

Metodi

- Un metodo è una porzione di codice a cui si associa un **nome**.
- Un'istruzione eseguita da un metodo può essere:
 - ▣ **invocazione**: esecuzione di un metodo
 - ▣ **assegnamento**: cambia lo stato di un campo
 - ▣ **ripetizione**: esegue istruzioni più volte
 - ▣ **selezione**: valuta condizioni ed esegue alternative
 - ▣ **eccezione**: gestisce situazioni anomale

Metodi

- Per richiedere a un oggetto di eseguire un'operazione bisogna mandargli un **messaggio** opportuno.
- Bisogna specificare:
 - ▣ il messaggio che si invia
 - ▣ l'oggetto che deve eseguirlo (parametro implicito che viene identificato come **this**)

```
rif_oggetto. nome_metodo (argomenti [...]);
```

Metodi

- La definizione di un metodo è composta di due parti: intestazione + corpo.

```
public int unMetodo(int x, int y)
```

← intestazione

```
{ if (x<0) return -1;  
  else  
  { this.potenza = Math.pow(x,y) ;  
    return x;  
  }  
};
```

← corpo

- L'intestazione è quanto precede le graffe.
- Il corpo è quanto vi è contenuto e descrive l'implementazione del metodo.

Metodi

- L'intestazione fornisce le seguenti informazioni su come utilizzare il metodo:
 - ▣ modificatori (opzionali): final, static, public, ...
 - ▣ tipo di ritorno del metodo (primitivo / riferimento)
 - ▣ nome del metodo
 - ▣ lista dei parametri

Metodi

- Un sottoinsieme dell'intestazione è dato dal prototipo del metodo: comprende il tipo di ritorno, il nome, i parametri.
- Un sottoinsieme del prototipo è la firma: comprende solo il nome e i parametri.

```
public int unMetodo (x, y)
```

intestazione

prototipo

firma

Metodi

- Una classe può avere più metodi con lo stesso nome: **overloading** (sovraccarico) del metodo.
- Tuttavia, non possono esserci più metodi con la stessa **firma**.
 - ▣ In altre parole, due metodi possono avere lo stesso nome ma devono differire per il numero dei parametri (non basta un diverso tipo di ritorno)
- L'overloading viene risolto in fase di compilazione.

Metodi

- Per il compilatore il metodo è identificato dalla sua firma. L'unicità della firma garantisce la selezione non ambigua del metodo nell'ambito di una classe.
- L'uso dell'overloading consente di non dovere introdurre nomi diversi per metodi con funzionalità simili ma che operano su argomenti diversi.

Variabili locali

- Le variabili definite nel corpo di un metodo prendono il nome di **variabili locali**.
 - ▣ Non hanno mai indicazioni **private** o **public**.
 - ▣ Quando il metodo termina (ad esempio a seguito di un **return**) vengono distrutte
 - ▣ A differenza dei campi (variabili d'istanza) dell'oggetto non vengono automaticamente inizializzate.

Parametri

- Le variabili dichiarate nell'intestazione del metodo sono dette **parametri formali**.
 - ▣ Inoltre, c'è il parametro implicito `this`.
- Distinguiamo tra:
 - ▣ nomi dei parametri (usati relativamente al metodo) che sono i parametri formali
 - ▣ valori assunti dai parametri al di fuori da esso chiamati parametri attuali o argomenti
- I parametri vengono inizializzati ai valori degli argomenti quando il metodo viene chiamato.

Parametri

- Le variabili dei parametri sono in effetti come le altre variabili locali del metodo.
 - ▣ Quando il metodo termina, vengono distrutte anch'esse.
 - ▣ Possono anche essere modificate nel corpo.

```
public int unMetodo(int x, int y)
{ x = 5; //ammesso, ma pessimo stile
  return y;
};
```

- ▣ Farlo però è considerato pessimo stile: semanticamente **x** dovrebbe rimanere costante.

Tipo di ritorno

- Se il tipo di ritorno è void, il metodo termina con l'ultima istruzione.

- ▣ Altrimenti è obbligatorio usare un'istruzione:

return espressione

che restituisce un valore e termina l'esecuzione.

- ▣ Il compilatore controlla che il metodo esegua per forza un **return** e anche che l'espressione che segue **return** sia compatibile con il tipo di ritorno indicato per il metodo.

Invocazione

- Quando un oggetto invoca un metodo di un altro si parla di chiamata di metodo esterno.
- Se invece un oggetto invoca un suo proprio metodo si tratta di chiamata di metodo interno.
- In questo caso non è necessario usare **this**: si parla di **autoreferenziazione**.
 - Un metodo d'istanza può accedere ai campi dell'oggetto (variabili d'istanza) senza usare **this**.
 - Solo se non c'è una variabile locale con quel nome.

