Java

il sistema di I/O

G. Prencipe
prencipe@di.unipi.it

Introduzione

- La gestione del sistema di I/O è una parte fondamentale di qualsiasi linguaggio di programmazione
- In questa lezione approfondiremo la gestione dell'I/O in Java, descrivendo le principali classi coinvolte

Streams

- Le librerie per I/O usano spesso l'astrazione di *stream*, che rappresenta una sorgente o una destinazione di dati come un oggetto capace di produrre o ricevere dati in forma di *flusso*
- Per ricevere informazioni, un programma apre uno stream verso una sorgente (un file, la memoria, una socket) e legge le informazioni sequenzialmente, come mostrato in figura

Streams

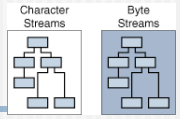
- Analogamente, un programma può inviare informazioni a una destinazione esterna aprendo uno stream verso di essa e scrivendo le inforamzioni sequenzialmente, come mostrato in figura

Streams

- Indipendentemente dal tipo dei dati e da dove provengano o dove essi siano diretti, gli algoritmi per leggere e scrivere sequenzialmente dati sono essenzialmente gli stessi

Per leggere	Per scrivere
open uno stream	**open** a stream
while ci sono dati	**while** ci sono dati
read dati	**write** dati
close lo stream	**close** lo stream

Java e gli streams



- Al solito, anche gli streams in Java sono oggetti
- Il pacchetto **java.io** contiene una collezioni di classi che supportano algoritmi per leggere e scrivere su stream
  - Per utilizzare queste classi bisogna importare **java.io**
- Le classi relative agli stream sono divise in due gerarchie, basate sul tipo di dati letti e scritti (caratteri o bytes), come mostrato in figura

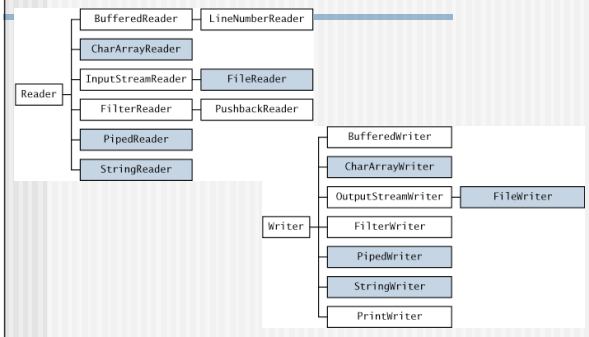
Classi **Reader** e **Writer**

- Derivano da **Object**, quindi non sono "imparentate" con **InputStream** e **OutputStream**
  - Furono introdotte successivamente a **InputStream** e **OutputStream**, ma non le hanno rimpiazzate in alcun modo
- **Reader** e **Writer** sono le superclassi astratte per gli stream di caratteri in **java.io**
  - Gli stream di caratteri sono stream di caratteri di 16-bit

Reader e **Writer**

- **Reader** offre una parziale implementazione per gli stream in lettura e **Writer** per quelli in scrittura
- Le sottoclassi di **Reader** e **Writer** implementano stream specializzati e sono divisi in due categorie
  - Quelle che leggono da e scrivono verso diverse sorgenti e destinazioni, rispettivamente (mostrate in grigio nella figura che segue)
  - Quelle che effettuano qualche tipo di trattamento dei dati (mostrate in bianco)

Reader e Writer -- gerarchia



Reader e Writer

- Molti programmi dovrebbero utilizzare queste classi per leggere e scrivere informazioni testuali
- Il motivo è legato al fatto che esse possono trattare caratteri dello standard *Unicode* (a 16 bit), mentre gli stream di bytes (**InputStream** e **OutputStream**) sono limitati ai bytes da 8-bit dell'*ISO-Latin-1*

