Java

espressioni regolari

G. Prencipe
prencipe@di.unipi.it

Espressioni regolari

- Le librerie per trattare espressioni regolari sono state aggiunte in JDK1.4
- Servono per processare testo
- Permettono di specificare complessi schemi di testo (*pattern*) che possono essere ricercati in una stringa

Espressioni regolari

- Vediamo un piccolo insieme di possibili operatori utili a costruire espressioni regolari
  - B: il carattere B
  - \xhh: carattere con valore esadecimale oxhh
  - \t: tab
  - \n: newline
  - \r: ritorno carrello

Espressioni regolari

- La potenza delle espressioni regolari viene fuori con le classi di caratteri
  - .: rappresenta qualsiasi carattere
  - [abc]: qualsiasi carattere tra a, b e c
  - [^abc]: qualsiasi carattere tranne a, b, e c
  - [a-zA-Z]: qualsiasi carattere da a a z e da A a Z
  - [abc[hij]]: qualsiasi carattere tra a, b, c, h, i, e j
  - [a-z&&[hij]]: o h o i o j (intersezione)
  - \s: il carattere spazio bianco (spazio, tab, newline, ritorno carrello)
  - \S: [^\s]

Espressioni regolari

- \d: una cifra [0-9]
- \D: una non-cifra [^0-9]
- \w: un carattere in una parola [a-zA-Z_0-9]
- \W: [^\w]

- Es.: la parola Rodolfo viene "matchata" dalle seguenti espressioni regolari
  - Rodolfo, [rR]odolfo, [rR][aeiou][a-z]ol.o
- Maggiori dettagli nella documentazione della classe **java.util.regex.Pattern**

Nota

- In Java il carattere **** è utilizzato come carattere di escape
  - Serve a dire che il carattere successivo a **** non deve essere interepretato come semplice carattere, ma ha un significato *speciale*
- Quindi, **** va a sua volta considerato *speciale*
- Cioè, se vogliamo scrivere l'espressione regolare che indica un carattere in una parola, dobbiamo scrivere **\\w**
  - Il primo **** ci dice che il carattere successivo (\) è speciale

Quantificatori

- Un quantificatore descrive il modo con cui un pattern "assorbe" il testo in input
- Può essere
  - Greedy: cerca quanti più match è possibile (del pattern nel testo)
  - Riluttante: si specifica con un ? in fondo all'espressione e cerca il minimo numero di caratteri necessari nel testo per soddisfare il pattern

Quantificatori

Greedy	Riluttanti	Match
X?	X??	X, una o nessuna
X*	X*?	X, zero or più
X+	X+?	X, una o più
X{n}	X{n}?	X, esattamente n volte
X{n,}	X{n,}?	X, almeno n volte
X{n,m}	X{n,m}?	X, almeno n ma non più di m volte

Espressioni regolari

- È consigliabile racchiudere le espressioni X tra parentesi, per essere sicuri di specificarle nel modo giusto
  - abc+ è diverso da (abc)+
- Le classi che gestiscono espressioni regolari in Java sono **Pattern** e **Matcher** in **java.util.regex**

La classe **Pattern**

- Un oggetto **Pattern** rappresenta una espressione regolare
- Il metodo statico **compile()** serve a mettere una espressione regolare in un oggetto **Pattern**

La classe **Matcher**

- Un oggetto **Matcher** è creato invocando il metodo **matcher()** di **Pattern** con argomento una **CharSequence** (interfaccia implementata da **String**)
  - È il motore che effettua il **match** tra il **Pattern** e l'argomento passato
- Il risultato viene acceduto tramite i metodi di **Matcher**

La classe **Matcher**

- Tra i suoi metodi abbiamo
  - **boolean matches()**: **true** se la il **Pattern** matcha interamente la **CharSequence** in input
  - **boolean lookingAt()**: come la precedente, ma il match non deve esistere sull'intera **CharSequence**
  - **boolean find()**: cerca di trovare la successiva sottosequenza della **CharSequence** che matcha il **Pattern**
    - È come un iteratore che si muove in avanti lungo la **CharSequence** alla ricerca di **Pattern**

