Classi e interfacce di utilità

Sezioni

Nelle API di Java, per diversi tipi `T` è comune incontrare una classe di nome `Ts` (di seguito vedremo ad esempio le classi `Objects`, `Arrays` e `Collections`). Tali classi (che non possono essere istanziate, avendo solo il costruttore di default e con visiilità privata), sono dei "contenitori" di metodi statici di utilità che riguardano il tipo `T`.

Questa sezione presenta un paio di tali classi e alcuni dei loro metodi che possono risultare molto utili per la prova pratica; inoltre richiama brevemente le interfacce che definiscono come *comparare* gli oggetti secondo le API di Java.

La classe `Objects`

La classe `java.util.Objects` contiene alcuni metodi statici di utilità generale che possono essere adoperati per tutti gli oggetti, indipendentemente dal loro tipo.

Sovrascrivere `hashCode`

Nel caso in cui si intendano sovrascrivere i metodi `equals` e `hashCode` di un oggetto, il metodo `hash` (che è *variadico*) può risultare molto comodo.

Se `equals` viene sovrascritto come congiunzione dell'uguaglianza di (un sottoinsieme degli) attributi dell'oggetto (diciamo `attr_1`, `attr_2`, `attr_N`), allora `hashCode` può essere sovrascritto come

```
@Override
public int hashCode() {
  return Objects.hash(attr_1, attr_2, attr_N);
}
```

piuttosto che implementando direttamente la ricetta proposta nell'Item 11 del Capitolo 3 del libro di testo "Effective Java".

Gestire i `null`

Il metodo `requireNonNull` consente di verificare se una espressione è `null` e, nel caso, sollevare una `NullPointerException` col messaggio indicato; ad esempio

```
Objects.requireNonNull(espressione, "Messaggio");
```

può essere usato invece di:

```
if (espressione == null)
  throw new NullPointerException("Messaggio");
```

Dal momento che qualora l'espressione non sia `null` il metodo ne restituisce il valore, esso può essere convenientemente usato in un assegnamento o invocazione di metodo; ad esempio

```
variabile = Objects.requireNonNull(espressione, "Messaggio");
```

può essere usato invece di:

```
if (espressione == null)
  throw new NullPointerException("Messaggio");
variabile = espressione;
```

e similmente

```
Objects.requireNonNull(espressione, "Messaggio").metodo();
```

può essere usato invece di:

```
if (espressione == null)
  throw new NullPointerException("Messaggio");
espressione.metodo();
```

Possono risultare comodi anche i metodi statici `equals`, `toString` e `hashCode` che possono essere usati anche su riferimenti `null`; ad esempio

```
String stringa = Objects.toString(oggetto);
boolean uguali = Objects.euqlals(questo, quello);
int hash = Objects.hashCode(oggetto);
```

possono essere rispettivamente usati invece di

```
String stringa = oggetto == null ? "null" : oggetto.toString();
boolean uguali = questo == null ? quello == null : questo.equals(quello);
int hash = oggetto == null ? 0 : oggetto.hashCode();
```

Controllare indici e intervalli

In molte circostanze può capitare di dover controllare se un indice (o un intervallo di indici interi, che può essere specificato dandone gli estremi, oppure l'estremo sinistro e la dimensione) è contenuto in un segmento iniziale dei numeri naturali (specificato tramite la sua dimensione).

Il metodo `checkIndex` e le sue varianti possono essere comodamente utilizzati a tale scopo: nel caso la condizione sia soddisfatta, essi restituiscono il valore dell'indice (o il limite inferiore dell'intervallo), viceversa sollevano una `IndexOutOfBoundException`.

Le interfacce `Comparable` e `Comparator`

Se si è interessati a definire una *relazione d'ordine* sugli oggetti di una certo tipo sono possibili due strategie:

- se gli oggetti sono dotati di un ordinamento *naturale*, generalmente si rendono *comparabili* facendo in modo che il loro tipo lo realizzi implementando l'interfaccia `Comparable`,
- se viceversa si vogliono tenere in considerazione più ordinamenti, si ricorre di volta in volta ad un *comparatore* diverso, ottenuto implementando opportunamente l'interfaccia `Comparator`.

Le due interfacce descritte prescrivono rispettivamente l'implementazione di un metodo `compareTo` (che compara l'oggetto corrente con un altro oggetto del medesimo tipo), o di un metodo `compare` (che compara due oggetti dello stesso tipo tra loro).

