    public static void main(String [] args)
    { Thread t1 = new Thread (
        new Runnable() {public void run()
            {Object obj1 = new Object();
            System.out.println(theMap.putIfAbsent("5",obj1));}});

        t1.start();

        Thread t2 = new Thread (new Runnable() {public void run()
            {Object obj2 = new Object();
            System.out.println(theMap.putIfAbsent("5",obj2));}});

        t2.start();}}
```

- le Collection JAVA supportano diversi tipi di iteratori
 - si distinguono riguardo al comportamento di una collezione in presenza di “concurrent modification”
 - cosa accade quando la collezione viene modificata, mentre un iteratore la sta scorrendo, e questa modifica arriva “dall'esterno” dell'iteratore?
- fail-fast
 - se c'è una modifica strutturale (inserzione, rimozione, aggiornamento), dopo che l'iteratore è stato creato, l'iteratore la rileva e solleva una ConcurrentModificationException
 - fallimento immediato dell'operatore, per evitare comportamenti non deterministici
 - la maggior parte delle collezioni “non-concurrenti” sono fail-fast
Vector, ArrayList, HashMap, ed altre....

- **fail-safe** (“snapshot”) introdotti in JAVA 1.5 con le concurrent collections
 - creano una copia della collezione, al momento della creazione dell'iteratore e lavorano su questa copia
 - non sollevano `ConcurrentModificationException`
 - l'iteratore accede ad una versione non aggiornata della collezione
 - `CopyOnWriteArrayList`, ...
- **weakly consistent** introdotti in JAVA 1.5 con le concurrent collections
 - l'iteratore e modifiche operano sulla stessa copia
 - no `ConcurrentModificationException`, comportamento fail-safe
 - l'iteratore considera gli elementi che esistevano al momento della costruzione dell'iteratore e può riflettere le modifiche che sono avvenute dopo la costruzione dell'iteratore, anche se non è garantito
 - `ConcurrentHashMap`, ...

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 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)

FAIL SAFE: COPYONWRITEARRAYLIST

- come risulta evidente dal nome, effettua una copia dell'array tutte le volte che viene effettuata una operazione di modifica (add, set, etc..)
- “**snapshot style iterator**”: usa un riferimento ad una copia dello stato dell'array nel momento in cui l'iteratore è creato
 - l'array riferito non viene mai modificato durante la vita dell'iteratore: l'iteratore non cattura le modifiche effettuate dopo la sua creazione
 - thread-safe: ogni thread lavora su una propria copia
 - fail safe: non solleva `ConcurrentModificationException`
- operazione di copia molto costosa
 - è adatto quando ci sono più accessi in lettura che modifiche

FAIL SAFE: COPYONWRITEARRAYLIST

```
import java.util.*;

import java.util.concurrent.CopyOnWriteArrayList;

public class FailSafeDemo {

    public static void main(String[] args) {

        List<String> numList = new
            CopyOnWriteArrayList<String>();

        numList.add("1"); numList.add("2");
        numList.add("3"); numList.add("4");
        //This thread will iterate the list

        Thread thread1 = new Thread(){

            public void run(){

                try{

                    Iterator<String> i = numList.iterator();

                    while (i.hasNext())

                        { System.out.println(i.next());

                          Thread.sleep(1000); }

                    catch(Exception e)

                        {e.printStackTrace();}} }

            thread1.start();

            // This thread will try to add to the
            collection,

            // while the collection is iterated by another
            thread.

            Thread thread2 = new Thread(){

                public void run(){

                    try{

                        Thread.sleep(2000);

                        // adding new value to the shared
                        list

                        numList.add("5");

                        System.out.println("new value added
                            to the list");

                        System.out.println("List " +
                            numList);

                    }catch(Exception e){

                        e.printStackTrace();}

                    } };

                thread2.start(); }}
```

FAIL SAFE: COPYONWRITEARRAYLIST

1

2

3

new value added to the list

List [1, 2, 3, 4, 5]

4

- thread 1 stampa i valori 1,2,3 e 4, l'iteratore non intercetta il valore 5 aggiunto dal Thread2
- thread 2 ha una copia della `CopyOnWriteArrayList` e il valore 5 viene aggiunto su questa copia

WEAKLY CONSISTENCY: CONCURRENT HASH MAP

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.”

- l'iteratore
 - non clona la struttura al momento della creazione
 - la collezione può catturare le modifiche effettuate sulla collezione dopo la sua creazione
 - non solleva ConcurrentModificationException
- alcuni metodi, `size()` e `isEmpty()`
 - possono restituire un valore “approssimato”
 - “weakly consistent behaviour”

UN ITERATORE 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 : 7
```

INPUT/OUTPUT IN JAVA

- I/O: reperire informazioni da una sorgente esterna o inviare ad una sorgente esterna
 - *file system*: files e directories
 - *connessioni di rete*
 - *keyboard*: `System.in`, `System.out`, `System.error`
 - *in-memory buffers* (array)
 - “vista” di un buffer di memoria come una sorgente o destinazione esterna.
 - un programma legge da un file csv.
 - per ottimizzare l’accesso ai dati si legge tutto il file in un buffer, in memoria centrale.
 - l’interfaccia verso il modulo che gestisce i dati deve rimanere la solita
- di particolare interesse per il corso:
 - connessioni di rete modellate come streams
 - in-memory buffers per la generazione di pacchetti UDP.

