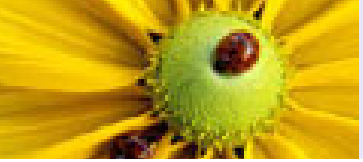
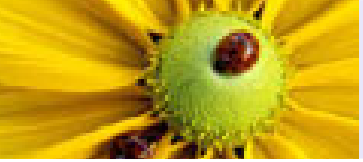
Linguaggi di Programmazione I

Prof. Marco Faella
mailto://m.faella@unina.it
http://wpage.unina.it/mfaella
Materiale didattico elaborato con i Proff. Sette e Bonatti

30 marzo 2023

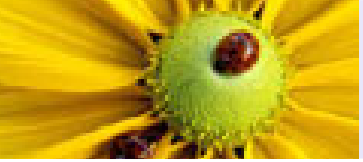


Modificatori



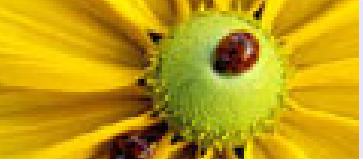
Modificatori

- I modificatori sono parole riservate che danno al compilatore informazioni sulla natura del codice, dei dati, delle classi



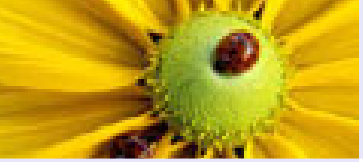
Modificatori

- I modificatori sono parole riservate che danno al compilatore informazioni sulla natura del codice, dei dati, delle classi
- Un gruppo di modificatori, detti *di accesso*, specificano in quali contesti lessicali è visibile quell'elemento

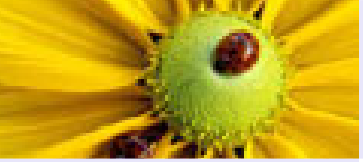


Modificatori

- I modificatori sono parole riservate che danno al compilatore informazioni sulla natura del codice, dei dati, delle classi
- Un gruppo di modificatori, detti *di accesso*, specificano in quali contesti lessicali è visibile quell'elemento
- Altri modificatori possono essere usati, in combinazione con i precedenti, per qualificare quell'elemento



Modificatori di accesso

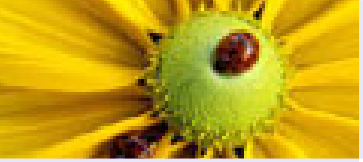


Generalità

I modificatori di accesso sono:

- `public`
- `protected`
- `private`

Inoltre, se non è presente alcun modificatore, parleremo di "accesso di default"



Generalità

I modificatori di accesso sono:

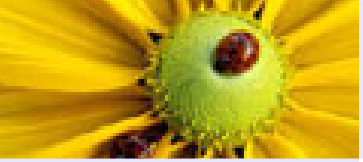
- `public`
- `protected`
- `private`

Inoltre, se non è presente alcun modificatore, parleremo di "accesso di default"

I seguenti elementi possono essere dotati di modificatore di accesso:

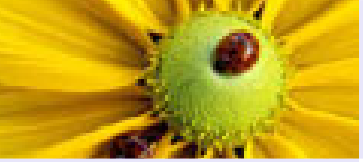
- Classi (anche interfacce ed enumerazioni)
- Attributi
- Metodi e costruttori

In particolare, le variabili locali non supportano modificatori di accesso, perché la loro visibilità è già limitata al blocco corrente



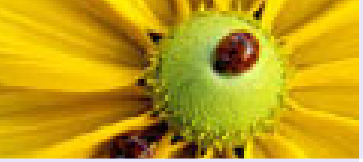
public

- È il modificatore più generoso. Accesso consentito ovunque



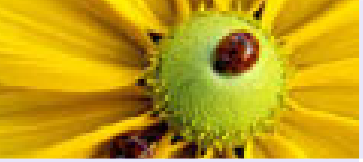
public

- È il modificatore più generoso. Accesso consentito ovunque
- Attenzione: l'accesso ad un attributo o un metodo `public` è subordinato all'accesso alla classe che lo contiene



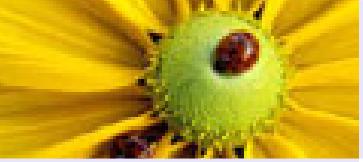
`private` (1)

- È il modificatore meno generoso



`private` (1)

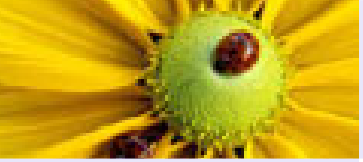
- È il modificatore meno generoso
- Può essere usato solo per attributi o metodi, non per classi top-level



private (1)

- È il modificatore meno generoso
- Può essere usato solo per attributi o metodi, non per classi top-level

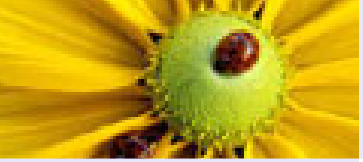
```
public class Complesso {
  private double reale, immag;

  public Complesso(double r, double i) {
    reale=r; immag=i;
  }
  public Complesso add(Complesso c) {
    return new Complesso(reale + c.reale,
                         immag + c.immag);
  }
}
```



private (1)

- È il modificatore meno generoso
- Può essere usato solo per attributi o metodi, non per classi top-level

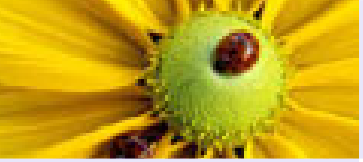
```
public class Complesso {
  private double reale, immag;

  public Complesso(double r, double i) {
    reale=r; immag=i;
  }
  public Complesso add(Complesso c) {
    return new Complesso(reale + c.reale,
                         immag + c.immag);
  }
}

public class Cliente {
  void usali() {
    Complesso c1 = new Complesso(1, 2);
    Complesso c2 = new Complesso(3, 4);
    Complesso c3 = c1.add(c2);
```



private (1)

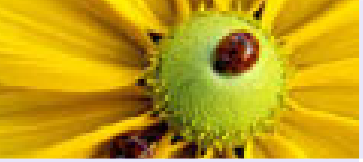
- È il modificatore meno generoso
- Può essere usato solo per attributi o metodi, non per classi top-level

```
public class Complesso {
  private double reale, immag;

  public Complesso(double r, double i) {
    reale=r; immag=i;
  }
  public Complesso add(Complesso c) {
    return new Complesso(reale + c.reale,
                         immag + c.immag);
  }
}

public class Cliente {
  void usali() {
    Complesso c1 = new Complesso(1, 2);
    Complesso c2 = new Complesso(3, 4);
    Complesso c3 = c1.add(c2);
    double d = c3.reale;                  // Illegale
  }
}
```



`private` (2)

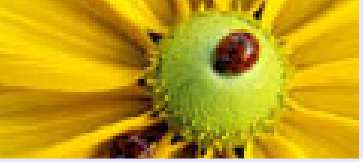
- Le variabili private possono essere nascoste anche allo stesso oggetto che le possiede



private (2)

