Laboratorio di Programmazione II

Corso di Laurea in Bioinformatica

Dipartimento di Informatica - Università di Verona

Sommario

- Metodi e Classi
- Definizione di Metodi
- Esercizi

Modularizzazione ed Astrazione

Modularizzazione

Dividere un programma in moduli

- Modulo: parte di un programma con funzionalita' precise
  - **Servizi esportati**: funzionalita' che gli altri moduli usano
  - **Interfaccia**: modalita' di uso delle funzionalita'
  - **servizi importati**: funzionalita' di cui il modulo e' **cliente**
  - **struttura interna**: come sono realizzate le funzionalita' (non interessante per il clienti)
- Modularizzazione, cruciale per: leggibilita', estensibilita', riusabilita'.

Example (La classe String)

La classe String gestisce le operazioni su stringhe,

- **Servizi esportati**: e.g., rendere una stringa maiuscola
- **Interfaccia**: e.g., *toUpperCase()*
- **Struttura interna**: implementazione del metodo.

Astrazione

Modularizzazione e Astrazione

Modello che prescinde da dettagli non rilevanti per la soluzione di un problema

- **Astrazione sulle operazioni**: metodi, *cosa* si deve fare e non *come*
- **Astrazione su oggetti**: classi, *oggetti* con proprieta' simili appartengono alla stessa classe

Definizione di Metodi Statici

Definire metodi

Metodi e moduli

Metodo definito da

- **Servizi esportati**: *cosa* viene realizzato dal metodo;
- **Interfaccia**: *intestazione*: signature + ritorno;
- **Servizi importati**: altri metodi o classi usate dal metodo;
- **Struttura interna**: corpo del metodo (codice java che realizza il servizio esportato).

Definire metodi **statici** I

Sintassi

Metodi che **NON** hanno un oggetto di invocazione

- Intestazione:

```
public static tipoRisultato nomeMetodo(param.)
```

  - public: modificatore di accesso
  - static: indica che il metodo e' statico
  - tipoRisultato: tipo di risultato del metodo (anche void)
  - nomeMetodo: nome del metodo
  - param.: lista di tipo e nome dei parametri formali, separati da virgola; ogni parametro e' una variabile. La lista puo' essere vuota.

- Blocco:

```
{
    istruzioni
}
```

Definire metodi **statici** II

Semantica

- Il corpo del metodo specifica le istruzioni che devono essere eseguite.
- I parametri formali passano le informazioni necessarie all'esecuzione del metodo. I parametri formali vengono utilizzati all'interno del metodo esattamente come delle variabili **gia' inizializzate**.
- I parametri formali sono inizializzati dal metodo chiamante con il valore dei **parametri attuali**.
- Il risultato del metodo (se presente) e' il valore dell'invocazione del metodo e viene restituito al metodo chiamante.

Definire metodi **statici**: Esempio

Example (II metodo *main*)

```
public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s = sc.nextLine();
        System.out.println(s.charAt(s.length()-1)
         +s.substring(1,s.length()-1)
         +s.charAt(0));
}
```

- public: accessibile dall'esterno della classe
- static: non ha un oggetto di invocazione
- void: non restiuisce risultati
- String[] args: un parametro array di String

Definire metodi **statici**: EsempioII

Example (II metodo *primaUltima*)

```
public class PrimaUltimaPar{
    public static void primaUltima(String s){
        System.out.println(s.charAt(s.length()-1)
         +s.substring(1,s.length()-1)
         +s.charAt(0));
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String my_s = sc.nextLine();
        primaUltima(my_s);
    }
}
```

Nota: il parametro **formale** s viene inizializzato con il valore del parametro **attuale** my_s

Risultato di un metodo

Istruzione *return*

- se un metodo ha un valore di ritorno allora **deve** eseguire l'istruzione *return*
- sintassi

  `return espressione`
- il valore dell'espressione deve essere compatibile con il tipo di ritorno del metodo

Example (return)

```
public static String primaUltima(String s){
      String res = s.charAt(s.length()-1)
      +s.substring(1,s.length()-1)
      +s.charAt(0));
      return res;
}
```

Istruzione *return* per metodi *void*

Tipo di ritorno *void* ed istruzione *return*

- se un metodo ha un valore di ritorno *void* non deve necessariamente contenere una istruzione *return*
- l'istruzione *return* puo' essere inclusa:

  `return;`
- l'istruzione *return* fa terminare il metodo (le istruzioni seguenti non vengono eseguite)