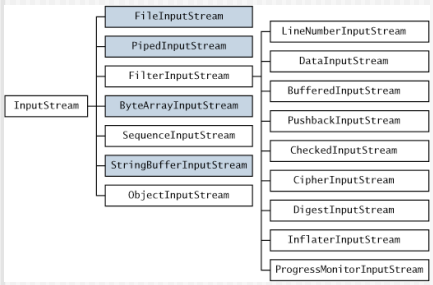
Stream di bytes a 8-bit

- Per leggere e scrivere bytes a 8 bit, i programmi dovrebbero utilizzare gli stream di bytes, gestiti dai discendenti delle superclassi **InputStream** e **OutputStream**
  - Questi stream sono tipicamente utilizzati per leggere dati in formato binario, come ad esempio immagini e suoni
- **InputStream** e **OutputStream** forniscono una parziale implementazione per questi tipi di stream
- Due delle classi che gestiscono stream di byte, **ObjectInputStream** e **ObjectOutputStream**, sono utilizzate per la *serializzazione* (che tratteremo in seguito)

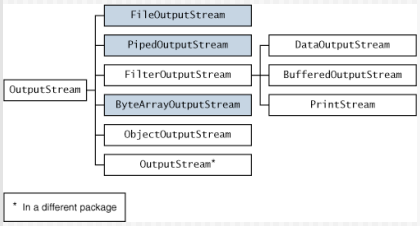
Stream di bytes a 8-bit

- Come per Reader e Writer, le sottoclassi di InputStream e OutputStream forniscono la gestione specializzata per gli stream di bytes, e sono suddivise in due categorie
- Gestione delle letture e scritture di dati, e trattamento degli stream

Stream di bytes -- gerarchia



Stream di bytes -- gerarchia



Metodi principali

- **Reader** e **InputStream** definiscono metodi simili ma per diversi tipi di dato
- Per esempio, **Reader** contiene i seguenti metodi per leggere caratteri e array di caratteri
  - int read()
  - int read(char cbuf[])
  - int read(char cbuf[], int offset, int length)
- **InputStream** definisce gli stessi metodi ma per leggere bytes e array di bytes
  - int read()
  - int read(byte cbuf[])
  - int read(byte cbuf[], int offset, int length)

Metodi principali

- Inoltre, sia **Reader** che **InputStream** forniscono metodi per
  - *marcare* una locazione nello stream
  - *saltare* alcuni dati in input, e
  - *resettare* la posizione corrente nello stream

Metodi principali

- Analogamente, **Writer** e **OutputStream** procedono parallelamente
- **Writer** definisce i seguenti metodi per scrivere caratteri e array di caratteri
  - int write(int c)
  - int write(char cbuf[])
  - int write(char cbuf[], int offset, int length)
- E **OutputStream** definisce gli stessi metodi, ma per i bytes
  - int write(int c)
  - int write(byte cbuf[])
  - int write(byte cbuf[], int offset, int length)

Metodi principali

- Tutti gli streams — readers, writers, input streams, e output streams — sono *automaticamente aperti* al momento della creazione
- Per *chiudere* uno stream, si invoca il suo metodo **close()**
- Un programma dovrebbe chiudere uno stream appena ha terminato di utilizzarlo, in modo da liberare risorse di sistema

Classe **InputStream**

- **InputStream** può rappresentare classi che producono input da diverse sorgenti
- Esse possono essere
  - Array di **byte**s
  - Una **String**a di oggetti
  - Un file
  - Un *pipe* (si inseriscono dati da un lato del "tubo" e vengono fuori dall'altro)
  - Una sequenza di altri *stream*, in moso da poterli riunirre in un unico *stream*
  - Altre sorgenti, tipo connessione Internet

Classe **InputStream**

- Ognuna di queste sorgenti ha associata una sottoclasse di **InputStream**
- Inoltre, **FilterInputStream** è anche un tipo (sottoclasse) di **InputStream** e fornisce una classe per poter aggiungere attributi o interfacce agli stream di input

Tipi di **InputStream**

- Ecco alcune delle sottoclassi esistenti
  - **ByteArrayInputStream**
    - Permette di utilizzare un buffer in memoria come **InputStream**
  - **StringBufferInputStream**
    - Converte **String**he in **InputStream**
  - **FileInputStream**: per leggere da file