- Le variabili private possono essere nascoste anche allo stesso oggetto che le possiede

```
class Complesso {
  private double reale, immag;
}

class SubComplesso extends Complesso {
  SubComplesso(double r, double i) {
    reale = r;                          // Illegale
  }
}
```

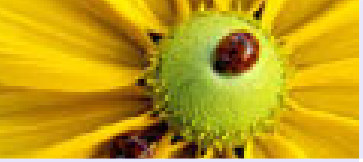


private (2)

- Le variabili private possono essere nascoste anche allo stesso oggetto che le possiede

```
class Complesso {
  private double reale, immag;
}

class SubComplesso extends Complesso {
  SubComplesso(double r, double i) {
    reale = r;                          // Illegale
  }
}
```

- La classe SubComplesso eredita gli attributi della superclasse, MA quegli attributi possono essere usati solo dal codice della classe Complesso

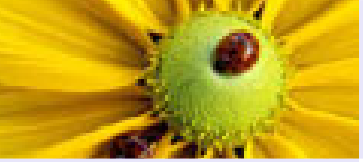


private (2)

- Le variabili private possono essere nascoste anche allo stesso oggetto che le possiede

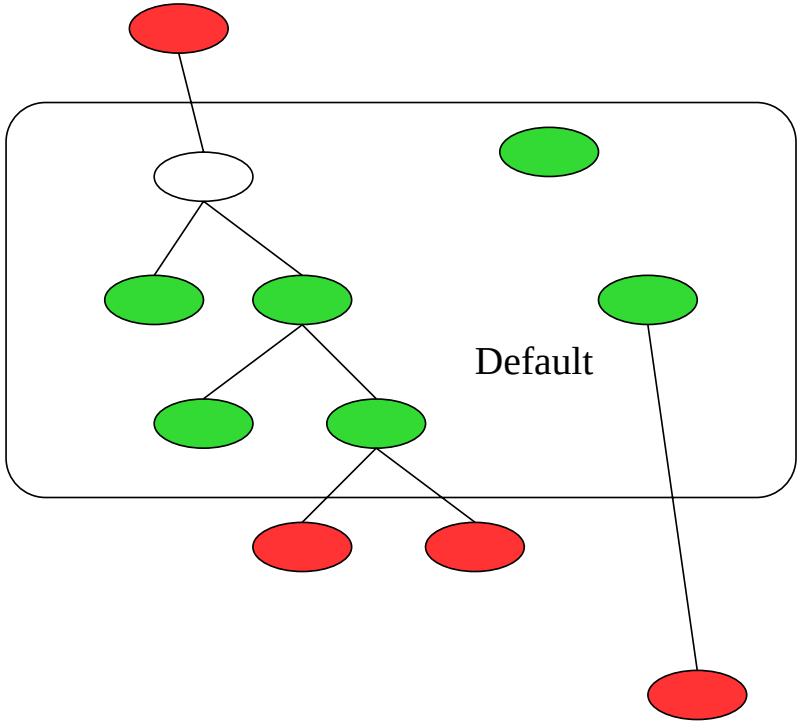
```
class Complesso {
  private double reale, immag;
}

class SubComplesso extends Complesso {
  SubComplesso(double r, double i) {
    reale = r;                          // Illegale
  }
}
```

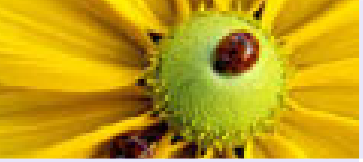
- La classe SubComplesso eredita gli attributi della superclasse, MA quegli attributi possono essere usati solo dal codice della classe Complesso



Default

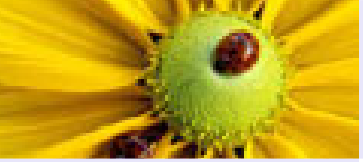
- In mancanza di un modificatore di accesso, l'elemento è visibile al codice che si trova nello stesso pacchetto





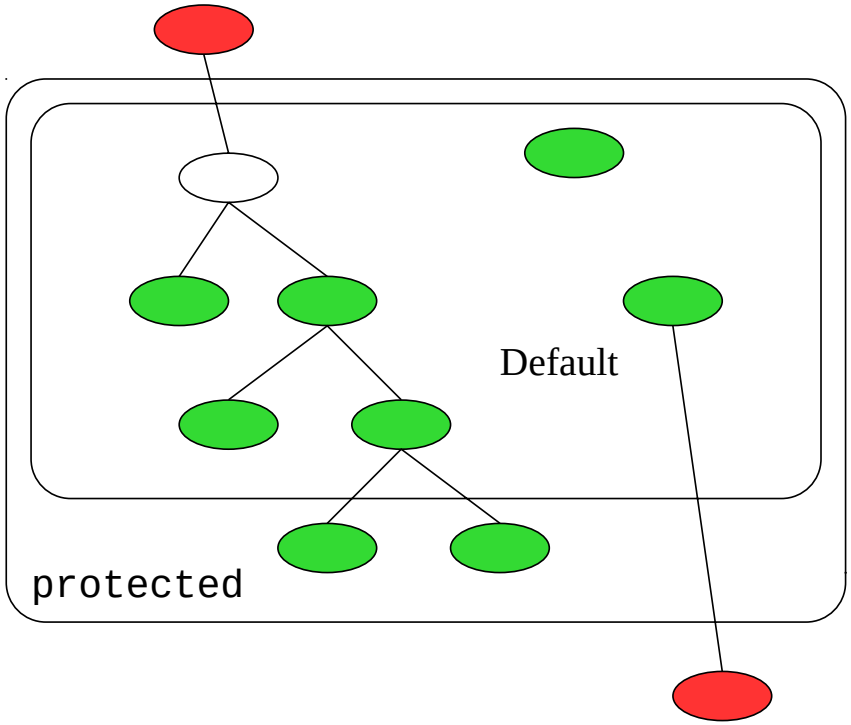
`protected` (1)

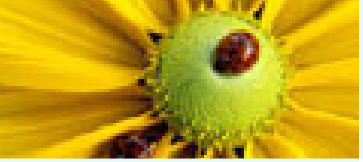
- Accesso consentito al codice dello stesso pacchetto e...



`protected` (1)

- Accesso consentito al codice dello stesso pacchetto e...
- ...a tutte le sottoclassi in pacchetti diversi, *ma solo per ereditarietà*
- Una sottoclasse in un altro pacchetto può accedere ad un membro `protected` della superclasse solo attraverso un riferimento ad un oggetto del proprio tipo (come `this`), o di un suo sottotipo.



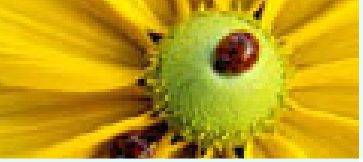


protected (2)

Esempio:

```
package p1;

public class A {
  protected int x = 7;
}
```

```
package p2;
import p1.A;

public class B extends A {
  public void testProt() {
    System.out.println(x); // Ok: x e' ereditato
    // uso implicito di this
```

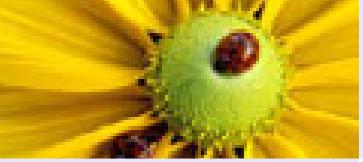


protected (2)

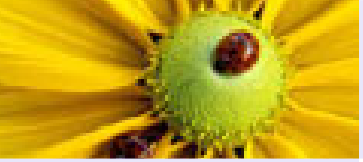
Esempio:

```
package p1;

public class A {
  protected int x = 7;
}
```

```
package p2;
import p1.A;

public class B extends A {
  public void testProt() {
    System.out.println(x); // Ok: x e' ereditato
    // uso implicito di this
    A a = new A();
    System.out.println(a.x); // Errore di comp.
    // a non e' di tipo B, ne' di sottotipo di B.
    B b = new B();
    System.out.println(b.x); // Ok: accesso
    // attraverso oggetto di tipo B.
  }
}
```

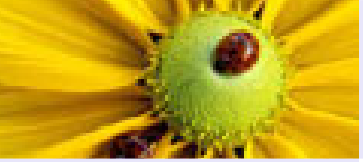


Altri modificatori



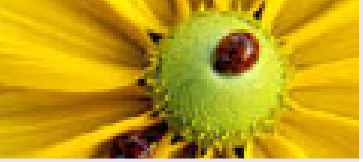
final

- Si applica a classi, metodi e variabili (non a costruttori).



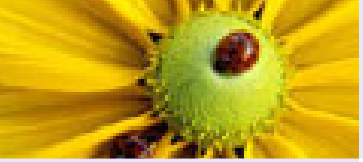
final

- Si applica a classi, metodi e variabili (non a costruttori).
- Il significato varia da contesto a contesto, ma l'essenza è: un elemento `final` non può essere modificato.



final

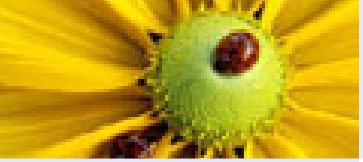
- Si applica a classi, metodi e variabili (non a costruttori).
- Il significato varia da contesto a contesto, ma l'essenza è: un elemento `final` non può essere modificato.
  - Una classe `final` non può essere estesa, cioè non può avere sottoclassi.



final

- Si applica a classi, metodi e variabili (non a costruttori).
- Il significato varia da contesto a contesto, ma l'essenza è: un elemento `final` non può essere modificato.
  - Una classe `final` non può essere estesa, cioè non può avere sottoclassi.
  - Una variabile `final` è una costante, cioè può solo essere inizializzata.

```
void f(final int n) {
   final int m = n;
   n = 0;          // Err. di comp.
   m = 1;          // Err. di comp.
   final int a;
   a = 2;
}
```

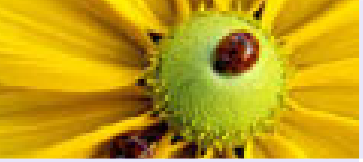


final

- Si applica a classi, metodi e variabili (non a costruttori).
- Il significato varia da contesto a contesto, ma l'essenza è: un elemento `final` non può essere modificato.
  - Una classe `final` non può essere estesa, cioè non può avere sottoclassi.
  - Una variabile `final` è una costante, cioè può solo essere inizializzata.

```
void f(final int n) {
   final int m = n;
   n = 0;          // Err. di comp.
   m = 1;          // Err. di comp.
   final int a;
   a = 2;
}
```

  - Un metodo `final` non può essere sovrascritto (overridden) in una sottoclasse

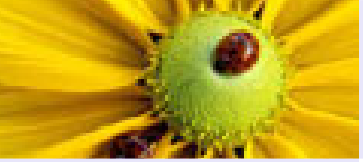


final

- Si applica a classi, metodi e variabili (non a costruttori).
- Il significato varia da contesto a contesto, ma l'essenza è: un elemento `final` non può essere modificato.
  - Una classe `final` non può essere estesa, cioè non può avere sottoclassi.
  - Una variabile `final` è una costante, cioè può solo essere inizializzata.

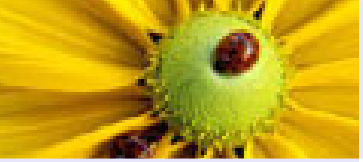
```
void f(final int n) {
   final int m = n;
   n = 0;          // Err. di comp.
   m = 1;          // Err. di comp.
   final int a;
   a = 2;
}
```

  - Un metodo `final` non può essere sovrascritto (overridden) in una sottoclasse
  - Non ha senso attribuire `final` a un costruttore, perché esso non è ereditato dalle sottoclassi e quindi non è mai sovrascrivibile



final

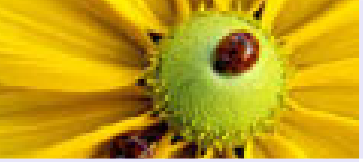
- Attenzione: un riferimento `final` ad un oggetto non può essere modificato (cioè riassegnato ad un altro oggetto), ma l'oggetto a cui esso fa riferimento sì!



final

- Attenzione: un riferimento `final` ad un oggetto non può essere modificato (cioè riassegnato ad un altro oggetto), ma l'oggetto a cui esso fa riferimento sì!

```
final Impiegato luca = new Impiegato("Luca", 1500);
luca = new Impiegato("Luca", 1600); // Err. di comp.
luca.setSalario(1600);                 // OK
```



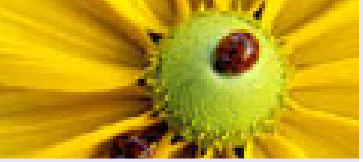
final

- Attenzione: un riferimento `final` ad un oggetto non può essere modificato (cioè riassegnato ad un altro oggetto), ma l'oggetto a cui esso fa riferimento sì!

```
final Impiegato luca = new Impiegato("Luca", 1500);
luca = new Impiegato("Luca", 1600); // Err. di comp.
luca.setSalario(1600);                 // OK
```

- Attenzione: un array `final` non può essere riassegnato, ma il contenuto dell'array può essere modificato!

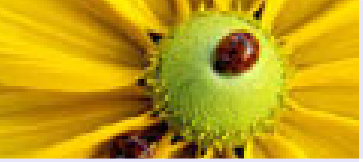
```
final int[] numeri = new int[10];
numeri = new int[20]; // Err. di comp.
numeri[0] = 77;       // OK
```



abstract

- Si applica solo a classi e a metodi.
- Un metodo `abstract` non possiede corpo (";" invece di "`{...}`"):

```
abstract void getValore();
```

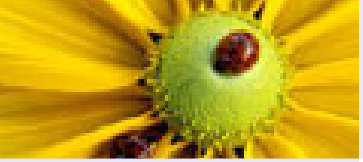


abstract

- Si applica solo a classi e a metodi.
- Un metodo `abstract` non possiede corpo (";" invece di "`{...}`"):

```
abstract void getValore();
```

- Una classe DEVE essere marcata `abstract` se:
  - essa contiene almeno un metodo `abstract`, oppure

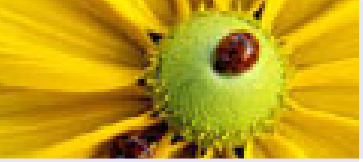


abstract

- Si applica solo a classi e a metodi.
- Un metodo `abstract` non possiede corpo (";" invece di "`{...}`"):

```
abstract void getValore();
```

- Una classe DEVE essere marcata `abstract` se:
  - essa contiene almeno un metodo `abstract`, oppure
  - essa eredita almeno un metodo `abstract` per il quale non fornisce una realizzazione, oppure

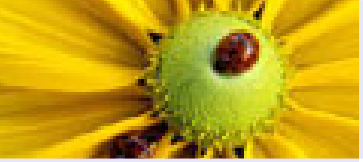


abstract

- Si applica solo a classi e a metodi.
- Un metodo `abstract` non possiede corpo (";" invece di "{...}"):

```
abstract void getValore();
```

- Una classe DEVE essere marcata `abstract` se:
  - essa contiene almeno un metodo `abstract`, oppure
  - essa eredita almeno un metodo `abstract` per il quale non fornisce una realizzazione, oppure
  - essa dichiara di implementare una interfaccia [vedremo in seguito], ma non fornisce una realizzazione di tutti i metodi di quell'interfaccia.

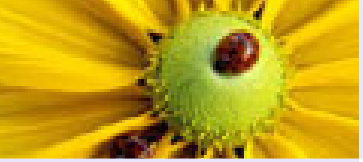


abstract

- Si applica solo a classi e a metodi.
- Un metodo `abstract` non possiede corpo (";" invece di "`{...}`"):

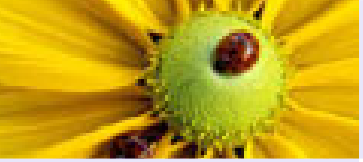
```
abstract void getValore();
```

- Una classe DEVE essere marcata `abstract` se:
  - essa contiene almeno un metodo `abstract`, oppure
  - essa eredita almeno un metodo `abstract` per il quale non fornisce una realizzazione, oppure
  - essa dichiara di implementare una interfaccia [vedremo in seguito], ma non fornisce una realizzazione di tutti i metodi di quell'interfaccia.
- Qualsiasi classe PUÒ essere marcata `abstract`, anche se non contiene metodi astratti. Il compilatore impedirà di istanziarla.
- In un certo senso, `abstract` è opposto a `final`: una classe (o metodo) `final` non può essere specializzata; una classe (o metodo) `abstract` esiste solo per essere specializzata.



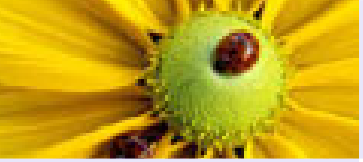
static

- Si applica ad attributi, metodi ed anche a blocchi di codice esterni ai metodi



static

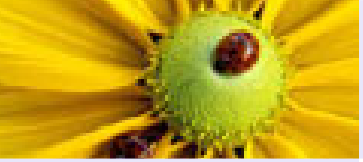
- Si applica ad attributi, metodi ed anche a blocchi di codice esterni ai metodi
- In generale, una caratteristica `static` appartiene alla classe, non alle singole istanze: essa è unica, indipendentemente dal numero (anche zero) di istanze di quella classe.



- Attributi `static`

- L'inizializzazione di un attributo `static` avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {
  static int x = 0;
  Ecstatic () { x++; }
}
```

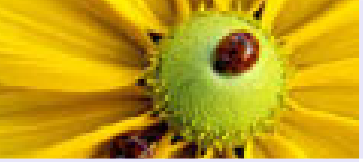


- Attributi `static`

- L'inizializzazione di un attributo `static` avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {
  static int x = 0;
  Ecstatic () { x++; }
}
```

- L'accesso ad un attributo `static` di una classe può avvenire (con la dot-notation):
  - o partendo da un riferimento ad una istanza di quella classe,

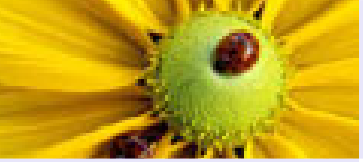


- Attributi `static`

- L'inizializzazione di un attributo `static` avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {
  static int x = 0;
  Ecstatic () { x++; }
}
```

- L'accesso ad un attributo `static` di una classe può avvenire (con la dot-notation):
  - o partendo da un riferimento ad una istanza di quella classe,
  - o partendo dal nome stesso della classe.



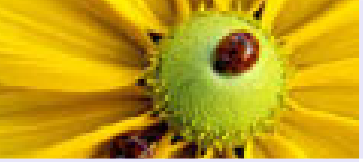
- Attributi `static`

- L'inizializzazione di un attributo `static` avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {
  static int x = 0;
  Ecstatic () { x++; }
}
```

- L'accesso ad un attributo `static` di una classe può avvenire (con la dot-notation):
  - o partendo da un riferimento ad una istanza di quella classe,
  - o partendo dal nome stesso della classe.

```
System.out.println(Ecstatic.x);
```



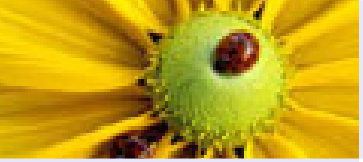
- Attributi `static`

- L'inizializzazione di un attributo `static` avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {
  static int x = 0;
  Ecstatic () { x++; }
}
```

- L'accesso ad un attributo `static` di una classe può avvenire (con la dot-notation):
  - o partendo da un riferimento ad una istanza di quella classe,
  - o partendo dal nome stesso della classe.

```
System.out.println(Ecstatic.x);
Ecstatic e = new Ecstatic();
e.x = 100;
```



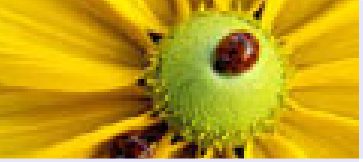
- Attributi `static`

- L'inizializzazione di un attributo `static` avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {
  static int x = 0;
  Ecstatic () { x++; }
}
```

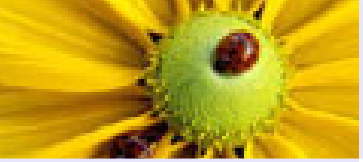
- L'accesso ad un attributo `static` di una classe può avvenire (con la dot-notation):
  - o partendo da un riferimento ad una istanza di quella classe,
  - o partendo dal nome stesso della classe.

```
System.out.println(Ecstatic.x);
Ecstatic e = new Ecstatic();
e.x = 100;
Ecstatic.x = 100;
```



- Metodi `static`

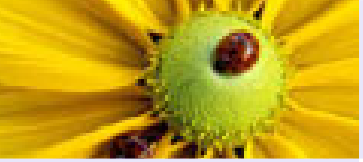
- Appartengono alla classe e non alle singole istanze



- Metodi `static`

- Appartengono alla classe e non alle singole istanze
- Si possono invocare a partire dal nome della classe:

```
class Test {
   public static void f() { ... }
}
...
Test.f();
```

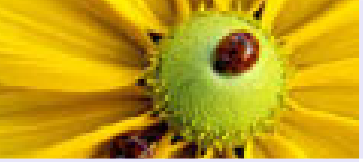


- Metodi `static`

- Appartengono alla classe e non alle singole istanze
- Si possono invocare a partire dal nome della classe:

```
class Test {
   public static void f() { ... }
}
...
Test.f();
```

- Non posseggono il riferimento `this`

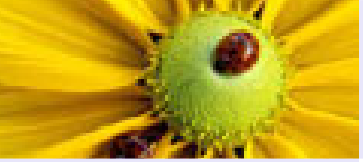


- Metodi `static`

- Appartengono alla classe e non alle singole istanze
- Si possono invocare a partire dal nome della classe:

```
class Test {
   public static void f() { ... }
}
...
Test.f();
```

- Non posseggono il riferimento `this`
- Esempio: il metodo `main`

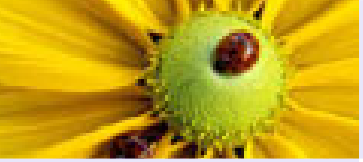


- Metodi `static`

- Appartengono alla classe e non alle singole istanze
- Si possono invocare a partire dal nome della classe:

```
class Test {
   public static void f() { ... }
}
...
Test.f();
```

- Non posseggono il riferimento `this`
- Esempio: il metodo `main`
- Hanno binding statico e non possono essere sovrascritti (overridden)

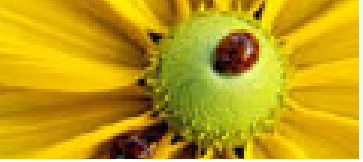


- Metodi `static`

- Appartengono alla classe e non alle singole istanze
- Si possono invocare a partire dal nome della classe:

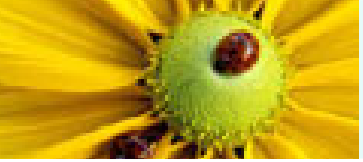
```
class Test {
   public static void f() { ... }
}
...
Test.f();
```

- Non posseggono il riferimento `this`
- Esempio: il metodo `main`
- Hanno binding statico e non possono essere sovrascritti (overridden)



Esempio di binding

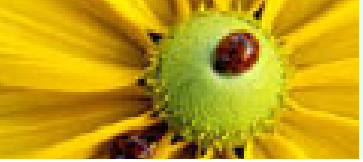
```
class A {

  public static void f() {
    System.out.println("f di A");
  }

  public void g() {
    System.out.println("g di A");
  }
}


class B extends A {

  // mascheramento, non overriding
  public static void f() {
    System.out.println("f di B");
  }

  @Override
  public void g() {
    System.out.println("g di B");
  }
}
```



Esempio di binding

```
class A {

  public static void f() {
    System.out.println("f di A");
  }

  public void g() {
    System.out.println("g di A");
  }
}


class B extends A {

  // mascheramento, non overriding
  public static void f() {
    System.out.println("f di B");
  }

  @Override
  public void g() {
    System.out.println("g di B");
  }
}
```

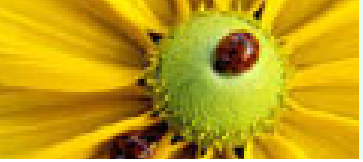
```
class C {
 public static void main(...){
    A a = new A();
    B b = new B();
    A x = b;

    /* binding statico */
    A.f();  /* stampa "f di A" */
    B.f();  /* stampa "f di B" */
    a.f();
```



Esempio di binding

```
class A {

  public static void f() {
    System.out.println("f di A");
  }

  public void g() {
    System.out.println("g di A");
  }
}


class B extends A {

  // mascheramento, non overriding
  public static void f() {
    System.out.println("f di B");
  }

  @Override
  public void g() {
    System.out.println("g di B");
  }
}
```

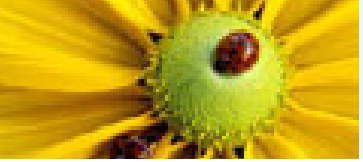
```
class C {
 public static void main(...){
    A a = new A();
    B b = new B();
    A x = b;

    /* binding statico */
    A.f();  /* stampa "f di A" */
    B.f();  /* stampa "f di B" */
    a.f();  /* stampa "f di A" */
    b.f();
```



Esempio di binding

```
class A {

  public static void f() {
    System.out.println("f di A");
  }

  public void g() {
    System.out.println("g di A");
  }
}


class B extends A {

  // mascheramento, non overriding
  public static void f() {
    System.out.println("f di B");
  }

  @Override
  public void g() {
    System.out.println("g di B");
  }
}
```

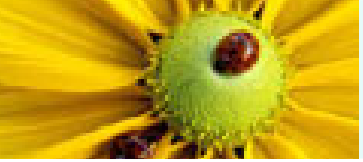
```
class C {
 public static void main(...){
    A a = new A();
    B b = new B();
    A x = b;

    /* binding statico */
    A.f();  /* stampa "f di A" */
    B.f();  /* stampa "f di B" */
    a.f();  /* stampa "f di A" */
    b.f();  /* stampa "f di B" */
    x.f();
```



Esempio di binding

```
class A {

  public static void f() {
    System.out.println("f di A");
  }

  public void g() {
    System.out.println("g di A");
  }
}


class B extends A {

  // mascheramento, non overriding
  public static void f() {
    System.out.println("f di B");
  }

  @Override
  public void g() {
    System.out.println("g di B");
  }
}
```

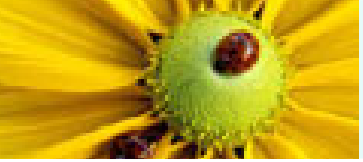
```
class C {
 public static void main(...){
    A a = new A();
    B b = new B();
    A x = b;

    /* binding statico */
    A.f();  /* stampa "f di A" */
    B.f();  /* stampa "f di B" */
    a.f();  /* stampa "f di A" */
    b.f();  /* stampa "f di B" */
    x.f();  /* stampa "f di A" */

    /* binding dinamico */
    a.g();
```



Esempio di binding

```
class A {

  public static void f() {
    System.out.println("f di A");
  }

  public void g() {
    System.out.println("g di A");
  }
}


class B extends A {

  // mascheramento, non overriding
  public static void f() {
    System.out.println("f di B");
  }

  @Override
  public void g() {
    System.out.println("g di B");
  }
}
```

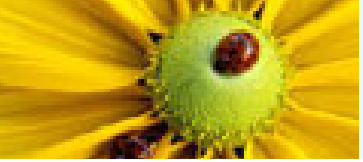
```
class C {
 public static void main(...){
    A a = new A();
    B b = new B();
    A x = b;

    /* binding statico */
    A.f();  /* stampa "f di A" */
    B.f();  /* stampa "f di B" */
    a.f();  /* stampa "f di A" */
    b.f();  /* stampa "f di B" */
    x.f();  /* stampa "f di A" */

    /* binding dinamico */
    a.g();  /* stampa "g di A" */
    b.g();
```



Esempio di binding

```
class A {

  public static void f() {
    System.out.println("f di A");
  }

  public void g() {
    System.out.println("g di A");
  }
}


class B extends A {

  // mascheramento, non overriding
  public static void f() {
    System.out.println("f di B");
  }

  @Override
  public void g() {
    System.out.println("g di B");
  }
}
```

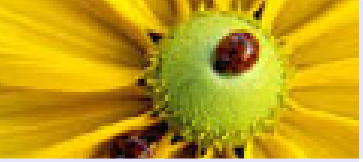
```
class C {
 public static void main(...){
    A a = new A();
    B b = new B();
    A x = b;

    /* binding statico */
    A.f();  /* stampa "f di A" */
    B.f();  /* stampa "f di B" */
    a.f();  /* stampa "f di A" */
    b.f();  /* stampa "f di B" */
    x.f();  /* stampa "f di A" */

    /* binding dinamico */
    a.g();  /* stampa "g di A" */
    b.g();  /* stampa "g di B" */
    x.g();
```



Esempio di binding

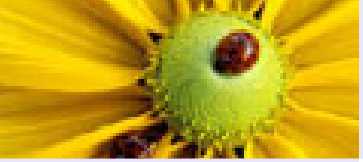
```
class A {

  public static void f() {
    System.out.println("f di A");
  }

  public void g() {
    System.out.println("g di A");
  }
}


class B extends A {

  // mascheramento, non overriding
  public static void f() {
    System.out.println("f di B");
  }

  @Override
  public void g() {
    System.out.println("g di B");
  }
}
```

```
class C {
 public static void main(...){
    A a = new A();
    B b = new B();
    A x = b;

    /* binding statico */
    A.f();  /* stampa "f di A" */
    B.f();  /* stampa "f di B" */
    a.f();  /* stampa "f di A" */
    b.f();  /* stampa "f di B" */
    x.f();  /* stampa "f di A" */

    /* binding dinamico */
    a.g();  /* stampa "g di A" */
    b.g();  /* stampa "g di B" */
    x.g();  /* stampa "g di B" */
  }
}
```



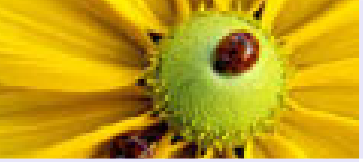
- Blocchi di inizializzazione `static`

- Una classe può contenere blocchi di codice marcati `static`



- Blocchi di inizializzazione `static`

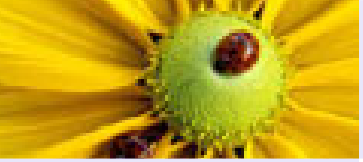
- Una classe può contenere blocchi di codice marcati `static`
- Tali blocchi sono eseguiti una sola volta, nell'ordine in cui compaiono, quando la classe viene caricata in memoria



- Blocchi di inizializzazione `static`

- Una classe può contenere blocchi di codice marcati `static`
- Tali blocchi sono eseguiti una sola volta, nell'ordine in cui compaiono, quando la classe viene caricata in memoria

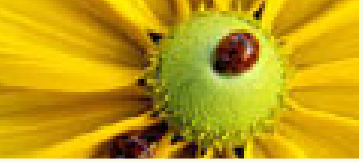
```
public class EsempioStatic {
  private static double d=1.23;

  static {
    System.out.println("Primo blocco static: d=" + d++);
  }

  public static void main(String[] args) {
    System.out.println("main: d=" + d++);
  }

  static {
    System.out.println("Secondo blocco static: d=" + d++);
  }
}
```



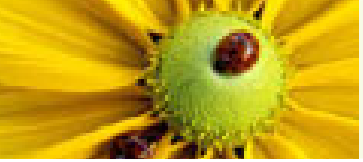
- Blocchi di inizializzazione `static`

- L'uso tipico consiste nell'inizializzare gli attributi statici

```
public class Esempio {
   private static final int SIZE = 100;
   private static int[] numeri = new int[SIZE];

   static {
      for (int i=0; i<SIZE; i++) {
         numeri[i] = i;
      }
   }

   ...
}
```



Questionario



D 1

Quale dei seguenti frammenti viene correttamente compilato e stampa "Uguale" in esecuzione?

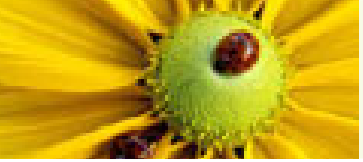
A.
```
Integer x = new Integer(100);
Integer y = new Integer(100);
if (x == y) {
  System.out.println("Uguale");
}
```

B.
```
int x=100;
Integer y = new Integer(100);
if (x == y) {
  System.out.println("Uguale");
}
```

C.
```
int x=100; float y=100.0F;
if (x == y) {
  System.out.println("Uguale");
}
```

D.
```
String x = new String("100");
String y = new String("100");
if (x == y) {
  System.out.println("Uguale");
}
```

E.
```
String x = "100";
String y = "100";
if (x == y) {
  System.out.println("Uguale");
}
```



D 1

Quale dei seguenti frammenti viene correttamente compilato e stampa "Uguale" in esecuzione?

A.
```
Integer x = new Integer(100);
Integer y = new Integer(100);
if (x == y) {
  System.out.println("Uguale");
}
```

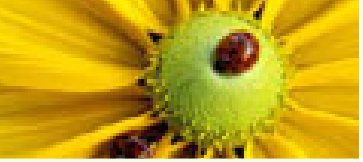
B.
```
int x=100;
Integer y = new Integer(100);
if (x == y) {
  System.out.println("Uguale");
}
```

C.
```
int x=100; float y=100.0F;
if (x == y) {
  System.out.println("Uguale");
}
```

D.
```
String x = new String("100");
String y = new String("100");
if (x == y) {
  System.out.println("Uguale");
}
```

E.
```
String x = "100";
String y = "100";
if (x == y) {
  System.out.println("Uguale");
}
```

C. E. – (E) funziona a causa di una ottimizzazione del compilatore, che riusa lo stesso oggetto quando vede due litterali `String` uguali.



D 2

Quali delle seguenti dichiarazioni sono illegali?

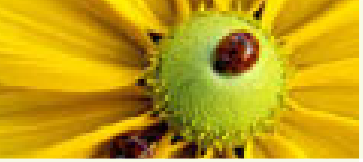
A. `default String s;`

B. `transient int i = 41;`

C. `public final static native int w();`

D. `abstract double d;`

E. `abstract final double cosenoIperbolico();`



D 2

Quali delle seguenti dichiarazioni sono illegali?

A. `default String s;`

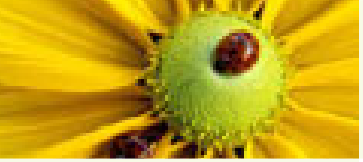
B. `transient int i = 41;`

C. `public final static native int w();`

D. `abstract double d;`

E. `abstract final double cosenoIperbolico();`

A. D. E.

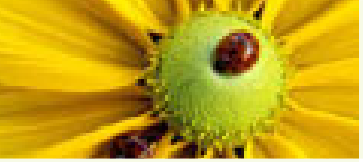


D 3

Quale delle seguenti affermazioni è vera?

A. Una classe `abstract` non può avere metodi `final`.

B. Una classe `final` non può avere metodi `abstract`.



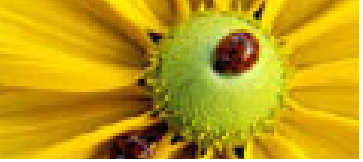
D 3

Quale delle seguenti affermazioni è vera?

A. Una classe `abstract` non può avere metodi `final`.

B. Una classe `final` non può avere metodi `abstract`.

B. – Una classe che contiene metodi `abstract` deve essere anch'essa `abstract`, ma ciò è in contraddizione con il modificatore `final`.



D 4

Qual è la minima modifica che rende il seguente codice compilabile?

```
 1. final class Aaa {
 2.   int xxx;
 3.   void yyy() { xxx=1; }
 4. }
 5.
 6. class Bbb extends Aaa {
 7.   final Aaa fref = new Aaa();
 8.   final void yyy() {
 9.     System.out.println(
10.       "In yyy()");
11.     fref.xxx = 12345;
12.   }
13. }
```

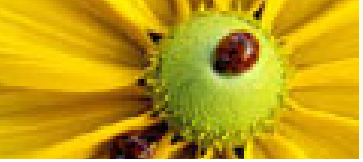
A. Alla linea 1, rimuovere `final`.

B. Alla linea 7, rimuovere `final`.

C. Rimuovere la linea 11.

D. Alle linee 1 e 7, rimuovere `final`.

E. Nessuna modifica è necessaria.



D 4

Qual è la minima modifica che rende il seguente codice compilabile?

```
 1. final class Aaa {
 2.   int xxx;
 3.   void yyy() { xxx=1; }
 4. }
 5.
 6. class Bbb extends Aaa {
 7.   final Aaa fref = new Aaa();
 8.   final void yyy() {
 9.     System.out.println(
10.       "In yyy()");
11.     fref.xxx = 12345;
12.   }
13. }
```

A. Alla linea 1, rimuovere `final`.

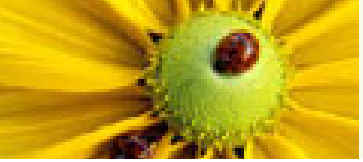
B. Alla linea 7, rimuovere `final`.

C. Rimuovere la linea 11.

D. Alle linee 1 e 7, rimuovere `final`.

E. Nessuna modifica è necessaria.

A.



D 5

Riguardo al codice seguente, quale affermazione è vera?

```
 1. class Roba {
 2.   static int x = 10;
 3.   static { x += 5; }
 4.
 5.   public static void main(String[] args) {
 6.     System.out.println("x=" + x);
 7.   }
 8.
 9.   static { x /= 5; }
10. }
```

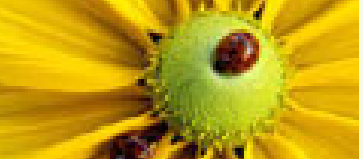
A. Le linee 3 e 9 non sono compilate, poiché mancano i nomi di metodi e i tipi ri ritorno.

B. La linea 9 non è compilata, poiché si può avere solo un blocco top-level `static`.

C. Il codice viene compilato e l'esecuzione produce `x=10`.

D. Il codice viene compilato e l'esecuzione produce `x=15`.

E. Il codice viene compilato e l'esecuzione produce `x=3`.



D 5

Riguardo al codice seguente, quale affermazione è vera?

```
 1. class Roba {
 2.   static int x = 10;
 3.   static { x += 5; }
 4.
 5.   public static void main(String[] args) {
 6.     System.out.println("x=" + x);
 7.   }
 8.
 9.   static { x /= 5; }
10. }
```

A. Le linee 3 e 9 non sono compilate, poiché mancano i nomi di metodi e i tipi ri ritorno.

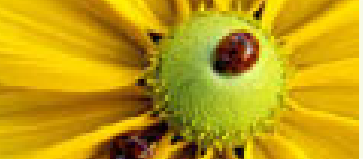
B. La linea 9 non è compilata, poiché si può avere solo un blocco top-level `static`.

C. Il codice viene compilato e l'esecuzione produce `x=10`.

D. Il codice viene compilato e l'esecuzione produce `x=15`.

E. Il codice viene compilato e l'esecuzione produce `x=3`.

E.



D 6

Rispetto al codice seguente, quale affermazione è vera?

```
 1. class A {
 2.   private static int x=100;
 3.
 4.   public static void main(
 5.       String[] args); {
 6.     A hs1 = new A();
 7.     hs1.x++;
 8.     A hs2 = new A();
 9.     hs2.x++;
10.     hs1 = new A();
11.     hs1.x++;
12.     A.x++;
13.     System.out.println(
14.       "x = " + x);
15.   }
16. }
```

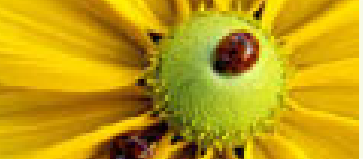
A. La linea 7 non compila, poiché è un riferimento `static` ad una variabile `private`.

B. La linea 12 non compila, poiché è un riferimento `static` ad una variabile `private`.

C. Il programma viene compilato e stampa `x = 102`.

D. Il programma viene compilato e stampa `x = 103`.

E. Il programma viene compilato e stampa `x = 104`.



D 6

Rispetto al codice seguente, quale affermazione è vera?

```
 1. class A {
 2.   private static int x=100;
 3.
 4.   public static void main(
 5.       String[] args); {
 6.     A hs1 = new A();
 7.     hs1.x++;
 8.     A hs2 = new A();
 9.     hs2.x++;
10.     hs1 = new A();
11.     hs1.x++;
12.     A.x++;
13.     System.out.println(
14.       "x = " + x);
15.   }
16. }
```

A. La linea 7 non compila, poiché è un riferimento `static` ad una variabile `private`.

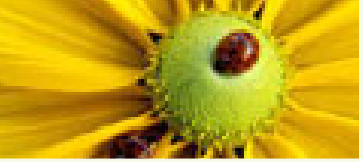
B. La linea 12 non compila, poiché è un riferimento `static` ad una variabile `private`.

C. Il programma viene compilato e stampa `x = 102`.

D. Il programma viene compilato e stampa `x = 103`.

E. Il programma viene compilato e stampa `x = 104`.

E.

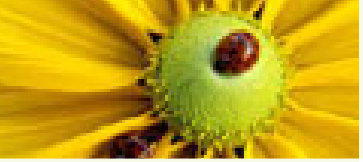


D 7

Dato il codice seguente:

```
1. class SuperC {
2.   void unMetodo() { }
3. }
4.
5. class SubC extends SuperC {
6.   void unMetodo() { }
7. }
```

1. Quali modificatori di accesso possono essere legalmente dati ad `unMetodo` alla linea 2, lasciando il resto del codice inalterato?
2. Quali modificatori di accesso possono essere legalmente dati ad `unMetodo` alla linea 6, lasciando il resto del codice inalterato?



D 7

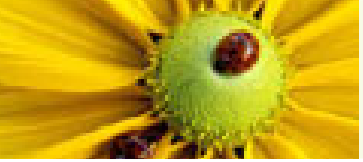
Dato il codice seguente:

```
1. class SuperC {
2.   void unMetodo() { }
3. }
4.
5. class SubC extends SuperC {
6.   void unMetodo() { }
7. }
```

1. Quali modificatori di accesso possono essere legalmente dati ad `unMetodo` alla linea 2, lasciando il resto del codice inalterato?
2. Quali modificatori di accesso possono essere legalmente dati ad `unMetodo` alla linea 6, lasciando il resto del codice inalterato?

---

1. Alla linea 2, il metodo può essere `private`, visto che nella sottoclasse viene sovrapposto da un metodo con visibilità di pacchetto.
2. Alla linea 6, il metodo può essere `protected` oppure `public`, visto che sovrappone un metodo con visibilità di pacchetto.



D 8

```
 1. package abcd;
 2.
 3. public class SupA {
 4.   protected static int count=0;
 5.   public SupA() { count++; }
 6.   protected void f() {}
 7.   static int getCount() {
 8.     return count;
 9.   }
10. }
```

```
 1. package abcd;
 2.
 3. class A extends abcd.SupA {
 4.   public void f() {}
 5.   public int getCount() {
 6.     return count;
 7.   }
 8. }
```

Riguardo ai codici a sinistra, quale affermazione è vera?

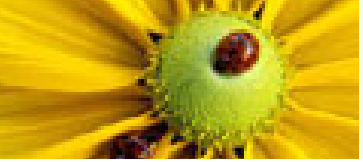
A. La compilazione di `A.java` fallisce alla linea 4, poiché il metodo `f()` è `protected` nella superclasse e `A` è nello stesso pacchetto di `SupA`.

B. La compilazione di `A.java` fallisce alla linea 4, poiché il metodo `f()` è `protected` nella superclasse e `public` nella sottoclasse.

C. La compilazione di `A.java` fallisce alla linea 5, poiché il metodo `getCount()` è `static` nella superclasse e non può essere sovrapposto da un metodo non-`static`.

D. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo `f()` su una istanza di `A`.

E. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo `getCount` su una istanza di `SupA`.



D 8

```
 1. package abcd;
 2.
 3. public class SupA {
 4.   protected static int count=0;
 5.   public SupA() { count++; }
 6.   protected void f() {}
 7.   static int getCount() {
 8.     return count;
 9.   }
10. }
```

```
 1. package abcd;
 2.
 3. class A extends abcd.SupA {
 4.   public void f() {}
 5.   public int getCount() {
 6.     return count;
 7.   }
 8. }
```

Riguardo ai codici a sinistra, quale affermazione è vera?

A. La compilazione di `A.java` fallisce alla linea 4, poiché il metodo `f()` è `protected` nella superclasse e `A` è nello stesso pacchetto di `SupA`.

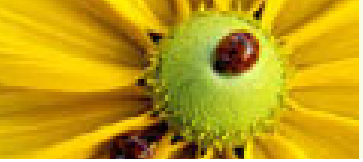
B. La compilazione di `A.java` fallisce alla linea 4, poiché il metodo `f()` è `protected` nella superclasse e `public` nella sottoclasse.

C. La compilazione di `A.java` fallisce alla linea 5, poiché il metodo `getCount()` è `static` nella superclasse e non può essere sovrapposto da un metodo non-`static`.

D. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo `f()` su una istanza di `A`.

E. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo `getCount` su una istanza di `SupA`.

C.



D 9

Riguardo ai codici seguenti, quale affermazione è vera?

```
 1. package abcd;
 2.
 3. public class SupA {
 4.   protected static int count=0;
 5.   public SupA() { count++; }
 6.   protected void f() {}
 7.   static int getCount() {
 8.     return count;
 9.   }
10. }
```

```
 1. package ab;
 2.
 3. class A extends abcd.SupA {
 4.   A() { count++; }
 5.
 6.   public static void main(
 7.       String[] args) {
 8.     System.out.print(
 9.       "Prima:" + count);
10.     A a = new A();
11.     System.out.println(
12.       " Dopo:" + count);
13.     a.f();
14.   }
15. }
```

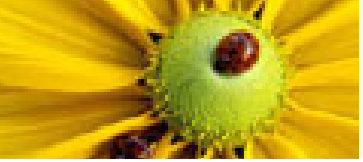
A. Il programma viene compilato e stampa: `Prima:0 Dopo:2`

B. Il programma viene compilato e stampa: `Prima:0 Dopo:1`

C. La compilazione di `A` fallisce alla linea 4.

D. La compilazione di `A` fallisce alla linea 13.

E. Il programma viene compilato, ma viene lanciata una eccezione alla linea 13.



D 9

Riguardo ai codici seguenti, quale affermazione è vera?

```
 1. package abcd;
 2.
 3. public class SupA {
 4.   protected static int count=0;
 5.   public SupA() { count++; }
 6.   protected void f() {}
 7.   static int getCount() {
 8.     return count;
 9.   }
10. }
```

```
 1. package ab;
 2.
 3. class A extends abcd.SupA {
 4.   A() { count++; }
 5.
 6.   public static void main(
 7.       String[] args) {
 8.     System.out.print(
 9.       "Prima:" + count);
10.     A a = new A();
11.     System.out.println(
12.       " Dopo:" + count);
13.     a.f();
14.   }
15. }
```

A. Il programma viene compilato e stampa: `Prima:0 Dopo:2`

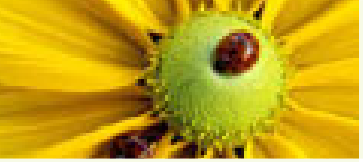
B. Il programma viene compilato e stampa: `Prima:0 Dopo:1`

C. La compilazione di `A` fallisce alla linea 4.

D. La compilazione di `A` fallisce alla linea 13.

E. Il programma viene compilato, ma viene lanciata una eccezione alla linea 13.

A.



D 10

Si consideri la classe seguente:

```
1. public class Test1 {
2.   public float unMetodo(float a, float b) {
3.   }
4.
5. }
```

Quali dei seguenti metodi possono lecitamente essere inseriti alla linea 4?

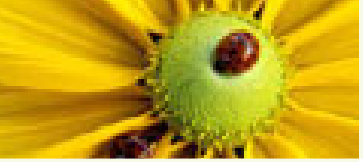
A. `public int unMetodo(int a, int b) { }`

B. `public float unMetodo(float a, float b) { }`

C. `public float unMetodo(float a, float b, int c) { }`

D. `public float unMetodo(float c, float d) { }`

E. `private float unMetodo(int a, int b, int c) { }`



D 10

Si consideri la classe seguente:

```
1. public class Test1 {
2.   public float unMetodo(float a, float b) {
3.   }
4.
5. }
```

Quali dei seguenti metodi possono lecitamente essere inseriti alla linea 4?

A. `public int unMetodo(int a, int b) { }`

B. `public float unMetodo(float a, float b) { }`

C. `public float unMetodo(float a, float b, int c) { }`

D. `public float unMetodo(float c, float d) { }`

E. `private float unMetodo(int a, int b, int c) { }`

A. C. E. – B e D potrebbero essere overriding non overloading.