Lezione 9

Proprietà e metodi, un'analisi più approfondita

Diversi concetti legati all'uso delle proprietà e dei metodi di un oggetto ancora devono essere chiariti. Questa lezione prosegue il discorso da dove lo interrompe la precedente, esaminando l'inizializzazione in linea delle proprietà, il sovraccarico dei metodi e dei costruttori, il controllo dell'accesso, i membri statici e numerosi altri argomenti collegati.

9.1 - Inizializzazione in linea delle proprietà

Come rimarcato nella lezione precedente, esistono due maniere per inizializzare le proprietà contenute in una classe:

1. Mediante l'inizializzazione automatica, che attribuisce ad ogni proprietà un valore di default dipendente dal tipo.
2. Inizializzando manualmente ogni singola proprietà dall'interno dei costruttori della classe.

Esiste poi una terza tecnica, spesso conveniente, detta *inizializzazione in linea*. Si osservi il codice che segue:

```
class Rettangolo {  
  
    int larghezza = 150;  
    int altezza = 100;  
  
}
```

Le due proprietà *larghezza* e *altezza*, in questo modo, vengono sempre inizializzate secondo i valori specificati in linea, ogni volta che viene istanziato un oggetto di tipo *Rettangolo*. Le inizializzazioni in linea sono valutate indipendentemente dai costruttori: la presenza di uno o più costruttori non influisce sui valori attribuiti in linea, che restano sempre e comunque validi. Tuttavia, un costruttore può sovrascrivere il valore precedentemente assegnato da una inizializzazione in linea. Questo è reso possibile dal fatto che le inizializzazioni in linea vengono sempre eseguite prima del costruttore:

```
class Rettangolo {  
  
    int larghezza = 150;  
    int altezza = 100;  
  
    Rettangolo() {}  
  
    Rettangolo(int l, int a) {  
        larghezza = l;  
        altezza = a;  
    }  
  
}
```

Richiamando il costruttore privo di argomenti, si ottiene un rettangolo di dimensioni standard,

secondo i dettami delle inizializzazioni in linea. Al contrario, servendosi del secondo costruttore disponibile, è possibile specificare nuovi valori che sovrascriverranno quelli predefiniti:

```
// Rettangolo 150 x 100
Rettangolo r1 = new Rettangolo();
// Rettangolo personalizzato
Rettangolo r1 = new Rettangolo(300, 200);
```

Le inizializzazioni in linea possono essere eseguite con ogni tipo di proprietà:

```
class Esempio {

    int a = 60;
    double b = 5.5;
    char c = 'a';
    boolean d = true;
    String e = "ciao";
    int[] f = new int[15];
    Rettangolo g = new Rettangolo(200, 150);

}
```

9.2 - Proprietà final

La parola chiave *final* può essere applicata alle proprietà presenti in una classe, ottenendo su di esse lo stesso effetto che si riscontra con le comuni variabili:

```
class TestFinal {

    final int i = 6;

}
```

La proprietà *i*, una volta inizializzata, non potrà più essere modificata. Bisogna svolgere un'ulteriore considerazione sulle proprietà *final*: devono necessariamente essere inizializzate esplicitamente, in linea oppure dall'interno dei costruttori della classe. Il seguente codice è pertanto corretto:

```
class TestFinal {

    final int i;

    TestFinal(int n) {
        i = n;
    }

}
```

La classe che segue, invece, contiene un errore:

```
class TestFinal {

    final int i;
```

```
}
```

L'errore che si riscontra, a tempo di compilazione, è trascritto di seguito:

```
variable i might not have been initialized
```