Classe **OutputStream**

- Include le classi che servono a stabilire dove inviare l'output
  - Array di **byte**s, un file o un "pipe"
- Inoltre, **FilterOutputStream** è anche un tipo (sottoclasse) di **InputStream** e fornisce una classe per poter aggiungere attributi o interfacce agli stream di output

Tipi di **OutputStream**

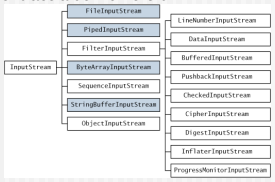
- Tra le sottoclassi di **OutputStream** troviamo
  - **ByteArrayOutputStream**
    - Permette di utilizzare un buffer in memoria come **OutputStream**
  - **FileOutputStream**: per scrivere su file

Classe **FilterInputStream**

- È una sottoclasse di **InputStream**
- Un **FilterInputStream** contiene al suo interno un qualche altro **InputStream** (il suo costruttore infatti ne prende uno come argomento), e lo utilizza come sorgente di dati
- Può trasformare i dati provenienti da questa sorgente o fornisce funzionalità aggiuntive per il loro trattamento
- Serve a leggere dati provenienti da un altro stream

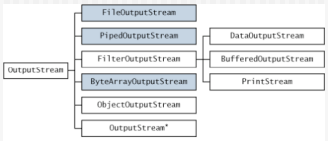
Classe FilterInputStream

- Ad esempio, una sua sottoclasse è **DataInputStream**
  - Essa permette di leggere differenti tipi di dati primitivi (offre i metodi **readByte()**, **readFloat()**, ecc.)
- Altre sottoclassi modificano il modo con cui agisce l'**InputStream** che contiene
  - Se con buffer o no, se deve tener traccia del numero di linee che sta leggendo, ecc.



Classe FilterOutputStream

- È una sottoclasse di **OutputStream**
  - È la controparte per le scritture di **FilterInputStream**
- In **DataOutputStream** troviamo **writeByte()**, **writeFloat()**,ecc.
- Serve a scrivere dati in un formato che possa essere letto da qualche altro stream



Classe FilterOutputStream

- Una sua sottoclasse è **PrintStream**
- Il suo scopo è di stampare dati primitivi e **String**he in un formato leggibile da un umano
  - Diverso da **DataOutputStream**, che semplicemente mette elementi su uno stream in un formato che **DataInputStream** può riconoscere

Classe PrintStream

- I due metodi importanti in **PrintStream** sono **print()** e **println()**
  - Sono definiti per tutti i tipi
- Non lancia **IOException** e situazioni d'errore devono essere controllate manualmente con il metodo **checkError()** che controlla lo stato di una variabile interna
- Non gestisce le interruzioni di linea in modo indipendente dalla piattaforma
  - Problemi risolti in **PrintWriter** nella gerarchia di **Writer**
    - Ricordiamo infatti che **Writer** è successiva a **OutputStream**

Gli "stream reader"

- **InputStreamReader** e **OutputStreamWriter** (nella gerarchia di **Reader** e **Writer**) sono delle classi che fungono da "ponte" tra l'approccio a bytes e quello a caratteri
- Ad esempio, **InputStreamReader** converte da **InputStream** a **Reader**

Corrispondenza

- Quasi tutte la classi Java originali per gestire stream hanno un corrispondente in **Reader** e **Writer** per fornire compatibilità con lo standard *Unicode*
- Ci sono delle situazioni in cui la soluzione corretta prevede l'utilizzo di **InputeStream** e **OutputStream** orientati ai **byte**
  - Come nelle librerie **java.util.zip**
- In generale si consiglia l'utilizzo di **Reader** e **Writer**
  - Se si rende necessario l'utilizzo dell'approccio a **byte**, ci saranno errori in compilazione, e quindi si ricorre a **InputStream** e **OutputStream**

Corrispondenza

- Questa tabella mostra la corrispondenza fra le due gerarchie

Sources & Sinks: Java 1.0 class	Corresponding Java 1.1 class
InputStream	Reader adapter: InputStreamReader
OutputStream	Writer adapter: OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
(no corresponding class)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

