

Sistemi per l'Elaborazione dell'Informazione 2

Adattatori e decoratori

Lo scopo di questa esercitazione è la creazione di diversi adattatori e decoratori basati sulla gerarchia di classi per l'input/output, contenuta nel package `java.io`.

Come sapete, la gerarchia di input/output è basata su due gruppi di classi: le sottoclassi di `InputStream` e `OutputStream` servono a leggere e scrivere *byte*, mentre le sottoclassi di `Reader` e `Writer` servono a leggere e scrivere *caratteri*. Come i caratteri vengano codificati e decodificati in byte è deciso da una classe ponte, `InputStreamReader` (che in mancanza di informazioni utilizza la codifica d'ufficio della piattaforma). Per esempio, un `FileInputStream` legge informazioni da un file, mentre un `FileWriter` scrive caratteri su un file. Vari decoratori forniscono servizi aggiuntivi (memorie tampone, lettura a righe, ecc.).

Gli attributi statici `in`, `out` ed `err` della classe `System` forniscono un `InputStream` e due `PrintStream` associati alla tastiera e al terminale, in maniera simile alle variabili globali `stdin`, `stdout` e `stderr` del C.

IMPORTANTE: quando scrivete classi per delega *non c'è bisogno di scrivere i metodi a mano*. Basta selezionare l'attributo a cui sono destinate le deleghe e utilizzare la voce opportuna nel sottomenu contestuale "Source" di Eclipse.

Classe di prova

Scrivete una classe con il solo `main()` che legge un file di testo da tastiera e lo stampa, una riga alla volta, numerando le righe. Utilizzate `LineNumberReader` per decorare un `InputStreamReader` che adatta `System.in`. Provate a eseguirlo da Eclipse. Provate poi a modificare la classe in modo che utilizzi un `FileReader` che legge un file specificato sulla riga di comando.

Maiuscolizzatore

Il primo decoratore serve ad aggiungere a un `Reader` la funzionalità seguente: tutti i caratteri alfabetici letti devono diventare maiuscoli. Dovete fare l'implementazione tramite delega: dichiarate innanzitutto che la vostra classe estende `Reader`, e utilizzate l'opportuna voce del menu di Eclipse per generare i mozziconi dei metodi da implementare; riempiteli quindi con le deleghe e le elaborazioni opportune. Provate ora a inserire il vostro decoratore (in che punto?) nella classe di prova e controllate che il risultato sia quello desiderato. Provate in altri punti e spiegatevi il risultato. I più arditi possono anche provare a realizzare decoratori più complessi, che ad esempio convertono tutti i caratteri in maiuscolo a eccezione di quelli all'inizio di ogni parola, oppure che riconoscono opportune codifiche nell'input, sostituendo ad esempio ogni coppia di caratteri formata da un backquote e da una vocale nella corrispondente vocale con l'accento grave, e lavorando in modo simile per gli accenti acuti, codificati tramite un quote (cioè la coppia 'a diventerà â, cosiccome la coppia 'e diventerà é.

Filtro per caratteri stampabili

A volte è utile cercare di vedere "che cosa c'è" in un file binario, stampando tutto ciò che è stampabile in ASCII. Scrivete un adattatore (da `InputStream` a `Reader`) che prende in considerazione solo i caratteri

ASCII stampabili (codice tra 32 e 126, inclusi). Provate a inserirlo nella classe di prova (in che punto?) e lanciatelo su qualche file binario.

Come un iteratore

Scrivete tre adattatori (sempre per composizione) che espongono, rispettivamente, un `InputStream`, un `Reader` e un `BufferedReader` come iteratori. Gli iteratori devono restituire byte, caratteri o stringhe fino alla fine del file.

Per pulizia, i tre adattatori dovrebbero essere *classi annidate* di una *fabbrica concreta*. Dovete cioè definire una classe `IOIteratorAdapters` che contiene tre metodi di fabbricazione statici (per esempio, `asIterator()`) che accettano, rispettivamente, un `Reader`, un `BufferedReader` e un `InputStream`. All'interno della classe, definite classi private statiche annidate (`private static class ...`) che vengono restituite dai costruttori. In questo modo la specifica implementazione degli adattatori è completamente nascosta all'utente.

Suggerimento: la difficoltà dell'esercizio sta nel fatto che per poter determinare se un file è terminato bisogna necessariamente tentare di effettuare un'operazione di lettura, ritrovandosi eventualmente con un carattere che non può essere "rimandato indietro". L'uso dei metodi `mark()` e `reset()` vi aiuterà notevolmente a risolvere questo problema quando lavorerete con `BufferedReader`, mentre per le altre due classi dovrete cavarvela da soli.

Modificate ora la vostra classe di prova in maniera che l'utente possa specificare tramite un'opzione se vuole stampare i byte, i caratteri o le righe di un file. Isolate il ciclo di stampa in maniera che sia uno solo *indipendentemente dal tipo di stampa richiesto* (utilizzando gli iteratori di cui sopra).

CachedIterator

Un `Iterator` non ha modo di essere riletto più volte. A volte è utile però poter reimpostare un iteratore in modo che possa emettere di nuovo il suo output. Per questo occorre una memoria tampone che memorizzi i risultati emessi dall'iteratore.

Invece di inserire una logica del genere in ogni iteratore, conviene creare un decoratore che aggiunge un metodo `reset()`, e la responsabilità di restituire, a fronte di ogni chiamata a `reset()`, l'output emesso dall'iteratore.

Una prima implementazione banale, non pigra, svuota l'iteratore passato al costruttore e lo memorizza in un vettore (o, meglio, in una `java.util.ArrayList`, visto che non sapete a priori la lunghezza). La classe mantiene una variabile `pos` che dà la posizione corrente nella lista.

I più arditi possono cimentarsi con un'implementazione molto più interessante, che memorizza in maniera pigra i risultati dell'iteratore soggiacente solo se richiesti dall'utente. Al momento della creazione non verrà dunque istanziato nulla. La logica è però decisamente più complessa perché per sapere se è possibile emettere ancora elementi è necessario controllare non solo se la memoria tampone è esaurita, ma anche se l'iteratore soggiacente è esaurito.

In entrambi i casi, è fondamentale scrivere un `main()` di prova che controlli che i risultati dell'iteratore siano coerenti da un `reset()` all'altro.

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.