Lezione 3

giovedì 11 ottobre 2018 15:33

Inizializzazione dei campi statici

Prima possibilità:

```
class A {
    static int x = 10;        // come var. istanza
     :
}
```

Seconda possibilità: usare un blocco static

```
class Esempio {
    ≡
    static {                  // ← viene eseguito una sola volta nel momento in cui la classe viene caricata.
        ≡
    }
}
```

Esempio relativo a blocco static

PA1819_bloccostatic

Metodi statici

Metodi "della classe" che non lavorano sulle singole istanze della classe

I metodi statici

- non hanno il riferimento this all'oggetto implicito
- possono usare solo campi statici e/o altri metodi statici
- possono lavorare su istanze della classe a patto di riceverne il riferimento (o possono restituire oggetti della classe a patto di crearli)

Nota: il metodo main è statico

```
class A {
    int x;
    static int y;

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

```
        A.y = 10;
        A.x = 5;  // errore
         x = 5;   // errore
        A a = new A();
        a.x = 5;  // OK
          ;
    }
}
```

Dentro la classe un metodo statico può essere invocato mediante il suo nome "semplice".

Da fuori la classe un metodo statico può essere invocato con la forma

NomeDellaClasse.nomeDelMetodo()

```
class Veicolo {
    static int contatore;
    int id;
    String proprietario;
    Veicolo (String p) {
        proprietario = p;
        id = ++contatore;
    }
    static int quantiVeicoli() {
        return contatore;
    }
    public static void main(String[] args) {
        Veicolo v1 = new Veicolo("Mario");
        Veicolo v2 = new Veicolo("Gino");
        int q = Veicolo.quantiVeicoli();
        :
    }
}
```

I Package

I package sono dei raggruppamenti di classi

Sono utili per

- raggruppare classi all'interno di unità logiche
- definire uno spazio di nomi
- controllare in modo fine l'accesso a classi, metodi, e variabili.

Per "infilare" una classe all'interno di un package all'inizio del file sorgente dobbiamo scrivere

```
package nomedelpackage;

class A {
    =
    =
    =
}
```

A.java

```
package nomedelpackage;
class B {
    =
    =
    =
}
```

B.java

```
package pack1;
class X {
    =
}
```

il nome completamente qualificato (o completo) della classe è pack1.X

Utile per risolvere "omonimie"

```
package p1;
class A {
    =
}
```

```
package p2;
class A {
    =
}
```

Se scrivo p1.A

p2.A

I package possono avere una struttura annidata

```
package pack1.pack2;
class Y {
    =
}
```

il nome completamente qualificato è pack1.pack2.Y

Se una classe non specifica la propria appartenenza a un package allora appartiene al package di default (il package senza nome)

EVITARE (OK solo per piccoli esempi)

Tutte le classi delle librerie standard appartengono a package.

java.lang	contiene le classi base del linguaggio (per esempio String)
java.io	contiene classi utili a fare ingresso/uscita
java.net	classi utili per accedere alla rete
⋮	

Lo stesso viene fatto nel caso di librerie prodotte da terzi

Convenzione: usare i nomi di dominio al contrario

com.ibm.parser. ...
com.oracle.lib. ...
it.unipi.mylib. ...

Una classe può usare
- tutte le classi del suo stesso package
- le sole classi public di altri package

Per rendere una classe public:

```
public class Esempio {
        ≡
}
```

Per le classi top-level ci sono solo due livelli: public / non public

Esempio:

pack1 pack2

```
public class A {
    pack1.B b = new pack1.B(); // OK
    pack2.C c = new pack2.C(); // OK
    pack2.D d = new pack2.D(); // ERR.
      ⋮
}
```

A.java

```
package pack1;
class B {
    pack1.A a = new pack1.A(); // OK
    pack2.C c = new pack2.C(); // OK
    pack2.D d = new pack2.D(); // ERR
      ⋮
}
```

B.java

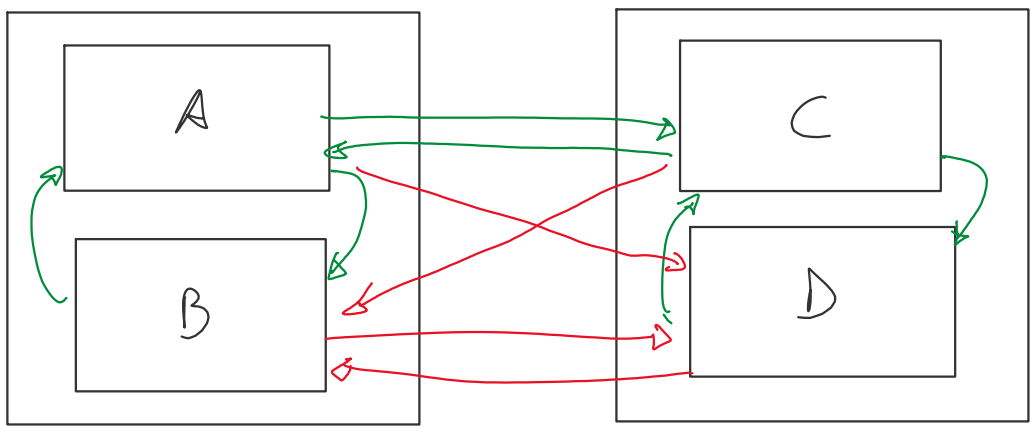
```
package pack2;
public class C {
    pack1.A a = new pack1.A(); // OK
    pack1.B b = new pack1.B(); // ERR
    pack2.D d = new pack2.D(); // OK
      ⋮
}
```

C.java

