Programmazione Orientata agli Oggetti

Java Input/Output (Stream)

Sommario

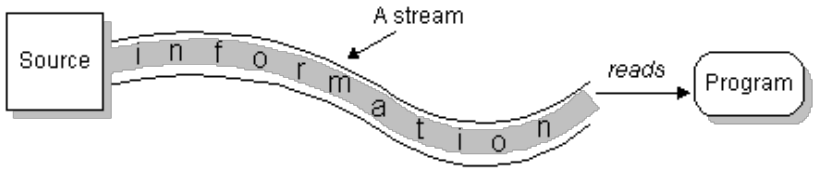
- Gestione Input/Output in Java
- Introduzione agli stream
  - caratteri vs byte : charset ed encoding
  - **`InputStreamReader`**
  - Gerarchie per la gestione I/O in **`java.io`**:
    - **`Reader/Writer`**
    - **`InputStream/OutputStream`**
- Esternalizzazione delle costanti
  - **`java.util.Properties`**
- Accesso alle risorse esterne
  - Risoluzione dei nomi logici di risorse
  - **`Class`** vs **`ClassLoader`**

Gestione dell'I/O in Java

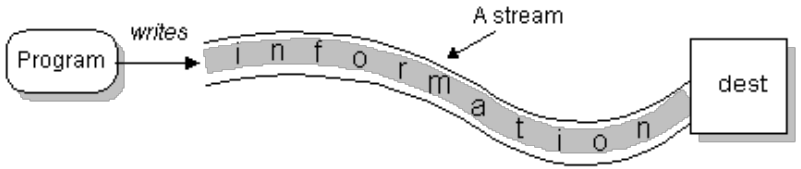
- Il package **`java.io.*`** di Java
  - datato, alcuni storici limiti, ma tuttora popolare (e necessario...)
- La più importante astrazione per la manipolazione dell'I/O è quella di *stream (flusso)*
  - «Stream»: *collezione di dati a cui è possibile accedere sequenzialmente, accedendo ad un dato solo dopo aver acceduto al precedente nella sequenza*
- N.B. Parliamo di stream per la gestione dell'I/O
  - ✓ da non confondere con gli *Stream* di Java 8 nel JCF

Stream di Input / Output

- Uno stream può essere di
  - *Input:* se permette di leggere i dati prodotti da una «sorgente» (stringa, file, connessione di rete...)



  - *Output:* se permette scrivere dati diretti ad una «destinazione» (di nuovo: stringa, file, connessione ...)



Stream: Caratteri vs Byte

- Uno stream di byte può essere interpretato in molti modi
  - Stream di caratteri (testi)
  - Stream binari (es. immagini, eseguibili, .class, ecc. ecc.)
- Una delle possibili e più frequenti interpretazioni di uno stream di byte è come stream di caratteri, che per importanza meritano una gestione specializzata
- Per accedere agli stream esistono classi che agiscono da «lettori» e «scrittori» estendendo queste classi astratte
  - Stream di caratteri: ***`Reader`*** e ***`Writer`***
  - Stream di byte: ***`InputStream`***, ***`OutputStream`***
- Molti tipi di sorgenti e destinazioni:
  - Stringhe: **`StringReader`**, **`StringWriter`**
  - File: **`FileReader`**, **`FileWriter`**, **`FileInputStream`**

Reader : *InputStream* = char : byte

- Per convertire uno stream di byte in uno di caratteri è necessario:
  - fissare un insieme di caratteri disponibili («charset»)
  - definire un algoritmo di conversione (codifica/«encoding») dello stream di byte in uno stream di caratteri
- Java internamente gestisce una *variante* della codifica UTF-16 per il charset *unicode* (a 16 bit) per permettere al tipo primitivo **`char`** di coprire gli alfabeti più diffusi
  - Vedi **`javadoc java.lang.Character`**
- Nonostante tutto questo fosse decisamente innovativo per l'epoca in cui Java è nato e si è diffuso, l'avvento del Web ha reso indispensabile il supporto di ancora altre codifiche

