

Filesystem

Sezioni

- [La traccia](#)
- [Soluzione](#)

La traccia

Scopo della prova è progettare e implementare una gerarchia di oggetti utili a realizzare una *shell*, ossia un semplice interprete di comandi, per un *filesystem* elementare. Come è ben noto, il filesystem è una delle componenti fondamentali del sistema operativo e il suo compito è gestire la memorizzazione delle informazioni (nelle memorie di massa).

Il filesystem

Assuma per semplicità che nel **filesystem** esistano solo due tipi di *entry*: i *file* e le *directory*, le caratteristiche principali di questi elementi sono:

- ogni **entry** ha un *nome* (dato da una stringa non vuota),
- i **file** hanno anche una *dimensione* (data da un intero positivo),
- le **directory** possono *contenere* altre entry (ciascuna entry può essere contenuta in una sola directory), la *dimensione* di una directory è la somma delle dimensioni delle entry che contiene;
- la *radice* di un filesystem è una directory che contiene (anche indirettamente) tutte le entry del filesystem.

Queste definizioni implicano, tra l'altro, che il filesystem può essere descritto da un *albero orientato e radicato* i cui nodi sono le entry, la radice coincide con quella del filesystem ed esiste un arco orientato tra due nodi se e solo se il primo contiene il secondo (ragione per cui i nodi interni possono essere solo directory, mentre le foglie possono essere sia file che directory vuote).

Un **path** è un elenco ordinato di nomi di entry in cui solo l'ultima delle quali può essere un file. Ogni entry di un filesystem è identificata da un path, detto *assoluto*, corrispondente all'elenco dei nomi delle entry lungo il percorso che conduce dalla radice alla entry nell'albero sopra descritto; i path che sono suffissi di un path assoluto sono detti *relativi* (e corrispondono ai path delle directory che si trovano lungo il percorso che va dalla radice alla entry identificata dal path assoluto).

Le primitive e i comandi

Il filesystem mette a disposizione una serie di *primitive* che consentono di:

- reperire una entry dato il suo path assoluto,
- creare una directory dato il suo path assoluto,
- creare un file dato il suo path assoluto e una dimensione,
- elencare il contenuto di una directory dato il suo path assoluto,
- ottenere la dimensione di una entry dato il suo path assoluto.

Una **shell** per il filesystem è un interprete in grado di ricevere dei *comandi* (con eventuali *parametri*) in forma testuale e di invocare, secondo necessità, le opportune primitive del filesystem per eseguirli. La shell legge il flusso di ingresso una linea alla volta, la prima stringa di ogni linea è un comando, le restanti parti della linea sono i parametri. Qualora il comando (o la sua esecuzione) causi degli errori, la shell emette opportune informazioni — ma non interrompe mai la sua esecuzione (e non solleva eccezioni).

Per comodità la shell conserva un riferimento a una tra le directory del filesystem che viene chiamata la *directory corrente* dell'interprete; essa è inizialmente la radice del filesystem e può essere modificata e visualizzata con degli appositi comandi. Ogni comando che accetta un percorso come parametro può accettare un percorso assoluto, oppure un percorso relativo alla directory corrente, nel senso che essa dovrà

essere usata come prefisso per rendere assoluto il percorso indicato dal parametro prima di adoperarlo in una delle primitive del filesystem.

I comandi (e relativi parametri) dell'interprete sono:

- `ls [path]` che elenca il contenuto della directory indicata dal path, o della directory corrente,
- `size [path]` che riporta la dimensione della entry indicata dal path, o della directory corrente,
- `mkdir path` che crea la directory indicata dal path,
- `mkfile path size` che crea il file indicato dal path con la dimensione specificata,
- `cd [path]` che modifica la directory corrente in quella indicata dal path, o nella radice del filesystem,
- `pwd` che stampa il nome della directory corrente;

il parametro `path` (che può essere assente se riportato tra quadre, nel qual caso il comando si riferisce alla directory corrente) è un percorso (assoluto o relativo), mentre `size` è un intero positivo.