Una discussione esaustiva di queste interfacce esula dagli scopi di questo documento, chi volesse approfondire è invitato a consultare la documentazione delle API e a leggere l'Item 14 del Capitolo 3 del libro di testo "Effective Java".

Può essere però utile richiamare alcuni metodi (di default e statici) di `Comparator` che consentono di ottenere dei comparatori d'uso comune:

- il metodo di default `reversed` che consente di ottenere il comparatore corrispondente all'ordine inverso;
- i metodi statici `naturalOrder` e `reverseOrder` che restituiscono rispettivamente i comparatori dell'ordine naturale e del suo inverso;
- i metodi statici `nullsFirst` e `nullsLast` che restituiscono i comparatori ottenuti dal comparatore specificato che, in aggiunta, considerano i riferimenti `null` rispettivamente minori o maggiori di ogni altro valore.

Osservate che i metodi relativi all'ordine naturale non hanno argomento, devono pertanto inferire il tipo del comparatore da restituire o dal contesto, come ad esempio in

```
Comparator<Integer> DA_GRANDE_A_PICCOLO = Comparator.reverseOrder();
DA_GRANDE_A_PICCOLO.compare(1, 2)
```

```
1
```

dove il tipo è dedotto da quello deella variabile a cui assegnare il risultato, oppure da uno *hint*, come in

```
Comparator.<Integer>reverseOrder().compare(2, 1)
```

```
-1
```

Un esempio di uso

Si consideri una classe che rappresenti un orario della mattina (a prescindere dall'opportunità di sviluppare un tipo del genere, accennato qui a solo a titolo esemplificativo). Una implementazione minimale di tale tipo è data da

```
class OrarioMattina implements Comparable<OrarioMattina> {
  private static final String[] NUMERO_A_PAROLE = {"mezzanotte", "una", "due", "tre",
"quattro", "cinque", "sei", "sette", "otto", "nove", "dieci", "undici", "dodici"};
  public final int ore, minuti;
  public OrarioMattina(final int minutiDaMezzanotte) {
    if (minutiDaMezzanotte < 0 || minutiDaMezzanotte >= 12 * 60 ) throw new
IllegalArgumentException();
    ore = minutiDaMezzanotte / 60;
    minuti = minutiDaMezzanotte % 60;
  }
  public String ore() {
    return NUMERO_A_PAROLE[ore];
  }
  @Override
  public String toString() {
    if (minuti == 0) return ore();
    return ore() + " e " + minuti;
  }
  @Override
  public int compareTo(OrarioMattina altro) {
    int result = Integer.compare(ore, altro.ore);
    if (result == 0) result = Integer.compare(minuti, altro.minuti);
    return result;
  }
  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof OrarioMattina)) return false;
    final OrarioMattina altro = (OrarioMattina)obj;
    return ore == altro.ore && minuti == altro.minuti;
  }
  @Override
  public int hashCode() {
    return Objects.hash(ore, minuti);
  }
}
```

Il `toString` della classe indca le ore in parole, seguite dai minuti (se non pari a 0). Gli oggetti della classe sono *comparabili* secondo l'ordine naturale dello scorrere del tempo; si osservi che (come da specifiche dell'interfaccia) l'implementazione di `compareTo` porta con se la necessità di implementare `equals` e `hashCode` in modo coerente.

Dati due orari di questo tipo

```
OrarioMattina
  colazione = new OrarioMattina(8 * 60 + 30),
  merenda = new OrarioMattina(10 * 60);
```

è possibile confrontarli secondo l'ordine naturale come segue

```
colazione.compareTo(merenda) < 0
```

```
true
```

con l'atteso risultato che la colazione si fa prima della merenda.

Se ora volessimo confrontarli in base all'ordine lessicografico delle loro rappresentazioni testuali potremmo definire (qui facendo uso di una *classe anonima*) il *comparatore*

```
final Comparator<OrarioMattina> LESSICOGRAFICO_ORE = new Comparator<>() {
    @Override
    public int compare(OrarioMattina primo, OrarioMattina secondo) {
      return primo.ore().compareTo(secondo.ore());
    }
  };
```

secondo quest'ultimo, l'ordine tra i due orari scelti in precedenza si ribalta

```
LESSICOGRAFICO_ORE.compare(colazione, merenda) > 0
```

```
true
```

in quanto "sette" viene lessicograficamente dopo "dieci" (dato che la "s" è dopo la "d" nell'ordine alfabetico).

La classe `Arrays`

La classe **`java.util.Arrays`** contiene alcuni metodi statici di utilità generale che riguardano gli array.

Per i metodi illustrati di seguito sono state scelte le segnature con argomenti di tipo *generico* o `Object`, osservate però che di ciascuno di essi esiste (per ragioni di semplicità ed efficienza) una versione sovraccaricata per ciascun *tipo primitivo* di argomento (come è ovvio sarà il compilatore a scegliere la segnatura adatta di volta in volta, sulla scorta del tipo apparente degli argomenti).

Il metodo `toString`

Può capitare molte volte di dover emettere il contenuto di un array sotto forma di stringa, purtroppo l'implementazione del metodo `toString` di `Object` ereditata dagli array non è particolarmente leggibile; è però possibile usare il metodo **`toString`** di questa classe per ottenere una rappresentazione molto semplice; ad esempio

```
int[] arr = new int[] {1, 2, 3, 4};
Arrays.toString(arr) + " è più leggibile di " + arr
```

```
[1, 2, 3, 4] è più leggibile di [I@5052d5e3
```

Riempire o copiare

Usando il metodo **`fill`** è possibile riempire (un segmento) di un array con un valore di *default*; ad esempio

```
String[] slot = new String[6];
Arrays.fill(slot, 0, 3, "primi tre");
Arrays.fill(slot, 4, 6, "ultimi due");
Arrays.toString(slot)
```

```
[primi tre, primi tre, primi tre, null, ultimi due, ultimi due]
```

Esistono diversi metodi per ottenere una copia di un array (non tutti realizzati tramite la classe `Arrays`); è necessario ricordare che se gli elementi dell'array non sono di tipo primitivo, tutti effettuano una *copia per indirizzo*, ossia copiano solo i riferimenti degli elementi dall'array d'origine a quello copiato, senza però copiare gli elementi stessi. Questo significa, tra l'altro, che se gli elementi sono di tipo *mutabile* attraverso la copia dell'array è possibile modificare gli elementi dell'array di origine (ovviamente a prescindere dal fatto che i riferimenti in cui sono memorizzati gli array siano dichiarati come `final`!).

Il primo modo di ottenere una copia è data dal metodo **`copyOf`**; osservate che tale metodo può produrre una copia con un numero di elementi maggiore di quello dell'originale (popolando con `null` le posizioni aggiuntive). Questo può essere molto utile nel caso in cui, memorizzando valori in un array, si stia per eccederne la dimensione: sarà sufficiente copiarlo in uno di dimensione doppia e quindi procedere. Di ciascun metodo esistono anche delle versioni sovraccaricate senza i limiti del segmento (che vengono assunti coincidere con l'inizio e la fine dell'array). Con il metodo **`copyOfRange`** è invece possibile ottenere una copia di (un segmento) di un array; ad esempio

```
String[] subslot = Arrays.copyOfRange(slot, 2, 5);
Arrays.toString(subslot)
```

```
[primi tre, null, ultimi due]
```

Una delle versioni sovraccaricate di **`copyOf`** può essere usata per effettuare una sorta di "casting" tra array i cui elementi siano l'uno il sottotipo dell'altro. Come è ben noto, anche se `S` è sottotipo di `T` ed è certo che gli elementi di un array `t` di tipo `T[]` siano tutti di tipo concreto `S`, non è possibile effettuare il cast di tale array come `(S[])t`; per fare un esempio

```
Number[] numeri = new Number[] {1, 2, 3};
try {
  Integer[] interi = (Integer[])numeri;
} catch (ClassCastException e) {
  System.err.println(e);
}
```

```
java.lang.ClassCastException: class [Ljava.lang.Number; cannot be cast to class
[Ljava.lang.Integer; ([Ljava.lang.Number; and [Ljava.lang.Integer; are in module java.base
of loader 'bootstrap')
```

solleva una eccezione: evidentemente, il cast non può avvenire solo sul riferimento, perché la cosa avesse senso, dovrebbe venir applicato anche elemento per elemento (ad esempio con un ciclo); ma si può anche usare `copyOf` nella versione che accetta una istanza di **`Class`** per determinare il tipo degli elementi

```
Integer[] interi = Arrays.copyOf(numeri, numeri.length, Integer[].class);
Arrays.toString(interi)
```

```
[1, 2, 3]
```

Per concludere, vale la pena ricordare anche due modi di ottenere una copia indipendenti dalla classe `Arrays`.

Il più elementare è usare il metodo `clone` dell'array stesso. Il secondo è usare il metodo statico **`arraycopy`** della classe `System`. Questo metodo invece di restituire la copia in nuovo array, copia i riferimenti da un array sorgente ad uno destinazione (che deve essere già allocato e della dimensione opportuna); ad esempio

```
int[] positivi = new int[] {1, 2, 3, 4, 5, 6, 7};
int[] negativi = new int[] {-1, -2, -3, -4, -5, -6, -7};
System.arraycopy(positivi, 2, negativi, 1, 3);
```

ha l'effetto di copiare 3 elementi dalla posizione 2 di `positivi` alla posizione 1 di `negativi`

```
Arrays.toString(negativi)
```

```
[-1, 3, 4, 5, -5, -6, -7]
```

Adattare la dimensione di un array

Vogliamo raccogliere in un array di `long` di nome `pows` gli elementi dell'insieme $\{n < 10^{12} | n = 2^{2k}, k \geq 0\}$; supponendo di non conoscere a priori la cardinalità dell'insieme, possiamo riempire iterativamente l'array, inizialmente di dimensione 1, raddoppiandone la dimensione ogni volta che il numero `i` di potenze individuate ne uguaglia la lunghezza

```
long[] pows = new long[1];
long n = 0;
int i = 0, k = 0;
while ((n = (long)Math.pow(2, 2 * k++)) < 1_000_000_000_000L) {
  if (i == pows.length) pows = Arrays.copyOf(pows, pows.length * 2);
    pows[i++] = n;
}
pows = Arrays.copyOf(pows, i);
Arrays.toString(pows)
```

```
[1, 4, 16, 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576, 4194304, 16777216,
67108864, 268435456, 1073741824, 4294967296, 17179869184, 68719476736, 274877906944]
```

Osservate come, oltre al modo comodo di scrivere la costante 10^{12} come `1_000_000_000_000L` interponendo per leggibilità il separatore `_`, al termine del riempimento, si possono eliminare le posizioni rimaste vuote dell'array copiandolo in uno di dimensione esattamentepari al numero totale di potenze individuate.

Confrontare

Il metodo **`equals`** può essere usato per decidere se due array contengono lo stesso numero di elementi e tutti gli elementi in posizione corrispondente risultano uguali (secondo il metodo `equals` del tipo degli element dell'array, o la comparazione con `==` nel caso di tipi primitivi).

In modo analogo, per i tipi primitivi e quelli che sono *comparabili* il metodo **`compare`** permette di determinare l'*ordine lessicografico* tra i due array, basandosi sull'ordine naturale degli elementi. Nel caso in cui gli elementi non siano *comparabili* (o si voglia utilizzare un ordine diverso da quello naturale), esiste una versione sovraccaricata del metodo **`compare`** che accetta un **`Comparator`** come argomento.

Ordinare e cercare

Dato un vettore, è possibile ordinarlo *in loco* secondo l'*ordine naturale* dei suoi elementi tramite il metodo **`sort`**, oppure specificando esplicitamente un *comparatore* con la versoine sovraccaricata **`sort`**.

Riutilizzando la classe `OrarioMattina` della sezione precedente, ad esempio

```
OrarioMattina[] orari = new OrarioMattina[] {colazione, merenda, new OrarioMattina(5 * 60
+ 10)};
Arrays.sort(orari);
Arrays.toString(orari)
```

```
[cinque e 10, otto e 30, dieci]
```

permette di ordinare gli orari secondo l'ordine naturale, mentre

```
Arrays.sort(orari, LESSICOGRAFICO_ORE);
Arrays.toString(orari)
```

```
[cinque e 10, dieci, otto e 30]
```

li ordina secondo l'ordine lessicografico dell'ora (in parole). La versione in cui è possibile specificare il comparatore può essere utile per invertire l'orine; ad esempio

```
Arrays.sort(orari, LESSICOGRAFICO_ORE.reversed());
Arrays.toString(orari)
```

```
[otto e 30, dieci, cinque e 10]
```

oppure, basandosi sull'ordine naturale,

```
Arrays.sort(orari, Comparator.reverseOrder());
Arrays.toString(orari)
```

```
[dieci, otto e 30, cinque e 10]
```

Dato un vettore ordinato, è possibile cercare la posizione di un elemento nel vettore (o scoprire se non è contenuto nel vettore), usando la *ricerca dicotomica*, tramite il metodo **`binarySearch`** che si basa sull'ordine naturale, o la versione sovraccaricata di **`binarySearch`** che consente di specificare un comparatore (che, evidentemente, deve essere il medesimo che era stat usato per ordinare l'array prima della ricerca).

Un esempio di ricerca e inserimento

Come esempio della ricerca, consideriamo l'array `cifreOrdinate` che contenga le parole corrispondenti alle cifre decimali ordinato lessicograficamente, ottenuto come

```
String[] cifreInParole = new String[] {"zero", "un", "due", "quattro", "cinque", "sei",
"sette", "otto", "nove" };
String[] cifreInParoleOrdinate = cifreInParole.clone();
Arrays.sort(cifreInParoleOrdinate);
Arrays.toString(cifreInParoleOrdinate)
```

```
[cinque, due, nove, otto, quattro, sei, sette, un, zero]
```

Usiamo la ricerca per ottenere l'inversa della funzione che mappa `i` in `cifreInParoleOrdinate[i]`, ossia la funzione `cifraAPosizione` che mappa la parola `cifra` corrispondente a una cifra in parole nell'indice `i` tale che `cifreInParoleOrdinate[i].equals(cifra)` sia vero

```
static final int cifraAPosizione(final String cifra) {
  int valore = Arrays.binarySearch(cifreInParoleOrdinate, cifra);
  if (valore < 0) throw new IllegalArgumentException();
  return valore;
}
```

che si comporta come atteso

```
cifraAPosizione("zero")
```

```
8
```

dato che "z" è certamente l'ultima lettera dell'alfabeto.

Come è facile accorgersi, ci siamo scordati del tre, cercandolo infatti otteniamo un valore negativo dell'indice!

```
int idx = Arrays.binarySearch(cifreInParoleOrdinate, "tre");
idx
```

```
-8
```

Secondo il contratto del metodo di ricerca, un risultato negativo non solo indica che l'elemento non è stato trovato, ma suggerisce la posizione `pos` dove dovrebbe venir inserito nell'array secondo la formula `idx = -pos - 1` (che ovviamente rende `idx` sempre negativo); usando questa informazione è possibile sistemare la mancanza

```
int pos = -idx - 1;
pos
```

```
7
```

A questo punto è sufficiente allocare un nuovo array `corretto` con una posizione in più, copiare dall'array `cifreInParoleOrdinate` le posizioni fino a `pos` esclusa, aggiungere in tale posizione di `corretto` la stringa `"tre"` e quindi copiare le rimanenti `cifreInParoleOrdinate.length - pos` posizioni da `cifreInParoleOrdinate` in `corretto` a partire da `pos + 1`

```
String[] corretto = new String[cifreInParoleOrdinate.length + 1];
System.arraycopy(cifreInParoleOrdinate, 0, corretto, 0, pos);
corretto[pos] = "tre";
System.arraycopy(cifreInParoleOrdinate, pos, corretto, pos + 1,
cifreInParoleOrdinate.length - pos);
Arrays.toString(corretto)
```

```
[cinque, due, nove, otto, quattro, sei, sette, tre, un, zero]
```

Nota

A maggior conferma del fatto che nessuna delle conoscenze di questo documento è strettamente necessaria al superamento della prova pratica, quanto mostrato sin qui consente di costruire e mantenere un array (eventualmente ordinato) di dimensione adattabile facendo uso soltanto di concetti elementari (appresi dall'insegnamento di "Programmazione" del primo anno) che, peraltro, possono essere molto facilmente implementati anche usando esclusivamente array e cicli `for`.

Con un array di dimensione adattabile è molto semplice implementare buona parte dei comportamenti delle collezioni che saranno illustrate nella seguente sezione (può essere un ottimo esercizio provare a farlo!). Ai fini del superamento della prova pratica, le *liste* possono essere banalmente sostituite da un array di dimensione adattabile, così come lo possono gli *insiemi* (e sufficiente prestare attenzione ai duplicati); persino le *mappe* possono essere implementate senza scomodare nessuna struttura dati tra quelle più sofisticate (incontrate nell'insegnamento di "Algoritmi e strutture dati" del secondo anno), ma usando semplicemente due array "paralleli"; tale implementazione può essere persino resa ragionevolmente efficiente se le chiavi sono comparabili!
