

## Sistemi per l'Elaborazione dell'Informazione 2

### Strategie, osservatori e osservabili

#### Un tuffo nel passato: interi in ordine lessicografico

Per scaldarci cominciamo con un esercizio sulle collezioni. Un `SortedSet` mantiene gli elementi contenuti nell'ordine naturale (cioè quello implementato dall'interfaccia `Comparable`). È però possibile costruire un `SortedSet` specificando al tempo di costruzione un `Comparator` (definito in `java.util`), cioè una strategia di confronto.

Dovete creare un `SortedSet` che ordina lessicograficamente gli interi, cioè come stringhe, e non seguendo il valore numerico. Per farlo, dovete solo creare una classe che implementa `Comparator`, assume che gli argomenti passati al suo metodo `compare()` siano `Integer`, e restituisca il risultato corretto. Notate che tale valore è facilmente ottenibile convertendo gli interi in stringhe e utilizzando il metodo `compareTo()` di `String`.

Create a questo punto una classe di prova in cui aggiungete all'insieme alcuni interi, e provate a stampare l'insieme stesso, magari utilizzando un iteratore. Provate a variare il `Comparator`: per esempio ordinando i numeri in prima istanza per lunghezza, e ordinando numeri con la stessa lunghezza lessicograficamente.

#### Visita di una stringa

Come esempio di visita, creiamo una classe `StringUtils` che contiene metodi statici di utilità per stringhe. Questa classe conterrà un metodo statico `applyTransform()` che accetta come argomenti una stringa e un'istanza di `CharTransform`, un'interfaccia specificata da voi. L'interfaccia contiene un solo metodo `transform()` che trasforma un carattere, passato come argomento, in un altro carattere. Il metodo `applyTransform()` applica a sua volta il metodo `transform()` della `CharTransform` che gli viene passata a tutti i caratteri della stringa e restituisce la stringa così ottenuta.

Provate a scrivere implementazioni di `CharTransform` che rendono i caratteri maiuscoli, minuscoli, e che implementano la crittografia a rotazione di Cesare (trasformano un carattere di codice ASCII  $x$  in un carattere di codice  $(x+k) \bmod 65536$ , dove  $k$  è un intero fornito all'istanza di `CharTransform` al tempo di costruzione; attenti ai moduli negativi). Ancora, potete realizzare un'implementazione di `CharTransform` che rende i caratteri rispettivamente maiuscoli e minuscoli con probabilità  $p \in [0, 1]$  e  $1 - p$ , dove anche in questo caso  $p$  viene specificato al tempo di costruzione. Notate che solo le prime due implementazioni si prestano a diventare singoletti. Applicate due trasformate crittografiche inverse di seguito (cioè con  $k$  e  $-k$ ) e controllate che il risultato sia la stringa originale. Per fornire input di prova al programma, potete leggere stringhe da console adattando `System.in` con un `InputStreamReader` e decorando il `Reader` risultante con un `BufferedReader`. Alternativamente, potete leggere un file di testo decorando, come al solito, un `FileReader`.

#### Visite osservate

Nell'ipotesi che `applyTransform()` venga usato su stringhe di grandi dimensioni, è utile controllarne il funzionamento passo passo. Per fare questo, faremo un uso molto semplificato di `Observer`: non ci

sarà in effetti nessun `Observable` istanziato effettivamente, e l'`Observer` assumerà che a ogni chiamata l'`Observable` (cioè il processo di visita della stringa) sia effettivamente cambiato. Faremo inoltre riferimento alle versioni di `Observable` e `Observer` contenute nel package `java.util`.

Aggiungiamo quindi una versione polimorfa di `applyTransform` che prende, oltre alla stringa e alla trasformazione, un `Observer`. A ogni iterazione del ciclo, notificiamo l'osservatore del fatto che un nuovo carattere è stato elaborato. Lo facciamo invocando il suo metodo `update()`, passando `null` come `Observable` (dato che è sottinteso) e la frazione di stringa elaborata come secondo argomento (il prototipo di `update()` in `Observer` richiede di passare un'istanza della classe `Object`, ma è possibile utilizzare l'autoboxing e specificare un valore `double`, da trattare opportunamente nel corpo di `update()`).

