Il metodo `clone`

Il metodo `clone` serve per copiare gli oggetti

Si può invocare su quasi tutti gli oggetti delle classi predefinite

```
  public static void main(String args[]) {
    Point p, q;
    p=new Point(2,3);
    q=(Point) p.clone();
  }
```

Perchè il cast? Poi vediamo

`clone` e `Object`

È un metodo di `Object`

È definito `protected`

Quindi: si può usare solo:

- nello stesso package di `Object`
  (che è `java.lang`)
- nelle classi derivate

Restrizione su `clone`

Il `clone` di `Object` non si può usare nelle nostre classi:

```
class CloneObject {
  public static void main(String args[]) {
    Object o=new Object();
    Object s;

    s=o.clone();
  }
}
```

Viene dato questo errore in compilazione:

```
... clone() has protected access ...
```

Definizione di `clone`

Nella classe `Object`, è definito in questo modo:

```
protected Object clone()
```

Quindi:

1. ritorna un `Object`
2. è protected

Come clonare oggetti predefiniti

Non tutti gli oggetti sono clonabili:

```
String a, b;

a=new String("abcd");
b=(String) a.clone(); // errore
```

Per gli oggetti clonabili: il metodo ritorna un `Object`, quindi va fatto il cast:

```
Point p, q;
p=new Point(2,3);
q=(Point) p.clone();
```

Definire il metodo `clone`

Prendiamo la classe `Studente` e definiamo `clone`

```
class Studente {
  String nome;
  int anno;
}
```

Clonazione: primo tentativo

Facciamo overloading

```
class Studente {
  String nome;
  int anno;

    // errore logico
  Studente clone() {
    ...
  }
}
```

Non va bene: `clone` deve fare overloading del metodo di `Object`

Clonazione: secondo tentativo

Come per `equals`, il metodo deve avere esattamente la stessa intestazione del metodo di `Object`:

```
class Studente {
  String nome;
  int anno;

  Object clone() {
    ...
  }
}
```

Non funziona nemmeno cosí:

```
clone() in Studente cannot override
clone() in java.lang.Object;
attempting to assign weaker access
privileges; was protected
```

A parole: si possono allargare i diritti, ma non restringere

Va dichiarato `protected` oppure `public`

Clonazione: terzo tentativo

Lo dichiariamo `public` (si poteva anche dichiarare `protected`, se serviva):

```
class Studente {
  String nome;
  int anno;

  public Object clone() {
    ...
  }
}
```

Ora funziona: scriviamo il corpo

Clonazione: terzo tentativo

Il codice completo è:

```
class Studente {
  String nome;
  int anno;

  public Object clone() {
      // non del tutto corretto ...
    Studente s=new Studente();
    s.nome=this.nome;
    s.anno=this.anno;
    return s;
  }
}
```

Va bene per clonare oggetti `Studente`

Non va tanto bene se `Studente` ha sottoclassi

`clone` e `equals` per sottoclassi

Supponiamo che `Studente` abbia 10 sottoclassi
(Non è una cosa insolita che una classe abbia 10 sottoclassi)

Scriviamo i metodi `equals` e `clone` come detto prima in ognuna di queste 11 classi

Ora, `Studente` viene modificato con l'aggiunta di una campo `cod_fiscale`

Vanno modificate tutte e 11 le classi

Non solo: altri programmatori potrebbero aver realizzato altre sottoclassi senza che io me ne accorga; in queste sottoclassi, `equals` e `clone` non funzionano

---

Ereditarietà

Sia per `equals` che per `clone`

In una classe derivata, è opportuno definire `equals` e `clone` in termini della loro definizione nella sovraclasse:

```
class Borsista extends Studente {
  int stipendio;

  public String toString() {
    return "["+this.nome+","+this.anno+","+this.stipendio+"]";
  }

  public boolean equals(Object o) {
    Borsista b;

    if(!super.equals(o))
      return false;

    b=(Borsista) o;

    if(this.stipendio!=b.stipendio)
      return false;

    return true;
  }
}
```

`super.equals` fa la parte dei controlli relativi alle componenti ereditate

In questo modo, se si modifica `Studente`, non vanno modificate le sottoclassi

---

Clonazione in classi derivate

Faccio la stessa cosa:

```
class Borsista extends Studente {
  int stipendio;

  ...

  public Object clone() {
    Borsista b=(Borsista) super.clone();

    b.stipendio=this.stipendio;

    return b;
  }
}
```