Filtri

- Anche **Reader** e **Writer** prevedono l'utilizzo di filtri analoghi a **FilterInputStream** e **FilterOutputStream**
- La differenza è nella organizzazione delle classi
- Ecco una tabella che mostra approssimativamente le corrispondenze esistenti

Filtri

- Nota: se si utilizza **readLine()** in **DataInputStream** si ottiene un *warning* (*deprecated*)

Filters: Java 1.0 class	Corresponding Java 1.1 class
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter (abstract class with no subclasses)
BufferedInputStream	BufferedReader (also has readLine( ))
BufferedOutputStream	BufferedWriter
DataInputStream	Use DataInputStream (Except when you need to use readLine( ), when you should use a BufferedReader)
PrintStream	PrintWriter
LineNumberInputStream (deprecated)	LineNumberReader
StreamTokenizer	StreamTokenizer (use constructor that takes a Reader instead)
PushBackInputStream	PushBackReader

Classi non modificate

- Queste classi sono rimaste invariate da Java1.0 a Java1.1
  - **DataOutputStream**
  - **File**
  - **RandomAccessFile**
  - **SequenceInputStream**

Classe RandomAccessFile

- È utilizzata per file che contengono record di dimensione nota
- Permette di muoversi tra i vari record (**seek()**), leggere o modificarli
- I record non devono necessariamente essere della stessa dimensione
  - Bisogna essere in grado di determinare quanto sono grandi e dove sono nel file

Classe RandomAccessFile

- Non è sottoclasse di nessuna delle gerarchie viste (discende da **Object**)
  - Questo perché accede ai file in maniera diversa da quella offerta dagli stream
  - Implementale interfacce **DataInput** e **DataOutput**
- Ha metodi per determinare dove ci si trova nel file (**getFilePointer()**), per muoversi nel file **(seek()**), e determinare la lunghezza del file (**length()**)
- Il costruttore richiede un secondo argomento per stabile se il file deve essere aperto in sola lettura o in lettura e scrittura

Utilizzo tipico degli stream

- Da quanto visto finora, è possibile combinare tutte le classi a disposizione in tanti modi
- In genere, comunque, solo alcune combinazioni sono utilizzate
- Vediamo un esempio che mostra la creazione e l'utilizzo di tipiche configurazioni I/O

Esempio

1. Per aprire un file per leggere caratteri, si usa un **FileReader** con una **String**a o un **File** per rappresentare il nome del file
   1. Per velocizzare le operazioni, si utilizza un buffer con **BufferedReader**
   2. Essa fornisce **readLine()**, che al termine del file restituisce **null**
   3. Per leggere input da Console si utilizza **System.in**, che è un **InputStream**
      1. **BufferedReader** necessita di un **Reader** e quindi **InputStreamReader** è il "ponte" che sistema tutto

Esempio

```
public class IOStreamDemo {
  private static Test monitor = new Test();
  // Lancia eccezioni su Console
  public static void main(String[] args) throws IOException {
  // 1. Legge input per linee
    BufferedReader in = new BufferedReader(
      new FileReader("IOStreamDemo.java"));
    String s, s2 = new String();
    while((s = in.readLine())!= null)
      s2 += s + "\n";
    in.close();
// 1b. Legge da standard input
    BufferedReader stdin = new BufferedReader(
            new InputStreamReader(System.in));
    System.out.print("Enter a line:");
    System.out.println(stdin.readLine());
```

Esempio

2. **StringReader** (sottoclasse di **Reader**) rappresenta uno stream di caratteri la cui sorgente è una **String**a
   1. Il metodo **read()** legge un carattere e lo restituisce come **int**

Esempio

```
// 2. Input da memoria -- s2 è stato definito in 1.
   StringReader in2 = new StringReader(s2);
   int c;
   while((c = in2.read()) != -1)
     System.out.print((char)c);
```

Esempio

3. Per leggere dati formattati, si usa **DataInputStream** (orientata ai **byte**)
   1. Quindi si utilizzano le classi **InputStream**
   2. Per leggere una **String**a con queste classi, bisogna convertirla in array di **byte**s che viene passato a **ByteArrayInputStream**
   3. La fine del file è determinata catturando **EOFException**
      1. Infatti ogni byte è considerato input da leggere, e non si può utilizzare alcun carattere particolare come "ultimo"
   4. Alternativamente si può utilizzare il metodo **available()** per stabilire quanti caratteri sono ancora disponibili

