Definizione di metodi in Java

Un *metodo* in Java definisce un'operazione ad alto livello (*sottoprogramma*) che consente di manipolare dati e oggetti. Durante la computazione effettuata da un programma, un'operazione può essere applicata piú volte e su argomenti di volta in volta diversi. In tal caso è una buona tecnica di programmazione organizzare la computazione in maniera modulare individuando quelle operazioni (o *funzionalità*) che possono essere definite in modo piú astratto per operare su argomenti diversi. Come abbiamo già detto, ogni metodo è contenuto in una classe. Una classe può contenere la definizione di vari metodi. I metodi contengono dichiarazioni ed istruzioni. Un metodo può richiedere degli argomenti su cui operare e restituisce un valore che è il risultato della sua computazione oppure non restituisce alcun valore, ma semplicemente il metodo effettua delle modifiche sullo stato della macchina (per esempio, il metodo può modificare una qualche caratteristica di un oggetto o stampare una sequenza di caratteri).

Supponiamo di voler scrivere un semplice programma Java che calcola il prodotto di due numeri naturali tramite l'algoritmo basato sulla somma. Poiché durante una computazione può essere necessario applicare piú volte l'operazione del prodotto, è opportuno definire un metodo che, dati due numeri naturali, calcola e restituisce il risultato del loro prodotto. Il metodo `main` (necessario per avere un programma Java eseguibile) può limitarsi a prendere in input i numeri da sommare (forniti dall'utente), invocare l'applicazione del metodo che implementa l'operazione di prodotto e, per esempio, provvedere a visualizzare il risultato.

Tipicamente conviene organizzare le varie classi con i loro metodi nel modo seguente. Si definisce *una classe per file* (con la corrispondenza tra i nomi già ricordata). I vari metodi sono definiti in una o piú classi, ed il `main` viene messo in una classe separata di test (o utilizzo). Ad esempio, possiamo definire la classe `Mult` nel file `Mult.java` contenente un metodo `mult` definito come segue:[1]

```
public class Mult {
  public static int mult (int x, int y) {
    int z = 0;
    while (y > 0) {
      z = z+x;
      y = y-1;
    }
    return z;
  }
}
```

e poi definiamo una classe `MultTest` nel file `MultTest.java` contenente il seguente metodo `main`:

```
public class MultTest {
  public static void main (String[] args) {
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    System.out.println(Mult.mult(x,y));
  }
}
```

Il metodo `mult` definisce l'operazione di moltiplicazione di due numeri naturali tramite la somma, mentre il metodo `main` prende in input due valori, chiama il metodo `mult` su tali

[1]Nel seguito anche le classi verranno definite pubbliche tramite la parola riservata `public`. Nota che in un file può essere definita al piú una classe pubblica ed il suo nome deve corrispondere al nome del file.

valori e poi ne stampa il risultato. In questo esempio, il metodo `mult` è quindi chiamato (o invocato) dal metodo `main`, che agisce da metodo chiamante.

Vediamo come sono stati definiti questi due metodi. L'intestazione del metodo `mult` è

```
public static int mult (int x, int y)
```

da cui deriviamo le seguenti informazioni:
- il metodo è definito pubblico (tipicamente i nostri metodi saranno definiti pubblici, in modo che possano essere invocati da qualsiasi punto del programma);
- il metodo è statico (si dice anche che è un *metodo di classe*) in quanto dipende solo dalla classe in cui è definito e non dall'oggetto su cui verrà applicato;
- il metodo restituisce un valore di tipo intero (denotato dal tipo `int` dopo la parola riservata `static`);
- `mult` è il nome (un identificatore) del metodo;
- il metodo si aspetta due valori in ingresso, entrambi di tipo intero, denotati da `(int x, int y)`. Questi sono i *parametri formali* del metodo, specificati mediante una lista di coppie della forma *Tipo ParametroIde*, dove *Tipo* è il tipo (pre-definito in Java o definito dall'utente) del parametro corrispondente, il cui nome è l'identificatore denotato da *ParametroIde*. Occorre ricordare che un metodo può non avere parametri, ma le parentesi vanno sempre scritte (sia in fase in definizione che in fase di chiamata del metodo).

Il corpo della definizione del metodo è una sequenza di dichiarazioni locali al metodo (la variabile intera `z`) ed istruzioni che codificano in Java l'algoritmo della moltiplicazione di due numeri naturali tramite la somma. Il codice del metodo termina con l'istruzione

```
return z;
```

la cui sintassi è data dalla parola riservata `return` seguita da un'espressione il cui tipo è compatibile con quello del risultato del metodo (definito nell'intestazione). Data un'espressione `E`, la semantica dell'istruzione `return E` consiste nel valutare `E`, ottenendo il suo valore v_E, e nel restituire il controllo dell'esecuzione al metodo chiamante passandogli il valore v_E come risultato della chiamata del metodo. Infatti, nel metodo `main`, dopo aver opportunamente convertito i due valori in input ed averli memorizzati nelle variabili `x` ed `y`, si ha la chiamata del metodo `mult` come segue:

```
Mult.mult(x,y)
```

Il metodo `mult` è definito `static`, quindi la sua invocazione si effettua premettendo il nome della classe in cui è definito (seguito da un punto) al nome del metodo. Questa chiamata è (sintatticamente) un'espressione di tipo `int` (il tipo del valore restituito da `mult`) ottenuto applicando il metodo `mult` della classe `Mult` sugli argomenti, detti *parametri attuali*, dati dalle espressioni `x` ed `y` (che sono, in particolare, degli identificatori di variabili).