La rappresentazione testuale

L'interprete rappresenta i path concatenandone i nomi separati tra loro tramite il carattere `:`, indicando tale carattere anche all'inizio nel caso il path sia assoluto. Sono per esempio due path, il primo dei quali assoluto:

```
:quattro:sette
quattro:sette:otto:nove
```

Negli elenchi di entry, per distinguere i file dalle directory, al nome dei primi viene fatta seguire la dimensione (racchiusa tra tonde) e a quelli delle directory un asterisco. Nell'esempio seguente

```
uno*
tre(50)
quattro*
```

sono directory la prima e terza entry, mentre è un file (di dimensione 50), il secondo. Infine, per consentire di distinguere meglio i comandi impartiti (nel flusso di ingresso) dall'output prodotto (nel flusso d'uscita), l'interprete prefissa ogni riga di output con `>>>`.

Quella riportata di seguito è una sessione di interazione con l'interprete

```
ls
mkdir uno
ls
>>> uno*
mkfile due 10
mkfile tre 20
ls
>>> uno*
>>> due(10)
>>> tre(20)
size
>>> 30
cd uno
ls
pwd
>>> :uno
mkdir quattro
mkfile quattro:cinque 30
size
>>> 30
cd
size
>>> 60
```

dove, secondo quanto illustrato, le righe senza prefisso corrispondono ai comandi impartiti e quelle che iniziano con `>>>` sono l'output prodotto dell'interprete; al termine dell'esecuzione una rappresentazione testuale dell'albero del filesystem è

```
├─ uno*
│   └─ quattro*
│       └─ cinque(30)
├─ due(10)
└─ tre(20)
```

(osservi che la realizzazione del comando per ottenere tale rappresentazione non è richiesta, ma potrebbe essere un'interessante estensione facoltativa).

Cosa è necessario implementare

Al termine della realizzazione della gerarchia di classi che rappresentano le entità in gioco, dovrà scrivere una classe di test che legga dal flusso di ingresso i comandi per l'interprete ed emetta il risultato della loro esecuzione nel flusso d'uscita. Ad esempio, se il flusso di ingresso contiene

```
ls
mkdir uno
ls
mkfile due 10
mkfile tre 20
ls
size
cd uno
ls
pwd
mkdir quattro
mkfile quattro:cinque 30
size
cd
size
```

il programma emette

```
>>> uno*
>>> uno*
>>> due(10)
>>> tre(20)
>>> 30
>>> :uno
>>> 30
>>> 60
```

nel flusso d'uscita — osservi che il comando `ls` impartito su directory vuote non produce output.

Soluzione

Le entry: file e directory

Dal momento che il *nome* è specificato come attributo comune di *file* e *directory* può convenire implementare le **entry** con una classe astratta, lasciando alle sottoclassi concrete l'onere di implementare il calcolo della *size*. La rappresentazione è pertanto limitata ad una stringa

```
public final String name;

protected Entry(final String name) {
    if (Objects.requireNonNull(name).isEmpty())
        throw new IllegalArgumentException("Il nome non può essere vuoto.");
    this.name = name;
}

\[sorgente\]
```

il cui invariante è non essere `null` e non essere vuota fatto che può essere controllato in costruzione e non dovrà più essere verificato (dato che, riguardo a questa porzione di informazioni, la classe è immutabile).