Esempio

```
// 3. Input formattato da memoria
   try {
     DataInputStream in3 = new DataInputStream(
       new ByteArrayInputStream(s2.getBytes()));
     while(true)
       System.out.print((char)in3.readByte());
   } catch(EOFException e) {
     System.err.println("End of stream");
   }
```

Esempio

4. Per scrivere dati in un file si può creare un **FileWriter**
   1. Analogamente alle letture, per velocizzare le operazioni si utilizza un buffer con **BufferedWriter**
   2. Per facilitare la formattazione si trasforma il tutto in **PrintWriter**
      1. In questo modo il file creato è leggibile come un normale file di testo

Esempio

```
// 4. File output
  try {
    BufferedReader in4 = new BufferedReader(
      new StringReader(s2));
    PrintWriter out1 = new PrintWriter(
      new BufferedWriter(new FileWriter("IODemo.out")));
    int lineCount = 1;
    while((s = in4.readLine()) != null )
      out1.println(lineCount++ + ": " + s);
    out1.close();
  } catch(EOFException e) {
    System.err.println("End of stream");
  }
```

Esempio

5. **PrintWriter** formatta i dati in modo che siano leggibili da un umano
   1. Per formattarli in modo che siano recuperabili da un altro stream si utilizza **DataOutputStream**
      1. Sottoclasse di **OutputStream**
   2. In questo modo Java garantisce che i dati verranno recuperati correttamente un **DataInputStream** indipendentemente dalla piattaforma
   3. Per scrivere una **String**a in modo da non creare problemi a **DataInputStream** è di utilizzare la codifica *UTF-8* (variazione di Unicode che memorizza i caratteri in 2 **byte**s)

Esempio

```
// 5. Memorizzare dati
  try {
    DataOutputStream out2 = new DataOutputStream(
      new BufferedOutputStream(
        new FileOutputStream("Data.txt")));
    out2.writeDouble(3.14159);
    out2.writeUTF("That was pi");
    out2.writeDouble(1.41413);
    out2.writeUTF("Square root of 2");
    out2.close();
    DataInputStream in5 = new DataInputStream(
      new BufferedInputStream(
        new FileInputStream("Data.txt")));
```

Esempio

```
// Deve usare DataInputStream per i dati
    System.out.println(in5.readDouble());
    // Solo readUTF() recupera
    // Java-UTF String correttamente
    System.out.println(in5.readUTF());
    // Legge Double e String
    System.out.println(in5.readDouble());
    System.out.println(in5.readUTF());
  } catch(EOFException e) {
    throw new RuntimeException(e);
  }
```

Esempio

6. La classe **RandomAccessFiles** è isolata dalle altre (a parte il fatto che implementa **DataInput** e **DataOutput**)
   1. Quindi non la si può combinare con le caratteristiche di **InputStream** e **OutputStream**

Esempio

```
// 6. Leggere/scrivere con RandomAccesFiles
  RandomAccessFile rf =
    new RandomAccessFile("rtest.dat", "rw");
  for(int i = 0; i < 10; i++)
    rf.writeDouble(i*1.414);
  rf.close();
  rf = new RandomAccessFile("rtest.dat", "rw");
  rf.seek(5*8);
  rf.writeDouble(47.0001);
  rf.close();
  rf = new RandomAccessFile("rtest.dat", "r");
  for(int i = 0; i < 10; i++)
    System.out.println("Value " + i + ": " +
      rf.readDouble());
  rf.close();}} ///:~
```

Esercizio

- Scrivere un programma **FileToString.java** che prende in input (come argomento al programma) il nome di un file e lo trasforma in **String**a
  - Stampare la **String**a ottenuta
- Come prova lanciare il programma passando come argomento **FileToString.java**

