Programmazione 2

Undicesima esercitazione

18/05/2022

Classi anonime

Classi anonime

- Spesso, capita di dover scrivere delle classi che vengono istanziate e usate una sola volta nel programma
- Ci è capitato ad esempio negli esercizi sui `Comparator` (w8)

```
Gatto[] gatti = new Gatto[3];
...
Arrays.sort(gatti, new CodaComparator());
```

Classi anonime

- Se la classe "è breve", è conveniente sostituirla con una classe anonima
  NOTA: le classi anonime possono anche avere diversi metodi o variabili
- Abbiamo già visto un esempio, sempre con un `Comparator` (w9)

```
Ordina.perCriterioUtente(risorse,
    new Comparator<RisorsaWeb>() {
        public int compare(Risorsa a, Risorsa b) {  }
    }
);
```

Classi anonime

- La sintassi **somiglia** alle solite chiamate a costruttore:
  - Operatore **new**
  - Nome dell'interfaccia da implementare o superclasse da estendere
  - Parentesi con argomenti per il costruttore (nel caso della superclasse)
  - Il codice della classe (metodi e variabili)

```
Ordina.perCriterioUtente(risorse,
   new Comparator<RisorsaWeb>() {
     // variabili e metodi
   }
);
```

Espressioni lambda

Espressioni lambda

- Alternativa sintattica ancora più breve delle classi anonime
- Le espressioni lambda ci permettono di definire in modo conciso un metodo "usa e getta" da passare come argomento a un altro metodo
- Sono molto utili per sostituire classi anonime che implementano interfacce funzionali, come negli esempi precedenti

```
String[] parole = {"casa", "armadio", "zenzero"};
Arrays.sort(parole,
    (String a, String b) -> { return a.compareTo(b); }
);
```

Esercizio: espressioni lambda (massimo 10 minuti)

- Scaricate da E-Learning i file Gatto.java e TestLambda.java

- Scrivete le due espressioni lambda per ordinare l'array di oggetti Gatto in base al loro nome e in base alla lunghezza della loro coda

```
Arrays.sort(gatti, /* LA TUA LAMBDA QUI */);
```

Method references

Method references

- Sostituiscono le espressioni lambda, quando queste non farebbero altro che passare gli argomenti ricevuti a un metodo già esistente
- Si presentano in quattro varianti:
  - Passare gli argomenti a un metodo statico: `NomeClasse::nomeMetodo`
  - Passare gli argomenti a un metodo d'istanza: `nomeReference::nomeMetodo`
  - Passare gli argomenti a un costruttore: `NomeClasse::new`
  - Passare gli argomenti a un metodo di istanza di un oggetto qualsiasi `NomeTipo::nomeMetodo`

Method references: esempio di "oggetto qualsiasi"

- Riprendiamo un esempio precedente:

```
String[] parole = {"casa", "armadio", "zenzero"};
Arrays.sort(parole, (a, b) -> a.compareTo(b) );
```

- La chiamata equivalente quando usiamo i method references, è:

```
Arrays.sort(parole, String::compareTo);
```

- Il metodo viene applicato al primo argomento della lambda, e riceve gli argomenti successivi (se presenti)

Paradigma funzionale: first-class functions

Paradigma funzionale: first-class functions

- Le nostre lambda (o method references) possono essere usate come argomenti per un metodo, o assegnate a variabili
- Le lambda diventano di fatto un nuovo tipo di valore
- Il package java.util.function mette a nostra disposizione le interfacce che ci permettono di usare le lambda alla stregua degli altri oggetti

```
import java.util.function.Function;
Function<String,String> f1 = s -> s.toLowerCase();
// f1 = String::toLowerCase;
String s2 = f1.apply("CIAO"); // "ciao"
```

Esercizio: `java.util.function` (massimo 5 minuti)

- Scaricate da E-Learning il file TestFirstClass.java

- Trovate, nella documentazione del package `java.util.function`, l'interfaccia da usare come tipo del criterio di stampa

```
java.util.function./* NOME INTERFACCIA */ miaLambda
java.util.function./* NOME INTERFACCIA */ criterio
```

Java streams

Java streams

- Il package java.util.stream definisce classi e interfacce per costruire delle "pipeline" di operazioni su flussi di valori, potenzialmente infiniti
- Una pipeline si compone di:
  - Una sorgente (`Collection`, funzione generatrice, un canale di I/O, ...)
  - Una sequenza di operazioni intermedie da applicare ai suoi valori
  - Una operazione terminale
- Ogni operazione intermedia genera un nuovo stream per la successiva
- Una volta eseguita l'operazione terminale, la pipeline è consumata
  - Per eseguire altre operazioni sulla sorgente, occorre ri-convertirla in stream

Java streams: esempio

- Un esempio di pipeline:
  - Uso un array di `String` come sorgente
  - Estraggo le iniziali da ogni elemento;
  - Converto le iniziali in maiuscolo
  - Rimuovo i duplicati
  - Termino stampando a video lo stream finale

Java streams: esempio

```
String[] sorgente = { "albero", "casa", "palla",
"cane", "aereo" };

java.util.Arrays.stream(sorgente)
                .map(s -> s.substring(0, 1).toUpperCase())
                .distinct()
                .forEach(e -> System.out.println(e));

// Output: ACP
```

Java streams

- Ogni operazione produce un risultato, e non modifica la sorgente
- Le operazioni intermedie sono generalmente *lazy*, ovvero vengono applicate solo sui valori necessari per proseguire
- Di conseguenza, gli stream possono essere infiniti
- Ogni elemento dello stream viene visitato una sola volta

Java streams

- Alcune importanti operazioni intermedie:
  - map: genera un nuovo stream applicando una trasformazione a ogni elemento dello stream corrente
  - filter: il nuovo stream conterrà solo gli elementi che aderiscono a un criterio
  - concat: accoda due stream
  - distinct: rimuove duplicati
- Alcune importanti operazioni terminali:
  - count: rende il numero di elementi nello stream
  - forEach: applica una funzione con side-effect agli elementi dello stream
  - collect: utile per riconvertire uno stream in una collezione

Java streams

- Modi interessanti per generare stream:
  - Da un oggetto `Collection` usando il suo metodo `stream()`
  - Da un array, con `Arrays.stream(Object[])`
  - Da un file, con `BufferedReader.lines()`
  - Dai metodi statici dei vari stream

- In particolare, tra questi ultimi:
  - `Stream.iterate(T seed, UnaryOperator<T> f)`
  - `IntStream.range(int startInclusive, int endExclusive)`

Java streams: esercizio

Recupero da una vecchia esercitazione: Metodi generici

Recupero da una vecchia esercitazione: Metodi generici

Vogliamo un metodo che sfrutti le classi generiche:

```
class Utils {
    static int misura(java.util.List<T> l) {
        return l.get(1).toString().length();
    }
}
```

Recupero da una vecchia esercitazione: Metodi generici

Vogliamo un metodo che sfrutti le classi generiche:

```
class Utils {
    static int misura(java.util.List<T> l) {
        return l.get(1).toString().length();
    }
}
```

```
Utils.java:2: error: cannot find symbol
```

Recupero da una vecchia esercitazione: Metodi generici

Vogliamo un metodo che sfrutti le classi generiche:

```
class Utils<T> {
    static int misura(java.util.List<T> l) {
        return l.get(1).toString().length();
    }
}
```

Recupero da una vecchia esercitazione: Metodi generici

Vogliamo un metodo che sfrutti le classi generiche:

```
class Utils<T> {
    static int misura(java.util.List<T> l) {
        return l.get(1).toString().length();
    }
}
```

```
Utils.java:2: error: non-static type variable T cannot
            be referenced from a static context
```

Recupero da una vecchia esercitazione: Metodi generici

Vogliamo un metodo che sfrutti le classi generiche:

```
class Utils {
    static <T> int misura(java.util.List<T> l) {
        return l.get(1).toString().length();
    }
}
```

Ripasso Collection

Ripasso Collection

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Esercitazione

Esercitazione: Stack (1/2)

- Creare una classe Stack, basata su `ArrayList`, che permetta la gestione di uno stack di elementi generici. Implementare i metodi:
  - Costruttore
  - push(elemento)
  - pop()
  - get(indice)
  - size()
  - clear()

Esercitazione: Stack (2/2)

- Creare una classe Parentesi, che sfrutti la classe Stack per validare sequenze di parentesi
- Esempi di sequenze valide:
  - ()
  - (())
  - ((()(())))
- Esempi di sequenze non valide:
  -)(
  - ())
  - (