9.3 - Sovraccarico dei metodi

Java consente il *sovraccarico dei metodi*. Un metodo si dice sovraccarico quando ne esistono più versioni che differiscono tra loro almeno in uno degli argomenti accettati. In sostanza, è possibile creare più metodi con lo stesso nome, purché restituiscano sempre il medesimo tipo e differiscano nella lista dei parametri. Il seguente esempio dimostra la caratteristica:

```
class TestSovraccarico {

    void test() {
        System.out.println("Senza argomenti");
    }

    void test(int a) {
        System.out.println("Con un argomento int: " + a);
    }

    void test(double a) {
        System.out.println("Con un argomento double: " + a);
    }

    void test(int a, double b) {
        System.out.println(
            "Con due argomenti int e double: " + a + ", " + b
        );
    }

    void test(String a, boolean b) {
        System.out.println(
            "Con due argomenti String e boolean: " + a + ", " + b
        );
    }

    public static void main(String[] args) {
        TestSovraccarico mioTest = new TestSovraccarico();
        mioTest.test();
        mioTest.test(5);
        mioTest.test(2.3);
        mioTest.test(1, 1.56);
        mioTest.test("ciao", true);
    }
}
```

La classe *TestSovraccarico* contiene il metodo *main()*. Pertanto, dopo la compilazione, può essere avviata e sperimentata. L'output prodotto è mostrato di seguito:

```
Senza argomenti
```

```
Con un argomento int: 5
Con un argomento double: 2.3
Con due argomenti int e double: 1, 1.56
Con due argomenti String e boolean: ciao, true
```

Come è possibile osservare, sono state implementate cinque differenti versioni del metodo `test()`. Nel momento in cui il metodo viene richiamato, il gestore dell'esecuzione decreta automaticamente quale versione attivare, in base al numero e al tipo degli argomenti specificati in linea.

9.4 - Sovraccarico dei costruttori

I costruttori, proprio come i metodi, possono essere sovraccaricati. Sono già stati esaminati diversi esempi di costruttori sovraccaricati. Vi è comunque un'importante precisazione che è d'uopo riportare: un costruttore può ricorrere ad un altro costruttore della medesima classe. Si prenda in considerazione la seguente classe:

```
class Triangolo {

    int lato1;
    int lato2;
    int lato3;

    Triangolo(int a) {
        lato1 = a;
    }

    Triangolo(int a, int b) {
        lato1 = a;
        lato2 = b;
    }

    Triangolo(int a, int b, int c) {
        lato1 = a;
        lato2 = b;
        lato3 = c;
    }

}
```

La classe *Triangolo* contiene ben tre costruttori. Il primo permette di impostare solamente le dimensioni di un lato, il secondo ne imposta due, mentre il terzo può impostare tutti e tre i lati. In realtà, il secondo costruttore contiene implicitamente anche il primo, giacché l'operazione

```
lato1 = a;
```

viene svolta in ambo i casi. Il terzo, addirittura, ingloba tutto il codice degli altri due costruttori:

```
lato1 = a;
lato2 = b;
```

In casi come questo, dove un costruttore esegue come prima cosa le operazioni che possono svolgere altri costruttori con meno compiti, è possibile avvantaggiarsi di un secondo uso della

parola chiave *this*. La classe appena vista è equivalente a quella che segue:

```
class Triangolo {  
  
    int lato1;  
    int lato2;  
    int lato3;  
  
    Triangolo(int a) {  
        lato1 = a;  
    }  
  
    Triangolo(int a, int b) {  
        this(a);  
        lato2 = b;  
    }  
  
    Triangolo(int a, int b, int c) {  
        this(a, b);  
        lato3 = c;  
    }  
  
}
```

Quando si richiama il primo costruttore, tutto avviene secondo la norma:

```
Triangolo t1 = new Triangolo(5);
```

Alla proprietà *lato1*, quindi, sarà normalmente assegnato il valore 5, mentre per *lato2* e *lato3* saranno impiegati i rispetti valori predefiniti (0). Quando si impiega il secondo costruttore, che accetta due argomenti, si impostano le dimensioni di due lati del triangolo. L'assegnazione del valore di *a* alla proprietà *lato1*, ad ogni modo, non viene svolta all'interno del secondo costruttore. Per l'inizializzazione di *lato1*, ci si appella al primo costruttore disponibile, tramite la chiamata:

```
this(a);
```

Il concetto è ancora più evidente quando si passa all'analisi del terzo costruttore. In questo caso, per l'inizializzazione dei primi due lati, ci si appella al secondo costruttore della serie:

```
this(a, b);
```

Il costruttore a due argomenti, a sua volta, si appella al costruttore con un solo argomento. La convenienza di queste invocazioni a catena, in un esempio così spicciolo, non è troppo evidente. I migliori vantaggi sono ottenibili lavorando con costruttori complessi e ricchi di codice.