JAVA STREAM E CHANNEL

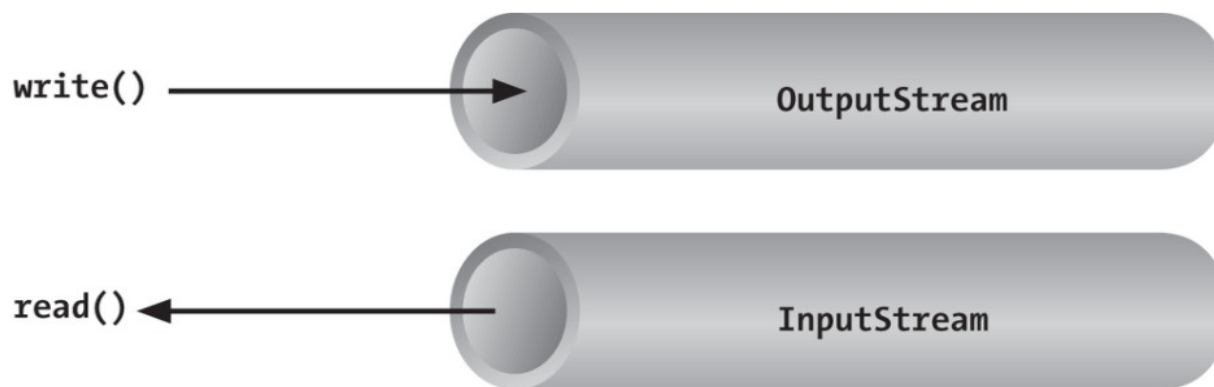
- definizione di un insieme di **astrazioni per la gestione dell'I/O**: una delle parti più complesse di un linguaggio
- diversi tipi di device di input/output: se il linguaggio dovesse gestire ogni tipo di device come caso speciale, la complessità sarebbe enorme
 - necessità di **astrazioni opportune** per rappresentare una device di I/O
- in JAVA, la prima astrazione definita è basata sul concetto di **stream (o flusso)**
- altre astrazioni per l'I/O
 - File: per manipolare descrittori di files
 - Channels (NIO)
 - ...

JAVA PACKAGES PER I/O

- programmare 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 da JDK 1.0
 - basato su stream
 - lavora su bytes e caratteri
 - **JAVA NIO API: (New IO)** funzionalità simili a `java.io`, ma con basato su canali e con comportamento non bloccante: migliori performance in alcuni scenari. Introdotta in JDK 1.4
 - **JAVA NIO.2**: alcuni ulteriori miglioramenti rispetto a JAVA NIO

L'ASTRAZIONE DEGLI STREAM

- uno stream rappresenta una connessione tra un programma JAVA ed un dispositivo esterno (file, buffer di memoria, connessione di rete,...)
- un flusso di informazione di lunghezza illimitata



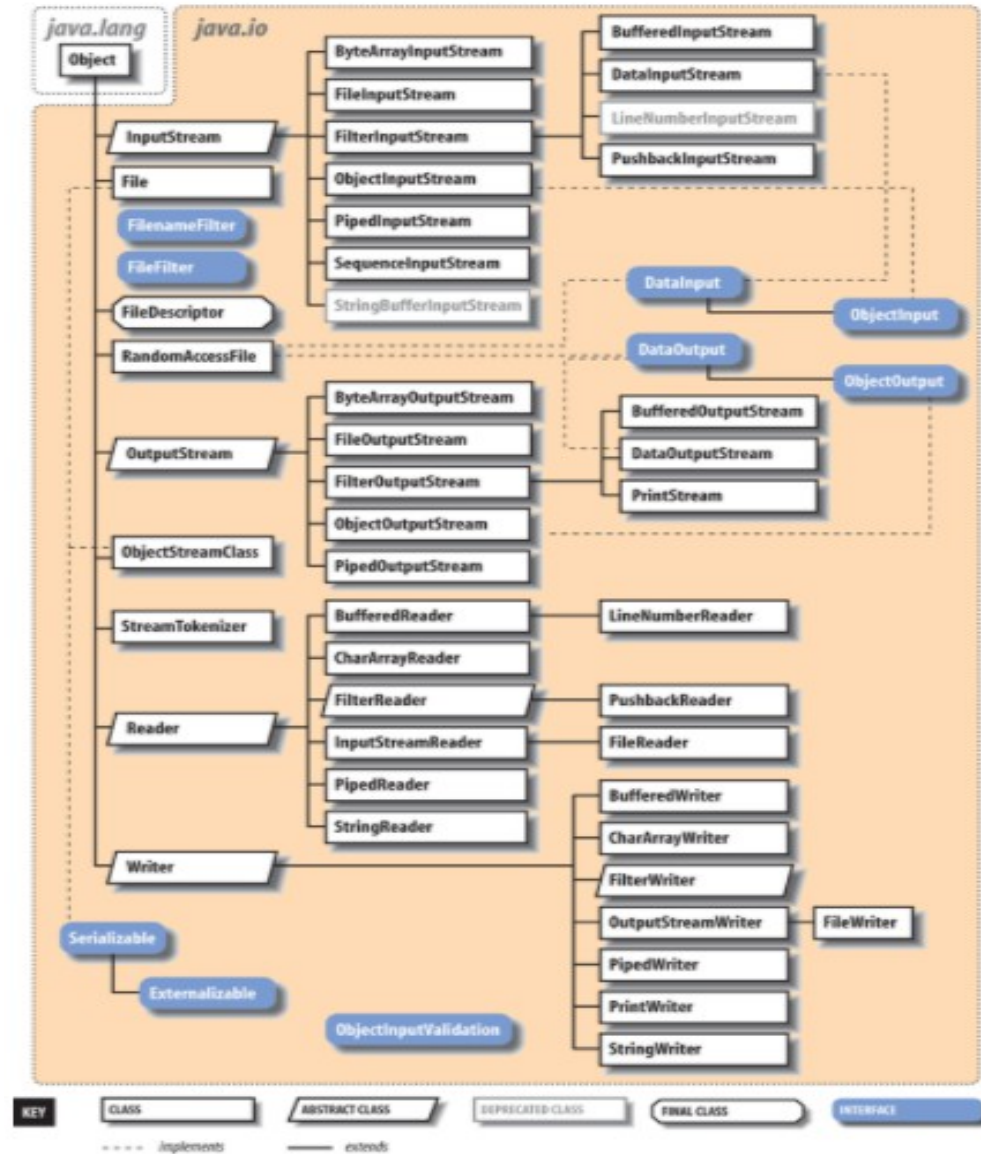
- un “tubo” tra una sorgente ed una destinazione
 - dal programma ad un dispositivo e viceversa
- l'applicazione inserisce dati o li legge ad/da un capo dello stream
- i dati fluiscono da/verso la destinazione