Standard I/O

- Il termine *standard I/O* si riferisce al concetto di Unix di un singolo stream di informazione che viene usato da un programma
- Tutti gli input dei programmi possono provenire dallo *standard input*, gli output essere diretti allo *standard output*, e gli errori allo *standard error*

Standard I/O

- Java fornisce standard I/O tramite **System.in**, **System.out** e **System.err**
- **System.out** e **System.err** sono dei **PrintStream** (derivate dal filtro applicato a **OutputStream**)
  - Possono essere utilizzate direttamente tramite i metodi **print()** e simili
- **System.in** invece è un **InputStream**
  - Per usarlo si crea con esso un **BufferedReader**

```
BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in))
```

Standard I/O

- Come detto, **System.out** è un **PrintStream** (è nella gerarchia degli **OutputStream**)
- È possibile trasformarlo in **PrintWriter** (per utilizzarlo nella gerarchia dei **Writer**)
  - Si utilizza un costruttore di **PrintWriter** che prende **OutputStream** come argomento

```
PrintWriter out = new PrintWriter(System.out, true)
```

  - Il costruttore **PrintWriter** che prende anche **boolean** serve per fare il *flush* automatico dei caratteri nello stream (altrimenti si potrebbe non vedere l'output)

Ridirezionare lo standard I/O

- La classe **System** permette di ridirezionare gli stream di standard input, output e error
  - **setIn(InputStream)**
  - **setOut(PrintStream)**
  - **setErr(PrintStream)**
- Ridirezionare l'output è utile quando si hanno grandi quantità di output
- Ridirezionare l'input è utile per testare uno stesso insieme di comandi da passare a un programma

Compressione

- La libreria Java per l'I/O contiene classi che supportano la lettura e la scrittura di stream compressi
- Queste classi sono parti delle gerarchie di **InputStream** e **OutputStream**
  - Le librerie di compressione lavorano con **byte**s e non caratteri
  - Nel caso in cui si sia forzati a passare ai caratteri, bisogna sfruttare le classi "ponte" **InputStreamReader** e **OutputStreamWriter**

Compressione

- Le classi **GZIPOutputStream** e **GZIPInputStream** servono a scrivere in un file dati compressi e leggere dati compressi con GZIP
- Il loro utilizzo è semplice: si racchiudono gli stream di output/input in queste classi e poi si utilizzano come negli I/O non compressi
- Vediamo un esempio

Esempio

```
BufferedReader in = new BufferedReader(
    new FileReader(args[0]));
BufferedOutputStream out = new BufferedOutputStream(
  new GZIPOutputStream(
    new FileOutputStream("test.gz")));
System.out.println("Writing file");
int c;
while((c = in.read()) != -1)
  out.write(c);
in.close();
out.close();
System.out.println("Reading file");
BufferedReader in2 = new BufferedReader(
  new InputStreamReader(new GZIPInputStream(
    new FileInputStream("test.gz"))));
String s;
while((s = in2.readLine()) != null)
  System.out.println(s);  }} ///:~
```

Compressione

- Esistono altre funzionalità legate alla compressione
  - Come ad esempio la compressione di più file
- Maggiori informazioni disponibili nella documentazione di Java

JAR

- Il formato Zip è utilizzato anche negli Java ARchive (JAR), che è un modo per collezionare un gruppo di file in un unico file compresso
- File JAR sono indipendenti dalla piattaforma

JAR

- Sono particolarmente utili con il Web
  - Senza JAR, un browser deve fare ripetute richieste per ottenere tutti i file necessari a far girare una applet
  - Con i JAR invece si invia tutto in un unico file compresso
- Un JAR consiste di una collezione di file zippati insieme a un "manifesto" che li descrive
- L'utilità **jar** distribuita con la JDK automaticamente comprime i file
- Se si crea un archivio JAR utilizzando l'opzione **0**, l'archivio può essere inserito nel CLASSPATH e utilizzato da Java

Serializzazione

- La *serializzazione* degli oggetti permette di prendere un oggetto che implementa l'interfaccia **Serializable** e trasformarlo in una sequenza di **byte**s
  - Crea un'immagine dell'oggetto
- Successivamente è possibile prendere questa sequenza e ricomporla per rigenerare l'oggetto di partenza
- È molto utile nello scambio di oggetti su rete
  - Ci si scambia dati che poi sono ricostruiti in oggetti sulla piattaforma specifica