Ad ogni modo, la chiamata a *this()* deve sempre rappresentare la prima istruzione presente in un costruttore che vuole avvantaggiarsi di tale tecnica. Facendo altrimenti, si commette un errore. Se si fosse strutturato il terzo costruttore della classe *Triangolo* al seguente modo

```
Triangolo(int a, int b, int c) {
```

```
lato3 = c;
this(a, b); // Errato!
}
```

si sarebbe ottenuto l'errore di compilazione sotto riportato:

```
call to this must be first statement in constructor
```

9.5 - Passaggio di parametri e restituzione dei valori di ritorno

Il passaggio dei parametri e la restituzione di un valore di ritorno sono operazioni fondamentali che meritano un maggiore grado di approfondimento. Come per l'assegnazione, anche il trasferimento di informazioni da e verso un metodo agisce diversamente in base al tipo del valore trattato. I tipi predefiniti, al solito, sono trattati per valore, mentre gli oggetti vengono gestiti per riferimento. Ad esempio, quando si fornisce ad un metodo un argomento di tipo *int*, il valore specificato viene letto e copiato in una nuova variabile, creata automaticamente all'interno del blocco di codice collegato al metodo. Modificando la variabile locale creata per lo scopo, dunque, il valore originario non subisce alcuna modifica. Ecco una dimostrazione:

```
class PassaggioPerValore {

    void incrementa(int i) {
        i++;
    }

    public static void main(String[] args) {
        PassaggioPerValore test = new PassaggioPerValore();
        int a = 5;
        test.incrementa(a);
        System.out.println(a);
    }

}
```

L'output mostrato è:

5

L'esempio dimostra come la variabile *i*, interna al metodo *incrementa()*, sia una copia di *a*, e non un suo alias.

Con gli oggetti, il comportamento è diametralmente opposto:

```
class Rettangolo {

    int larghezza;
    int altezza;

    Rettangolo(int l, int a) {
        larghezza = l;
        altezza = a;
    }

}
```

```

}

class PassaggioPerRiferimento {

    void modifica(Rettangolo r) {
        r.larghezza /= 2;
        r.altezza /= 2;
    }

    public static void main(String[] args) {
        PassaggioPerRiferimento test = new PassaggioPerRiferimento();
        Rettangolo rett = new Rettangolo(150, 100);
        test.modifica(rett);
        System.out.println("Larghezza: " + rett.larghezza);
        System.out.println("Altezza: " + rett.altezza);
    }
}

```

L'output mostrato è:

```

Larghezza: 75
Altezza: 50

```

Il parametro *r*, interno al metodo *modifica()* della classe *PassaggioPerRiferimento*, è un alias dell'oggetto *rett* creato nel *main()* del programma. Modificando il parametro, si varia anche il contenuto dell'oggetto ricevuto in gestione, ottenendo ciò che in gergo è chiamato *side effect* (*effetto collaterale*).

Le medesime considerazioni possono essere svolte in merito ai valori di ritorno: i tipi predefiniti vengono restituiti per valore, mentre gli oggetti sono trasmessi per riferimento.

9.6 - Ricorsione

Java supporta le tecniche *ricorsive*. Un metodo si dice ricorsivo quando, in determinate condizioni, richiama se stesso. Il più classico esempio della ricorsione è il calcolo del fattoriale di un intero non negativo. Il fattoriale di un intero *n* è dato dal prodotto di tutti i numeri compresi tra 1 e *n*, estremi inclusi:

1! (si legge "uno fattoriale") = 1

2! = 2 * 1 = 2

3! = 3 * 2 * 1 = 6

4! = 4 * 3 * 2 * 1 = 24

...

