# Programmazione a Oggetti
# Modulo B

## Lezione 13

Dott. Alessandro Roncato

**18/03/2013**

# Riassunto

- Controller
- Model View Controller
- Observer (Listner)
- GUI

# Listener  Multipli

- Come facciamo a gestire più pulsanti con la stessa GUI?
- Creiamo più pulsanti
- Mettiamo in ascolto la stessa GUI per tutti i pulsanti
- Controlliamo nell'evento quale pulsante lo ha generato

# Esempio ==

```
public class ClienteFrame extends JFrame implements ActionListner
{
  JButton crea= new Jbutton("Crea");
  JButton elimina = new JButton("Elimina");
  Jlabel esito = new JLabel("");
  public MyJFrame(){
    crea.addActionListener(this);
    elimina.addActionListener(this);
    ...}
  public void actionPerformed(ActionEvent evt){
    if (evt.getSource()==crea) {…}
    else {…}
  }
}
```

# Identità oggetti

- Se devo controllare se due **valori** sono uguali uso l'operatore ==
- Esempio: 2==3 è falso, 2==2 è vero
- Per gli **oggetti** c'è da fare attenzione in quanto oltre all'operatore == c'è il metodo `equals` definito in ogni oggetto

# == e `equals`

- Dati x e y due riferimenti a oggetti, controllo se x e y fanno riferimento allo stesso oggetto con l'operatore ==, mentre controllo se x e y fanno riferimento a due oggetti uguali con `x.equals(y)`.

# Esempio

```
public class Prodotto {
  String descrizione;
  double prezzo;
  public boolean equals(Object o) {
   if (o==null) return false;
   if(! o instanceof Prodotto) return false;
   Prodotto p = (Prodotto) o;
   if (descrizione!=null  && !
           descrizione.equals(p.descrizione))
     return false;
   if(descrizione==null && p.descrizione!=null)
     return false;
   if (prezzo!=p.prezzo) return false;
   …
   return true;
}
```

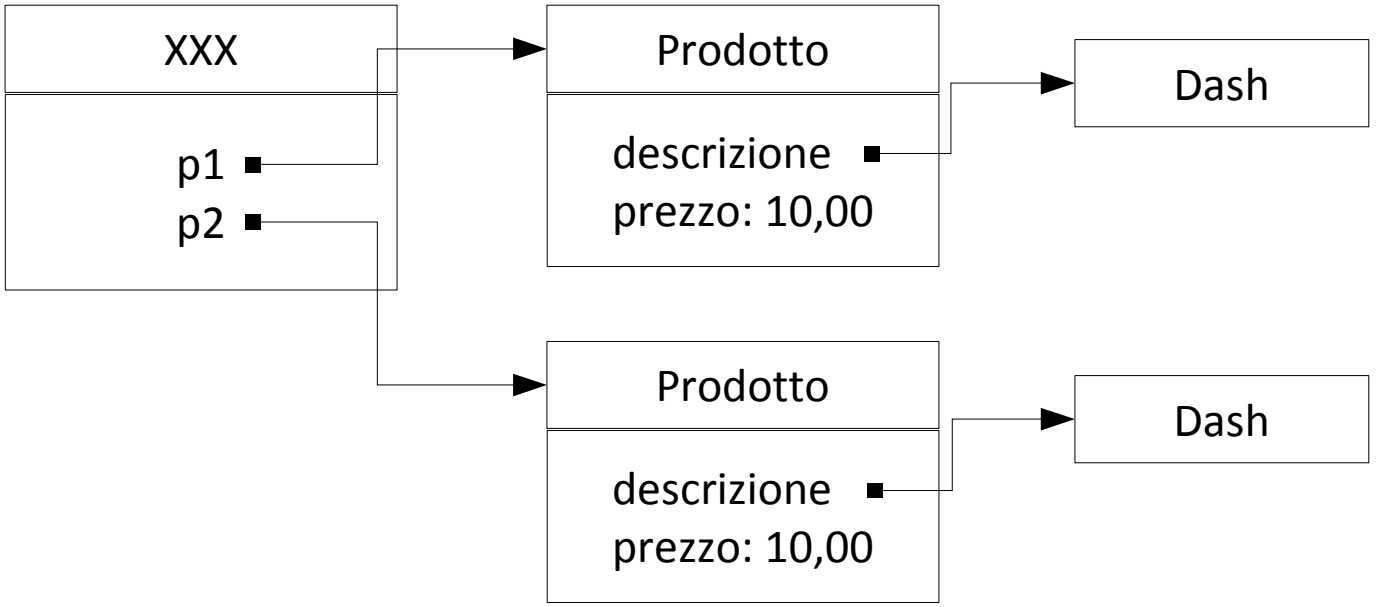
# Esempio

```
public class XXX {
   Prodotto p1 = new Prodotto();
   Prodotto p2 = new Prodotto();

   public ... qualcheMetodo(){

      p1==p2          //è sempre falso
      p1.equals(p2) // è vero se i descrizione
                    // prezzo etc sono uguali
   }
}
```