```
package pack2;
class D {
    pack1.A a = new pack1.A(); // OK
    pack1.B b = new pack1.B(); // ERR
    pack2.C c = new pack2.C(); // OK
      ⋮
}
```

D.java

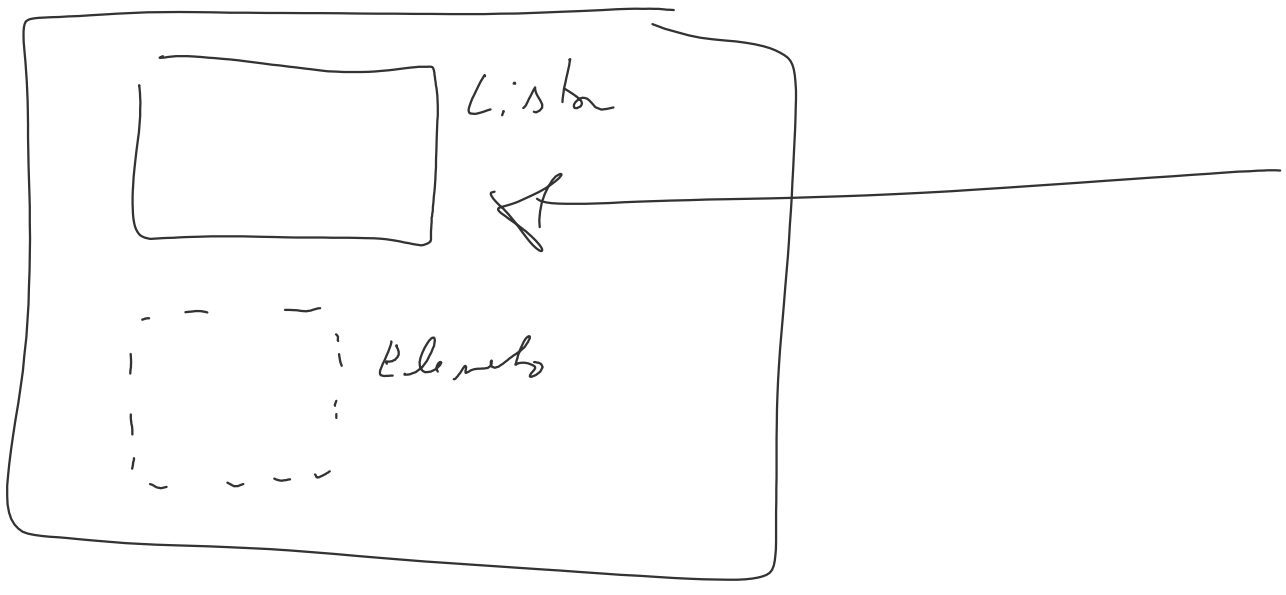
pack1 pack2



Public o non public ?

* se la classe definisce l'"interfaccia" del package con il mondo esterno → public
* se la classe è legata solamente all'implementazione → non

Contesto



L'esempio precedente è "prolisso" (tutti i nomi di classe sono completamente qualificati)

Per classi che sono nello stesso package posso usare il nome semplice

Per le classi di altri package posso usare l'istruzione import

```
import  pack1.A;
import  pack1.*;
```

← tutte le classi del package pack1

Per le classi importate è possibile usare il nome semplice.