$n! = n * (n - 1) * (n - 2) * \dots * 1$

Per convenzione, è stabilito che:

0! = 1

Il seguente metodo, che può essere incluso in una classe qualsiasi, esegue il calcolo in maniera iterativa, mediante un ciclo *for*, senza servirsi della ricorsione:

```
int fattorialeIterativo(int n) {
    int risultato = 1;
    for (int i = 1; i < n; i++)
        risultato *= i;
    return risultato;
}
```

Nel calcolo del fattoriale, tuttavia, è possibile scorgere una proprietà ricorsiva:

1! = 1

2! = 2 * 1!

3! = 3 * 2!

...

n! = n * (n - 1)!

L'aver scorto tale proprietà consente di realizzare il seguente metodo, basato sulla ricorsione:

```
int fattorialeRicorsivo(int n) {
    if (n == 0) return 1;
    else return n * fattorialeRicorsivo(n - 1);
}
```

Le tecniche ricorsive, benché talvolta dispendiose di risorse, sono spesso più eleganti, più ordinate e più facili da concepire, una volta entrati nell'ottica di questa metodologia.

Un'altra nota tecnica ricorsiva consente di ordinare in maniera crescente gli elementi contenuti in un array di interi. La seguente classe ingloba un metodo di ordinamento basato su tale tecnica, dimostrandone inoltre il funzionamento:

```
class OrdinaArray {

    void ordina(int[] a, int start) {
        if (start >= a.length) return;
        int indiceMinimo = start;
        for (int i = start; i < a.length; i++)
            if (a[i] < a[indiceMinimo])
                indiceMinimo = i;
        int temp = a[start];
        a[start] = a[indiceMinimo];
        a[indiceMinimo] = temp;
        // Istruzione ricorsiva.
        ordina(a, start + 1);
    }
}
```

```

public static void main(String[] args) {
    OrdinaArray test = new OrdinaArray();
    int[] mioArray = {
        12, 6, 76, 2, 0, 34, 17
    };
    test.ordina(mioArray, 0);
    for (int i = 0; i < mioArray.length; i++)
        System.out.println(mioArray[i]);
    }
}

```

L'array creato all'interno del metodo *main()* è inizializzato con elementi disposti in ordine sparso. La chiamata a *ordina()* modifica la disposizione degli elementi presenti nell'array, applicando un side effect all'oggetto passato come argomento. Al termine dell'elaborazione, gli elementi del vettore saranno perfettamente ordinati, come dimostra l'output prodotto dal codice:

```

0
2
6
12
17
34
76

```

La tecnica impiegata per l'ordinamento, tutto sommato, è semplice. Si ragiona osservando i passi elencati di seguito:

1. Il primo elemento del vettore viene confrontato con tutti quelli successivi, alla ricerca di valori minori di esso.
2. Terminato il confronto, si prende il più piccolo tra i minori che sono stati individuati (se ne esiste almeno uno) e lo si scambia con il primo elemento del vettore.
3. La tecnica viene eseguita nuovamente (passo ricorsivo), agendo però sul sottovettore che va dal secondo all'ultimo elemento del vettore corrente. Quando il secondo e l'ultimo elemento del vettore corrente coincidono, il lavoro è concluso e la tecnica non deve più essere eseguita nuovamente.

9.7 - Controllo dell'accesso

Lo scopo ed il pregio di classi ed oggetti dovrebbe a questo punto essere evidente. Ciascuna classe fornisce automaticamente una serie di oggetti programmabili, che lo sviluppatore può impiegare nei propri codici in maniera semplice e sicura. Affinché ciò avvenga, quando si sviluppa una classe, è necessario distinguere due categorie di membri (dove con la parola *membri* si intendono sia le proprietà sia i metodi della classe): i membri di interfaccia ed i membri ad uso interno. I primi si rivolgono a quei programmatori che sono utenti della classe, ed il loro scopo è fornire una maniera logica, coerente, semplice e sicura per l'impiego degli oggetti. Un programmatore di questa categoria deve sempre domandarsi cosa faccia una determinata classe, e non come lo faccia. I membri ad uso interno, al contrario, servono allo sviluppatore della classe per realizzare delle routine interne che permettano il corretto funzionamento del codice. Di solito, è necessario tracciare un confine netto tra le due

categorie, per fare in modo che solo l'interfaccia di una classe sia esposta. Il suo delicato contenuto, che altrimenti potrebbe essere sfruttato in maniera non prevista, deve rimanere celato.