Qui abbiamo un problema

Ereditare `clone`

```
  public static void main(String args[]) {
    Borsista s=new Borsista();
    Borsista q;

    q=(Borsista) s.clone();
  }
```

In **esecuzione** viene dato questo errore:

```
Exception in thread "main" java.lang.ClassCastException
        at Borsista.clone(Borsista.java:23)
        at ProvaBorsista.main(ProvaBorsista.java:6)
```

La radice del problema

Facendo `super.clone()` nella classe `Borsista`, viene invocato il metodo `clone` di `Studente`

In `Studente`, il metodo `clone` è definito come un metodo che crea un oggetto `Studente`

Il cast a `Borsita` fallisce

Ereditare `clone` da `Object`

Il metodo standard di clonazione è questo: ogni oggetto invoca il `clone` della sovraclasse

Il `clone` di `Object` fa la creazione del nuovo oggetto e ci copia le componenti del vecchio

Il `clone` di `Object` crea un oggetto della stessa classe di quello di partenza, non (necessariamente) un oggetto `Object`

Invocare `clone` di `Object`

La versione complessiva è complicata

Il concetto di base (che non funziona) è che ogni `clone` invoca il `clone` della sovraclasse

```
class Studente {
  ...
        //  ancora non funziona...
  public Object clone() {
    return super.clone();
  }
}
```

Lo stesso vale per classi derivate:

```
class Borsista extends Studente {
  ...
        //  ancora non funziona...
  public Object clone() {
    return super.clone();
  }
}
```

`CloneNotSupportedException`

I metodi `clone` scritti sopra non funzionano:

```
Studente.java:35: unreported exception
java.lang.CloneNotSupportedException; must be caught or
declared to be thrown
    return super.clone();
```

Questo è dovuto al modo in cui è realizzato il metodo `clone` di `Object`

Metodo `clone` di `Object`

Il metodo `clone` di `Object` è definito in questo modo:

- se l'oggetto appartiene a una classe che *non* implementa l'interfaccia `Cloneable`, lancia una eccezione `CloneNotSupportedException`
- altrimenti, fa una copia esatta dell'oggetto

Questo ha due conseguenze:

- occorre fare `try-catch` dell'eccezione
- la classe deve implementare l'interfaccia `Cloneable`

Nota:

- non fare `try-catch` produce un errore in compilazione
  (il metodo `super.clone()` dichiara di lanciare un eccezione che però non viene catturata)
- non implementare l'interfaccia produce un errore in esecuzione
  (perchè `super.clone()` in questo caso lancia una eccezione)

Notare che il primo errore è sintattico: se un metodo dichiara (con `throw`) di poter lanciare una eccezione, questa va rilanciata o catturata

Che poi il metodo lanci effettivamente l'eccezione o no, in fase di compilazione non è verificabile

---

`Cloneable`: motivazione

Perchè tutte queste complicazioni?

Soluzioni alternative:

- l'interfaccia `Cloneable` contiene `clone` mentre `Object` no
- la `clone` di `Object` fa la copia e basta

Problemi delle due soluzioni:

- ogni classe deve fare la creazione del nuovo oggetto e la copiatura (dato che una interfaccia può solo contenere l'intestazione di un metodo, il metodo va poi realizzato completamente nella classe)
- la clonazione non ha senso per tutti gli oggetti

Clonazione usando `clone` di `Object`

```
class Studente implements Cloneable {
  ...

  public Object clone() {
    try {
      return super.clone();
    }
    catch(CloneNotSupportedException e) {
      return null;
    }
  }
}
```

- la classe deve implementare `Cloneable`
- dato che `clone` di `Object` ha `throws CloneNotSupportedException`, va catturata

Clonazione in classi derivate

Se `Borsista` estende `Studente`:

```
class Borsista extends Studente {
  ...

  public Object clone() {
    return super.clone();
  }
}
```

Commenti:

non serve `implements Cloneable`
: dato che `Studente` implementa l'interfaccia `Cloneable`, anche `Borsista` la implementa automaticamente in modo indiretto

non serve catturare l'eccezione
: viene già fatto nel `clone` di `Studente`

Chi clona l'oggetto?

Quando si invoca `clone` di `Borsista`:

1. viene invocato `clone` di `Studente`
2. che invoca `clone` di `Object`

È sempre `clone` di `Object` che fa la copia!