# Diagramma oggetti

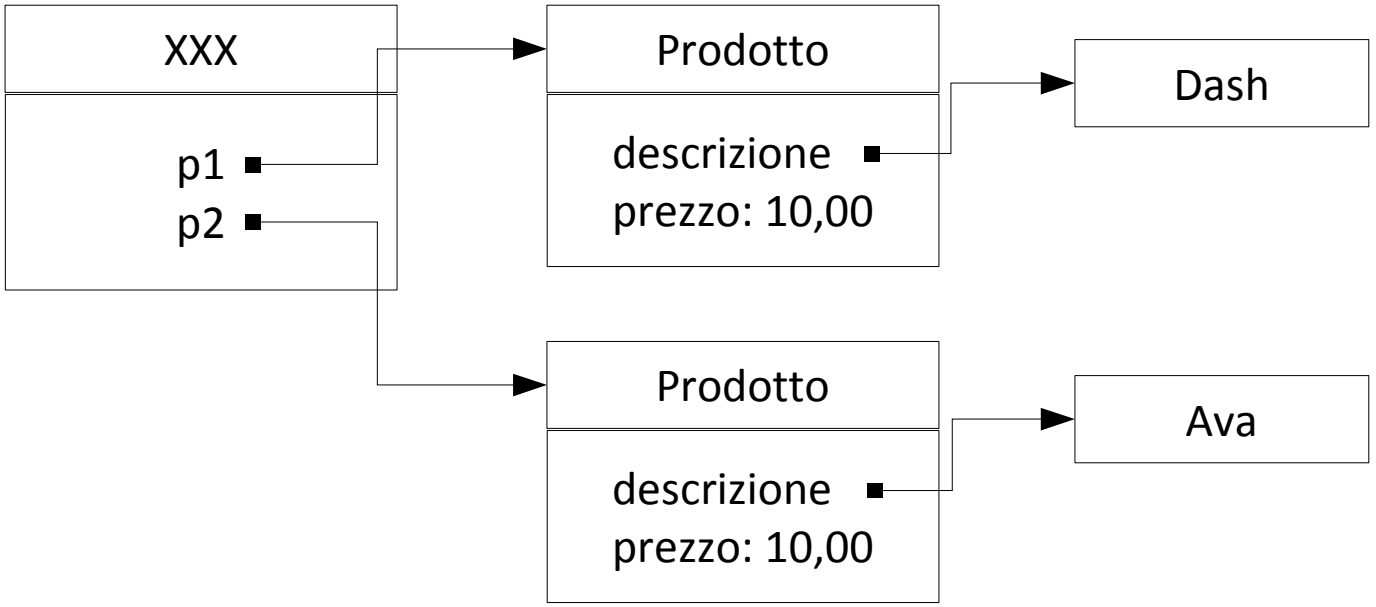



```
p1==p2
p1.equals(p2)
```

**FALSE**

**TRUE**

# Diagramma oggetti



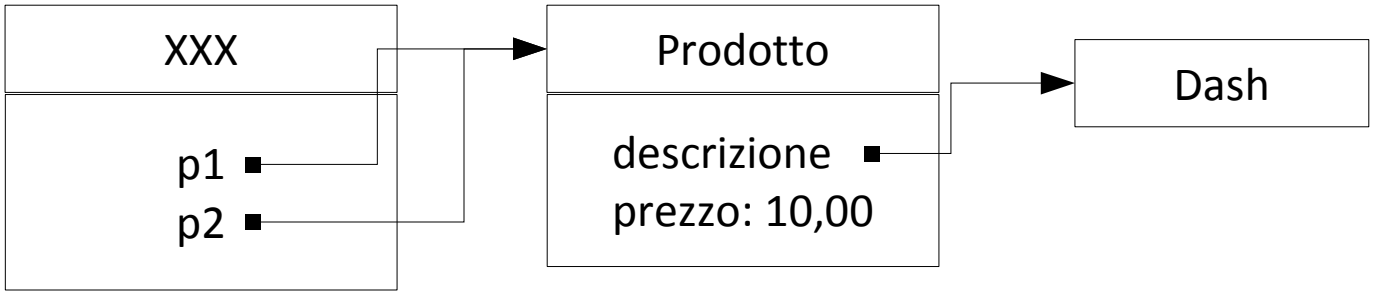

```
p1==p2                    FALSE
p1.equals(p2)             FALSE
```

# Esempio

```
public class XXX {
   Prodotto p1 = new Prodotto();
   Prodotto p2 = p1;

   public ... qualcheMetodo(){
      p1==p2          //è sempre vero
      p1.equals(p2) // è sempre vero
   }
}
```

# Diagramma oggetti




```
p1==p2
p1.equals(p2)
```

**TRUE**
**TRUE**

== implica equals

# equals

- equals è definito nella classe `Object` e quindi ereditato da tutte le altre classi!
- Il metodo `equals` della classe `Object` ritorna true solo se gli oggetti sono gli stessi (`x.equals(y)` sse `x==y`)

.

# Sovrascrittura equals

- Quando il comportamento standard del metodo `equals` non si comporta come vogliamo lo possiamo riscrivere nella classe
- La riscrittura comporta anche la riscrittura del metodo `hashCode` (altro metodo definito dalla classe `Object`).

# Esempio equals

```
class Generica {
   private String name;
   private int i;
   [...]
   public boolean equals(Object o) {
       if (this == o) return true;
       if (o==null) rerturn false;
       if (!(o instanceof Generica)) return false;
       Generica g = (Generica) o;
       if (name != null ? !name.equals(g.name) : g.name !=
null) return false;
       if (i != g.i) return false;
       [...]
       return true;
}
```

**Nota: se si riscrive il metodo equals, bisogna riscrivere il metodo hashCode in modo tale che se due oggetti sono equals anche l'hashCode sia uguale**

}

# Esempio equals

```
class Albero {
   private String info;
   private Albero sinistro;
   private Albero destro;

   public boolean equals(Object o) {
      if (this == o) return true;
      if (!(o instanceof Albero)) return false;
      Albero a = (Albero) o;
      if (info != null && !info.equals(g.info))
          return false;
      if (info==null && g.info != null) return false;
      if(sinistro.equals(a.sinistro)
                && destro.equals(a.destro))
          return true;
      else return false;
        }
      public int hashCode() {
            …}
```

# String equals

```
String s1=”prova”;
String s2=”pro”;

s2 += “va”;

System.out.println(s1);
System.out.println(s2);

System.out.println(s1==s2);
```

**prova**
**prova**

**false**

# String equals

```
String s1=”prova”;
String s2=”prova”;
```

```
System.out.println(s1);          prova
System.out.println(s2);          prova

System.out.println(s1==s2);      true
```

# Integer equals

```
Integer i1=127;
Integer i2=127;
```

```
System.out.println(i1);
System.out.println(i2);

System.out.println(i1==i2);
```

**127**

**127**

**true**

# Integer equals

```
Integer i1=345;
Integer i2=345;
```

```
System.out.println(i1);          345
System.out.println(i2);          345

System.out.println(i1==i2);      false
```

# Perché questi esempi?

- Bisogna capire a fondo la differenza tra == e equals
- Purtroppo alcune volte la il comportamento è così complesso da testare che non è facile trovare eventuali errori
- Per esempio, nel caso degli interi se uso == al posto di equals me ne accorgo dopo il numero 128!!!

# equals

- Di solito quando vogliamo testare l'uguaglianza tra due oggetti abbiamo bisogno di usare il metodo `equals` e NON l'operatore ==
- I casi in cui abbiamo bisogno  di usare l'operatore == riguardano il confronto dell'identità dell'oggetto

# Esempio ==

```
public class MyJFrame extends JFrame implements
ActionListner{
  JButton creaProdotto= new Jbutton();
  JButton creaCliente = new JButton();
  Jlabel esitoJ = new JLabel("");
  public MyJFrame(){
    creaProdotto.setText("Crea");
    creaProdotto.addActionListener(this);
    CreaCliente.setText("Crea");
    creaCliente.addActionListener(this);
    ...}
  public void actionPerformed(ActionEvent evt){
    if (evt.getSource()==creaProdotto) {…}
    else {…}
  }
}
```

# Lista circolare

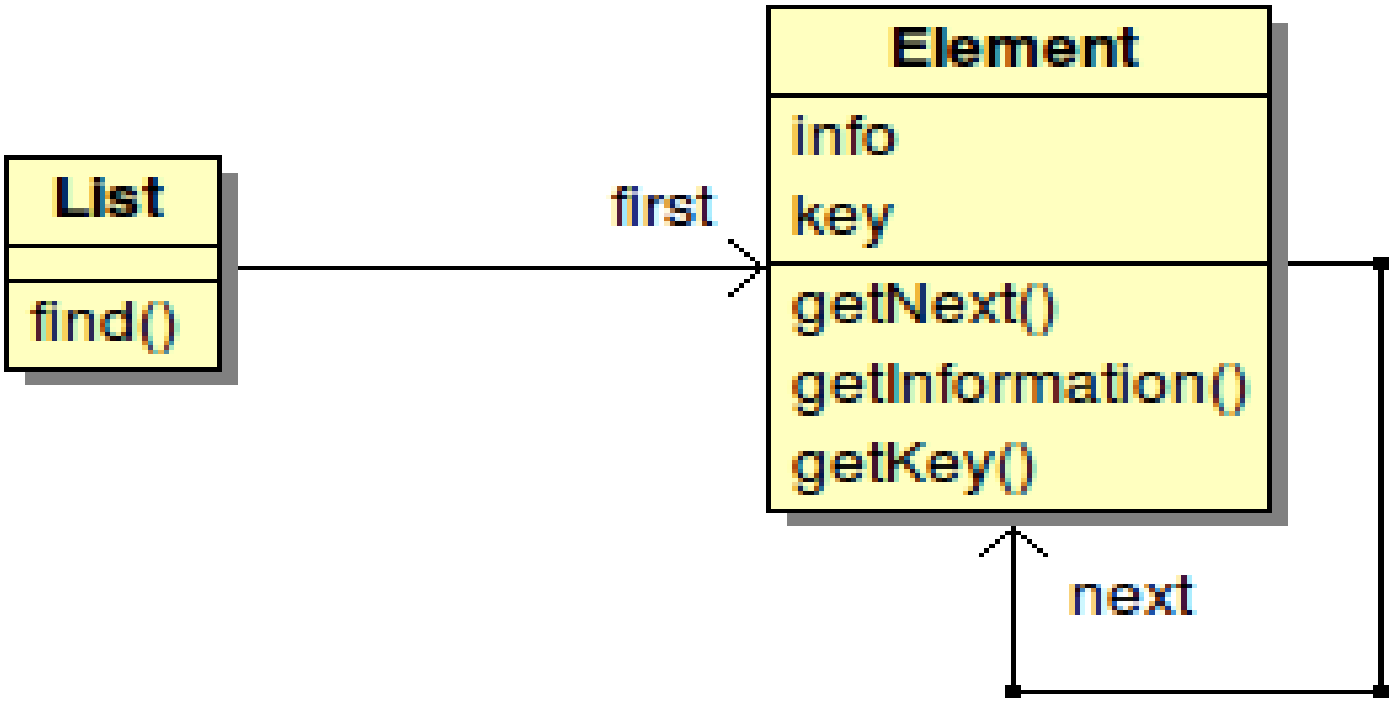
```
public class Element {
    Element next;
    Key key;
    Information info;
    public Element getNext(){
      return next;}
    public Key getKey(){
      Return key;}
    public Information getInformation() {
      return info;}
    …
}
```

# Lista circolare

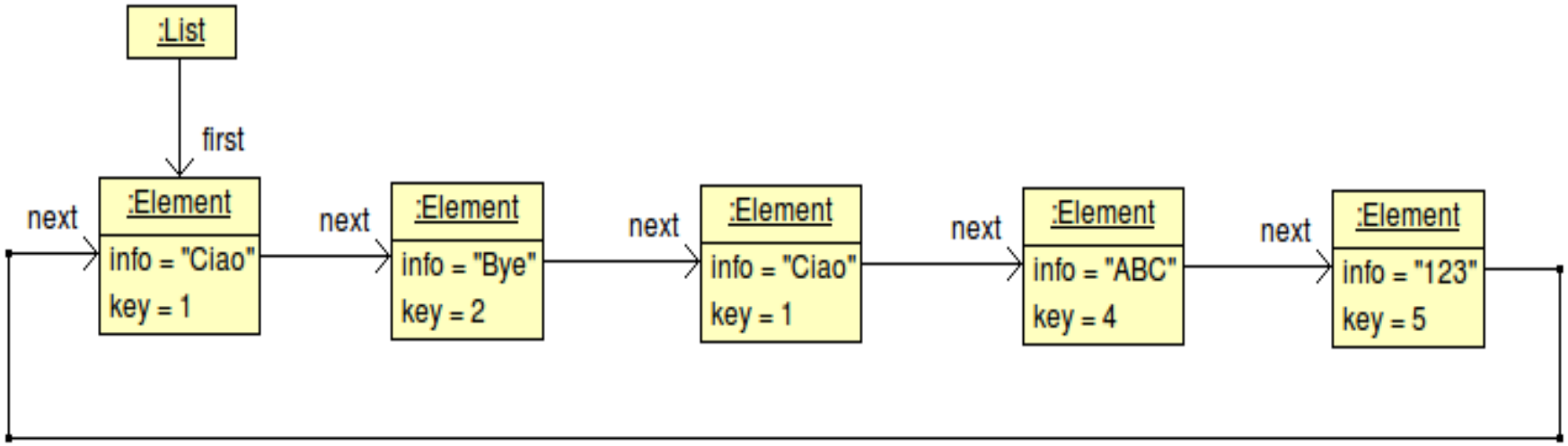
```
Public class List {
   Element first;

   public Information find(Key key){
      Element cursor=first;
      if (cursor==null)
        return null;
      do {
          if (cursor.getKey().equals(key))
            return cursor.getInformation();
          cursor=cursor.getNext();
      }
      while (!(cursor==first))
     return null;
}}
```

# Diagramma Classi



# Diagramma oggetti



# In pratica

- Non usare il controllo == per String, Integer, Date e tutti gli oggetti “valore” di Java
- Per gli oggetti riferimento, ponderare bene se serve controllare se l'oggetto è lo stesso (==) oppure se l'oggetto deve essere uguale ad un'altro (`equals`)

# Perché sovrascrivere hashCode

- Quando usiamo le collezioni (es. HashMap, HashSet etc.) queste usano i metodi equals e hashCode per confrontare gli oggetti della collezione.
- Quindi, se vogliamo che la collezione gestisca correttamente chiavi e elementi, questi devono implementare correttamente i metodi equals e hashCode.

# Cliente

```
public class Cliente {
   String cf;
   String nome;
   ...
   public boolean equals(Object o){
     if (!o instance of Cliente)
        return false;
    Cliente c = (Cliente) o;
    if (cf.equals(c.cf)) return true;
    else return false;
   }
   public int hashCode(){
     return cf.hashCode();
   }
   …
}
```

# Uso cliente

```
public class Banca {
    Set<Cliente> clienti = new HashSet<Cliente>();

    public Cliente creaCliente(String cf, String  nome, …)
{
    Cliente c = new Cliente(cf,nome,...);
    if (clienti.contains(c)){
       return null;
    else {
      clienti.add(c);
      return c;
    }
}
```

# Cliente

- Un oggetto cliente e' identificato dal codice fiscale.
- In questo esempio abbiamo visto che sono possibili piu' istanze di oggetto che rappresentano lo stesso oggetto fisico.
- Alternativamente potremmo impedire che questo avvenga usando opportuni pattern (es. Manager).

# Persistenza

- In Java la persistenza degli oggetti può essere gestita in vari modi. Abbiamo già visto come potrebbe essere gestita la persistenza tramite Database sfruttando l'API più conosciuta (JDBC)
- Oggi vedremo una soluzione più semplice e nativa in Java che fa uso di file: **Seralizzazione**

# Serializzazione

Per poter serializzare un oggetto è sufficiente che l'oggetto implementi l'interfaccia `Serializable`;

Tale interfaccia non definisce metodi. Quindi è sufficiente indicare "`implements Serializable`" senza fare altro

Per scrivere su un file un'oggetto si utilizza il metodo `writeObject` della classe `ObjectOutputStream`, mentre per leggere il metodo `readObject` di `ObjectInputStream`

# Esempio Serializable

```
public class Punto  implements Serializable{
   double x,y,z;
   public Punto(double x, double y,double z){...
   public double getX(){return x;}
   public double distance(Punto p){...}
   …
}
```

# Esempio Scrittura

```
public boolean store(Serializable s){
 try
 {
   FileOutputStream fos = new FileOutputStream(filename);
   ObjectOutputStream out = new ObjectOutputStream(fos);
   out.writeObject(s);
   out.close();
   Fos.close();
   return true;
 }
 catch (Exception e){return false;}
}
```

# Esempio Lettura

```
public Object load(){
  Object res=null;
  try
  {
    FileInputStream fis = new FileInputStream(filename);
    ObjectInputStream in = new ObjectInputStream(fis);
    res= in.readObject();
    in.close();
    fis.close();
    return res;
  }
  catch (Exception e){return null;}
}
```

# Note Serializable

- La classe dell'oggetto al momento del salvataggio deve essere la stessa al momento del caricamento.
- Non basta che abbia lo stesso nome, deve essere fatta allo stesso modo.
- Quindi se dopo il salvataggio dell'oggetto Punto, aggiungiamo, modifichiamo o togliamo un attributo della classe Punto, al momento del caricamento verrà generata un'eccezione del tipo `java.io.InvalidClassException`

# Transient

- Automaticamente tutti gli attributi della classe vengono salvati nel file e recuperati durante il caricamento.
- Se un attributo di  non può essere salvato (Thread, Stream, Socket, etc.) oppure non vogliamo salvarlo, basta usare la  parola chiave `transient`.

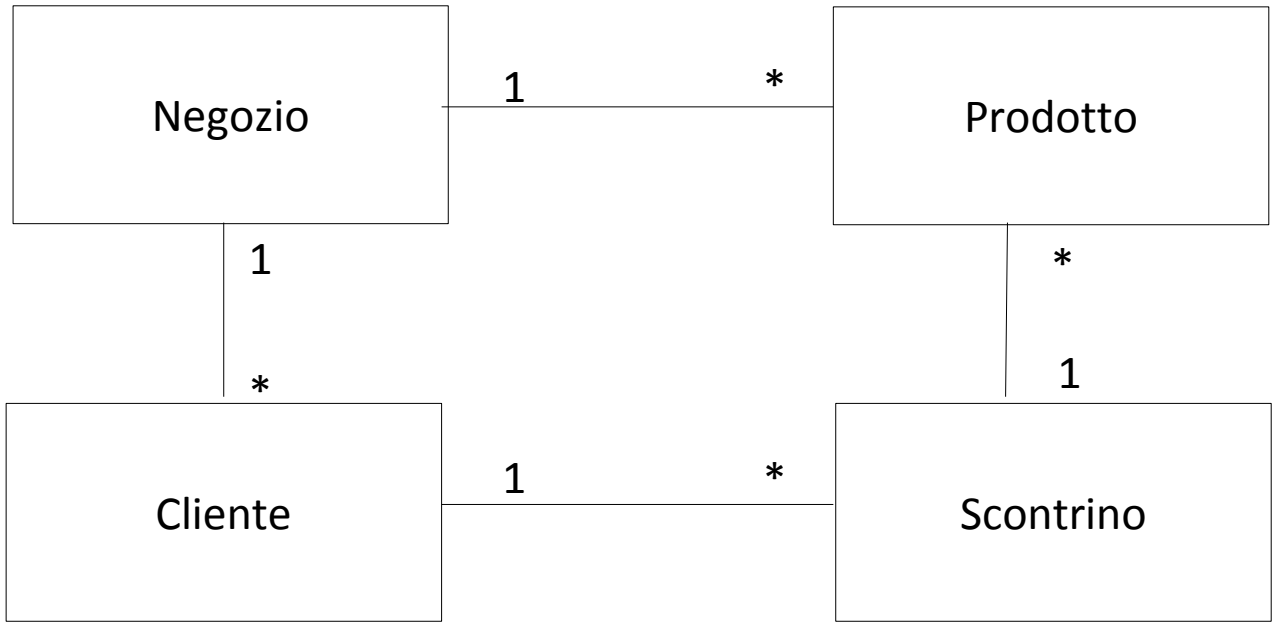
# Oggetti composti

```
public class Carrello  implements Serializable{
   Set<Prodotto> prodotti;
   ...
}
```

# Oggetti composti

- Gli oggetti collegati vengono salvati e recuperati contemporaneamente all'oggetto principale.
- Tutti gli oggetti conservano il proprio riferimento: quindi `o` il riferimento all'oggetto salvato e `n` è il riferimento all'oggetto recuperato, abbiamo `o==n` (che implica anche `o.equals(n)`)

# Riferimenti ciclici



# Riferimenti ciclici

- Se salviamo Negozio, vengono salvati una sola volta anche tutti gli altri oggetti collegati al Negozio.
- Quando carichiamo il Negozio, vengono caricati anche tutti gli altri oggetti collegati al Negozio.

# Come organizzare i dati

- Possiamo fare in modo che da un oggetto unico (Es. Negozio) siano “raggiungibili” tutti gli altri oggetti tramite i riferimenti.
- Quando carichiamo Negozio automaticamente vengono caricati anche tutti gli altri oggetti dell'applicazione.
- E quando salviamo Negozio, automaticamente vengono salvati anche tutti gli altri oggetti dell'applicazione.

# Esempio Caricamento

```
public static void main(String [] args){
   …
   Negozio negozio= (Negozio) load();
   if (negozio==null)
      negozio=new Negozio();
   …

}
```

# Esempio Scrittura

Nel Controller del caso d'uso “Esci dall'applicazione”. (Listner del pulsante “Esci”)

```
public void actionPerformed(ActionEvent evt){
  …
   store(negozio);//andrebbe controllato esito
   …
}
```

Come prototipo può bastare.

# domande