Una classe può sommariamente essere paragonata ad un'automobile. Per guidare un'automobile, non è necessario conoscere come funzioni il suo motore: all'utente della vettura viene semplicemente fornita un'interfaccia, costituita da un volante, tre pedali ed una leva del cambio. L'interfaccia permette di gestire il mezzo senza doversi curare di cosa ci sia sotto il cofano. Del motore, che è la parte celata del meccanismo, dovranno interessarsi esclusivamente gli addetti ai lavori. La metafora dell'automobile svela anche un altro importante vantaggio della programmazione orientata agli oggetti: il motore di una vettura può essere sostituito con un modello più efficiente, oppure una sua parte può essere rimpiazzata da un ricambio leggermente diverso da quello originario, ma la maniera di guidare l'automobile, attraverso la sua interfaccia, non subirà modifiche a seguito del cambio. Lo stesso avviene con le classi. Da una parte ci sono le proprietà ed i metodi di interfaccia, che consentono un elegante uso degli oggetti, mentre dall'altro ci sono i membri ad uso interno, ossia il motore della classe. La parte ad uso interno, con il tempo, può essere migliorata per incrementare le prestazioni e l'efficienza del codice, senza però sconvolgere l'interfaccia della classe. Una modifica al cuore interno di una classe, in sostanza, non richiede un aggiornamento a catena di tutti gli altri componenti che ne fanno uso. Per questo, Java è un linguaggio che permette di gestire facilmente i progetti di vaste dimensioni. Allo stesso modo, la protezione posta sul motore della classe fa in modo che in non addetti ai lavori non vadano a toccare, per errore, le parti più delicate del meccanismo. Java è un linguaggio che può garantire la sicurezza e la robustezza delle applicazioni.

Assenza dello specificatore

Prima che il significato delle parole *public* e *private* venisse illustrato, i membri delle classi fornite come esempio non hanno fatto uso di alcuno specificatore di accesso. Questo, in Java, è perfettamente legale. L'assenza dello specificatore causa una norma di accesso simile a quella dettata da *public*, anche se più ristretta. Ai membri privi di specificatore, infatti, possono accedere soltanto gli elementi appartenenti allo stesso *pacchetto* della classe che li contiene. All'esterno del pacchetto corrente, è come se tali membri fossero *private*. Giacché gli esempi mostrati sinora sono stati tutti riposti nella medesima *directory*, e quindi implicitamente nello stesso pacchetto, non è stato possibile ravvisare alcuna differenza sensibile. Questo corso si occuperà dei pacchetti successivamente.

Java prevede quattro differenti tipi di accesso ai membri di una classe, che si realizzano antepoendo degli specificatori di accesso alla dichiarazione di proprietà e metodi. Per il momento, solo due di essi sono rilevanti:

1. ***public***. Quando un membro è dichiarato *public*, è a tutti gli effetti parte dell'interfaccia pubblica della classe. Chiunque, da qualsiasi posizione ed in qualsiasi condizione, potrà avere accesso al membro così etichettato.
2. ***private***. Ossia l'inverso di *public*. I membri privati fanno parte del cuore funzionale della classe, e nessuno può accedervi. Hanno accesso ai membri privati di una classe solamente i codici contenuti all'interno della classe stessa.

Si consideri il seguente esempio:

```
class Rettangolo {  
  
    private int larghezza;  
    private int altezza;  
    private int area;  
  
}
```

```

public Rettangolo(int l, int a) {
    larghezza = l;
    altezza = a;
    calcolaArea();
}

public int getLarghezza() {
    return larghezza;
}

public int getAltezza() {
    return altezza;
}

public int getArea() {
    return area;
}

private void calcolaArea() {
    area = larghezza * altezza;
}
}

```

La classe *Rettangolo*, così definita, dispone di tre proprietà private: *larghezza*, *altezza* e *area*. Nessuna altra classe potrà accedere al contenuto di queste proprietà, ad eccezione della stessa classe *Rettangolo*. Il costruttore, infatti, vi accede liberamente, per fare in modo che le proprietà dell'oggetto possano essere correttamente inizializzate. Dall'esterno della classe, ad ogni modo, non è possibile scrivere:

```

Rettangolo r = new Rettangolo(10, 20);
System.out.println(r.larghezza); // Errato!