Metodi statici definiti in altre classi

Come chiamare metodi statici di altre classi

- Sintassi:

  ```
  NomeClasse.nomeMetodo(parametri);
  ```

- *NomeClasse* e' il nome della classe dove il metodo *nomeMetodo* e' definito.
- Il compilatore java (e la JVM) devono poter localizzare la classe *NomeClasse*.
  - istruzione *import* specifica quali classi considerare
  - il file *NomeClasse.java* si trova nella stessa directory della classe che la usa
  - per default Java cerca le classi nella directory corrente (concetto di CLASSPATH)

Metodi statici definiti in altre classi

Example (Uso metodi statici di altre classi)

```
import java.util.*;

/**
* Classe cliente per la classe PrimaUltimaPar
*/

public class ClientePUP{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String my_s = sc.nextLine();
        PrimaUltimaPar.primaUltima(my_s);
    }
}
```

Passaggio dei parametri

Passaggio parametri in java

- **Parametri Formali** specificati nella definizione del metodo
- `public static String primaUltima(String s){...}`
- **Parametri Attuali** specificati nella invocazione del metodo
- ```
  ...
  String my_s = sc.nextLine();
  PrimaUltimaPar.primaUltima(my_s);
  ...
  ```
- i parametri attuali e formali in java vengono legati per valore:
  - valutato parametro attuale (pa)
  - associa locazione di memoria per parametro formale (pf)
  - pf viene inizializzato con pa
- **NOTA:** questo legame puo' creare side effect (quando il parametro attuale e' un riferimento ad oggetto mutabile).
  ```

Esecuzione di un metodo: esempio

Example (cosa succede quando si esegue un metodo)

```
public static String ripeti(String pf){
    return pf+";"+pf;
}
...
public static void main(String[] args){
    String s = ripeti("A"+"T");
    System.out.println(s);
}
```

Esecuzione di un metodo

passi per eseguire un metodo

1. valutazione parametri attuali: valutata espressione A+T=AT
2. individuazione metodo da eseguire (in base alla signature): ripeti(String)
3. sospensione del chiamante: metodo main
4. allocazione memoria per parametri formali (e altre variabili): allocata memoria per pf
5. legame tra paramteri attuali e formali: pf = AT
6. esecuzone metodo: creata AT;AT e ritornata al chiamante
7. deallocata memoria per par. formali (e altre variabili): pf
8. risultato del metodo passato al chiamante: s = AT;AT
9. riprende esecuzione chiamante: stampa di s.

Modifica di un oggetto da parte di un metodo

Modifica dei parametri

- consideriamo il programma ModificaParametri.java
- la stringa rappresentata dalla variabile a non viene modificata
- questo perche' la modifica riguarda il parametro formale s che viene distrutto quando il metoto termina e non l'oggetto a cui s si riferisce (String sono oggetti immutabili)
- la stringa rappresantata dalla variabile b risulta modificata
- questo perche' viene modificato l'oggetto a cui b (ed sb) si riferisce quindi abbiamo il side-effect (StringBuffer sono oggetti mutabili)

Variabili Locali

Variabili Locali

- Variabile Locale: variabile definita all'interno di un metodo
- Tutte le variabili viste fino ad ora sono locali (e.g., al metodo main)
- In java esistono anche le variabili globali: sono variabili definite nella classe e precedute dalla parola chiave *static*.
- Concetti importanti per le variabili locali
  - campo di azione (scope)
  - tempo di vita

Scope delle variabili

Scope per variabili Locali

- insieme di moduli in cui la variabile e' visibile (i.e., puo' essere letta)
- Campo di azione: una variabile dichiarata in un blocco (...) e' visibile nel blocco ed in blocchi interni, non all'esterno.
- Il campo di azione di una variabile e' totalmente statico (puo' essere definito a tempo di compilazione).
- Visibility.java

Tempo di vita delle variabili

Tempo di vita per variabili Locali

- tempo in cui la variabile rimane accessibile in memoria
- variabili locali create ad ogni invocazione di un metodo e distrutte ogni volta che il metodo termina
- ad invocazioni diverse corrispondono nuove variabili
- il tempo di vita dipende dall'esecuzione del programma.
- VariableTime.java

Esercizi definizione di metodi

Esercizi sui metodi

UM1 realizzare il metodo needReplacing(...) della classe ProcessStringSB.java *Soluzione*: ProcessStringSB.sol

UM2 realizzare il metodo replaceAwithT(...) della classe ProcessStringSB.java *Soluzione*: ProcessStringSB.sol

UM3 realizzare la classe ProcessString che effettua le stesse operazioni di ProcessStringSB ma utilizza String e non StringBuffer *Soluzione*: ProcessString.java