Esempio

- Un tipico pezzo di codice che utilizza **Pattern** e **Matcher** è

```
Pattern p = Pattern.compile(espressReg);
Matcher m = p.matcher(charSequence);
```

Esercizio

- Scrivere una classe **FindDemo.java** che nel **main** spezzi una frase in parole
  - Usare come **CharSequence** una frase qualsiasi
  - Utilizzare **find()**, l'espressione regolare **\\w+** e **group()**

Gruppi

- I gruppi sono espressioni regolari racchiuse da parentesi tonde che possono essere richiamate utilizzando un numero di gruppo
  - Il Gruppo 0 indica tutta l'espressione, il Gruppo 1 l'espressione racchiusa tra le prime parentesi, ecc.
  - A(B(C))D
    - Gruppo 0=ABCD, Gruppo 1=BC, Gruppo 2=C

Gruppi

- L'oggetto **Matcher** ha metodi per ottenere informazioni sui gruppi
  - **groupCount()** restituisce il numero di gruppi nel **Pattern**. Il Gruppo 0 non è contato
  - **group()** restituisce la sottostringa della **CharSequence** catturata dal Gruppo **0** (l'intero Pattern) durante la precedente operazione di match
  - **group(int i)** restituisce la sottostringa della **CharSequence** catturata dal Gruppo **i** durante la precedente operazione di match

start() e end()

- Dopo un match che ha avuto successo, il metodo di **Matcher start()** restituisce l'indice del primo carattere matchato, e **end()** l'indice dell'ultimo carattere matchato

split()

- Il metodo **split(String regex)** nella classe **String** divide una stringa in accordo a una espressione regolare passata come argomento, e inserisce il risultato in un array di **String**he
- È un modo veloce per spezzare un testo rispetto un delimitatore
- Lo stesso metodo esiste anche in **Pattern**
- In **String** esistono (per semplificare le cose) anche metodi di **matches()**, **replaceFirst()** e **replaceAll()**

StringTokenizer

- Prima di JDK1.4 per dividere una stringa in parti si doveva ricorrere alla classe **StringTokenizer**
  - La **String**a viene spezzata in base a un *token*
  - È più semplice però fare la stessa cosa con le espressioni regolari e il metodo **split()**

Esercizio

- Scrivere **SplitDemo.java** il cui **main** divide una stringa rispetto ai **!**

Metodi replace

- I metodi di replace nella classe **Matcher** servono a rimpiazzare testo con altro testo
  - **replaceFirst(String replace)**: sostituisce la prima occorrenza del **Pattern** con **replace**
  - **replaceAll(String replace)**: sostituisce tutte le occorrenze del **Pattern** con **replace**
  - Metodi con le stesse funzionalità sono anche implementati in **String**

Metodi reset

- Un oggetto **Matcher** già esistente può essere applicato a una nuova **CharSequence** utilizzando i metodi **reset()**
  - **reset()** pone il **Matcher** all'inizio della sequenza corrente
  - **reset(charSeq)** sostituisce la sequenza corrente con **charSeq**

Espressioni regolari e I/O

- Gli esempi visti finora lavorano solo con stringhe costanti
- È utile riuscire a combinare espressioni regolari e file
- Vediamo un possibile impiego di espressioni regolari con file nella classe **File**

La classe **File**

- La classe **File** rappresenta nomi di file e directory
- Dato che i nomi dei file sono dipendenti dalla macchina, questa classe fornisce pathname indipendenti dalla macchina
  - *Abstract pathnames* (APN)
- Gli abstract pathnames hanno
  - Un prefisso che individua il sistema di provenienza del pathname
    - \\ per Windows, / per Unix
  - Una sequenza di 0 o più stringhe

Abstract pathnames

- Ogni nome in un APN rappresenta una directory, tranne l'ultimo
  - L'ultimo può rappresentare una directory o un file
- Un APN vuoto non ha prefisso o sequenza di stringhe
- Il metodo **getName()** in **File** restituisce il nome del file o della directory denotato dall'APN