Serializzazione

- È molto semplice serializzare un oggetto: è sufficiente implementare **Serializable** (che non ha metodi!!)
- Per utilizzare la serializzazione bisogna
  - Creare un oggetto **OutputStream** e racchiuderlo in un oggetto **ObjectOutputStream oos**
  - A questo punto si chiama **oos.writeObject(Object o)** e l'oggetto **o** è serializzato e inviato all'**OutputStream**
  - Simile per la lettura (**ObjectInputStream** e **readObject**)
  - Chiaramente in lettura si ottiene un riferimento a **Object**

Serializzazione

- Nella serializzazione di un oggetto vengono salvati i campi non-statici e non-transienti (vedremo dopo) dell'oggetto
- Inoltre quando si serializza un oggetto vengono salvati anche i suoi riferimenti ad altri oggetti (purchè implementino a loro volta **Serializable**)
- Vediamo un esempio: **Worm.java**

Serializzazione e .class

- Supponiamo di creare una classe **A Serializable**
  - Creiamo un suo oggetto **x**, serializziamolo e scriviamolo su un file
  - Poi recuperiamo l'oggetto **x** da questo file (deserializzando)
  - Se **A.class** non è nella stessa directory di **B** o nel CLASSPATH, il tentativo di deserializzazione produce una **ClassNotFoundException**
  - Cioè, è importante sottolineare che la JVM deve avere accesso ai **.class** degli oggetti da deserializzare

La parola chiave transient

- Non sempre si vuole serializzare tutto di un oggetto
  - Ad esempio, se un oggetto contiene dati sensibili (come una password)
- Per evitare di serializzare questi campi, è sufficiente dichiararli come **transient**

La classe Preferences

- JDK1.4 ha introdotto le *Preferences API* che automaticamente salvano e recuperano informazioni
  - **java.util.prefs**
- Il loro utilizzo è ristretto a piccoli insiemi di dati
  - È possibile conservare solo tipi primitivi e Stringhe, e la lunghezza di ogni singola **String**a non può superare gli 8K
  - Servono appunto per salvare preferenze degli utenti o configurazioni dei programmi

La classe Preferences

- Sono coppie **chiave-valore** memorizzate in una gerarchia di nodi
- Il metodo **statico userNodeForPackage(Class c)** serve a restituire uno di questi nodi per la classe **c**
- Il metodo **get(chiave, default)** restituisce il **valore** associato alla **chiave**; se non esiste la **chiave**, viene restituito il valore di **default**
- Il metodo **put(chiave, valore)** inserisce la coppia nel nodo
  - **put()** e **get()** per tutti i tipi primitivi e per le **String**he
    - **putInt()**, **putDouble()**, ecc.
- Vediamo un esempio

Esempio

```
public class PreferencesDemo {
  public static void main(String[] args) throws Exception {
    Preferences prefs = Preferences.userNodeForPackage(PreferencesDemo.class);
    prefs.put("Location", "Oz");
    prefs.put("Footwear", "Ruby Slippers");
    prefs.putInt("Companions", 4);
    prefs.putBoolean("Are there witches?", true);
    int usageCount = prefs.getInt("UsageCount", 0);
    usageCount++;
    prefs.putInt("UsageCount", usageCount);
    Iterator it = Arrays.asList(prefs.keys()).iterator();
    while(it.hasNext()) {
      String key = it.next().toString();
      System.out.println(key + ": "+ prefs.get(key, null));
    }
    System.out.println("How many companions does Dorothy have? " +
      prefs.getInt("Companions", 0));
  }} ///:~
```

Esercizi

1. Scrivere un programma **NumeraLinee.java** che prende come argomento (da linea di comando) i nomi di due file
   1. Legge il primo file riga per riga
   2. Scrive ogni riga nel secondo file, aggiungendo un numero di riga
   3. Stampa a console il contenuto del secondo file

Java

il sistema di I/O

fine