JAVA STREAMS: CARATTERISTICHE GENERALI

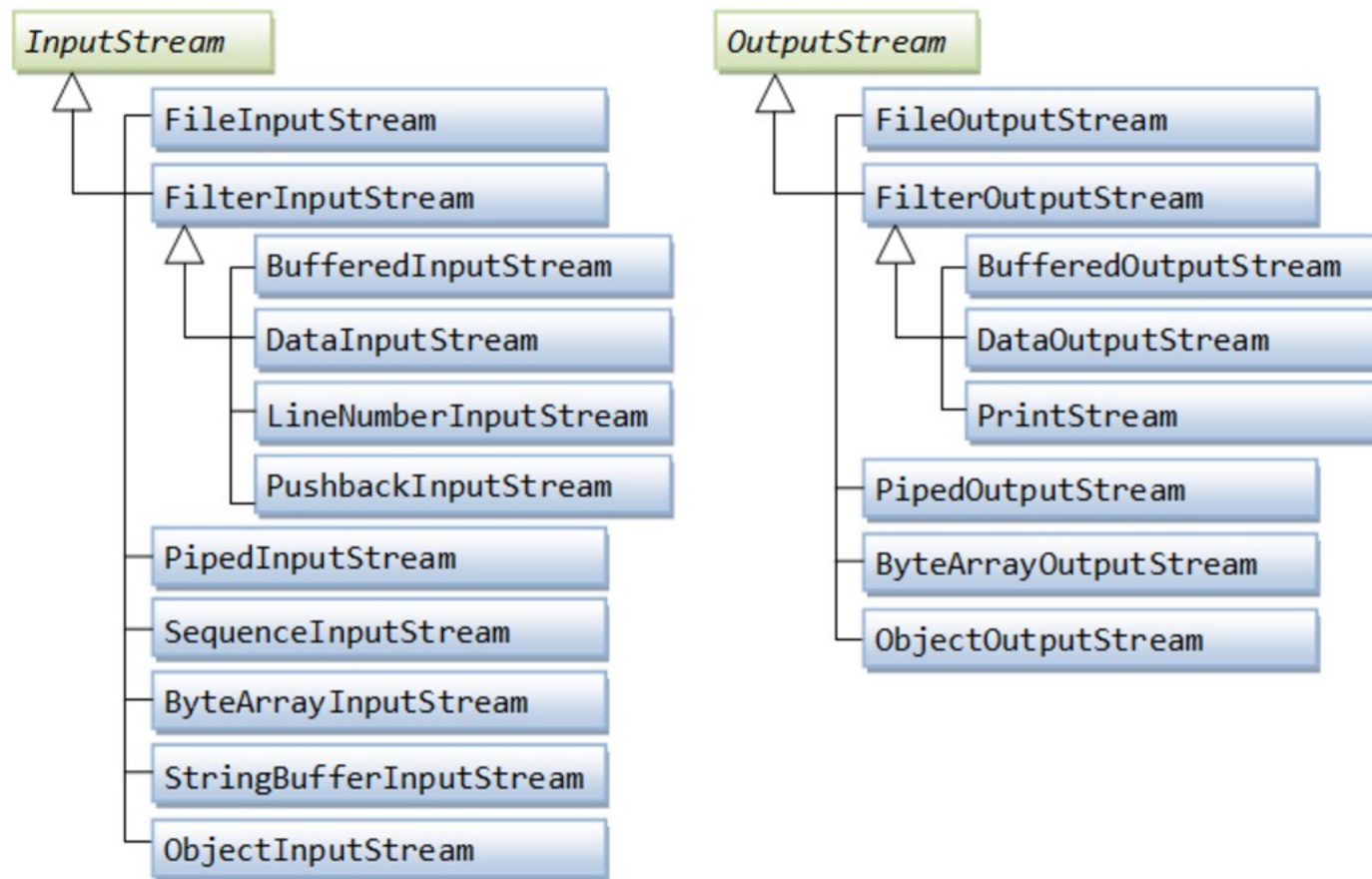
- accesso **sequenziale**
- mantengono l'**ordinamento FIFO**
- **one way**: read only oppure write only (a parte i file ad accesso random)
 - se un programma ha bisogno di dati in input ed output, è necessario aprire due stream, in input ed in output
- **bloccanti**: quando un'applicazione legge un dato dallo stream (o lo scrive) si blocca **finchè l'operazione non è completata**
- non è richiesta una corrispondenza stretta tra letture/scritture
 - una unica scrittura inietta 100 bytes sullo stream
 - i byte vengono letti con due write successive 'all'altro capo dello stream', la prima legge 20 bytes, la seconda 80 bytes)

LA GIUNGLA DELLE CLASSE IN JAVA IO

- 4 classi astratte fondamentali:
 - `InputStream`, `OutputStream`,
`Reader`, `Writer`
 - `Input/OutputStream`:
 - leggono e scrivono bytes
 - dati copiati byte a byte
 - `ObjectStreamClass`
 - non strutturati
 - `StreamTokenizer`
 - senza alcuna traduzione
 - ideale per leggere/scrivere raw data in binario
 - un'immagine
 - la codifica di un video.
- Readers/Writers per caratteri



