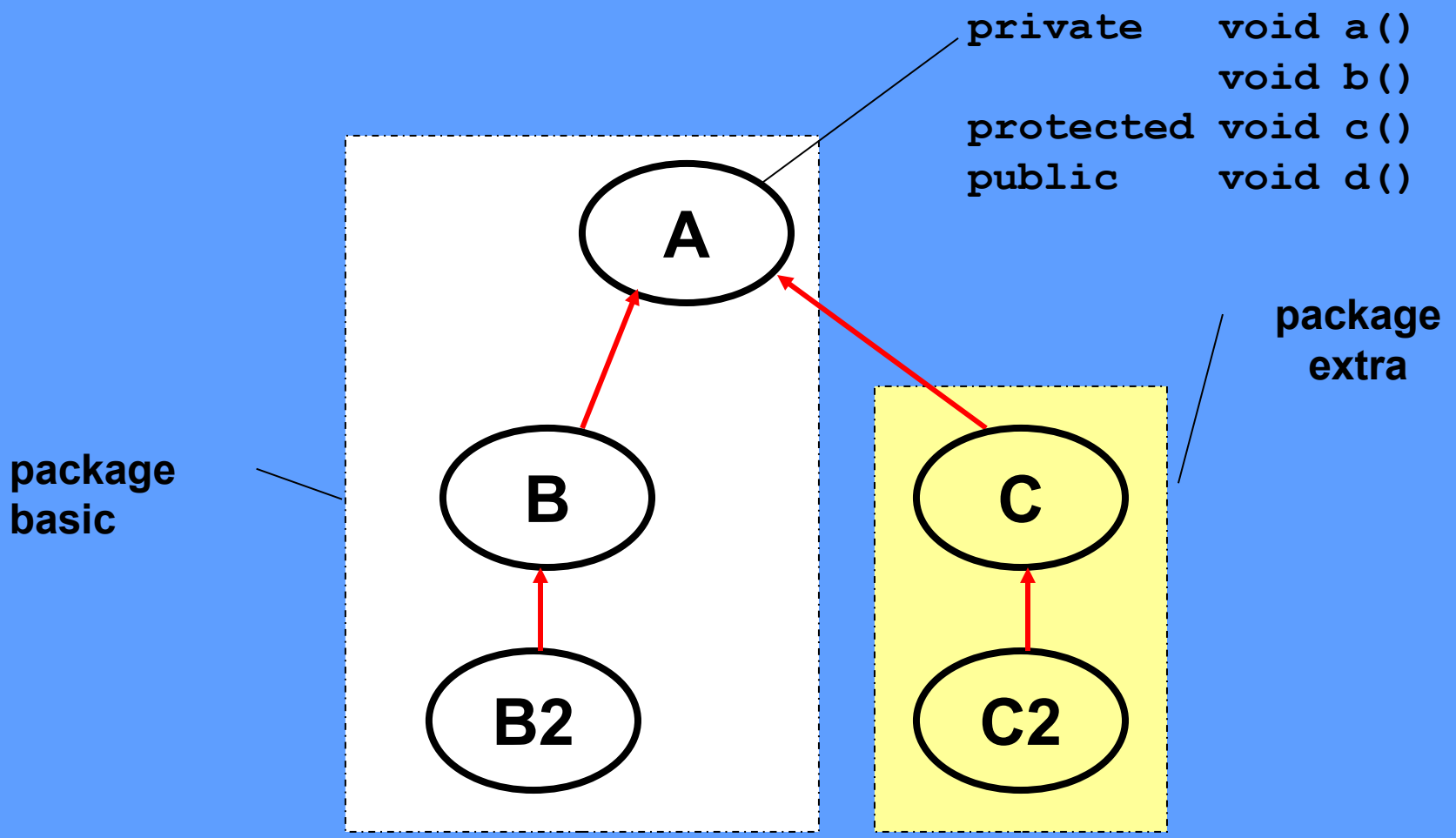
EREDITARIETÀ, VISIBILITÀ, POLIMORFISMO

L'uso dei *modificatori di visibilità* nella definizione di metodi coinvolti in *gerarchie di ereditarietà* può avere conseguenze sul polimorfismo.