```

Si otterrebbe un errore del tipo:

```

larghezza has private access in Rettangolo

```

Ad ogni modo, è possibile leggere il contenuto delle tre proprietà private servendosi dei metodi pubblici *getLarghezza()*, *getAltezza()* e *getArea()*. Questi tre membri lavorano da ponte tra l'interfaccia pubblica e la parte privata della classe: potendo accedere alle proprietà riservate, poiché membri di *Rettangolo*, i tre metodi si assumono la responsabilità di trasferire dei dati riservati all'esterno dell'oggetto. Tramite questo stratagemma, si è fatto in modo che le tre proprietà della classe risultino, in un certo senso, di sola lettura: l'utente della classe, dopo averle inizializzate, può solo recuperarle, ma non modificarle. Non è stato predisposto alcun metodo capace di intervenire in scrittura sulle tre proprietà private della classe.

Il metodo *calcolaArea()* è privato, ed è dunque destinato all'uso interno. Il costruttore lo richiama per inizializzare il valore della proprietà *area*. Dall'esterno della classe, ovviamente, non è possibile accedervi:

```

Rettangolo r = new Rettangolo(10, 20);
r.calcolaArea(); // Errato!

```

L'errore che si riceve è già noto:

```
calcolaArea() has private access in Rettangolo
```

Il costruttore è stato dichiarato *public*. Di solito, i costruttori sono sempre formulati in questo modo, poiché è necessario che le altre classi possano liberamente accedervi. Ad ogni modo, esistono anche delle eccezioni a questa norma. Ad esempio, si potrebbe definire un costruttore privato che gli altri costruttori della classe possono richiamare appellandosi a *this()*.

9.8 - Membri e blocchi statici

Ragionare completamente per oggetti può, in certi casi, essere limitativo. In alcune occasioni, ad esempio, è conveniente fare in modo che alcune proprietà o alcuni metodi possano essere impiegati indipendentemente da un oggetto istanziato. Si supponga, per riagganciarsi ad uno degli esempi precedenti, di avere la necessità di un metodo per il calcolo del fattoriale, da impiegare in più classi simultaneamente. Esistono tre soluzioni per questo problema. La prima consiste nel copiare il codice del metodo all'interno di ciascuna delle classi che ne deve fare uso. Questa è la peggiore soluzione del lotto, giacché completamente in antitesi con le norme della buona progettazione. La seconda soluzione consiste nel realizzare una classe che incapsuli tale metodo:

```
class ClasseFattoriale {  
  
    public int fattoriale(int n) {  
        if (n == 0) return 1;  
        else return n * fattorialeRicorsivo(n - 1);  
    }  
  
}
```

La soluzione è buona, ma nasconde una inefficienza: ogni volta che si deve calcolare un fattoriale, è necessario istanziare un oggetto di tipo *ClasseFattoriale*, con conseguente spreco di tempo e risorse:

```
ClasseFattoriale mioFattoriale = new ClasseFattoriale();  
int risultato = mioFattoriale.fattoriale(5);
```

Per risolvere il problema nel migliore dei modi, conviene impiegare una terza soluzione, basata su una classe contenente un *metodo statico*:

```
class ClasseFattoriale {  
  
    public static int fattoriale(int n) {  
        if (n == 0) return 1;  
        else return n * fattoriale(n - 1);  
    }  
  
}
```

Un metodo statico può essere invocato senza istanziare alcun oggetto, poiché può funzionare

indipendentemente da esso. D'ora in avanti sarà possibile attivare il metodo realizzato scrivendo semplicemente:

```
int risultato = ClasseFattoriale.fattoriale(5);
```

Anziché usare un oggetto istanziato, si è fatto ricorso ad una forma del tipo:

```
NomeClasse.nomeMetodoStatico(argomenti)
```

Grazie a questa tecnica, è possibile realizzare delle collezioni di metodi svincolati da qualsiasi oggetto. Le classi che li contengono, semplicemente, lavorano da raccoglitori. Ad ogni modo, una classe può avere, allo stesso tempo, sia metodi statici sia metodi non statici. Un'importante avvertenza: all'interno del corpo di un metodo statico non è possibile usare la parola chiave *this*, poiché non esiste alcun oggetto di invocazione cui far riferimento. Allo stesso modo, non è possibile accedere ai membri non statici della classe, poiché non esiste un oggetto corrente cui collegarli.