BYTE STREAM: GERARCHIA DI CLASSI



Input/OutputStream: sono classi astratte, diverse implementazioni

STREAM: CARATTERISTICHE GENERALI

- `InputStream`/`OutputStream`:
 - forniscono operazioni base
 - possono essere “associati” ad ogni tipo di device di input/output.
- implementazioni diverse per tipi diversi di I/O:
 - console: `System.out`, `System.in`
 - files: `FileInputStream`, `FileOutputStream` per leggere/scrivere dati da un file
 - in-memory buffers: per trasferimento di dati da una parte all'altra di un programma JAVA: `ByteArrayOutputStream`, `ByteArrayInputStream`
 - utili per generare uno stream di byte, per poi trasferirlo in un pacchetto UDP.
 - connessioni TCP
 - `PipeInputStream`: pipes

INPUTSTREAM

- **int** read()
 - legge un byte dallo stream come un intero unsigned tra 0 e 255
 - restituisce -1 se viene individuata la “fine dello stream” (derivato dal C)
 - solleva una `IOException` se c'è un errore di I/O
 - bloccante
 - **public int** read(**byte**[] bytes)
 - riempie il vettore passato in input con tutti i dati disponibili sullo stream fino alla capacità massima del vettore e restituisce il numero di byte letti
 - skip(**long** n), available(), close() e molti altri metodi
- ```
int waiting = System.in.available();
if (waiting > 0) {
 byte [] data = new byte [waiting];
 System.in.read(data);
 ...}
```
- diverse classi concrete implementano `InputStream`, `FileInputStream` legge un byte da un file, `System.in` legge byte a byte dalla keyboard



# BYTE STREAM: COPIARE UN'IMMAGINE

```
import java.io.*;

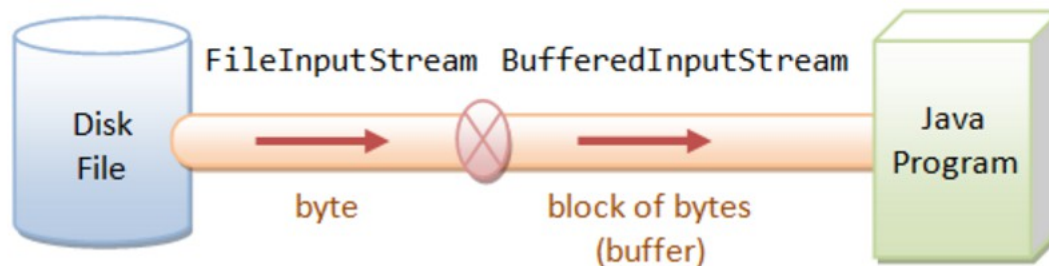
public class CopyImage {
 public static void main(String[] args){
 {InputStream is = null; OutputStream os = null;
 try { is = new FileInputStream(new File("immagine.jpg"));
 os = new FileOutputStream(new File("immaginenew.jpg"));
 byte[] buffer = new byte[8192]; int length;
 while ((length = is.read(buffer)) > 0) {
 os.write(buffer, 0, length); } }
 catch(Exception e){e.printStackTrace();}
 finally {
 if (is!=null)
 try {is.close();} catch(Exception e){e.printStackTrace();}
 if (os!=null)
 try {os.close();} catch(Exception e){e.printStackTrace();}
 }
 }
 }
}
```

# JAVA: FILTER STREAMS

- InputStream and OutputStream operano su “row bytes”
- classi filtro compiono trasformazioni sui dati a basso livello. Tipi di filtri:
- **Filter Stream**: trasformazioni effettuate
  - crittografia
  - compressione
  - buffering
  - traduzione dei dati in un formato a più alto livello
- **Readers Writes**
  - orientati al testo e permettono di decodificare bytes in caratteri
- I filtri possono essere **organizzati in catena**. Ogni elemento della catena
  - riceve dati dallo stream o dal filtro precedente
  - passa i dati al programma o al filtro successivo

# BUFFERED INPUT ED OUTPUT STREAM

- implementano una bufferizzazione per stream di input e di output,
- i dati vengono scritti e letti in blocchi di bytes, invece che un solo byte per volta
- miglioramento significativo della performance



```
FileOutputStream outputFile = new FileOutputStream("primitives.data");
```

```
BufferedOutputStream bufferedOutput = new BufferedOutputStream(outputFile);
```