Provate ora a scrivere diversi `Observer`. Prima di tutto, un `Observer` che stampa un punto ogni volta che viene richiamato, aggiungendo una parentesi quadra aperta la prima volta che viene chiamato e una parentesi quadra chiusa l'ultima volta che viene chiamato (cosa che sa quando l'argomento passato è esattamente 1). Provate a trasformare un testo di una lunghezza significativa, e guardate che cosa succede. Un altro `Observer` possibile stampa la percentuale di stringa elaborata ogni volta che viene elaborato il 10% del carico (dovete stampare, per ogni  $k$ , il primo valore passato all'`Observer` che supera  $k/10$  in forma di percentuale). Se l'output non vi soddisfa, chiedetevi se possono essere intervenuti errori di approssimazione e rimettete le cose a posto usando la classe `java.text.DecimalFormat`.

## Osservazione grafica

Proviamo ora a dare una veste grafica alla nostra applicazione. Realizzate una `Shell` che contiene una casella di testo in cui inserire il pathname di un file da trasformare (magari corredata da un pulsante che permette di riempire automaticamente la casella sfruttando un `FileDialog`), due combo che permettono di selezionare rispettivamente una `CharTransform` e un `Observer`, nonché un pulsante che una volta premuto legge il file, lo trasforma osservandolo in funzione delle selezioni nei `Combo` e visualizza il risultato in un'ulteriore casella di testo.

Già che ci siamo possiamo anche visualizzare graficamente l'andamento di una trasformazione. Per farlo, utilizziamo un `ProgressMonitor` dal package `javax.swing`. Un `ProgressMonitor` è un'interfaccia grafica molto semplice che visualizza l'andamento di una funzione che richiede tempo. Al momento della creazione viene richiesto un valore minimo e un valore massimo. In seguito, il metodo `setProgress()` viene utilizzato per dire a che punto dell'elaborazione (tra minimo e massimo) ci si trova.

Un `ProgressMonitor` non è un `Observer`, e quindi ci occorre un adattatore che implementi `Observer` e deleghi il metodo `update()` al metodo `setProgress()` di un `ProgressMonitor`. Dato che per noi un `Observer` riceve un `Double` tra 0 e 1, dovremo scalarne i valori. Il costruttore dell'adattatore deve creare il `ProgressMonitor` come segue:

```
progressMonitor =  
    new ProgressMonitor(null, "messaggio principale", "messaggio secondario", 0, 100);
```

In questo modo, il `ProgressMonitor` sa di doversi aspettare valori tra 0 e 100. Provate ora a sostituire il vostro osservatore testuale con quello grafico. Nota bene: un `ProgressMonitor` appare solo dopo che è passato qualche istante dal primo `setProgress()`. Per vederlo funzionare dovete trasformare un file ragionevolmente grande, o ritardare artificialmente il funzionamento di `applyTransform()` aggiungendo `Thread.sleep(n)` nel ciclo, che provoca una pausa di (approssimativamente)  $n$  millisecondi nell'elaborazione (notate che ciò richiede di gestire l'eccezione `InterruptedException`). Provate anche a migliorare l'interfaccia del `ProgressMonitor`, visualizzando di volta in volta nel messaggio secondario la percentuale di elaborazione portata a termine. Sulla pagina del corso trovate il collegamento a un file di grandi dimensioni, da utilizzare per verificare il corretto funzionamento delle classi.

## **Parte facoltativa**

Utilizzando UML scrivete i diagrammi delle classi e i diagrammi di sequenza per tutti gli esercizi risolti. Scrivete la documentazione JavaDoc per tutte le classi e i metodi coinvolti, evidenziando gli invarianti. Infine, scrivete dei test di funzionamento per tutte le classi coinvolte.