Information Hiding

Package

- Le classi sono raggruppate in **package**
  - Un package raggruppa classi definendo regole di visibilità
- Se una classe C è visibile nel package A, ma è dichiarata nel package B, questa viene denotata come B.C
  - Quindi si possono usare liberamente gli stessi nomi in package diversi, senza generare confusione

Compilation unit

- Un file che contiene la dichiarazione di una o più classi (o interfacce)
  - Una sola dichiarata pubblica (public class) e avente lo stesso nome del file
- C'è al più un solo metodo main
- Si può specificare il package di appartenenza (lo stesso per tutte)
  - Se non si specifica, si assume un package senza nome di default

Package

- Una directory che contiene una o più compilation unit
- Introduce un nuovo ambito di visibilità dei nomi:
  - Unit con lo stesso nome possono stare in package diversi
- Contiene un insieme di classi pubbliche ed un insieme di classi private al package (“**friendly**”)
- Le classi pubbliche si possono “importare” in altri package

Visibilità delle classi

- public
  - Sono visibili a tutti con import del package
  - Il file deve avere lo stesso nome
  - Al più una public class per ogni file
- "friendly”
  - Sono visibili solo all’interno dello stesso package/compilation unit
  - Possono stare in un file con altre classi

Esempio

```
package myTools.text;
public class TextComponent {
        . . .Zzz z;…
}
```

```
package myFigs.planar;
public class DrawableElement {
        . . .
}
class xxx {
        . . .
}
```

```
package myTools.text;
public class yyy {
        . . .
}
class Zzz{
        . . .
}
```

Visibilità di attributi e metodi

- Attributi e metodi di una classe vengono sempre ereditati e possono essere:
  - **public**
    - Sono visibili a tutti
  - **private**
    - Sono visibili solo all'interno della classe
    - Non sono visibili nelle sottoclassi
  - **protected**
    - Sono visibili alle classi nello stesso package
    - Sono visibili anche alle sottoclassi, anche in package diversi
  - "friendly"
    - Sono visibili alle classi nello stesso package
    - Sono visibili solo alle sottoclassi nello stesso package

Information hiding

- Una classe/attributo/metodo public è una promessa agli utilizzatori della classe
  - Sarà disponibile e non cambierà, perlomeno dal punto di vista degli utilizzatori della classe
- La promessa è molto vincolante
  - Meglio promettere poco!
- Tutte le proprietà per cui ci si vuole mantenere la possibilità di modifica o eliminazione devono essere private
  - Al massimo, ma solo se indispensabile, friendly
    - È meglio private per le proprietà “helper” di una classe
    - Se un attributo è friendly e qualcuno lo usa non possiamo più cambiarlo!!!

Information hiding

- È fortemente consigliato che gli attributi di una classe public siano private o friendly
  - Usare metodi per accedervi!
- I metodi che possono essere usati dagli utilizzatori "esterni" della classe dovrebbero essere public
  - Gli attributi friendly sono usati solo quando le classi all'interno dello stesso package devono avere accesso privilegiato
    - Esempio: una classe Lista deve usare una classe Nodo che implementa i nodi della Lista: è utile che Lista possa accedere ai campi di Nodo, ma gli utilizzatori di Lista non devono potere accedere a Nodo

Accesso ai membri private

- Le sottoclassi non possono accedere agli attributi (e metodi) private delle superclassi!
- Quindi è sbagliato scrivere:

```
public void accendi() {
      if (batterieCariche) accesa = true;
}
```

- …perché accesa è private nella superclasse!

Polimorfismo

Ereditarietà

- Una classe definisce un tipo
- Una sottoclasse (transitivamente) definisce un sottotipo
- Un oggetto del sottotipo è sostituibile a un oggetto del tipo
- Si distingue tra
  - Tipo statico: il tipo dichiarato
  - Tipo dinamico (o attuale): il tipo dell'oggetto attualmente assegnato

- Java garantisce che ciò non comprometta la **type safety**
- Il compilatore verifica che ogni oggetto venga manipolato correttamente **in base al tipo statico**
- Il linguaggio garantisce che a run time non sorgono errori se si opera su un oggetto **il cui tipo dinamico** è un sottotipo del tipo statico

Esempio

```
public class UsaAutomobile {
    public static void partenza(Automobile p) {
        if (p.puoPartire())
            p.accendi();
    }

    public static void main(String args[]) {
        // legale!!
        Automobile myCar = new AutomobileElettrica("T");

        partenza(myCar); //funziona anche con AutomobileElettrica
    }
}
```

Polimorfismo

- L'esempio precedente è un caso di **polimorfismo**
- Polimorfismo è la capacità per un elemento sintattico di riferirsi a elementi di diverso tipo
- In Java una variabile di un tipo riferimento T può riferirsi ad un qualsiasi oggetto il cui tipo sia T o un sottotipo di T
- Similmente un parametro formale di un tipo riferimento T può riferirsi a parametri attuali il cui tipo sia T o un sottotipo di T

Tipo dinamico e tipo statico

- Il tipo **statico** è quello definito dalla **dichiarazione**

```
Automobile myCar;
```

- Il tipo **dinamico** è definito dal **costruttore** usato per definirlo

```
AutomobileElettrica yourCar = new AutomobileElettrica();
```

- In Java, il tipo dinamico può essere sottotipo del tipo statico

```
Automobile myCar = new Automobile();
AutomobileElettrica yourCar = new AutomobileElettrica();
myCar = yourCar;
```

Assegnamento polimorfico

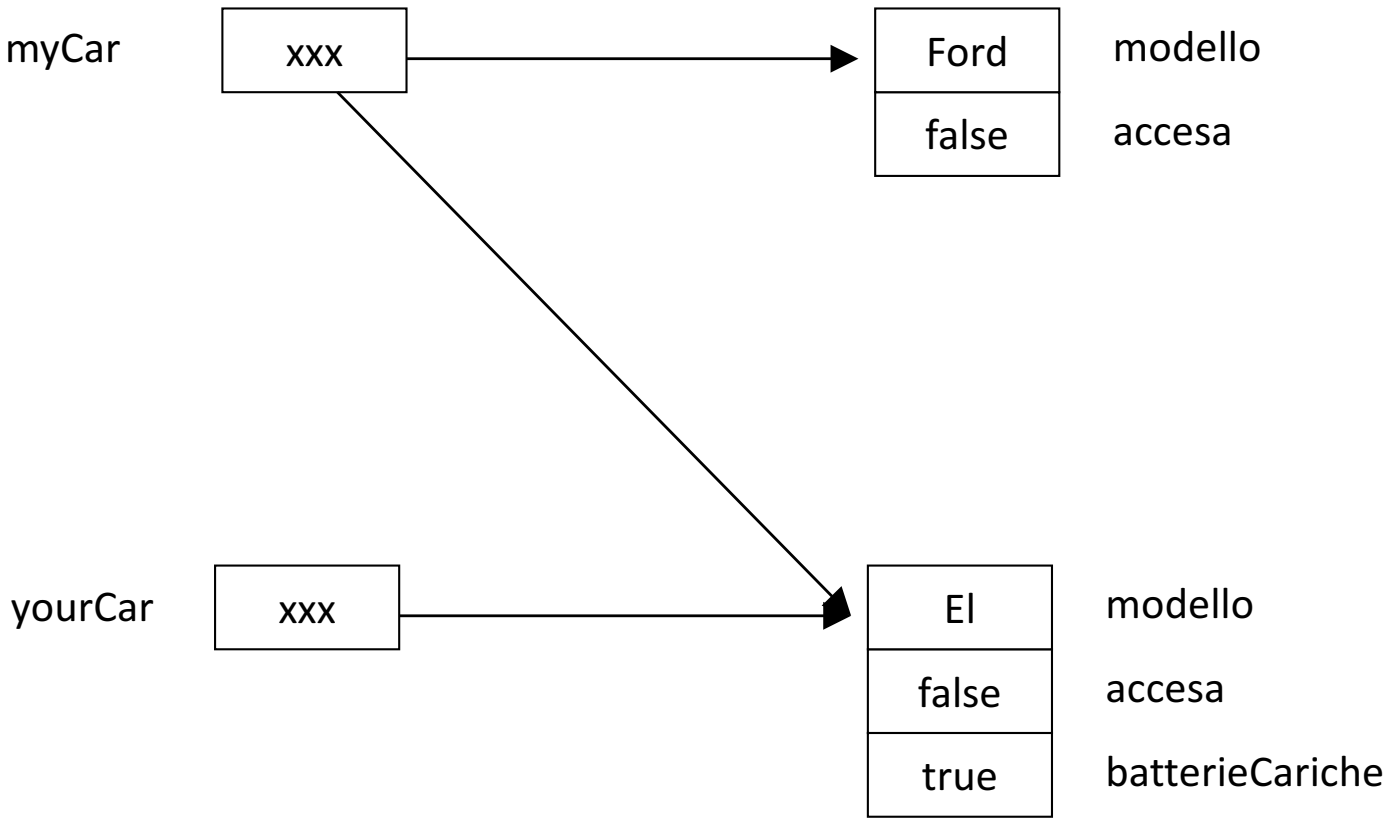
- A un oggetto di tipo statico T si può sempre assegnare un oggetto il cui tipo S è sottotipo di T (ma non viceversa)
- Questo consente che il tipo dinamico possa essere sottotipo di quello statico

```
Automobile myCar = new AutomobileElettrica("T");
```

- Il compilatore verifica che ogni oggetto venga manipolato correttamente **solo in base al tipo statico**
- La regola precedente garantisce che a runtime non sorgono errori se si opera su un oggetto il cui tipo dinamico è un sottotipo del tipo statico

Uso delle classi dell'esempio

```
AutomobileElettrica yourCar = new AutomobileElettrica("E1");
Automobile myCar = new Automobile("Ford");
myCar = yourCar;
```



Chiamata di un metodo

```
Automobile myCar = new AutomobileElettrica();
…
myCar.puoPartire(); //OK, chiama metodo di Automobile
myCar.accendi(); //OK, chiama metodo di AutomobileElettrica
myCar.ricarica(); //KO, ricarica non e' metodo di Automobile
AutomobileElettrica yourCar = new AutomobileElettrica ();
yourCar.ricarica(); //OK, chiama metodo di AutomobileElettrica
```

Polimorfismo e binding dinamico

- In Java, a fronte della invocazione x.f(x1,...,xn), l'implementazione scelta per il metodo f **dipende dal tipo dinamico** di x e **non** dal suo tipo statico
- Il legame (**binding**) tra il metodo invocato e il metodo attivato è dinamico, dipende dal tipo attuale dell'oggetto

Esempio

```
public class UsaAutomobile {

   public static void partenza(Automobile a) {
    a.accendi(); // solo a run time si conosce il tipo effettivo
   }

   public static void main(String args[]) {
    Automobile a1 = new Automobile("Ford");
    Automobile a2 = new AutomobileElettrica("T");
    a1.accendi();
    //a run-time chiama implementazione di Automobile
    a2.accendi();
    //a run-time chiama implementazione di AutomobileElettrica
    partenza(a2);
    }
}
```

Binding dinamico

```
public static void partenza(Automobile a) {
    a.accendi();
}
```

- Quale implementazione del metodo accendi() chiamare
  - Quella di Automobile o quella di AutomobileElettrica?
  - Il tipo effettivo del parametro a è noto solo quando accendi viene chiamato, cioè a runtime
- Il compilatore non genera il codice per eseguire il metodo, ma genera il codice che cerca l'implementazione giusta in base al tipo dinamico dell'oggetto e la esegue
  - Questo si chiama **dispatching**, che in Java avviene in maniera **dinamica**

Overloading e overriding

- Overriding non va confuso con overloading

```
public class Punto2D{
    public float distanza(Punto2D p){…}
}
public class Punto3D extends Punto2D {
    public float distanza(Punto3D p){…} //OVERLOADING!!!
}
```

  - Il metodo distanza di Punto3D ha un'intestazione diversa da quella di distanza dichiarato in Punto2D:

```
Punto2D p = new Punto3D();
p.distanza(p); //chiama Punto2D.distanza(Punto2D)
```

  - **Non** è overriding ➔ non si applica binding dinamico

Regola per chiamata metodi

- Il compilatore, quando trova una chiamata di un metodo x.m(p) **risolve staticamente l'overloading**, individuando la segnatura del metodo chiamato in base al tipo statico P del parametro attuale p e al tipo statico X di x
- Il **binding dinamico** si applica a **run-time**: il codice sceglie a runtime il metodo "più vicino" tra quelli che hanno il prototipo X.m(P) stabilito staticamente
  - Risale la gerarchia per cercare il metodo piu vicino

Esempio

```
public class Punto2D{public float distanza(Punto2D p){…}}
public class Punto3D extends Punto2D {public float distanza(Punto3D p){…}}

public void static void main(String[] args){
    Punto2D p1,p2;
    Punto3D p3;

    p1 = new Punto2D(3,7);
    p2 = new Punto3D(3,7, 4);
    System.out.println(p1.distanza(p2)); //metodo di Punto2D
    System.out.println(p2.distanza(p1)); //metodo di Punto2D
    p3 = new Punto3D(6,7, 5);
    System.out.println(p2.distanza(p3)); //metodo di Punto2D
    System.out.println(p3.distanza(p1)); //metodo di Punto2D
    System.out.println(p1.distanza(p3)); //metodo di Punto2D

    Punto3D p4 = new Punto3D(6,1, 5);
    System.out.println(p3.distanza(p4)); //metodo di Punto3D
}
```

Classi e metodi astratti

- Un metodo astratto è un metodo per il quale non viene specificata alcuna implementazione
- Una classe è astratta se contiene almeno un metodo astratto
- Non è possibile creare istanze di una classe astratta
- Le classi astratte sono molto utili per introdurre delle astrazioni di alto livello

Classi e metodi astratti

```
abstract class Shape {
   static Screen screen = new Screen();
   Shape(){}
   abstract void show();
}
class Circle extends Shape {
  void show() {
     …
  }
}
```

```
Shape s=new Shape(); //ERRATO
Circle c=new Circle(); //CORRETTO
Shape s=new Circle(); //CORRETTO
```

Classi e metodi final

- Se vogliamo impedire che sia possibile creare sottoclassi di una certa classe la definiremo final

```
final class C {...}
class C1 extends C //ERRATO
```

- Similmente, se vogliamo impedire l'overriding di un metodo dobbiamo definirlo final

```
class C {final void f(){…}
class C1 extends C {
        void f(){…} //ERRATO
}
```