La parola chiave *static* può essere applicata anche alle proprietà:

```
class RaccoltaCostantiMatematiche {  
  
    //...  
    public static final double PI_GRECO = 3.14;  
    // ...  
  
}
```

E' possibile accedere alla proprietà statica sopra presentata digitando:

```
RaccoltaCostantiMatematiche.PI_GRECO
```

Un ulteriore uso della parola chiave *static* riguarda i blocchi statici. I blocchi statici sono collezioni di istruzioni che vengono eseguite in maniera statica, non appena la classe viene caricata in memoria la prima volta:

```
class TestBlocchiStatici {  
  
    public static int test;  
  
    static {  
        int a = 5;  
        int b = 3;  
        test = a * b;  
    }  
  
}
```

La classe mostrata presenta una proprietà statica, *test*, non immediatamente inizializzata. La sua inizializzazione, infatti, avviene dinamicamente, all'interno del blocco statico posto al di sotto di essa. I blocchi statici vengono eseguiti non appena la classe è caricata in memoria, e possono manipolare e configurare gli altri membri statici dell'insieme. Scrivendo

```
System.out.println(TestBlocchiStatici.test);
```

si otterrà il risultato 15, poiché il blocco statico che inizializza la proprietà è stato eseguito prima ancora che l'accesso al membro sia stato soddisfatto. All'interno dei blocchi statici, ovviamente, non è possibile far riferimento ai membri non statici della classe o alla parola chiave *this*.

9.9 - Il metodo `main()`

Giunge finalmente il momento di esplorare l'etimologia del metodo `main()`, che sin dalle prime lezioni di questo corso è stato ampiamente sfruttato ma mai approfonditamente documentato. Tutto quello che è stato detto è che, per convenzione, il metodo `main()` costituisce il punto di avvio di un programma Java. La sua firma è la seguente:

```
public static void main(String[] args)
```

E' possibile osservare che:

1. Il metodo è *public*, poiché deve accedervi un elemento estraneo alla classe, ossia la macchina virtuale che deve avviare il software.
2. Il metodo è *static*, poiché lavora indipendentemente da ogni istanza della classe che lo contiene.
3. Il metodo è *void*, poiché non deve restituire nulla al codice chiamante.

Lo scopo dell'array di stringhe fornito come argomento al metodo, è riportare gli eventuali parametri immessi su riga di comando, durante l'avvio del software. Il seguente esempio può chiarire le idee:

```
class TestMain {  
  
    public static void main(String[] args) {  
        if (args.length == 0)  
            System.out.println(  
                "Nessun argomento sulla linea di comando..."  
            );  
        else {  
            System.out.println(  
                args.length + " argomenti ricevuti:"  
            );  
            for (int i = 0; i < args.length; i++)  
                System.out.println(" - " + args[i]);  
        }  
    }  
}
```

Si compili la classe, e quindi si avvii il programma dal prompt dei comandi con la classica istruzione:

```
java TestMain
```

L'output prodotto è:

```
Nessun argomento sulla linea di comando...
```

Si provi ora alla seguente maniera:

```
java TestMain 12 ciao 7.0 pippo
```

Il risultato sarà differente:

```
4 argomenti ricevuti:  
- 12  
- ciao  
- 7.0  
- pippo
```

L'array di stringhe fornito dalla macchina virtuale al metodo *main()*, in pratica, racchiude gli argomenti suggeriti al programma dalla riga di comando. La stragrande maggioranza dei software non basati su interfaccia a finestre sfrutta questa caratteristica per ottenere degli input dall'utente, reagendo poi diversamente secondo i valori riscontrati.