# VALUTARE EFFETTI BUFFERIZZAZIONE

```
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 bytesRead;
```

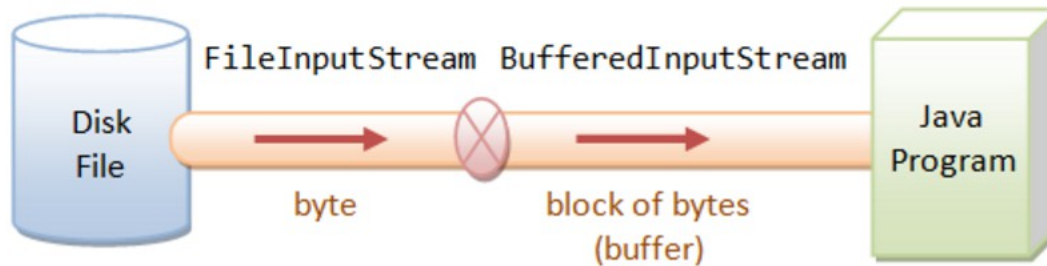
# VALUTARE EFFETTI BUFFERIZZAZIONE

```
// Read a raw byte, returns an int of 0 to 255.
while ((byteRead = in.read()) != -1)
{
 // Write the least-significant byte of int, drop the upper 3 bytes
 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



modificando il programma precedente come segue:

**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**

# SCRIVERE/LEGGERE OGGETTI DA STREAM

- gli oggetti esistono in memoria fino a che la JVM è in esecuzione:
  - per la loro persistenza al di fuori della JVM, occorre
    - creare una rappresentazione dell'oggetto indipendente dalla JVM
    - meccanismi di **serializzazione**
- ogni oggetto è caratterizzato da uno stato e da un comportamento
  - comportamento: specificato dai metodi della classe
  - stato: “vive” con l’istanza dell’oggetto
  - la serializzazione effettua il **flattening** dello **stato dell'oggetto**
  - la deserializzazione ricostruisce lo stato dell'oggetto

1 Object on the heap

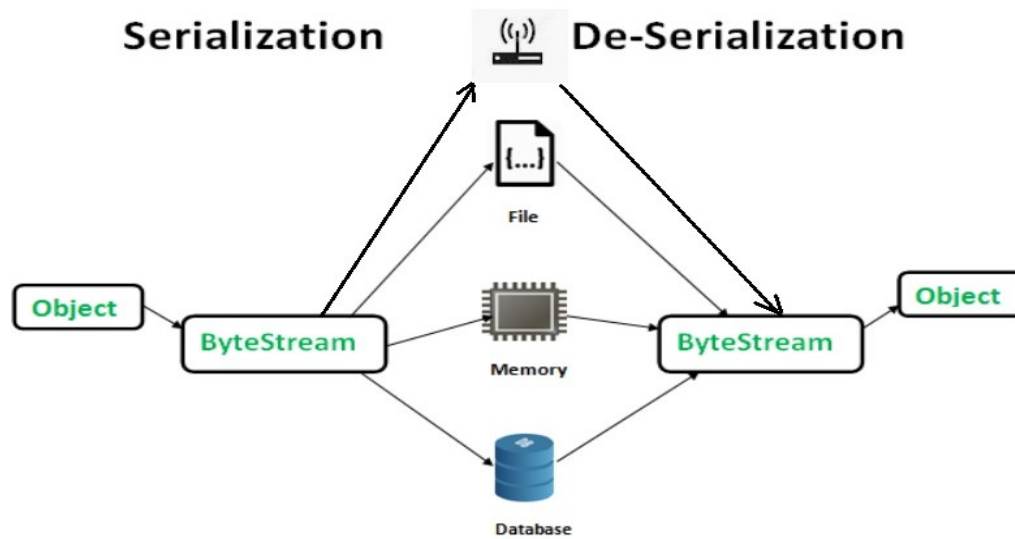


2 Object serialized



# PERSISTENZA ED INVIO DI OGGETTI

- 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



# SERIALIZZAZIONE JAVA: HOW TO DO

- **Serializable Interface**
  - per rendere un oggetto “persistente”, l'oggetto deve implementare l'interfaccia **Serializable**
  - marker interface: nessun metodo, solo informazione su un oggetto per il compilatore e la JVM
  - controllo limitato sul meccanismo di linearizzazione dei dati
  - tutti i tipi di dato primitivi sono serializzabili
  - gli oggetti, se implementano **Serializable**, sono serializzabili
    - a parte alcuni oggetti....(vedi slide successive)
- **Externizable Interface**
  - estende **Serializable**
  - consente creare un proprio protocollo di serializzazione
    - ottimizzare la rappresentazione serializzata dell'oggetto
    - implementazione metodi **readExternal** e **writeExternal**

# SERIALIZZAZIONE JAVA: HOW TO DO

```
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; } }
```

in rosso le parti relative alla serializzazione

**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

# SERIALIZZAZIONE JAVA: HOW TO DO

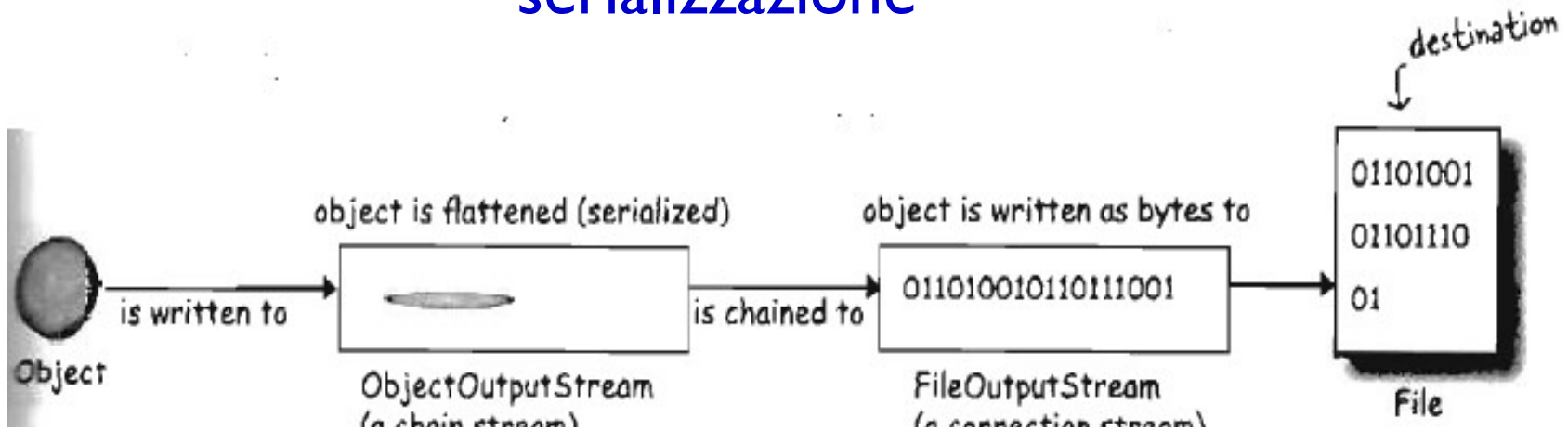
```
import java.io.*;