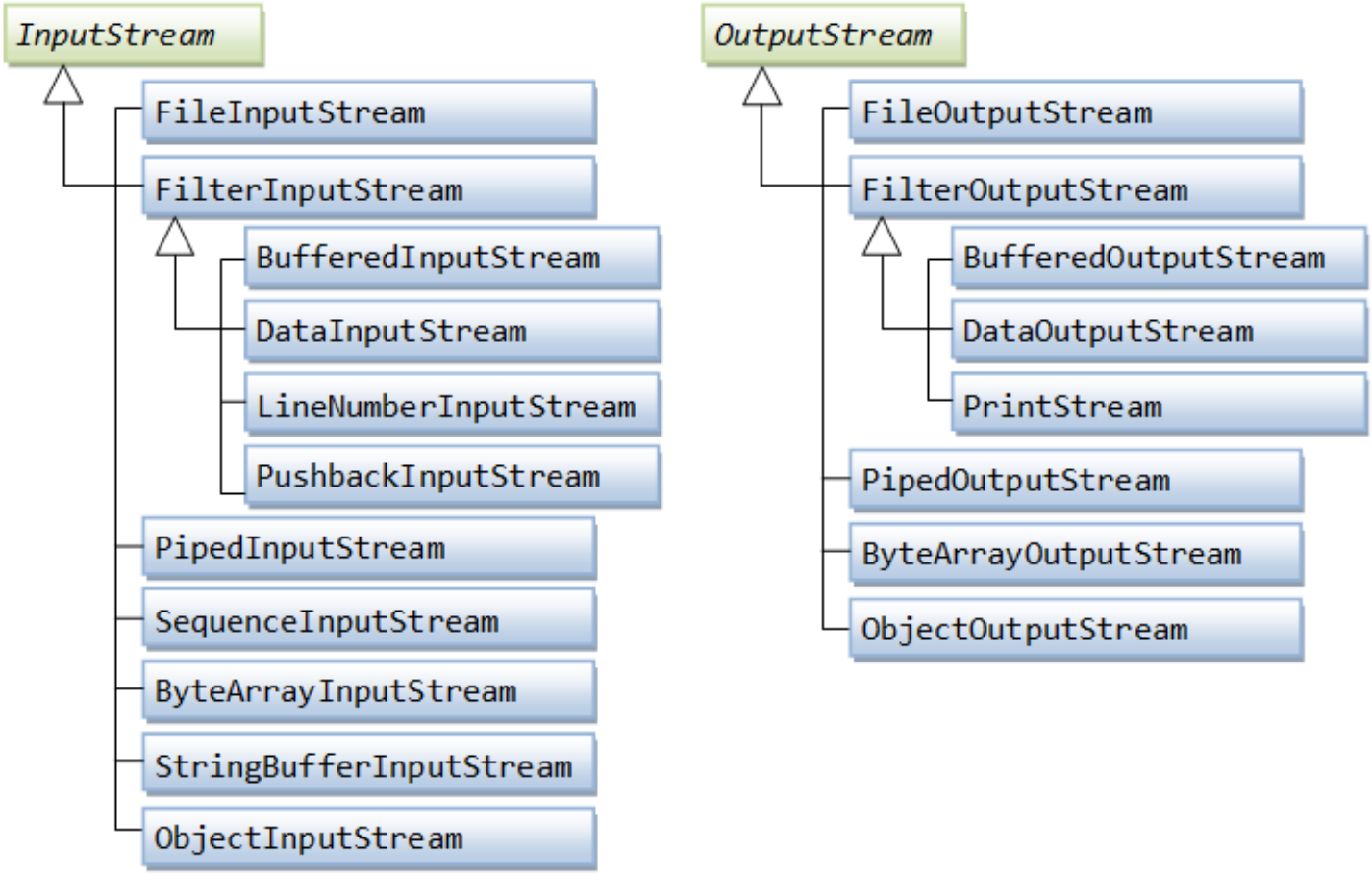
Stesso Charset Diversi Encoding

- Lo stesso charset unicode è rappresentato in diverse codifiche supportate anche da Java
  - UTF-8
    - ✓Forte diffusione perché codifica i primi 127 caratteri come la codifica ASCII, con un solo byte, ed è quindi più compatta
  - UTF-16
  - UTF-32
  - …

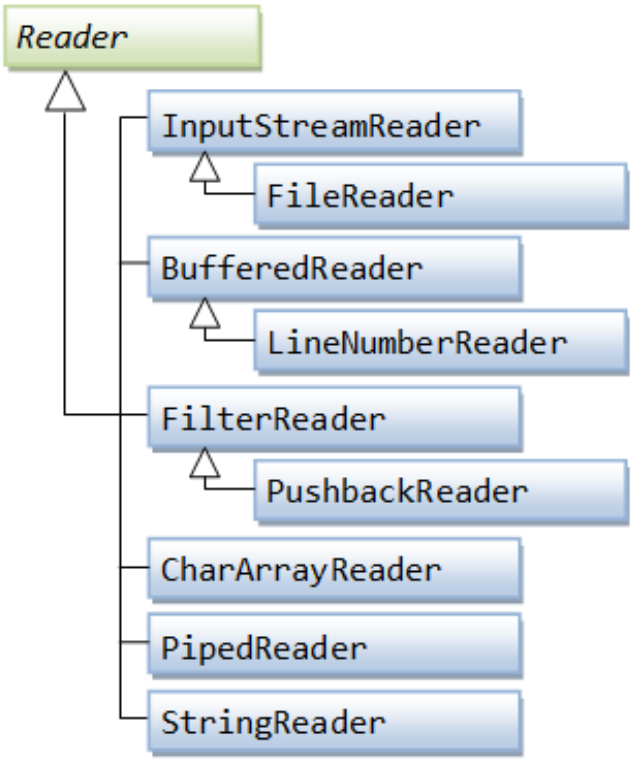
	A	א	好	不
Code point	U+0041	U+05D0	U+597D	U+233B4
UTF-8	41	D7 90	E5 A5 BD	F0 A3 8E B4
UTF-16	00 41	05 D0	59 7D	D8 4C DF B4
UTF-32	00 00 00 41	00 00 05 D0	00 00 59 7D	00 02 33 B4