```
package  pack1;
import   pack2.C;
public   class A {
        B  b = new B();
        C  c = new C();
```

Basta B perché è nello stesso package

← posso scrivere semplicemente C perché

```
}

package   pack1;
import    pack2.C;
class     B {
        A   a = new  A();
        C   c = new  C();
}

package  pack2;
import   pack1.A;
public  class  C {
        A   a = new  A();
        D   d = new  D();
}

package   pack2;
import    pack1.A;
class     D {
        A   a = new  A();
        C   c = new  C();
}
```

Quando usiamo la forma con * :
attenzione non vengono importati sottopackage

```
package   abc;
.
.
;
package   abc.def;
```

Non importa abc.def

```
import    abc.*;
import    abc.def.*;
```

Le classi del package java.lang
vengono importate automaticamente

In alcuni casi rimangono ambiguità:

```
java.util.Date
java.sql.Date
```

```
import java.util.*;
import java.sql.*;
  :
Date d = new Date();   Quale delle due?
```

```
java.util.Date d = ...
```

↖ In questi casi usare il nome completamente qualificato

Modificatori di accesso

Possono essere applicati ai metodi e alle variabili di una classe in modo da regolarne l'accesso.

Modificatori:

private	il metodo (o la variabile) può essere acceduto solo dalla stessa classe
no modificatore (friendly)	il metodo (o la variabile) può essere acceduto dalla stessa classe o da altre classi nello stesso package.
protected	il metodo (o la variabile) può essere acceduto dalla stessa classe, da altre classi nello stesso package, da sottoclassi in altro package.
public	il metodo (o la variabile) può essere acceduto da tutte le classi

```
package pack1;

public class Esempio {
        private int v1;
        int v2;
        protected int v3;
        public int v4;
        private void m1() {--}
        void m2() {...}
        protected void m3() {...}
        public void m4() {...}
}
```

```
package pack1;
class Uso1 {
        :
        Esempio e = new Esempio();
        e.v1 = ...;   // Errore
        e.v2 = ...;   // OK
        e.v3 = ...;   // OK
        e.v4 = ...;   // OK
        e.m1();       // Errore
        e.m2();       // OK
        e.m3();       // OK
        e.m4();       // OK
        :
}
```

```
package pack2;
import pack1.Esempio;
class Uso2 {
        :
        Esempio e = new Esempio();
        e.v1 = ...;   // Errore
        e.v2 = ...;   // Errore
        e.v3 = ...;   // Errore
        e.v4 = ...;   // OK
        e.m1();       // Errore
        e.m2();       // Errore
        e.m3();       // Errore
        e.m4();
```

```
}
```

Il costruttore di default è public

È possibile applicare i modificatori anche ai costruttori.

Vediamo un esempio di costruttore privato per realizzare il Singleton pattern*

* Di una data classe deve esistere una (ed una sola) istanza.

```
public class MiaClasse {
    private static MiaClasse o;

    private MiaClasse () {
    }
    public static MiaClasse dammiSingleton () {
        if (o == null)
            o = new MiaClasse();
        return o;
    }
    ;
}
```

R istanziazione tardiva (lazy)

```
class Prova {
    =
    MiaClasse m = MiaClasse.dammiSingleton();
    m. ...
    =
}
```

Classi e file

Soluzione più semplice (e da preferire)

- ogni classe in un file separato
- il file ha lo stesso nome della classe

```
public class MiaClasse {
        ≡                          --> MiaClasse.java
}
```

```
public class Prova {
        ≡                          --> Prova.java
}
```

```
public class Principale {
        ≡
    Secondaria ...
        ≡
}
public class Secondaria {
        ≡
}
```

Principale.java

Secondaria.java

$ javac Principale.java

Il compilatore comincia a compilare Principale.java e a un certo punto serve il tipo Secondaria

* se Secondaria.class non è presente: compila Secondaria.java e poi riprende a compilare Principale.java
* se Secondaria.class è presente: controlla la data di Secondaria.java e Secondaria.class Se la data di Secondaria.java è successiva ricompila Secondaria, altrimenti usa Secondaria.class

In un singolo file possono essere presenti le definizioni di più classi.

Vincoli:

- al più una delle classi può essere public
-

lo stesso nome della classe public.

```
class A {
    ≡
}
class B {
    ≡
}
public class C {
    ≡
}
```

C.java

↑ Può andare bene solo se A e B vengono usate esclusivamente da C.

Supponiamo di avere

```
public class D {
    A a ...
}
```

D.java

$ javac D.java
il compilatore cerca A.class
se non la trova prova a compilare A.java
non lo trova → errore

```
class X {
    ≡
}
class Y {
    ≡
}
```

Peppino.java →

javac Peppino.java
↓
X.class e Y.class

Se le classi appartengono a package:

```
package it.unipi.pack1;
public class A {
    ==
}
```

→ A.java

Creare una struttura di cartelle che replica la struttura del package

```
it/
  └→ unipi/
       └→ pack1/
            └→ A.java
```

Da riga di comando lavorare nella cartella che contiene la cartella it

```
$ javac it/unipi/pack1/A.java
```

viene prodotto A.class nella cartella it/unipi/pack1

Se A contiene un main

```
$ java it.unipi.pack1.A
```

l'interprete cerca A.class in una cartella pack1 contenuta in una cartella unipi, contenuta in una cartella it.