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();
 }}}}
```

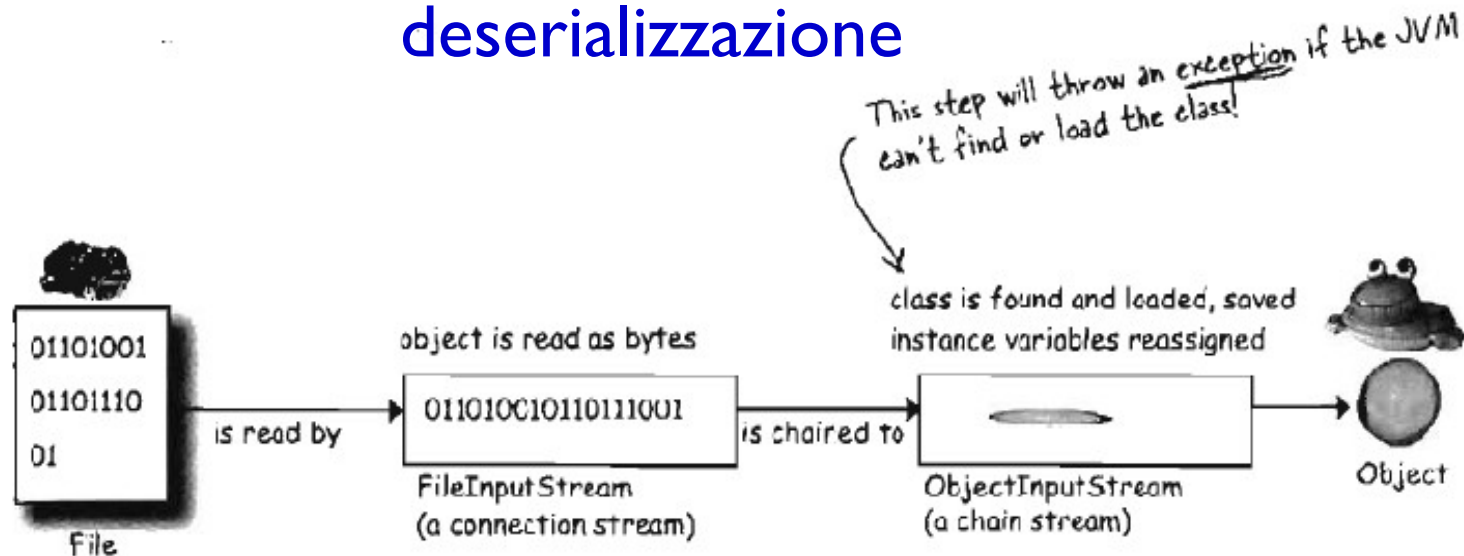
- la serializzazione vera e propria è gestita dalla classe `ObjectOutputStream`
- tale stream deve essere concatenato con uno stream di bytes, che può essere un `FileOutputStream`, uno stream di bytes associato ad un socket, uno stream di byte generato in memoria,...

# SERIALIZAZIONE E DESERIALIZAZIONE

## serializzazione



## deserializzazione



# SERIALIZZAZIONE E GERARCHIA DELLE CLASSI

- La serializzazione è un processo ricorsivo, l'oggetto serializzato può contenere altri oggetti
- quando un oggetto viene serializzato, si percorre la gerarchia delle superclassi e si salva lo stato di ogni classe, fino a che non si trova la prima classe non serializzabile

# DESERIALIZAZIONE

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

in rosso le parti relative alla **deserializzazione**

# DESERIALIZAZIONE

```
// 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

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

# DESERIALIZAZIONE

- 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 o una classe compatibile
- se non la trova viene sollevata una `ClassNotFoundException` ed il processo di deserializzazione viene abortito
- altrimenti, viene creato un nuovo oggetto sullo heap
  - lo stato di tutti gli oggetti serializzati viene ricostruito cercando i valori nello stream, senza invocare il costruttore (uso di Reflection)
  - si percorre l'albero delle superclassi fino alla prima superclasse non-serializzabile. Per quella classe viene invocato il costruttore



# COSA NON E' SERIALIZZABILE?

- oggetti contenenti riferimenti specifici alla JVM o al SO (JAVA native class)
  - `Thread`, `OutputStream`, `Socket`, `File`, non possono 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
- 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 e non `transient` si solleva una `notSerializableException`
  - **regola #2:** per rendere un oggetto persistente occorre marcare tutti i campi che non sono serializzabili come `transient`

# LA SERIALIZZAZIONE: “UNDER THE HOOD”

- la serializzazione è un meccanismo molto complesso e computazionalmente pesante
- La serializzazione standards di JAVA utilizza diversi meccanismi, che è interessante conoscere perchè utili anche nel caso di altri meccanismi di serializzazione
  - caching
  - controllo delle versioni
  - performance

# 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, ad esempio
  - mediante l'utilizzo di un compilatore diverso
  - a distanza di tempo rispetto al momento in cui è stata effettuata la serializzazione
  - su un computer diverso, a cui è stato mandato l'oggetto serializzato tramite una connessione di rete
- cosa succede se la classe utilizzata per la serializzazione cambia quando l'oggetto viene deserializzato?

# IL CONTROLLO DELLE VERSIONI

```
public class Employee {
 private String id;
 private String name;
 private int age;}
```

- supponiamo di serializzare i dati di un insieme di impiegati utilizzando la classe precedente
- successivamente la classe viene modificata come segue

```
public class Employee {
 private String id;
 private String name;
 private Date dateOfBirth; }
```


- si può utilizzare la classe modificata per deserializzare un oggetto che è stato serializzato con la classe prima della modifica?

## serialVersionUID (SUID)

- identificatore unico che identifica una classe
- utilizzato per verificare che la compatibilità tra le classi usate per la serializzazione e la deserializzazione, in fase di deserializzazione
- può essere gestito in due modi diversi
- **identificatore implicito**: generato dal compilatore, non specificato dall'utente
  - 64-bit hash (SHA) generato a partire dalla struttura della classe, durante la serializzazione (nome della classe, nomi delle interfacce, metodi e campi)
  - in alcuni casi, compilatori diversi possono generare identificatori diversi per la stessa classe
- **identificatore esplicito**: il programmatore gestisce esplicitamente la compatibilità delle classi, gestendo i loro identificatori. Ma come si generano gli identificatori?
  - dichiarato a scelta dal programmatore
  - usando l'IDE Eclipse
  - con un generatore a linea di comando

# serialVersionUID (SUID) IMPLICITO

- You write a Dog class

  
Dog.class


class version ID  
#343

- You serialize a Dog object using that class

  
Dog object

Object is stamped with version #343


- You change the Dog class

  
Dog.class

class version ID  
#728

- You deserialize a Dog object using the changed class

Object is stamped with version #343

  
Dog.class

class version is #728

- Serailization fails!!