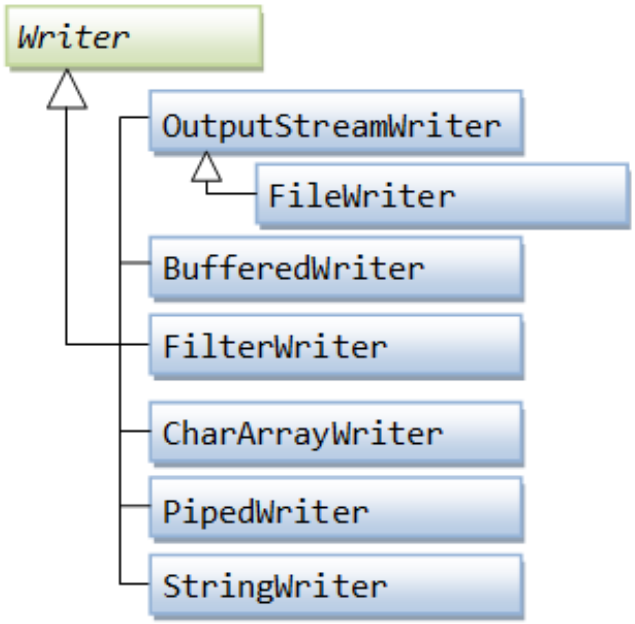
https://www.w3.org/International/articles/definitions-characters/

Gerarchia
InputStream/OutputStream

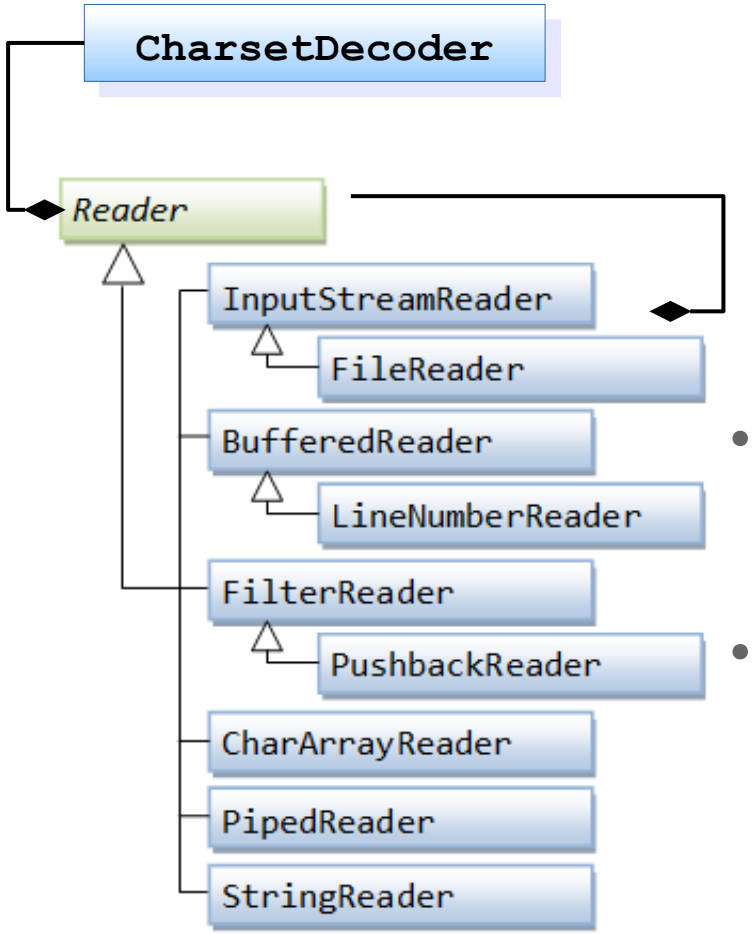


Gerarchia Reader/Writer





Reader/Writer vs InputStream/OutputStream: InputStreamReader



- Uno stream di caratteri presuppone un *encoding* per interpretare il sottostante stream di byte: se non specificato esplicitamente si usa un encoding di default fissato dalla piattaforma
  - ✓ per questo motivo spesso si lavora con stream di caratteri anche senza precisare l'encoding
- Il legame tra le due gerarchie di stream, è reso evidente da una classe in particolare: **`InputStreamReader`**
- Riceve nel costruttore un **`InputStream`** e la specifica di un encoding per interpretare lo stream di byte e convertirlo in un corrispondente stream di caratteri **`Reader`**:

```
Reader in = new InputStreamReader(
   new FileInputStream("test.txt"), "UTF-8")
);
```

Principali Metodi di `Reader/Writer`

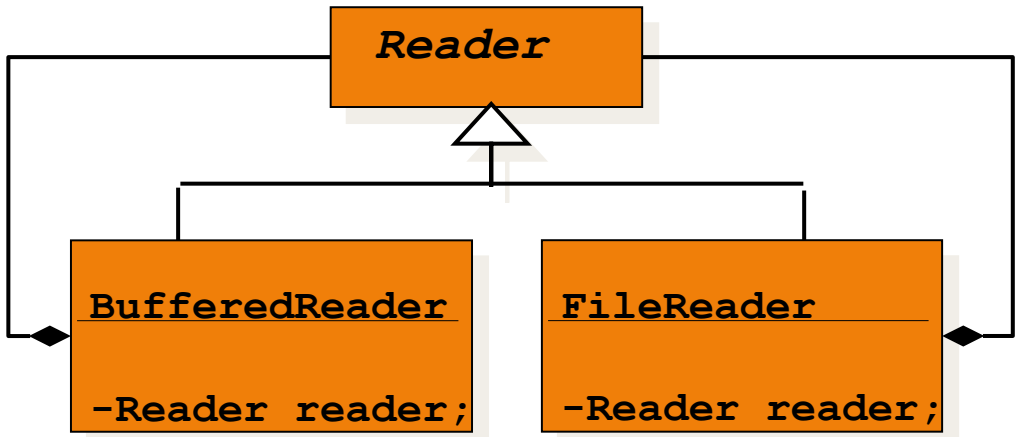
- I principali metodi di **`Reader`**
  - **`int read()`**
  - **`int read(char cbuf[])`**
  - **`int read(char cbuf[], int offset, int length)`**
- E i principali metodi di **`Writer`**
  - **`int write(int c)`**
  - **`int write(char cbuf[])`**
  - **`int write(char cbuf[], int offset, int length)`**
- Tutti questi metodi lanciano una checked exception: **`java.io.IOException`**
- Per ottenere le segnature dei corrispondenti metodi in **`InputStream`** ed **`OutputStream`** basta sostituire:

  `char` → **`byte`**

Composizione per Decorazione

- Anche se le funzionalità di base di un **`Reader`** (**`Writer`**) risultano piuttosto limitate (come si conviene ad un tipo astratto che poi bisognerà implementare in tante «forme» concrete), **`java.io`** offre diverse implementazioni per arricchirle
- Le funzionalità risultano così componibili tramite un meccanismo spesso detto di *decorazione*
  - i **`Reader`** concreti ospitano un riferimento ad un altro **`Reader`** (*decorato*) di cui adornano le funzionalità
  - Il reader decorato è "iniettato" tramite costruttore nel «decoratore»
- Ad es.

```
Reader br =
  new BufferedReader(
    new FileReader("file.txt")
  );
```



Lettura/Scrittura da/verso Stream

- Il protocollo di utilizzo degli *stream* è scontatamente il seguente
  - ✔ tranne per l'esplicita chiusura finale (necessaria perché richiesto dal protocollo adottato dal S.O. per la gestione delle risorse sottostanti), ricorda l'uso degli iteratori
  - ✔ in definitiva, si sta iterando sugli elementi dello stream

Reading

open a stream
while more information
  read information
close the stream

Writing

open a stream
while more information
  write information
close the stream

Esempio di Utilizzo `java.io.File`

- **`FileReader`** e **`FileWriter`** permettono di gestire stream da/verso file. Il seguente programma copia un file:

```
import java.io.*;
public class FileCopier {
    public static void main(String[] args)  {
        FileReader in = null;
        FileWriter out = null;
        try {
           in = new FileReader(args[0]);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            return;
        }
        try {
            out = new FileWriter(args[1]);
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }
    ...
```

Esempio di Utilizzo `java.io.File`

```
boolean eof = false;
int c;
try {
    while (!eof) {
        c = in.read();
        if (c != -1)
            out.write(c);
        else
            eof = true;
    }
} catch (IOException e) {
    e.printStackTrace();
    return;
}
finally {
    try {
        if (in != null)
            in.close();
        if (out!=null)
          out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
} } }
```