Invocazione

- Quando un oggetto **o1** invia un messaggio a un oggetto **o2** per invocare un suo metodo **m**:
 - ▣ **o1** smette di fare quello che stava facendo e si mette in uno **stato di attesa**
 - ▣ **o2** esegue il metodo **m** sulla base del messaggio ricevuto (parametri passati)
 - ▣ quando **o2** ha finito l'esecuzione delle istruzioni contenute nel corpo di **m**, **o2** passa il controllo a **o1** che riprende dal punto in cui si era fermato

Invocazione

- Questo modello di esecuzione Java viene detto a singolo **thread** (=filo) di esecuzione.
- Nel caso a singolo thread, quando due o più metodi sono attivati, solo uno è effettivamente in esecuzione (l'ultimo che è stato attivato)
- Java consente anche di scrivere programmi multi-thread, ossia che hanno più metodi in esecuzione simultanea (programmazione parallela).

Invocazione

- Si parla anche di **trasferimento di controllo**
 - ▣ nel caso a singolo thread, il controllo è unico.
 - ▣ solo l'oggetto con il controllo può eseguire azioni
 - ▣ quando un oggetto invoca un metodo di un altro oggetto, gli cede il controllo
 - ▣ quando un oggetto termina di eseguire il metodo, restituisce il controllo a chi glielo aveva ceduto.

Ricorsione

- È un caso particolare di trasferimento di controllo, in cui non solo **o1** e **o2** coincidono (quindi è una chiamata di metodo interno) ma il metodo **m** chiama sé stesso.
- Esempio: calcolo del fattoriale di un numero.
- `factorial(n)` viene calcolato come `n * factorial(n-1)`.
 - ▣ Attenzione a impostare bene le condizioni per una corretta ricorsione.

Ricorsione

```
public static int factorial(int n)
{ if (n < 0)
    throw new IllegalArgumentException();
  else
    if (n == 0) return 1;
    else
    { int result = n * factorial(n-1);
      return result;
    }
}
```

Eccezioni

- Che fare quando un metodo viene invocato con un parametro non consentito?
 - ▣ Si può impostare il metodo in modo che non faccia nulla e restituisca il controllo al chiamante, facendolo terminare con un'istruzione **return**.
 - ▣ O si può invocare **System.exit(codice)** che fa terminare il programma.
 - ▣ Oppure, alternativa più sofisticata: invocare un'**eccezione**.

Eccezione

□ Esempio di sintassi:

```
intestazione metodo(parametro)
{ if (parametro_non_valido)
    throw new IllegalArgumentException();
  else
  { prosegui..
  }
}
```

Eccezione

- A questo punto l'eccezione che abbiamo invocato deve essere gestita. Questo può essere fatto direttamente, oppure viene restituito il controllo al chiamante, e al limite terminato il programma.
- L'aspetto importante è che il chiamante può cercare di “fare il possibile” per gestire l'eccezione e risolvere il problema.

Metodi statici

- I metodi sono definiti all'interno di una classe.
- Vengono poi tipicamente invocati utilizzando una variabile riferimento che tiene traccia dell'istanza di un oggetto.
 - ▣ Questo rende possibile l'autoreferenziazione
 - ▣ Infatti durante l'esecuzione del metodo, **this** ha il valore di questa variabile.
- Ma se un metodo richiede un oggetto, come far partire il sistema (quando di oggetti ancora non ne esistono?)

Metodi statici

- La soluzione è usare un metodo speciale che non richiede oggetti e può essere usato in qualunque momento: un **metodo statico**.
 - ▣ Il metodo **main** deve essere statico
- Inoltre esistono casi in cui vogliamo dei servizi senza avere oggetti a cui chiederli.
 - ▣ se manipoliamo numeri
 - ▣ se ci serve un metodo generale che funzioni anche se non esistono oggetti di quella classe (e non vogliamo creare un oggetto dummy)

Metodi statici

- Il metodo **main** viene indicato alla JVM come primo metodo da eseguire.
 - ▣ Per questo dev'essere **static** , perché quando viene eseguito non ci sono ancora oggetti, e **public**, perché la JVM possa accederlo.
- In realtà ogni classe può avere il suo metodo **main** ma solo quello della classe che viene eseguita sarà l'entry point dell'applicazione.

Metodi statici

- Diversamente dagli altri metodi (d'istanza) i metodi statici destinano il loro messaggio alla classe, anziché a un oggetto.

`nome_classe . nome_metodo (argomenti [...]) ;`

- I metodi statici non hanno quindi il parametro implicito: non possono usare **this**.

Metodi statici

- I metodi non statici possono accedere a tutti i membri della classe, anche statici (es. le variabili di classe, che sono campi statici).
- Non vale il viceversa: un metodo statico può accedere solo membri statici. Quindi non può cambiare un campo non statico della classe.
 - ▣ La cosa è sensata, perché c'è una variabile per ogni istanza della classe: quale dovrebbe usare il metodo statico?

Metodi statici

- Ad esempio la classe `studente` potrebbe avere un campo statico: `static String corsoLaurea`
- Stiamo quindi dicendo che la variabile è una sola per tutta la classe `studente`
- Possiamo pensare di avere un metodo statico per cambiare questa variabile:
`public static void setCorso (String nome)`
- Impostare `studente.corsoLaurea = nome;` è corretto (metodo statico, campo statico).

Metodi statici privati

- A volte un metodo può:
 - ▣ essere non legato a un singolo oggetto
 - ▣ avere funzioni “interne”, cioè di utilità solo per la classe stessa dove è inserito
- Ad esempio, questo potrebbe capitare per metodi da usare nei costruttori. In questo caso, il metodo può essere **private static**.
 - ▣ **static** perché serve a tutta la classe,
private perché non serve al di fuori di essa.

Combinare metodi

- Esistono due modi per combinare più invocazioni di metodi.
- 1) Composizione
 - ▣ si usa il risultato di un'invocazione come **argomento** in un'altra invocazione

```
String a,b,c,d;
```

```
...
```

```
d = a.concat(b.concat(c));
```

eseguito dopo

eseguito prima

Combinare metodi

□ 2) Cascata

- si usa il risultato di un'invocazione come **oggetto destinatario** (parametro implicito) di un'altra invocazione di metodo

```
String a,b,c,d;
```

...

```
d = a.concat(b).concat(c);
```

eseguito prima

eseguito dopo

Passaggio dei parametri

- Quando si invoca un metodo, vengono passati i parametri. Dato che le variabili parametro vengono inizializzate ai valori degli argomenti (che vengono cioè “copiati”), di fatto stiamo realizzando un **passaggio per valore**.
- Questo meccanismo ha delle limitazioni, e Java non consente il passaggio per riferimento (come ad esempio in C).

Passaggio dei parametri

- Conseguenze in qualche modo diverse per tipi primitivi (numeri) o tipi riferimento (oggetti)
- La variabile del parametro formale viene inizializzata al valore del parametro attuale, e tra i due valori c'è totale indipendenza.
- Per entrambi il passaggio è per valore, e non si può cambiare il parametro attuale.
- Però possono sorgere effetti collaterali.

Passaggio dei parametri

- In effetti quando la variabile passata è di tipo riferimento, lei stessa non può essere modificata ma, tramite un effetto collaterale, lo può essere l'oggetto a cui essa punta.
- Anche se non è propriamente corretto, spesso si sente quindi dire che gli oggetti Java sono passati per riferimento.
 - La verità è che le variabili istanza contengono riferimenti e quindi non possono essere cambiate, ma può esserlo l'area a cui puntano.

Esempio

- Definiamo intanto un tipo `Numero` (doppione del tipo primitivo `int`, ma è un oggetto)

```
class Numero
{ private int num;
  public Numero(int x) {this.num = x;}
  public int getNum() {return this.num;}
  public void setNum(int x) {x = this.num;}
}
```

CAMPO

COSTRUTTORE

METODI (get, set)

Esempio

```
class Prova
{ public static void met(int k)
  { System.out.println(k); //stampa 5
    k++;
    System.out.println(k); //stampa 6
  }
  public static void main(String[] args)
  { int k;
    k = 5;
    System.out.println(k); //stampa 5
    met(k);
    System.out.println(k); //stampa? 5
  }
}
```

Esempio

```
class Prova
{ public static void met(Numero k)
  { System.out.println(k.getNum()); //stampa 5
    k = new Numero(6);
    System.out.println(k.getNum()); //stampa 6
  }
  public static void main(String[] args)
  { Numero k = new Numero(5);
    System.out.println(k.getNum()); //stampa 5
    met(k);
    System.out.println(k.getNum()); //stampa? 5
  }
}
```

Esempio

- In questa versione, bisogna ovviamente usare il metodo `get` per accedere `k` che è un `Numero` (non più `int`).
- Il metodo `met`, che viene invocato dal `main`, crea internamente a sé stesso un riferimento al numero 6. Questo riferimento sovrascrive il valore che `k` tiene dentro al metodo.
- Ma, terminato il metodo, la versione locale di `k` viene distrutta. Il `main` stamperà due volte 5.

Esempio

```
class Prova
{ public static void met(Numero k)
  { System.out.println(k.getNum()); //stampa 5
    k.setNum(6);
    System.out.println(k.getNum()); //stampa 6
  }
  public static void main(String[] args)
  { Numero k = new Numero(5);
    System.out.println(k.getNum()); //stampa 5
    met(k);
    System.out.println(k.getNum()); //stampa? 6
  }
}
```

Effetti collaterali

- Stavolta, il valore è stato cambiato.
- In effetti la variabile k è passata per valore (cosicché il parametro formale k viene a coincidere col parametro attuale k).
- Questo parametro in effetti è davvero trasmesso per valore.
- Solo che, essendo k un riferimento, nulla vieta che l'area di memoria a cui punta venga acceduta e cambiata: **effetto collaterale!**

Effetti collaterali

- L'oggetto `k` di tipo `Numero` viene modificato dal metodo `met` in cui è un parametro esplicito.
- Tutte le volte che un metodo modifica un oggetto che non sia il suo parametro implicito si parla di effetto collaterale.
- Il metodo `set` è basato su un unico effetto collaterale.

Effetti collaterali

- Un effetto collaterale può modificare due tipi di oggetti:
 - ▣ un parametro esplicito
 - ▣ un campo statico pubblico
- In generale, gli effetti collaterali sono tanto più indesiderabili quanto meno si riescono a tenere sotto controllo.
 - ▣ Se compaiono come errori di programmazione sono molto difficili da trovare.