La *modalità di passaggio dei parametri* in Java è il *passaggio per valore* (in inglese *call-by-value*). Ciò significa che vengono valutati i parametri attuali (nel nostro caso le variabili `x` ed `y` locali al `main`) ed i valori ottenuti vengono associati ai parametri formali[2], rispettando l'ordine in cui tali parametri sono stati definiti.

Quindi, si può compilare il programma tramite

```
javac Mult.java
javac MultTest.java
```

o semplicemente (se i due file sono nella stessa cartella) tramite

```
javac MultTest.java
```

[2]Nell'esempio anche i parametri formali del metodo `mult` sono denotati tramite `x` ed `y`, ma non vanno confusi con le variabili locali del `main`.

ovvero compilando solo il file in cui si trova il `main` (il compilatore provvede a compilare anche gli altri eventuali file da cui dipende il file con il `main` e genera anche i file .class degli altri). Quindi vengono generati i file `Mult.class` e `MultTest.class` e si può eseguire il programma mandando in esecuzione il file .class in cui si trova il `main`. Ad esempio:

```
java MultTest 4 8
32
```

Si ha che le variabili `x` ed `y` locali al `main` sono inizializzate con i valori interi `4` e `8` rispettivamente. Al momento della chiamata, tali valori sono passati al metodo `mult` ed associati ai parametri formali `x` ed `y` di `mult`. Il codice del metodo `mult` viene eseguito tenendo conto di tali legami. Il risultato calcolato e restituito dalla chiamata `Mult.mult(x,y)` viene infine stampato sul video tramite l'esecuzione del metodo `println`.

Possiamo definire altri metodi per la computazione di funzioni matematiche, mettendoli tutti nella stessa classe.[3] Definiamo, ad esempio, l'operazione di esponenziazione tramite la moltiplicazione.

```
public class Arith {
  public static int mult (int x, int y) {
    int z = 0;
    while (y > 0) {
      z = z+x;
      y = y-1;
    }
    return z;
  }

  public static int exp (int x, int y) {
    int z = 1;
    while (y > 0) {
      z = mult(x,z);
      y = y-1;
    }
    return z;
  }
}
```

Poiché il metodo `exp` invoca un metodo definito nella sua stessa classe, è possibile evitare di specificare la classe nella chiamata del metodo `mult`.

```
public class ArithTest {
  public static void main (String[] args) {
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    System.out.println(Arith.exp(x,y));
  }
}
```

```
javac ArithTest.java
java ArithTest 2 10
1024
```

[3] Tutte queste funzioni sono pre-definite in Java e contenute nella classe `Math`.

Esercizi

1. Scrivere un metodo in Java che, dati in ingresso due numeri $m, n \in \mathbb{N}$ tali che $m < n$, stampa tutti i numeri pari compresi tra m ed n, ovvero tutti i numeri x tali che x è pari e $m \leq x \leq n$.

```
public static void stampaPari (int m, int n) {
  int x;
  if (m%2 == 0)
    x = m;
  else
    x = m+1;
  while (x <= n) {
    System.out.println(x);
    x = x+2;
  }
}
```

   Supponendo che il metodo `stampaPari` sia definito nella classe `StampaPari`, un semplice programma di prova è il seguente:

```
public class StampaPariTest {
  public static void main (String[] args) {
    int m = Integer.parseInt(args[0]);
    int n = Integer.parseInt(args[1]);
    StampaPari.stampaPari(m,n);
  }
}
```

   Nota che il tipo del risultato del metodo `stampaPari` è `void`, quindi la chiamata del metodo `StampaPari.stampaPari(m,n);` è un'istruzione.

2. Scrivere un metodo che calcola il fattoriale di un numero naturale `n`. Si ricorda la definizione del fattoriale: $0! = 1$ ed $n! = n \times (n-1) \times \ldots \times 2 \times 1$ per $n \geq 1$. Se al metodo viene passato un numero intero negativo, il metodo restituisce `-1`.

```
public static int fattoriale (int n) {
  if (n<0) return -1;
  int f = 1;
  for (int i=1; i<=n; i++)
    f = f*i;
  return f;
}
```

3. Scrivere un metodo che calcola l'`n`-esimo numero di Fibonacci. Si ricorda la definizione dei numeri di Fibonacci: $fib(0) = 1$, $fib(1) = 1$, $fib(n) = fib(n-1) + fib(n-2)$ per $n \geq 2$. Se al metodo viene passato un numero intero negativo, il metodo restituisce `-1`.

```
public static int fibonacci (int n) {
  if (n<0) return -1;
  int fib, x, temp;
  fib = 1;
  x = 0;
```

```
  for (int i=1; i<=n; i++) {
    temp = x+fib;
    x = fib;
    fib = temp;
  }
  return fib;
}
```

4. Scrivere un metodo che, dato un intero `n`, stampa i primi `n` numeri di Fibonacci (incluso l'`n`-esimo numero).

5. Scrivere un metodo che, dato un intero `n`, stampa i numeri di Fibonacci minori di `n`.

6. Scrivere un semplice programma di prova in cui vengono utilizzati i due metodi definiti ai punti precedenti.