Altro Esempio: Contatore di Parole

```
public Map<String, Integer> countWordOccurrences(String fileName)
                                                    throws IOException {
    Map<String, Integer> word2count = new HashMap<>();
    BufferedReader reader =
        new BufferedReader(new FileReader(fileName));
    String line;
    while ((line = reader.readLine()) != null) {
        Scanner scannerLinea = new Scanner(line);
        while (scannerLinea.hasNext()) {
            String word = scannerLinea.next();
            int count = 0;
            if (word2count.containsKey(word))
                count = word2count.get(word);
            count = count + 1;
            word2count.put(word, count);
        }
    }
    reader.close();
    return word2count;
}
```

java.nio.*

- **java.io** datata; alcuni scelte progettuali discutibili
  - ✓ su tutte: **java.io.File** modella sia il concetto di percorso logico in un file-system che quello di file (come risorsa del S.O. che ospita la JVM)
- La distribuzione standard è stata progressivamente arrichita (da Java 4) del package **java.nio**
  - che in effetti introduce **java.nio.file.Path** per risolvere il problema di modellazione di **java.io.File**
- Alcune operazioni prima macchinose ora sono realizzabili con singoli metodi statici di utilità in **java.nio.file.Files**

Copiare File con `java.nio`

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class FileCopyTest {
    public static void main(String[] args)
                                    throws IOException {
        Path source = Paths.get("src.txt");
        Path destination = Paths.get("copied.txt");
        Files.copy(source, destination);
    }
}
```

Esercizio: Eccezioni e File nello Studio di Caso (1)

- Scrivere i dati necessari a creare un labirinto in un file di testo che viene caricato all'inizio del gioco
- Il file di testo segue una semplice (ma rigida) sintassi:

```
Stanze: biblioteca, N10, N11

Inizio: N10

Vincente: N11

Attrezzi: martello 10 biblioteca, pinza 2 N10

Uscite: biblioteca nord N10, biblioteca sud N11
```

Esercizio: Eccezioni e File nello Studio di Caso (2)

- Aggiungere un costruttore alla classe **`Labirinto`**
  - Inizializza il labirinto caricandolo da un file di testo
  - L'operazione di caricamento dei dati la deleghiamo ad una opportuna classe che incapsula e nasconde tutta la logica di parsing del file di testo
    - Si avvale di **`LabirintoBuilder`** (vedi HW3)
  - Ideare una opportuna logica di gestione delle eventuali anomalie introducendo anche «eccezioni di dominio»