The JVM says, "you can't teach an old Dog new code".

# CLASSI BACKWARD COMPATIBILI

- la classe usata per la serializzazione può essere modificata, ma essere sempre **backward-compatible**
  - aggiungere attributi e/o oggetti
  - trasformare attributi transient in non-transient
  - cambiare una variabile di istanza in static
  - e molti altri cambiamenti.....
- in fase di deserializzazione, il meccanismo di default semplicemente imposta i valori dei campi mancanti con valori di default
- il programmatore può “fixare” i valori dei campi aggiunti all'oggetto deserializzato
- modifiche che rendono la classe non backward-compatible
  - rimuovere attributi
  - trasformare attributi non-transient in transient

- il programmatore può generare esplicitamente serialVersionUID per le classi interessate alla serializzazione/deserializzazione
  - 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 esplicita di identificatori
  - mediante tool automatici di JAVA, dato il nome della classe restituiscono un identificatore unico della classe
- dichiarazione identificatori espliciti

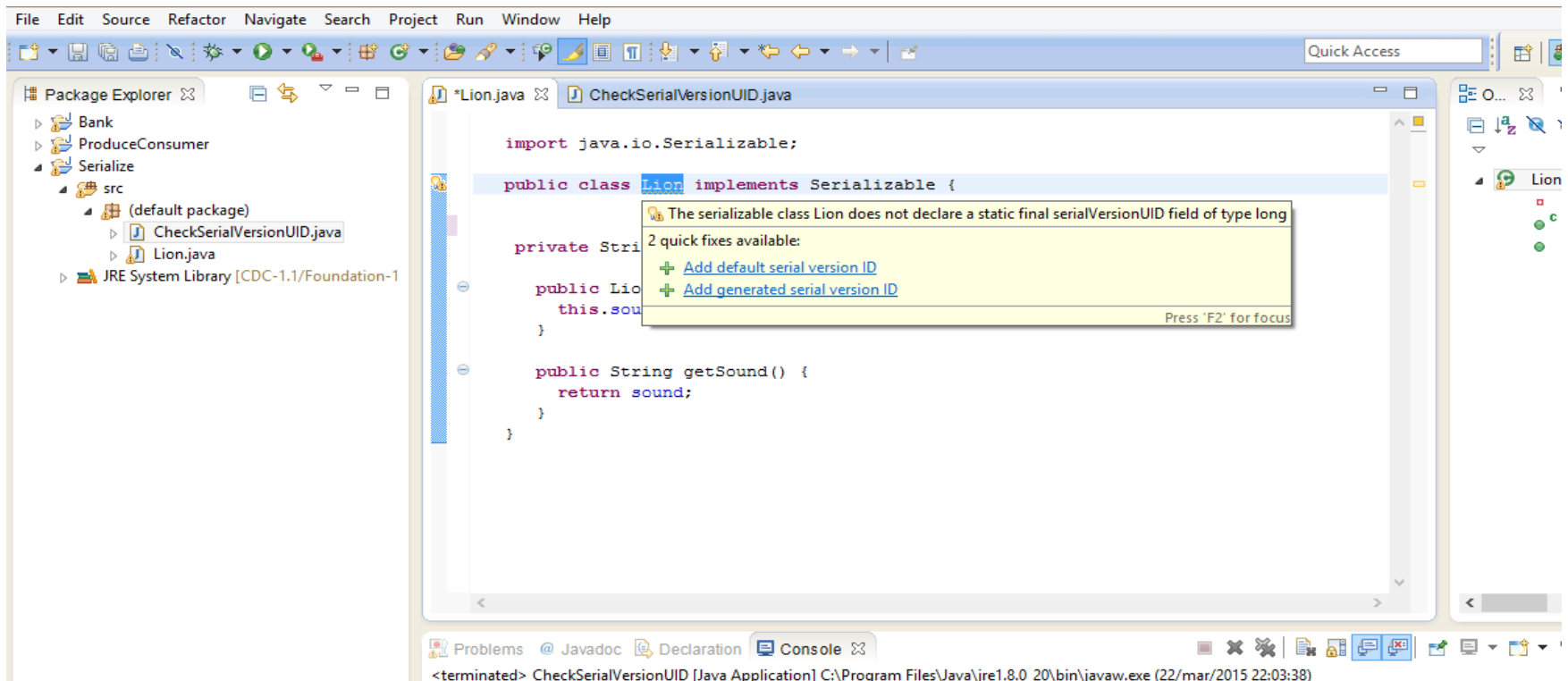
```
private static final long serialVersionUID = 42L;
```



# serialVersionUID ESPLICITO

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)