- **In Java, il late binding assicura che venga invocato il metodo definito nella classe dell'istanza corrente, o eventualm. in quella del suo antenato più prossimo**
- **Se il metodo invocato è pubblico o protetto, non ci sono mai problemi**
- **Se invece il metodo invocato è privato o package, ed è quindi invocato *indirettamente* tramite altri metodi, possono verificarsi situazioni "strane" e impreviste.**

UN ESEMPIO (1/3)

Si consideri questa gerarchia:



UN ESEMPIO (2/3)

Si supponga che la classe A definisca un metodo pubblico `test()` che invochi tutti gli altri:

```
package basic;
public class A {
 private void a(){          System.out.print("A:a()   "); }
 /* package */ void b() {  System.out.print("A:b()   "); }
 protected void c(){        System.out.print("A:c()   "); }
 public void d(){           System.out.print("A:d()   "); }
 public void test(){  a(); b(); c(); d();
  System.out.println(""); }
}
```

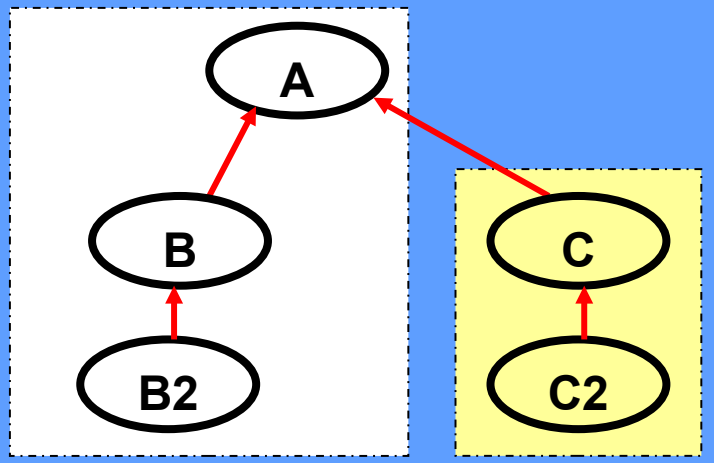
Se ora un main crea una istanza della classe A e invoca su essa il metodo pubblico `test()`, si ottiene ovviamente l'invocazione dei 4 metodi `a()`, `b()`, `c()`, `d()`.

UN ESEMPIO (3/3)

Si supponga ora che le classi B, C, B2, C2 ridefiniscano i quattro metodi `a()`, `b()`, `c()`, `d()`:

```
package basic;
public class B extends A { // analogamente per B2, C, C2
 private void a(){          System.out.println("B:a()  "); }
 /* package */ void b() {  System.out.println("B:b()  "); }
 protected void c(){        System.out.println("B:c()  "); }
 public void d(){           System.out.println("B:d()  "); }
}
```

Cosa succede adesso se un main crea un'istanza di tali classi e invoca su di esse il metodo pubblico `test()`, *che si appoggia su quei 4 metodi* ?



IL TEST

Consideriamo questo (apparentemente) semplice main:

```
import extra.*;
import basic.*;
public class Test {
 public static void main(String args[]) {
  A istanzaA = new A();
  B istanzaB = new B();     B2 istanzaB2 = new B2();
  C istanzaC = new C();     C2 istanzaC2 = new C2();
  istanzaA.test(); istanzaB.test(); istanzaB2.test();
                   istanzaC.test(); istanzaC2.test();
 }
}
```

Cosa succede ?

IL PROBLEMA (1/2)

Poiché Java supporta il polimorfismo, *ci aspetteremmo* che venissero *sempre e comunque invocate le versioni più recenti* di `a()`, `b()`, `c()`, `d()`, ottenendo quindi:

```
A:a()   A:b()   A:c()   A:d()                  // istanzaA.test()
B:a()   B:b()   B:c()   B:d()                  // istanzaB.test()
B2:a()  B2:b()  B2:c()  B2:d()                 // istanzaB2.test()
C:a()   C:b()   C:c()   C:d()                  // istanzaC.test()
C2:a()  C2:b()  C2:c()  C2:d()                 // istanzaC2.test()
```

E INVECE NO.

Tutto come previsto per quanto riguarda l'invocazione del metodo *protetto* `c()` e del metodo *pubblico* `d()` …

ma per gli altri c'è una sorpresa!

IL PROBLEMA (2/2)

In realtà, otteniamo questo:

```
A:a()   A:b()    A:c()    A:d()          // istanzaA.test()
A:a()   B:b()    B:c()    B:d()          // istanzaB.test()
A:a()   B2:b()   B2:c()   B2:d()         // istanzaB2.test()
A:a()   A:b()    C:c()    C:d()          // istanzaC.test()
A:a()   A:b()    C2:c()   C2:d()         // istanzaC2.test()
```

Ossia:

- **per il metodo privato `a()` viene sempre invocata la versione originale definita nella classe base A**
- **per il metodo con visibilità *package* `b()` viene invocata la versione originale *solo nel caso di istanze di classi non appartenenti allo stesso package*.**

COME SI SPIEGA CIÒ?

LA SPIEGAZIONE

Il motivo del comportamento "apparentemente strano" è da ricercarsi nel *significato profondo* dei modificatori di visibilità `public`, `protected`, (*package*), `private`.

Poiché le regole di visibilità (*scope rules*) di Java sono lessicali, l'esistenza di versioni alternative di un metodo dev'essere *determinabile staticamente,* a tempo di compilazione, *sulla base unicamente del "luogo" in cui le definizioni stesse sono (o non sono) scritte.*

Dunque,

- **una alternativa per il metodo `a()`, che è dichiarato *privato*, può essere cercata solo *nella classe medesima***
- **una alternativa per il metodo `b()`, che ha visibilità *package*, può essere cercata solo *nello stesso package***

CONSEGUENZA

Per questo motivo, l'invocazione del metodo `test()`

- **su una istanza di A, causa l'invocazione (ovviamente) di metodi definiti in A stessa**
- **su una istanza di B o B2, causa l'invocazione dei metodi `b()`, `c()` e `d()` ridefiniti in B o B2, e del metodo `a()` originale**
  - **la ridefinizione di `b()` è accessibile perché la classe B (o B2) è definita nello stesso package della classe A**
- **su una istanza di C o C2, causa l'invocazione dei metodi `c()` e `d()` ridefiniti in C o C2, e dei metodi `a()` e `b()` originali**
  - **la ridefinizione di `b()` stavolta non è accessibile perché la classe C (o C2) è definita in un altro package rispetto alla classe A**

```
A:a()   A:b()    A:c()    A:d()          // istanzaA.test()
A:a()   B:b()    B:c()    B:d()          // istanzaB.test()
A:a()   B2:b()   B2:c()   B2:d()         // istanzaB2.test()
A:a()   A:b()    C:c()    C:d()          // istanzaC.test()
A:a()   A:b()    C2:c()   C2:d()         // istanzaC2.test()
```

LA FINE DEL POLIMORFISMO?

Ovviamente, ciò equivale a dire che

- **il metodo `a()` non è polimorfo**
- **il metodo `b()` non è polimorfo fuori dal proprio package.**

Cosa significa, *la fine del polimorfismo?*

- **Sì.. *in un certo senso:* ma definire un metodo privato o package *ha esattamente il senso di renderlo inaccessibile*, e se è inaccessibile *da fuori non si deve vedere !* Siamo noi ad aver fatto questa scelta!**
- **No, in realtà: scegliere il polimorfismo significa fare una *scelta di apertura*, cioè l'esatto contrario di una limitazione di visibilità.**