```
public Labirinto(String nomeFile) {
    CaricatoreLabirinto c =
            new CaricatoreLabirinto(nomeFile);
    c.carica();
    this.stanzaIniziale = c.getStanzaIniziale();
    this.stanzaVincente = c.getStanzaVincente();
}
```

Esercizio: Eccezioni e File nello Studio di Caso (3)

- Implementare la logica necessaria per gestire diversi livelli del gioco,
  - Livelli di difficoltà via via crescenti

- Se si perde una partita, bisogna cominciare una nuova partita dal primo livello
- Ciascun labirinto è specificato da un file diverso di nome progressivo siffatto:
  - **`labirinto1.txt`**
  - **`labirinto2.txt`**
  - ...

Sviluppo nel Ciclo di Vita del Software

- Un corso incentrato quasi esclusivamente sullo sviluppo di codice rischia di far dimenticare che nel ciclo di vita di un'applicazione, lo sviluppo, per quanto sia importante, non è poi tutto...
- Bisogna saper distinguere e tener a mente ben separati
  - l'ambiente di sviluppo da quello di produzione in cui l'applicativo verrà eseguito
  - le operazioni che possono svolgere, con riferimento ad una certa applicazione, le diverse figure coinvolte nel suo ciclo di vita
    - sviluppatore
    - amministratore
    - utente finale
- Un corso incentrato esclusivamente sullo sviluppo, dove si usa sempre e soltanto l'ambiente di sviluppo e mai quello di produzione, può indurre molti a confondere ruoli ed ambienti!

Sviluppatore vs Amministratore

- Per questo motivo spesso è opportuno «spostare» informazioni fuori dal codice sorgente in file esterni al codice stesso
- Il primo è accessibile solo allo sviluppatore, il secondo anche agli amministratori
  - ✔ Le informazioni di configurazione devono poter essere cambiate da un amministratore (quindi NON dallo sviluppatore) che conosce i dettagli dell'ambiente di produzione ma non conosce affatto l'ambiente di sviluppo
  - ✔ Viceversa, lo sviluppatore deve evitare di cablare nel codice dipendenze verso l'ambiente di sviluppo che sarà ben diverso dall'ambiente di produzione finale

«Esternalizzazione» delle Costanti

- Sintomi di questi problemi si avvertono già con riferimento ad una problematica molto ben circonstanziata tecnicamente:
  - ✔ l'*esternalizzazione* delle costanti e delle parametri di configurazione
- Abbiamo già osservato come anche solo dichiarare apposite costanti **`final static`** migliora sia la leggibilità che la coesione del codice
  - Esempi di costanti già usate nello studio di caso:
    - il numero di CFU iniziali
    - il peso max della borsa
  - Altre informazioni di configurazione che si possono aggiungere: la specifica dei labirinti, magari con vari livelli di difficoltà<<

«Esternalizzazione» delle Costanti

- L’utilizzo di costanti ci ricorda esigenze delle altre figure coinvolte nel ciclo di vita di un applicativo
  - modificare il valore di una costante richiede la ricompilazione
  - operazione sgradita a tutti tranne che allo sviluppatore
  - ✔ il codice potrebbe anche non essere distribuito!
  - in molti casi non è nemmeno possibile pensare di riavviare un’applicazione

✔ Le costanti cablate nel codice possono essere utilizzate per fissare i valori di default (ovvero i valori da usare nel caso di non reperibilità delle informazioni di configurazione) >>

java.util.Properties

- Esigenze sufficientemente radicate da motivare classi dedicate alla gestione di informazioni persistenti (ad es. di configurazione) nella distribuzione standard sin Java 1.0 (N.B. quindi prima del JCF e **`java.util.Map`**)
- **`java.util.Properties`** è un sottotipo di **`Map<Object,Object>`** che gestisce una collezione di proprietà («property»), ovvero coppie di stringhe nome/valore
  - ✔ N.B. nonostante la tipizzazione, sia nome che valore sono di tipo **`java.lang.String`**
- Offre tre principali tipologie di servizi:
  - Metodi di accesso ai valori delle proprietà per nome
  - Gestione dei valori di default delle proprietà
  - Servizi di lettura/scrittura da file testuali/XML
- ✔ Esistono librerie più moderne come

http://commons.apache.org/proper/commons-configuration/

java.util.Properties: Accesso alle Proprietà

Metodi di accesso alla proprietà

`String getProperty(String key)`

- *Searches for the property with the specified key in this property list. If the key is not found in this property list, the default property list, and its defaults, recursively, are then checked. The method returns null if the property is not found.*

`Object setProperty(String key, String value)`

- *Calls the Hashtable method put. Provided for parallelism with the getProperty method. Enforces use of strings for property keys and values. The value returned is the result of the Hashtable call to put.*

java.util.Properties: Gestione Valori di Default

E' prevista la gestione di valori di default per le proprietà di valore non esplicitamente inizializzato. In particolare si può fornire un altro oggetto **`Properties`** contenente diversi valori di default al costruttore:

`Properties()`

- *Creates an empty property list with no default values.*

`Properties(Properties defaults)`

- *Creates an empty property list with the specified defaults.*

Inoltre il metodo **`getProperty()`** offre una versione sovraccarica:

`String getProperty(String key, String defaultValue)`

- *Searches for the property with the specified key in this property list. If the key is not found in this property list, the default property list, and its defaults, recursively, are then checked. The method returns null if the property is not found.*

java.util.Properties: Lettura/Scrittura da Testo/XML

`void load(InputStream inStream)`

- *reads a property list (key and element pairs) from the input byte stream.*

`void load(Reader reader)`

- *Reads a property list (key and element pairs) from the input character stream in a simple line-oriented format.*

`void loadFromXML(InputStream in)`

- *Loads all of the properties represented by the XML document on the specified input stream into this properties table.*

`void store(OutputStream out, String comments)`

- *writes this property list (key and element pairs) in this Properties table to the output stream in a format suitable for loading into a Properties table using the load(InputStream) method*

`void store(Writer writer, String comments)`

`void storeToXML(OutputStream os, String comment)`

`void storeToXML(OutputStream os, String comment, String encoding)`

java.util.Properties: Formato Testuale/XML (1)

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


public class EsempioProperties {
    public static void main(String[] args)
                        throws IOException {
        Properties prop = new Properties();
        prop.setProperty("cfu_iniziali",   "10");
        prop.setProperty("peso_max_borsa", "20");
        prop.store(new FileWriter("diadia.properties"),
                "Configurazione del gioco DIADIA");
        prop.storeToXML(new FileOutputStream("diadia.xml"),
                "Configurazione del gioco DIADIA");
    }
}
```

java.util.Properties: Formato Testuale/XML (2)

diadia.properties:

```
#Configurazione del gioco DIADIA
#Tue May 17 12:30:11 CEST 2016
peso_max_borsa=20
cfu_iniziali=10
```

vedere i javadoc di **java.util.Properties** per i dettagli sui formati

diadia.xml:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM
  "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Configurazione del gioco DIADIA</comment>
<entry key="peso_max_borsa">20</entry>
<entry key="cfu_iniziali">10</entry>
</properties>
```

Dipendenze da «Risorse»

- «Spostando» le informazioni di configurazione in file esterni al codice, si solleva un nuovo problema...
- Come sviluppare codice dipendente da «risorse» esterne senza che l'accesso dipenda dall'ubicazione del codice stesso?
  - *Risorsa:* qualsiasi contenuto (immagini, testi...) fuori dal sorgente ed in appositi file, ad es. ***diadia.properties***.
- Una NON-soluzione da EVITARE accuratamente:
  - `new FileReader("/home/rossi/diadia/diadia.properties")`
  - cabla nel codice compilato una assunzione sulla ubicazione di una risorsa che con tutta probabilità sarà violata al primo *deployment*
  - ✔Se anche non fosse ancora disponibile l'ambiente di produzione finale, almeno bisognerebbe provare l'esecuzione in ambienti diversi da quello di sviluppo
- ✔In generale è da evitare *sempre* l'utilizzo di percorsi *assoluti* (ovvero che dipendono dall'ambiente di sviluppo)

Nomi Logici delle Risorse

- La piattaforma Java supporta l'assegnazione di *nomi-logici* gerarchici alle risorse proprio per gestire queste problematiche
- Si basa sugli stessi meccanismi&metodi utilizzati per la risoluzione dei nomi delle classi
- Per questi motivi, i servizi necessari sono offerti da due classi
  - **`java.lang.Class`**
  - **`java.lang.ClassLoader`** (responsabile caricamento classi)

  … legate alle gestione delle classi
- *Entrambe* offrono questa coppia di metodi (ma non implementano alcuna interfaccia comune e la semantica differisce! >>)

`URL getResource(`*`String name`*`)`

- *Finds a resource with a given name*

`InputStream getResourceAsStream(`*`String name`*`)`

- *Returns an input stream for reading the specified resource*

Risoluzione Nomi Logici Risorse (1)

- Per la risoluzione dei nomi logici in risorse concrete, valgono le stesse regole per la risoluzione del *nome completamente qualificato* di una classe nella posizione del corrispondente file **`.class`**
  - **`it.uniroma3.dia.comandi.ComandoVai`** è il *nome completamente qualificato* della classe **`ComandoVai`** nel package **`it.uniroma3.dia.comandi`**
  - **`ComandoVai.class`** deve essere ubicata dentro una cartella di nome **`comandi`**, dentro uno cartella di nome **`dia`**, **`...`** , dentro una cartella di nome **`it`**.
  - La cartella di nome **`it`** *deve* essere raggiungibile a partire dal classpath noto alla JVM a tempo di esecuzione

Risoluzione Nomi Logici Risorse (2)

- Dato il nome logico di una risorsa, questa deve essere ubicata in un cartella posizionata seguendo lo schema già seguito per le classi
- Esistono alcune sottili differenze: a differenza dei nomi delle classi, i nomi logici delle risorse usano '**/**' come separatore, e possono essere:
  - *Relativi:* NON cominciano per '**/**'

    ad es. '**diadia.properties**'
  - *Assoluti:* cominciano per '**/**'

    ad es. '**/resources/diadia.properties**'

Class VS Classloader

- **`java.lang.Class`** risolve i percorsi
  - *relativi* a partire dalla posizione del file **`.class`** corrispondente all'oggetto **`Class`** a cui è associato
  - *assoluti* a partire dal classpath
- **`java.lang.ClassLoader`** risolve i percorsi
  - *assoluti*: non sono risolti!
    - ✓ quindi mai usarla con nomi logici che cominciano per '**`/`**'
  - *relativi* sono risolti a partire dal classpath (come fossero assoluti!)

`ClassLoader`: responsabile caricamento classi

✔ Ottenibile invocando il metodo di istanza:

`Class.getClassLoader()` (>>)

Esempi di Risoluzione (1)

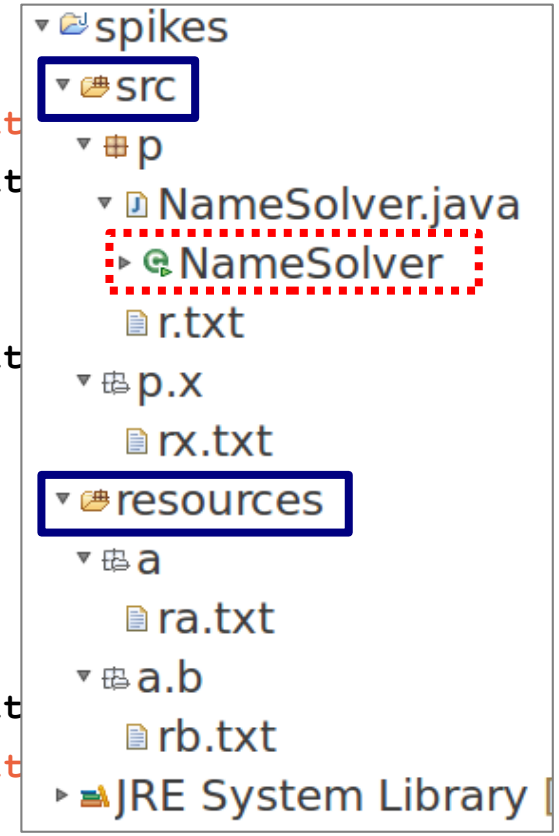
```
package p;
public class NameSolver {
  private static final String RNS_R = "p/r.txt",
  private static final String ABS_R = "/p/r.txt";
  private static final String REL_R = "r.txt";
  …// e molte altre ancora (>>)
  private void testResourceSolver() {
      Class<?> clazz = this.getClass();
      ClassLoader loader = clazz.getClassLoader();
      System.out.println("ClassLoader.getResource(\""+ABS_R+"\")=\t"+
                         loader.getResource(ABS_R));
      System.out.println("     Class.getResource(\""+ABS_R+"\")=\t"+
                         clazz.getResource(ABS_R));
      System.out.println("ClassLoader.getResource(\""+RNS_R+"\")=\t"+
                         loader.getResource(RNS_R));
    …
    … // e molte altre ancora (>>)
  }
  public static void main(String[] args) {
      new NameSolver().testResourceSolver();
  }
}
```

Esempi di Risoluzione (2)

