Programmazione 2

Ottava esercitazione

10/05/2023

Obiettivi dell’esercitazione

1. Classi anonime
2. Espressioni lambda

Classi anonime

Classi anonime

- Una classe anonima è una classe "locale" senza un nome assegnato.
- Si tratta di una classe definita e istanziata un'unica volta attraverso una singola espressione caratterizzata da una versione estesa della sintassi dell'operatore `new`.
- La sintassi per definire una classe anonima è identica a quella utilizzata per la chiamata di un costruttore, eccetto per il fatto che la chiamata al costruttore è seguita da un blocco di codice che estende la classe stessa (o implementa metodi nel caso si crei una istanza anonima di una interfaccia).

Classi anonime (sintassi)

A seconda che la classe anonima estenda una classe o implementi un'interfaccia si possono avere due varianti:

```
new ClassName ( <argomenti> ) { <corpo classe> }
```

oppure:

```
new InterfaceName () { <corpo classe> }
```

Quando si istanzia anonimamente una classe si deve usare uno dei costruttori disponibili, al quale vanno dunque passati gli opportuni argomenti.

Per istanziare una classe anonima basata su una interfaccia (che quindi la implementerà) è possibile fare uso solamente del costruttore *'default'* senza argomenti.

Classi anonime (esempio)

Possiamo utilizzare le classi anonime quando abbiamo la necessità di utilizzare una classe localmente ed una volta sola.

Consideriamo il seguente esempio, dove è data una interfaccia:

```
public interface TitledName {
  public String femaleTitle(String name);
  public String maleTitle(String name);
}
```

Classi anonime (esempio)

Il modo "classico" di istanziare un oggetto sarebbe:

```
BaseTitle frenchTitle = new FrenchTitle();
```

Classi anonime (esempio)

che dovrebbe essere corredato dalla definizione di una opportuna classe che implementi l'interfaccia:

```
public class FrenchTitle implements TitledName {
  @Override
  public String femaleTitle(String name) {
       return "Mademoiselle " + name;
  }
  @Override
  public String maleTitle(String name) {
       return "Monsieur " + name;
  }
}
```

Classi anonime (esempio)

mentre una istanza di un oggetto anonimo può essere ottenuta semplicemente ed in un sol colpo come:

```
BaseTitle englishTitle = new TitledName() {
  @Override
  public String femaleTitle(String name) {
        return "Ms " + name;
  }
  @Override
  public String maleTitle(String name) {
        return "Mister " + name;
  }
};
```

Classi anonime (esempio)

Nell'esempio possiamo notare che le classi anonime hanno caratteristiche precise:

- dichiarazione attraverso l'operatore new;
- interfaccia da implementare o classe da estendere;
- parentesi con gli argomenti per il costruttore (proprio come una classica espressione di istanziazione di una classe);
- il corpo della classe, in particolare, all'interno del corpo di definizione della classe anonima sono consentiti metodi, ma non istruzioni singole; naturalmente quando si istanzia anonimamente una interfaccia tutti i metodi devono essere implementati e allo stesso modo: tutti i metodi virtuali di una classe astratta andrebbero implementati.

Classi anonime (altre considerazioni)

Poiché la definizione di una classe anonima è un'espressione, deve essere parte di un'istruzione, motivo per cui c'è il ";" dopo l'ultima parentesi graffa chiusa.

Come per le classi locali, una classe anonima ha accesso a tutte le variabili della classe che la racchiude (anche a quelli privati).

Una classe anonima può essere definita anche dentro al corpo di un metodo, in tal caso sono visibili per la classe anonima solo le variabili `final` (ed `effectively final` da java 8).

Classi anonime (altre considerazioni)

In una classe anonima si possono dichiarare campi, metodi (anche se non implementano nessun metodo astratto della classe che si estende o dell'interfaccia che si implementa), classi locali ma non si possono definire costruttori.

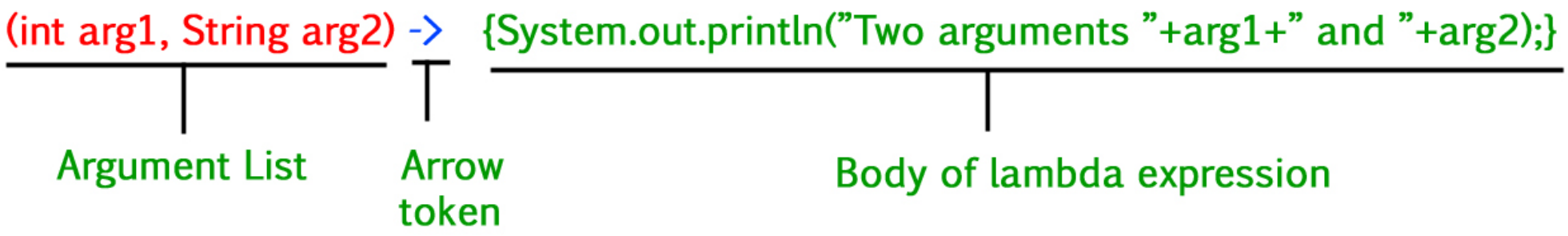
Le classi anonime sono una caratteristica del linguaggio Java utilizzata in numerosi contesti ma probabilmente il loro uso più comune è quando ci si trovano a realizzare interfacce grafiche dove il predominante approccio "a eventi" richiede la creazione di numerosi oggetti tutti simili (cioè tutti derivati di una determinata classe o interfaccia).

Espressioni lambda

Espressioni lambda

Java Lambda è una sintassi con cui possiamo più facilmente definire ed utilizzare funzioni (invece di dover dichiarare classi i cui oggetti hanno un metodo che rappresenta la nostra funzione).

Sono una alternativa alle **Classi Anonime** quando si debba implementare una *interfaccia funzionale*.



Espressioni lambda

*Cos'è un **interfaccia funzionale**?*

Un'interfaccia funzionale è un'interfaccia con un solo metodo. Queste interfacce rappresentano ciò che più si avvicina a una funzione.

Nel package `java.util.function` sono state introdotte molte interfacce funzionali.

Un esempio è l'interfaccia `Predicate<T>` con il solo metodo `boolean test(T)`.

`Predicate` rappresenta un predicato, una funzione ad un solo argomento che restituisce un valore booleano.

Espressioni lambda (**esempio**)

Vogliamo ordinare una lista di stringhe ignorando il primo carattere. Quindi la parola `casa` andrà messa prima di `armadio`.

È necessario riscrivere una funzione con un algoritmo di ordinamento specializzato a risolvere questo problema? NO.

Possiamo usare ad esempio la funzione `sort` della classe `java.util.Collections`

Espressioni lambda (**esempio**)

Il metodo `sort` è definito in due modi:

1)

```
static <T extends Comparable<? super T>> void sort(List<T> list)
```

che ordina la lista in base all'ordine naturale dei suoi elementi.

Utilizzando questo metodo, "armadio" verrebbe messo prima di "casa".

2)

```
static <T> void sort(List<T> list, Comparator<? super T>)
```

che ordina in base alle specifiche del comparatore

Espressioni lambda (**esempio**)

Utilizzando classi e oggetti potremo risolvere il problema così:

```
import java.util.*;

public class IgnoreFirstCharComparator implements Comparator<String> {

    /** compare ignoring the first character */
    @Override
    public int compare(String a, String b) {
        return a.substring(1).compareTo(b.substring(1));
    }

    public static void main(String[] args) {
        List<String> parole = Arrays.asList("casa", "armadio", "zenzero");

        // ordine naturale (String implementa Comparable)
        Collections.sort(parole);
        System.out.println(parole); // armadio, casa, zenzero

        // ordine personalizzato (IgnoreFirstCharComparator implementa Comparator)
        Collections.sort(parole, new IgnoreFirstCharComparator());
        System.out.println(parole); // casa, zenzero, armadio

    }

}
```

Espressioni lambda (**esempio**)

Utilizzando le classi anonime potremo risolvere il problema così:

```
        // ordine personalizzato (tramite classe anonima che implementa
        // o estende Comparator
        Collections.sort(parole, new Comparator<String>() {
            @Override
            public int compare(String a, String b) {
                return a.substring(1).compareTo(b.substring(1));
            }
        });
        System.out.println(parole); // casa, zenzero, armadio

    }
```

Espressioni lambda (**esempio**)

Utilizzando le espressioni lambda potremo risolvere il problema così:

```
        // ordine personalizzato (tramite lambda expression)
        Collections.sort(parole, (String a, String b) -> {
            return a.substring(1).compareTo(b.substring(1));
            }
        );
        System.out.println(parole); // casa, zenzero, armadio

    }
```

Espressioni lambda (**esempio**)

In base al contesto, è possibile omettere anche i tipi in input; se l'espressione viene calcolata in un’unica riga, anche parentesi grafe e return:

```
Comparator<String> ignoreFirstCharComparator =
        (a,b) -> a.substring(1).compareTo(b.substring(1)); // one-liner!
```

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, in base alla lunghezza della loro coda e in base al numero di caratteri del loro nome.

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

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 ("arbitrary object" nei tutorial Oracle): `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 chiamato sul primo argomento della lambda, e riceve gli argomenti successivi (se presenti)

Method references

Possiamo assegnare l'istanza di classe anonima a una variabile

```
Comparator<String> mioCom = new Comparator<String>() {
    public int compare(String a, String b) { ... }
};

Comparator<String> mioCom = (a, b) -> a.compareTo(b);

Comparator<String> mioCom = String::compareTo;
```

Paradigma funzionale: first-class functions

- Le classi anonime costruite intorno alle lambda (o method references) possono essere passate a un metodo, o assegnate a variabili
- Il package java.util.function definisce *interfacce funzionali* che saranno implementate dalle classi costruite intorno alle lambda

```
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
```

FINE