Il metodo **list()**

- Se **File f** rappresenta una directory, il metodo **list()** restituisce un array di **String**he contenente i nomi dei file e delle directory presenti in **f**
  - Altrimenti restituisce **null**
- È possibile farsi restituire da **list()** anche solo un sottoinsieme dei file nella directory (ad es. tutti i **.java**) utilizzando un *directory filter*
  - Stabilisce come selezionare gli oggetti **File** da visualizzare

Il metodo **list()**

- Il filtro si specifica invocando **list(FilenameFilter filtro)**
- **FilenameFilter** è una interfaccia con un solo metodo

  boolean accept(File dir, String name)
  - Stabilisce se un certo file (**name**) deve essere incluso in una lista di file
- Esso viene utilizzato da **list()** per stabilire di quali file nella directory recuperare il nome
- Quindi, per invocare **list()** con filtro bisogna fornire una implementazione di un **FilenameFilter** che stabilisce come deve essere fatto il filtro

Esempio filtro

```
public class DirList {
  public static void main(String[] args) {
    File path = new File(".");
    String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(new DirFilter(args[0]));
    Arrays.sort(list, new AlphabeticComparator());
    for(int i = 0; i < list.length; i++)
      System.out.println(list[i]);
  }}
```

Esempio filtro

```
class DirFilter implements FilenameFilter {
// Pattern è un contenitore per espressioni regolari
  private Pattern pattern;
  public DirFilter(String regex) {
    pattern = Pattern.compile(regex);
  }
  public boolean accept(File dir, String name) {
    return pattern.matcher(
      new File(name).getName()).matches();
  }} ///:~
```

Filtri

- Questo esempio si presta bene a mostrare l'uso delle classi annidate anonime
- Infatti, per ora l'implementazione di **FilenameFilter** è separata da **DirList**
- Vediamo

Esempio filtro

```
public class DirList2 {
  public static FilenameFilter filter(final String regex) {
    // Creazione di una classe anonima annidata in un metodo
    return new FilenameFilter() {
      private Pattern pattern = Pattern.compile(regex);
      public boolean accept(File dir, String name) {
        return pattern.matcher(
          new File(name).getName()).matches();
      }
    }; // Fine Classe anonima
  }
  public static void main(String[] args) {….}
```

Ma spingiamoci oltre

- Volendo, la costruzione della classe anonima si può addirittura passare direttamente come argomento a **list()**
- Vediamo

Esempio filtro

```
public class DirList3 {
  public static void main(final String[] args) {
    File path = new File("."); String[] list;
    if(args.length == 0)
      list = path.list();
    else
      list = path.list(new FilenameFilter() {
        private Pattern pattern = Pattern.compile(args[0]);
        public boolean accept(File dir, String name) {
          return pattern.matcher(
            new File(name).getName()).matches();}});….
```

Quindi….

- Le classi anonime possono essere utilizzate quando ci serve creare una classe per un utilizzo locale
- Rendono comunque il codice meno leggibile
- Quindi vanno usate con discrezione

Controllare e creare directory

- Un oggetto **File** può essere utilizzato anche per creare nuove directory o un intero path a una directory
- È possibile anche controllare le caratteristiche dei file (lettura/scrittura), controllare se un **File** rappresenta un file o una directory, e cancellare un **File**

Controllare e creare directory

- **isDirectory()**, **isFile()**, **getAbsolutePath()**, **canRead()**, **canWrite()**, **getName()**, **getParent()**, **length()**, **lastModified()**, **delete()**, **exists()**, **mkdirs()**, **renameTo()**
- Controllare la documentazione per informazioni più dettagliate

Esercizi

1. Scrivere **JGrep.java** che realizza il comando *grep*
   1. Dati come argomenti al programma un nome di file e una espressione regolare, si vogliono stampare le linee (e i relativi numeri di riga) nel file dove ci sono occorrenze della espressione regolare
   2. Si usino **FileReader** e **BufferedReader** per gestire il file

Java

espressioni regolari

fine