```
ClassLoader.getResource("/p/r.txt")=     null
      Class.getResource("/p/r.txt")=     file:/…/p/r.txt
ClassLoader.getResource("p/r.txt")=      file:/…/p/r.txt
      Class.getResource("p/r.txt")=      null
ClassLoader.getResource("r.txt")=        null
      Class.getResource("r.txt")=        file:/…/p/r.txt
ClassLoader.getResource("/p/x/rx.txt")= null
      Class.getResource("/p/x/rx.txt")=  file:/…/p/x/rx.txt
ClassLoader.getResource("p/x/rx.txt")=   file:/…/p/x/rx.txt
      Class.getResource("p/x/rx.txt")=   null
ClassLoader.getResource("x/rx.txt")=     null
      Class.getResource("x/rx.txt")=     file:/…/p/x/rx.txt
ClassLoader.getResource("/a/ra.txt")=    null
      Class.getResource("/a/ra.txt")=    file:/…/a/ra.txt
ClassLoader.getResource("a/ra.txt")=     file:/…/a/ra.txt
      Class.getResource("a/ra.txt")=     null
ClassLoader.getResource("/a/b/rb.txt")= null
      Class.getResource("/a/b/rb.txt")=  file:/…/a/b/rb.txt
ClassLoader.getResource("a/b/rb.txt")=   file:/…/a/b/rb.txt
      Class.getResource("a/b/rb.txt")=   null
```

Cartella nel classpath

Classe in esecuzione



Risorse Esterne nei Progetti Eclipse

- Per tenere separate le risorse dal codice sorgente è buona prassi creare una cartella apposita per contenerle
  - ✓ spesso chiamata **`resources`**
- E' però importante ricordarsi di includerla nel *classpath*
  - ✓ ovvero, nel *buildpath* di Eclipse

  per permettere la risoluzione a tempo di esecuzione dei nomi logici in risorse concrete
- Nell'esempio precedente la cartella **`resources`** svolge esattamente questo ruolo

Verifica «Preliminare» sulla Mancanza di Dipendenze Verso l'Ambiente di Sviluppo

- Creare un «jar eseguibile» (da Eclipse) contenente tutto il necessario all'esecuzione, salvarlo fuori dall'ambiente di sviluppo, ed eseguirlo:
  1. File → Export... → Runnable JAR file
  2. selezionare una «launch configuration» che specifichi la classe contente il **`main()`** (ad es. **`Gioco`** oppure **`DiaDia`** a seconda del nome della classe che contiene il metodo)
  3. selezionare il nome del file .jar, ad es. **`diadia.jar`**
  4. per eseguire (da console): **`java -jar diadia.jar`**
- *Avvertimento.* Attenzione ad un frequente motivo di molti mal di testa:
  - Il codice eseguibile nel **`.jar`** utilizza il metodo **`URL getResource()`** per ottenere un riferimento alla risorsa a partire dal suo nome logico
  - poi cerca di aprirla come se fosse un qualsiasi file del file-system
  - ➢ tuttavia un file impacchettato e compresso dentro un archivio non è accessibile come fosse un file ordinario!
  - ✔ Soluzione: usare invece **`InputStream getResourceAsStream()`** che permette di ottenere un accesso alla risorsa (come stream)
    - sia nel caso sia «interna» all'archivio **`.jar`**
    - sia nel caso sia presente come file nel file-system

Esercizio (Studio di Caso)

- Rivedere lo studio di caso al fine di rendere configurabile (mediante un file **`diadia.properties`** da distribuire assieme al codice stesso) il gioco senza ricompilarlo. Ad es. almeno:
  - Il numero di CFU iniziali
  - Il peso max della borsa
- N.B. Molte altre informazioni di configurazione potrebbero essere «rimaste nascoste» nel codice sottoforma di costanti letterali
- Rifattorizzare il codice affinché tutte le dipendenze verso le risorse (incluso quelle verso il file contenente la specifica del labirinto) non dipendano dall'ambiente di sviluppo
- Esportare l'applicazione in formato .jar e verificare che continui a funzionare nonostante la dipendenza verso le risorse "esterne"
- Provare quindi esecuzioni con diverse configurazioni (senza ricompilare il codice ma solo cambiando **`diadia.properties`**)