Copia superficiale e profonda

Se un oggetto contiene altri riferimenti a oggetti:

copia superficiale
: viene copiato solo l'oggetto

copia profonda
: viene copiato l'oggetto e tutti quelli collegati

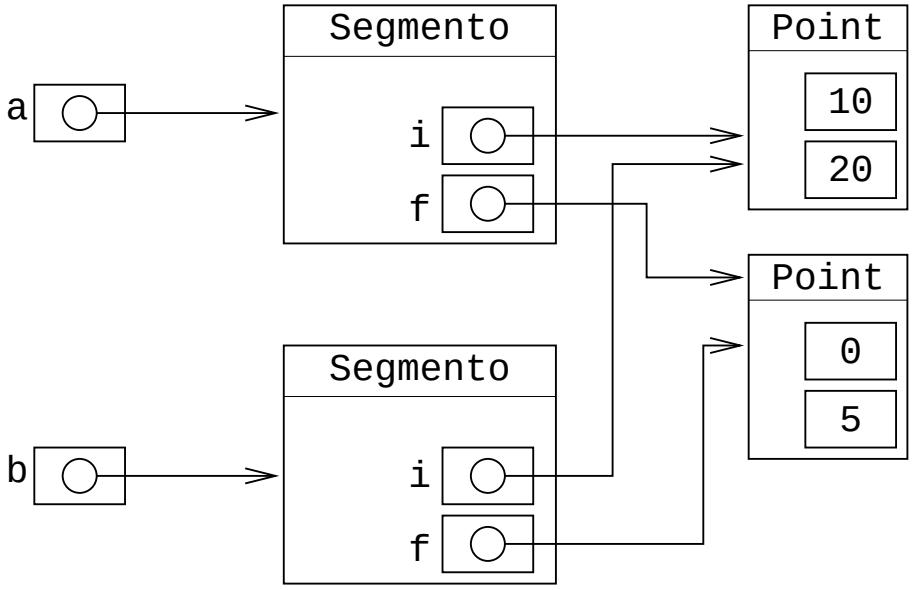
Esempio

Classe `Segmento`

```
import java.awt.*;

class Segmento {
  Point i, f;
}
```

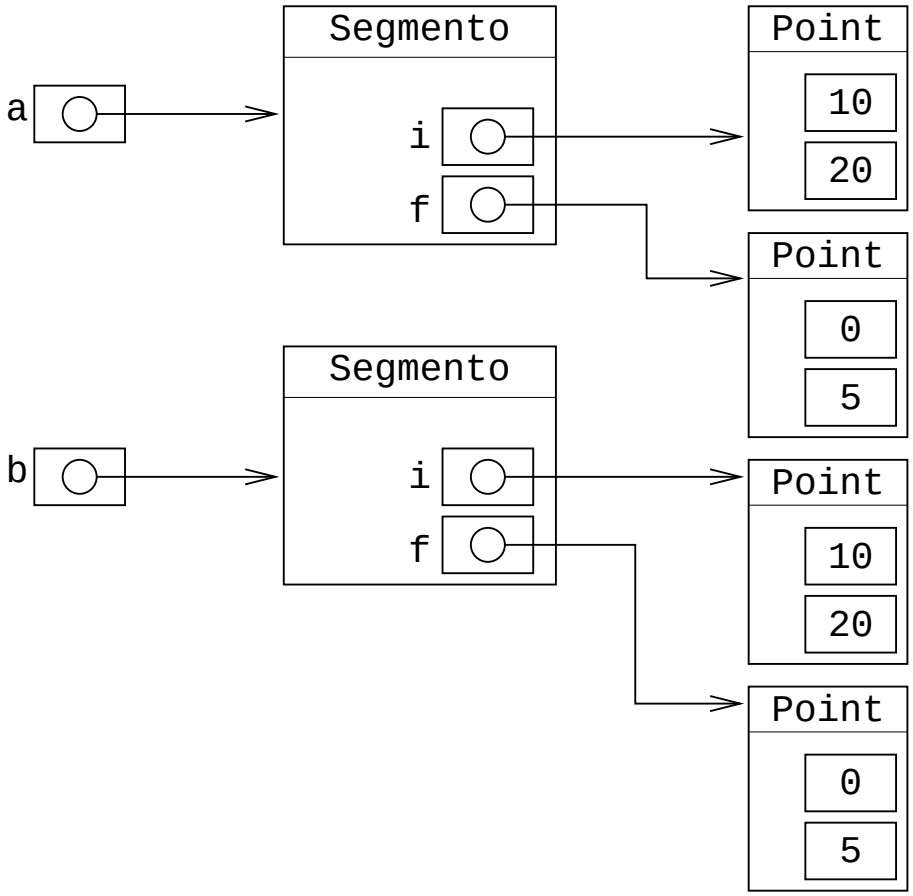
Cosa ci si aspetta

Confronto/clonazione superficiale:



Gli oggetti `Point` sono esattamente gli stessi

Confronto/clonazione profonda:



Gli oggetti `Point` non sono gli stessi, ma hanno gli stessi valori dentro

Uguaglianza superficiale

Si confrontano i riferimenti degli oggetti

Due oggetti `Segmento` sono `equals` se i loro compi sono esattamente uguali (contengono riferimenti agli stessi identici oggetti

```
import java.awt.*;

class Segmento {
  Point i, f;

        // equals superficiale
  public boolean equals(Object o) {
    if(o==null)
      return false;

    if(this.getClass()!=o.getClass())
      return false;

    Segmento s=(Segmento) o;

    return((this.i==s.i)&&(this.f==s.f));
  }
}
```

Confronto profondo

Gli oggetti vengono confrontati in base ai loro valori, non ai loro riferimenti:

```
import java.awt.*;

class Segmento {
  Point i, f;

  public boolean equals(Object o) {
    if(o==null)
      return false;

    if(this.getClass()!=o.getClass())
      return false;

    Segmento s=(Segmento) o;

    if(this.i==null) {
      if(s.i!=null)
        return false;
    }
    else if(!this.i.equals(s.i))
      return false;

    if(this.f==null) {
      if(s.f!=null)
        return false;
    }
    else if(!this.f.equals(s.f))
      return false;
```

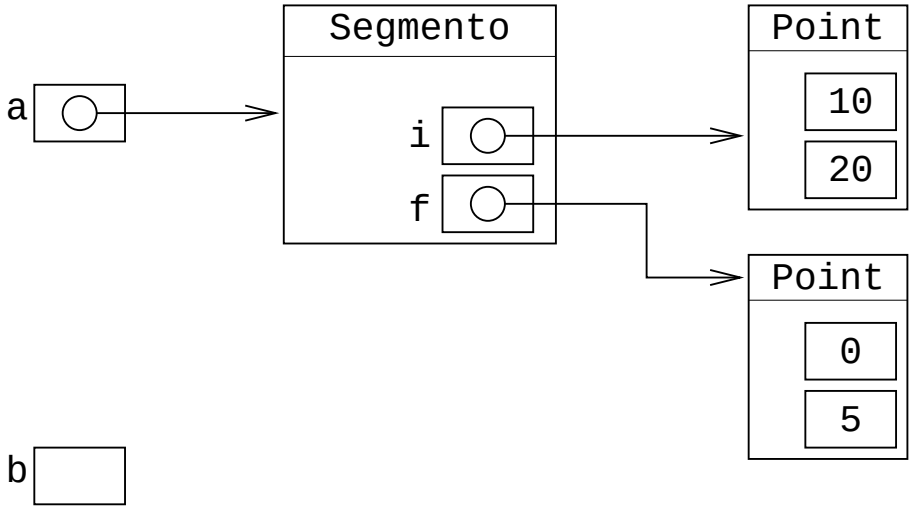
```
    return true;
  }
}
```

Copia superficiale

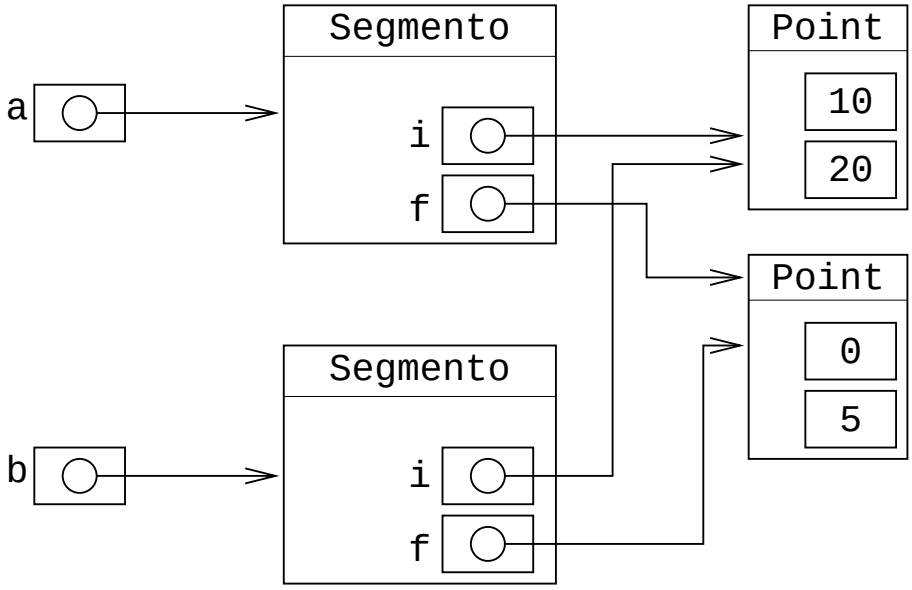
`clone` di `Object` fa la copia del solo oggetto

```
import java.awt.*;

class Segmento implements Cloneable {
  Point i, f;

  public Object clone() {
    try {
      return super.clone();
    }
    catch(CloneNotSupportedException e) {
      return null;
    }
  }
}
```

Prima della copia:

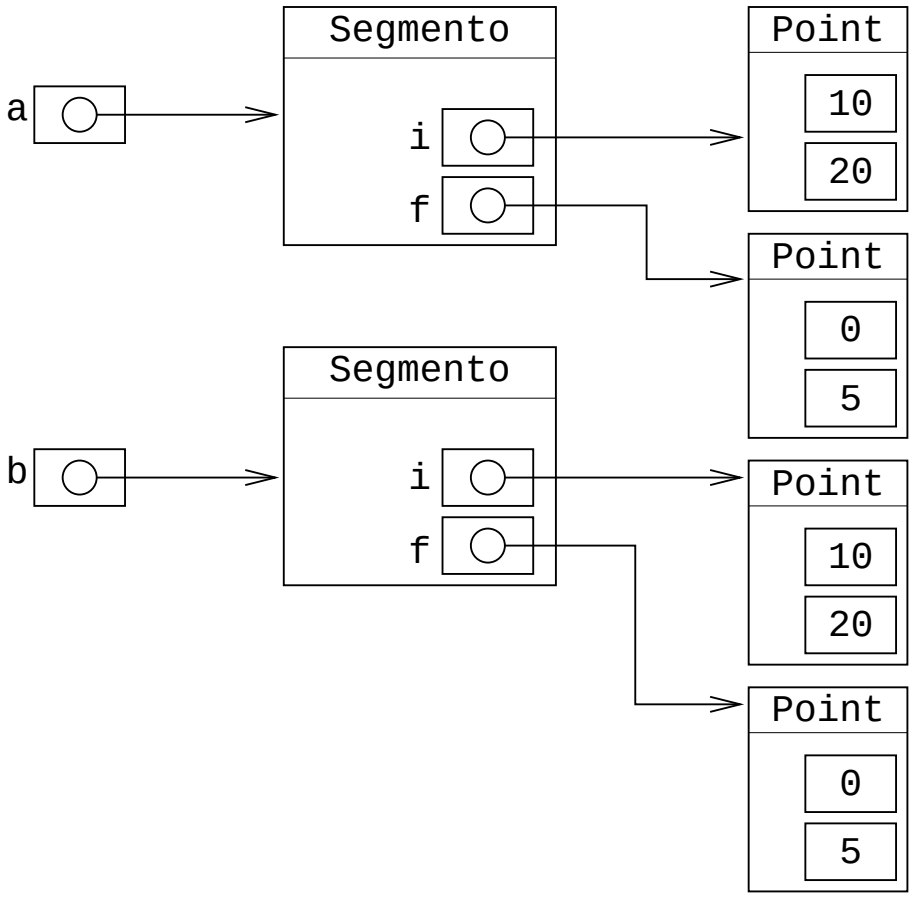


Dopo la copia superficiale:



Copia profonda

Ha senso richiedere una copia anche dei due oggetti `Point`



Questo non viene fatto da `clone` di `Object`

Va fatto manualmente:

Copia profonda: realizzazione

Occorre invocare `clone` su ognuno degli oggetti di cui voglio fare la copia

Notare che `super.clone` ha tipo di ritorno `Object`

Per poter accedere alle componenti, devo fare il cast

Lo stesso vale per le componenti

```
import java.awt.*;

class Segmento {
  Point i, f;

  ...

  public Object clone() {
    try {
      Segmento s;
      s=(Segmento) super.clone();

      s.i=(Point) this.i.clone();
      s.f=(Point) this.f.clone();

      return s;
    }
    catch(CloneNotSupportedException e) {
      return null;
    }
  }
}
```