Nell'interprete è necessario sapere se una entry corrisponde o meno ad una directory (per segnalare, ad esempio, se sia o meno un errore chiederne l'elencazione del contenuto, o di aggiungervi un'altra entry); per questa ragione aggiungiamo all'*entry* anche un metodo (astratto) `isDir`. La classe astratta avrà pertanto solo i seguenti metodi astratti:

```
public abstract boolean isDir();

public abstract int size();
```

[\[sorgente\]](#)

Per rendere concreta tale classe, la classe che implementa il **file** deve solo memorizzarne la dimensione

```
private final int size;

public File(final String name, final int size) {
    super(name);
    if (size <= 0) throw new IllegalArgumentException("La dimensione deve essere
positiva.");
    this.size = size;
}
```

[\[sorgente\]](#)

Si osserva che, sebbene l'attributo sia immutabile, non si avrebbe alcun vantaggio a renderlo pubblico dal momento che esiste un (omonimo) metodo osservazionale; d'altro canto non è possibile rimuovere tale metodo, dato che per il caso delle *directory* non è plausibile (come sarà discusso in seguito) memorizzare la dimensione in un attributo. Gli unici metodi da implementare sono sovrascritture di metodi delle superclassi:

```
@Override
public int size() {
    return size;
}

@Override
public boolean isDir() {
    return false;
}

@Override
public String toString() {
    return String.format("%s(%d)", name, size);
}
```

[\[sorgente\]](#)

Un po' più complicato è il discorso della **directory** che, oltre ad implementare i metodi astratti della superclasse astratta, deve consentire quantomeno di:

- aggiungere una entry,
- reperire una entry in esso contenuta, a partire dal suo nome,
- elencare le entry che contiene.

La rappresentazione più ragionevole appare essere data da una *lista* di *entry*. Oltre alle ovvie condizioni dell'invariante (che prescrivano che la lista non sia `null` e non contenga `null`) è necessario prestare attenzione che essa non contenga più *entry* con lo stesso nome, questo è banalmente vero in costruzione e verrà verificato nell'unico metodo mutazionale.

```
private final List<Entry> entries = new ArrayList<>();

public Directory(final String name) {
    super(name);
}
```

[\[sorgente\]](#)

I metodi più banali da implementare sono quelli che sovrascrivono metodi ereditati dalle superclassi. Il calcolo della dimensione avviene per scarico ricorsivo; si osserva che non è possibile memorizzare il valore in un attributo: sebbene il valore di tale attributo potrebbe essere aggiornato all'aggiunta di un *file*, non potrebbe viceversa esserlo qualora venisse aggiunto un file ad una *directory* contenuta nella presente.

```

@Override
public boolean isDir() {
    return true;
}

@Override
public int size() {
    int sum = 0;
    for (Entry e : entries) sum += e.size();
    return sum;
}

@Override
public String toString() {
    return String.format("%s*", name);
}

@Override
public Iterator<Entry> iterator() {
    return Collections.unmodifiableList(entries).iterator();
}

```

[\[sorgente\]](#)

L'ultimo metodo serve per far sì che questa classe implementi `Iterable<Entry>`, così che possa offrire un modo comodo di operare sul contenuto della *directory*. Si noti l'uso del metodo statico `unmodifiableList` di `Collections` che serve ad avvolgere l'elenco di entry della directory rendendolo una lista non modificabile (in modo da non esporre la rappresentazione consentendo che venga modificata dall'esterno).

```

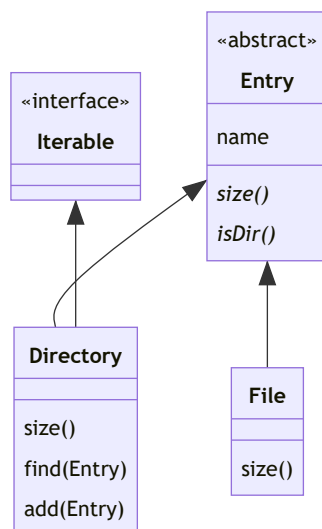
public Entry find(final String name) {
    Objects.requireNonNull(name);
    for (Entry e : entries) if (e.name.equals(name)) return e;
    return null;
}

void add(final Entry e) throws FileAlreadyExistsException {
    Objects.requireNonNull(e);
    if (find(e.name) != null)
        throw new FileAlreadyExistsException(
            "La directory contiene già una entry con lo stesso nome.");
    entries.add(e);
}

```

[\[sorgente\]](#)

Per concludere, la gerarchia realizzata fin qui è:



I path

Un **path** è una sequenza di stringhe non vuote che non contengono il *separatore* `:`, può essere *assoluto* (nel qual caso la sua rappresentazione testuale inizia col separatore), oppure *relativo*; non è necessaria alcuna altra restrizione, in particolare non è necessario che esista una relazione a priori tra un *path* e le *entry*: sarà compito del *filesystem* (a partire dal suo contenuto), dire se un *path* corrisponde ad una delle sue

entry, oppure no.

Per il momento, quindi, la rappresentazione più semplice è quella di conservare in una *lista* di stringhe che chiameremo *parts* e di un booleano che permetta di distinguere il caso *assoluto* da quello *relativo*. Esternamente a questa classe i *path* saranno costruiti a partire da stringhe, ma (come sarà chiaro in seguito) può essere utile avere un costruttore privato che accetti una lista di stringhe (e controlli l'invariante):

```
private final boolean isAbsolute;

private final List<String> parts;

private Path(final boolean isAbsolute, final List<String> parts) {
    this.isAbsolute = isAbsolute;
    this.parts = List.copyOf(parts);
    for (String p : parts) {
        if (p.isEmpty()) throw new InvalidPathException(p, "La componente è vuota.");
        if (p.indexOf(SEPARATOR) != -1)
            throw new InvalidPathException(p, "La componente contiene il separatore.");
    }
}
```

[\[sorgente\]](#)

Riguardo all'invariante, è necessario che:

- la lista *parts* non deve essere o contenere *null* (questo è verificato in costruzione grazie al metodo statico *copyOf* di *List*, che per giunta restituisce una lista immutabile che facilita il ragionamento sull'immutabilità della classe);
- nessuna stringa sia *null* o vuota.

Prima di procedere può essere utile aggiungere qualche costante, per il separatore, il percorso corrispondente alla radice del *filesystem* e il percorso *vouto*:

```
public static final String SEPARATOR = ":";

public static final Path ROOT = new Path(true, Collections.emptyList());

public static final Path EMPTY = new Path(false, Collections.emptyList());
```

[\[sorgente\]](#)

Al costruttore privato è quindi abbinato un metodo di fabbricazione che costruisce un *path* a partire da una stringa; si osservi che non sarebbe possibile ottenere lo stesso risultato con un costruttore che invochi quello privato (perché l'invocazione dovrebbe avvenire come prima istruzione, fatto che impedirebbe di gestire gli altri casi particolari).

```
public static Path fromString(final String path) {
    Objects.requireNonNull(path);
    if (path.isEmpty()) return EMPTY;
    final String[] parts = path.split(SEPARATOR);
    if (parts.length == 0) return ROOT;
    if (parts[0].isEmpty()) return new Path(true, Arrays.asList(parts).subList(1,
        parts.length));
    return new Path(false, Arrays.asList(parts));
}
```

[\[sorgente\]](#)

Le prime operazioni che è utile implementare sono quelle per suddividere un percorso nel suo prefisso (a meno dell'ultima componente) e nell'ultima componente (se presente); tali operazioni saranno utili ad esempio alla *shell* per decidere, dato il *path* di un *file* da creare, il nome della *directory* dove crearlo (prefisso) e del *file* (ultima componente). Data la scelta della rappresentazione, tali operazioni hanno una implementazione elementare:

```

    public boolean isAbsolute() {
        return isAbsolute;
    }

    public Path parent() {
        if (parts.isEmpty()) return this;
        return new Path(isAbsolute, parts.subList(0, parts.size() - 1));
    }

    public String name() {
        if (parts.isEmpty()) return null;
        return parts.get(parts.size() - 1);
    }
}

```

[\[sorgente\]](#)

Due operazioni che possono risultare comode riguardano «concatenazione» e «spezzamento» tra due percorsi; più formalmente, possono essere utili:

- la *risoluzione* di un percorso relativo rispetto ad uno assoluto (che corrisponde in sostanza a concatenare i due percorsi),
- la *relativizzazione* di un percorso rispetto ad un suo prefisso (che è sostanzialmente l'operazione inversa rispetto alla precedente);

La prima operazione, in particolare, sarà necessaria alla *shell* per conoscere il percorso assoluto su cui operare qualora tra gli argomenti di un comando sia presente un percorso relativo, che andrà risolto (come vedremo) rispetto al percorso assoluto della directory corrente.

L'implementazione di tali operazioni richiede un minimo di attenzione ai casi particolari, ma è ragionevolmente semplice:

```

    public Path resolve(final Path other) {
        if (Objects.requireNonNull(other).isAbsolute()) return other;
        final List<String> parts = new ArrayList<>(this.parts);
        parts.addAll(other.parts);
        return new Path(isAbsolute, parts);
    }

    public Path relativize(final Path other) {
        Objects.requireNonNull(other);
        if (!isAbsolute() && other.isAbsolute)
            throw new IllegalArgumentException(
                "Non si può relativizzare un path assoluto rispetto ad un relativo.");
        if (!parts.equals(other.parts.subList(0, parts.size())))
            throw new IllegalArgumentException("Il percorso non ha un prefisso in comune con questo.");
        return new Path(false, other.parts.subList(parts.size(), other.parts.size()));
    }
}

```

[\[sorgente\]](#)

Per finire, può essere comodo che il `Path` implementi un `Iterable<String>` sulle sue parti; i metodi sovrascritti pertanto sono:

```

@Override
public String toString() {
    return (isAbsolute ? SEPARATOR : "") + String.join(SEPARATOR, parts);
}

@Override
public Iterator<String> iterator() {
    return Collections.unmodifiableList(parts).iterator();
}

```

[\[sorgente\]](#)

Il filesystem

La rappresentazione del **filesystem** è del tutto elementare in quanto coincide con la sua *directory* radice, a cui verranno man mano aggiunti *file* e *directory* (a cui, a loro volta, potranno essere aggiunti *file* e *directory*).

```

private final Directory root = new Directory("ROOT");

```

[\[sorgente\]](#)

Non c'è in realtà bisogno d'altro invariante che di assicurare che `root` non sia `null` (che è vero per via dell'inizializzazione e non potrà mai cessare d'esserlo dal momento che il campo è `final`).

Le competenze del **filesystem** sono elencate direttamente nella traccia

- reperire una entry dato il suo path assoluto,
- ottenere la dimensione di una entry dato il suo path assoluto.
- elencare il contenuto di una directory dato il suo path assoluto,
- creare una directory dato il suo path assoluto,
- creare un file dato il suo path assoluto e una dimensione,

La prima è la più complessa da implementare: richiede che, a partire dalla radice, le componenti del path siano usate in sequenza per determinare (lungo il prefisso) le *directory* coinvolte e alla fine l'*entry*; può essere comodo avere una versione del metodo di ricerca che sollevi una eccezione nel caso l'ultima *entry* non sia essa stessa una *directory* (questo sarà utile per tutti i casi in cui si intende assicurarsi che tale sia il caso):

```
public Entry find(final Path path) throws FileNotFoundException {
    if (!Objects.requireNonNull(path, "Il path non può essere null").isAbsolute())
        throw new IllegalArgumentException("Il path deve essere assoluto.");
    Directory d = root;
    Entry e = d;
    for (final String p : path) {
        if (d == null)
            throw new FileNotFoundException("La parte prima di " + p + " non è una directory");
        e = d.find(p);
        if (e == null) throw new FileNotFoundException("Impossibile trovare l'entry " + p);
        d = e.isDir() ? (Directory) e : null;
    }
    return e;
}

public Directory findDir(final Path path) throws FileNotFoundException {
    if (!Objects.requireNonNull(path, "Il path non può essere null").isAbsolute())
        throw new IllegalArgumentException("Il path deve essere assoluto.");
    final Entry e = find(Objects.requireNonNull(path, "Il path non può essere null"));
    if (!e.isDir()) throw new FileNotFoundException("Non è una directory " + path);
    return (Directory) e;
}
```

[\[sorgente\]](#)

Le competenze «osservazionali» (restituire la dimensione di una *entry* o un iteratore sul contenuto di una *direcotry*) sono di facile implementazione, una volta dati i due metodi di ricerca appena sviluppati basterà delegare alle *entry*:

```
public Iterable<Entry> ls(final Path path) throws FileNotFoundException {
    if (!Objects.requireNonNull(path, "Il path non può essere null.").isAbsolute())
        throw new IllegalArgumentException("Il path deve essere assoluto.");
    return findDir(path);
}

public int size(final Path path) throws FileNotFoundException {
    if (!Objects.requireNonNull(path, "Il path non può essere null.").isAbsolute())
        throw new IllegalArgumentException("Il path deve essere assoluto.");
    return find(path).size();
}
```

[\[sorgente\]](#)

Un discorso simile vale per le competenze «mutazionali» (creare un *file* o *directory*), il punto cruciale è usare `findDir` su `path.parent()` per trovare la *directory* su cui effettuare l'inserimento e `path.name()` per individuare il nome della *entry* da aggiungere:


```

public void mkdir(final Path path) throws FileNotFoundException,
FileAlreadyExistsException {
    if (!Objects.requireNonNull(path, "Il path non può essere null.").isAbsolute())
        throw new IllegalArgumentException("Il path deve essere assoluto.");
    final String name = path.name();
    if (name == null) throw new IllegalArgumentException("Il percorso non può essere
vuoto");
    findDir(path.parent()).add(new Directory(name));
}

public void mkfile(final Path path, final int size)
throws FileNotFoundException, FileAlreadyExistsException {
    if (!Objects.requireNonNull(path, "Il path non può essere null.").isAbsolute())
        throw new IllegalArgumentException("Il path deve essere assoluto.");
    final String name = path.name();
    if (name == null) throw new IllegalArgumentException("Il percorso non può essere
vuoto");
    findDir(path.parent()).add(new File(name, size));
}

```

[\[sorgente\]](#)

La shell

La **shell** è molto semplice da realizzare: può fare affidamento sulle competenze del *filesystem* per svolgere il suo compito, oltre al *filesystem* è sufficiente che memorizzi la *directory corrente* (ossia il percorso assoluto di una *directory* del *filesystem*).

```

private final FileSystem fs;
private Path cwd;

public Shell(final FileSystem fs) {
    this.fs = Objects.requireNonNull(fs, "Il filesystem non può essere null.");
    cwd = Path.ROOT;
}

```

[\[sorgente\]](#)

L'invariante è elementare: gli attributi non devono essere `null` e `cwd` deve essere un *path* assoluto che corrispondere sempre ad una *directory*.

Dal momento che in molti comandi può essere specificato un *path* sia assoluto che relativo, è comodo avere una funzione privata che risolva questi ultimi rispetto alla *directory corrente*:

```

private Path resolve(final String path) {
    return cwd.resolve(Path.fromString(path));
}

```





[\[sorgente\]](#)

La funzione principale da implementare è l'*interprete* che, letta una riga per volta, la suddivide (con uno `Scanner`) nel comando e negli eventuali argomenti e agisca di conseguenza (tramite uno `switch`):

```

for (;;) {
    final String line = con.readLine();
    try (final Scanner s = new Scanner(line)) {
        final String cmd = s.next();
    }
}

```

 Stars
  16k
  DOI [10.5281/zenodo.5831781](https://doi.org/10.5281/zenodo.5831781)
 License [GPL v3](#)
 License [CC BY-SA 4.0](#)