# CONTROLLO DELLE VERSIONI: UN ESEMPIO

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

public class Employee implements Serializable {
 private static final long serialVersionUID = 1L;
 private String name;
 private String address;
 private byte age;

 public String getName() { return name; }
 public void setName(String name) { this.name = name;}
 public byte getAge() { return this.age;}
 public void setAge(byte age) { this.age = age;}
 public String getAddress() { return address;}
 public void setAddress(String address) { this.address = address;}
 public String whoIsThis()
 { StringBuffer employee = new StringBuffer();
 employee.append(getName()).append("is").append(getAge()).append("years
 old and lives at").append(getAddress());

 return employee.toString(); }}
```

# CONTROLLO DELLE VERSIONI: UN ESEMPIO

```
public class Writer {
 public static void main(String[] args) throws IOException {
 Employee employee = new Employee();
 employee.setName("Maria");
 employee.setAge((byte) 30);
 employee.setAddress("Via Bianchi");
 FileOutputStream fout = new FileOutputStream("./employee.obj");
 ObjectOutputStream oos = new ObjectOutputStream(fout);
 oos.writeObject(employee);
 oos.close();
 System.out.println("Process complete"); } }
```

# CONTROLLO DELLE VERSIONI: UN ESEMPIO

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
public class Reader {
 public static void main(String[] args) throws ClassNotFoundException,
 IOException {
 Employee employee = new Employee();
 FileInputStream fin = new FileInputStream("./employee.obj");
 ObjectInputStream ois = new ObjectInputStream(fin);
 employee = (Employee) ois.readObject();
 ois.close();
 System.out.println(employee.whoIsThis()); }}

```

# “GIOCARRE” CON L'ESEMPIO

- *prova 1*
  - eseguire prima Writer, che serializza l'oggetto con *serialVersionUID* = 1L
  - in seguito eseguire Reader, senza modificare la classe Employee
  - risultato: serializzazione e deserializzazione corrette, usano la stessa classe, con lo stesso ID, il programma stampa:

Maria is 30 years old and lives at Via Bianchi

- *prova 2*
  - eseguire prima Writer, che serializza l'oggetto con *serialVersionUID* = 1L
  - modificare quindi la classe Employee, aggiungendo un campo ed i metodi set e get relativi (vedi slide successiva)
  - in seguito eseguire Reader, che farà riferimento alla classe modificata
  - serializzazione e deserializzazione fanno riferimento a due classi diverse, ma con lo stesso *serialVersionUID*
  - il risultato è:

Maria is 30 years old and lives at Via Bianchi

# “GIOCARRE” CON L'ESEMPIO

```
import java.io.Serializable; import java.util.*;
public class Employee implements Serializable {
 private static final long serialVersionUID = 1L;
 private String name;
 private String address;
 private byte age;
 private Date dateOfBirth;
 public void setDate(Date date) {this.dateOfBirth = date;}
 public Date getDate() {return this.dateOfBirth;}

```

- modificare la classe dopo la serializzazione
- un nuovo campo dateOfBirth
- i metodo get e set relativi

# “GIOCARRE” CON L'ESEMPIO

- prova 3
  - eseguire prima Writer, senza specificare alcun *serialVersionUID* in Employee
  - modificare quindi la classe Employee, aggiungendo un campo e get e set relativi (come nella slide precedente), ma non aggiungere *serialVersionUID*
  - in seguito eseguire Reader, che farà riferimento alla classe modificata
  - serializzazione e deserializzazione fanno riferimento a due classi diverse, e con il *serialVersionUID* implicito e diverso
  - il risultato è:

```
Exception in thread "main" java.io.InvalidClassException: Employee; local class incompatible: stream classdesc serialVersionUID =
-7901079435486647298, local class serialVersionUID = 7127134952141211549
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 java.io.ObjectInputStream.readObject(Unknown Source)
at Reader.main(Reader.java:17)
```

# “GIOCARRE” CON L'ESEMPIO

- prova 4
  - eseguire prima Writer, specificando *serialVersionUID=1L* in Employee
  - modificare quindi la classe Employee, aggiungendo un campo e get e set relativi (come nella slide precedente), e modificare *serialVersionUID=2L*
  - in seguito eseguire Reader, che farà riferimento alla classe modificata
  - serializzazione e deserializzazione fanno riferimento a due classi diverse, e con un *serialVersionUID*, specificato esplicitamente dal programmatore e diverso
  - il risultato è:

```
Exception in thread "main" java.io.InvalidClassException: Employee; local class incompatible: stream classdesc serialVersionUID =
-7901079435486647298, local class serialVersionUID = 7127134952141211549
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 java.io.ObjectInputStream.readObject(Unknown Source)
at Reader.main(Reader.java:17)
```



# 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.
  - compilatori diversi generalo lo stesso serialVersionUID per la stessa classe
- per questo è consigliato di specificare comunque esplicitamente un `serialVersionUID`

# SERIALIZZAZIONE JAVA: SVANTAGGI

La serializzazione registra un descrittore dell'oggetto di dimensione significativa

- i “magic data”
  - `STREAM_MAGIC` = “acde”
  - `STREAM_VERSION` = versione della JVM
- i metadati che descrivono la classe associata all'istanza dell'oggetto serializzato
  - 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

# SERIALIZZAZIONE JAVA: SVANTAGGI

```
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
```

- la classe `SimpleClass` ha i 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
- per la memorizzazione di un oggetto di 20 bytes utilizzati circa 220 bytes!
  - molto costoso in termini di spazio utilizzato
  - alto consumo di banda, se l'oggetto deve essere spedito

# SERIALIZZAZIONE JAVA: SVANTAGGI

- oggetto serializzato aumenta molto di dimensione rispetto all'oggetto originario
- limitata interoperabilità
  - utilizzabile solo 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....

# ASSIGNMENT: FILE CRAWLER

- si scriva un programma JAVA che
  - riceve in input un *filepath* che individua una *directory D*
  - stampa le informazioni del contenuto di quella *directory* e, ricorsivamente, di tutti i file contenuti nelle sottodirectory di *D*
- il programma deve essere strutturato come segue:
  - attiva un *thread produttore* ed un insieme di *k thread consumatori*
  - il produttore comunica con i consumatori *mediante una coda*
  - il produttore visita ricorsivamente la *directory* data e, eventualmente tutte le sottodirectory e mette nella coda il nome di ogni *directory* individuata
  - i consumatori prelevano dalla coda i nomi delle *directories* e stampano il loro contenuto
  - la coda deve essere realizzata con una *LinkedList*. Ricordiamo che una *LinkedList* non è una struttura thread-safe. Dalle API JAVA “Note that the implementation is not synchronized. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally”