Ereditarietà (ultima)

Classi astratte

- **Un ibrido tra classi e interfacce**
- **Hanno alcuni metodi implementati normalmente, altri *astratti***
  - Un metodo astratto non ha implementazione

```
abstract class AccountTransaction
{
   . . .
   public abstract void esegui();
   . . .
}
```

  - Il metodo, e la classe sono **`abstract`**

Classi astratte

- **Le classi astratte possono comunque avere costruttori**
  - invocabili solo dai costruttori di sottoclasse

Classi astratte

```
abstract class AccountTransaction
{
  private BankAccount acct;

  public AccountTransaction(BankAccount a)
  {
     acct = a ;
  }

  public BankAccount getAcct()
  {
     return acct;
  }

  public abstract void esegui();

}
```

Sottoclassi concrete

```
class Prelievo extends AccountTransaction
{
  private double importo;

  public Prelievo(BankAccount a, double i)
  {
     super(a);
     importo = i;
  }

  public void esegui()
  {
     getAcct().preleva(i);
  }
}
```

Sottoclassi concrete

- **Le classi che estendono una classe astratta DEVONO implementare (sovrascrivere) tutti I metodi astratti**
  - altrimenti anche le sottoclassi della classe astratta sono astratte

Classi astratte

- **E’ possibile dichiarare astratta una classe priva di metodi astratti**
  - In questo modo evitiamo che possano essere costruiti oggetti di quella classe

Eventi del mouse

```
abstract class MouseAdapter implements MouseListener
{
   // un metodo per ciascun mouse event su una componente
   public void mousePressed(MouseEvent event) {/* donothing */};

   public void mouseReleased(MouseEvent event){/* donothing */};

   public void mouseClicked(MouseEvent event) {/* donothing */};

   public void mouseEntered(MouseEvent event) {/* donothing */};

   public void mouseExited(MouseEvent event)  {/* donothing */};

}
```

Continua…

Eventi del mouse

- **`MouseAdapter` è abstract e quindi non può essere istanziata direttamente, anche se tutti i suoi metodi sono concreti**
- **possiamo estenderla con un classe concreta**

```
class MyAdapter extends MouseAdapter
{
   /* gestisce solo gli eventi “click” */
   public void MouseClicked { . . . }
}
```

Metodi e (sotto) classi `final`

- **È possibile impedire la ridefinizione di un metodo in una sottoclasse, dichiarando il metodo `final`**
- **Oppure dichiarare non ridefinibili tutti i metodi della classe, definendo la classe stessa `final`**

Continua…

Metodi e (sotto) classi `final`

- **Due motivazioni**
  - Efficienza: metodi final hanno dispatch statico
  - Sicurezza

```
public class SecureAccount extends BankAccout
{
 public final boolean checkPassword(String pwd)
 { . . . }
}
```

Continua…

Controllo degli accessi

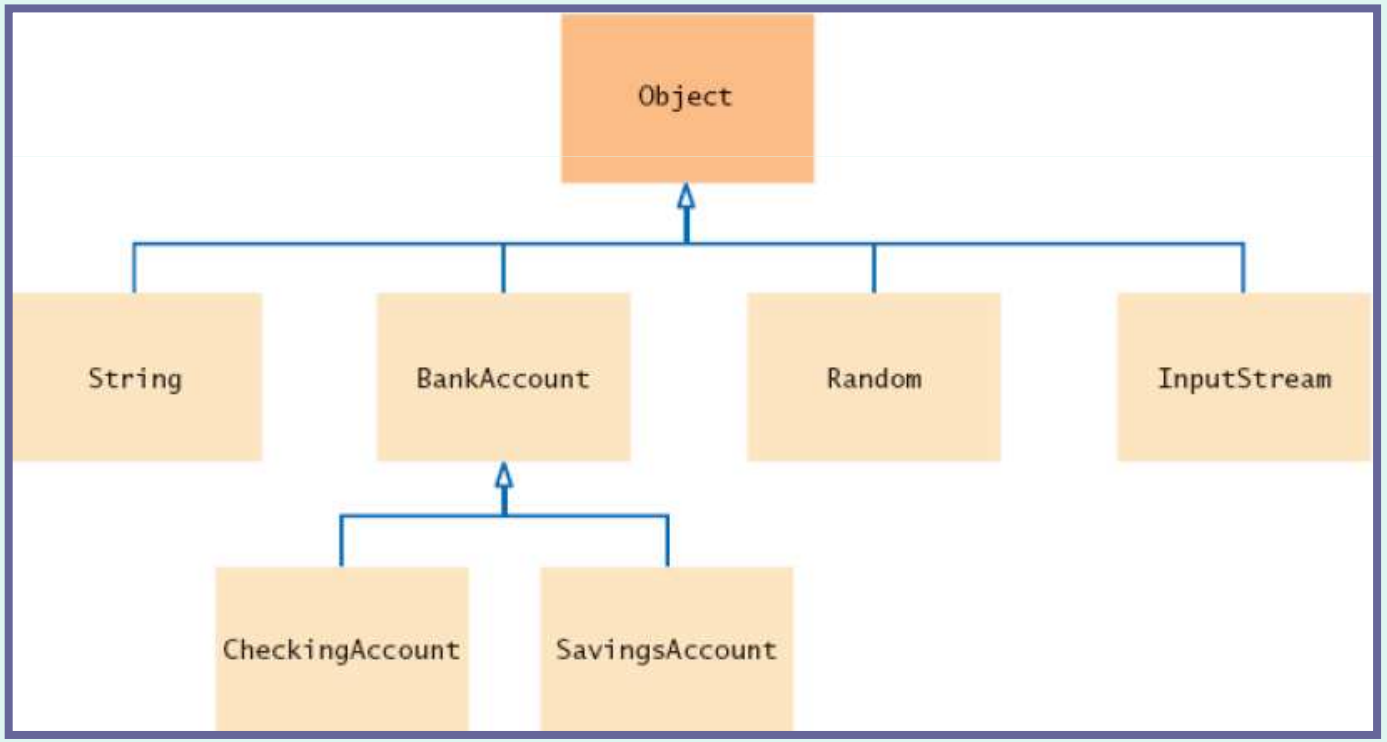
- **Java ha quattro livelli di accessibilità per gli elementi di una classe**
  - **`public`**
    - accessibile dai metodi di qualunque classe
  - **`private`**
    - accessibile solo dai metodi della proria classe
  - **`protected`**
    - accessibile alle sottoclassi
  - *package*
    - accessibile da tutte le classi dello stesso package
    - Il livello default, quando non specifichiamo alcun livello di accesso in modo eseplicito

Protected

```
abstract class AccountTransaction
{
  private BankAccount acct;

  protected AccountTransaction(BankAccount a)
  {
     acct = a ;
  }

  protected BankAccount getAcct()
  {
     return acct;
  }

  public abstract void esegui();

}
```

Object: la superclasse cosmica

- **Tutte le classi definite senza una clausola `extends` esplicita estendono automaticamente la classe `Object`**



Object: la superclasse cosmica

- I metodi più utili definiti da questa classe
  - `String toString()`
  - `boolean equals(Object otherObject)`
  - `Object clone()`
- Spesso *overridden* nelle classi di sistema e/o nelle classi definite da utente

`toString()`

- **`Object.toString()` restituisce una stringa ottenuta dalla concatenazione del nome della classe seguita dal codice hash dell'oggetto su cui viene invocato.**

Continua…

toString()

- invocato automaticamente tutte le volte che concateniamo un oggetto con una stringa
- ridefinito in tutte le classi predefinite per fornire una rappresentazione degli oggetti come stringhe

```
Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();
// s = "java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

Continua…

toString()

- **Possiamo ottenere lo stesso effetto nelle classi *user-defined*, ridefinendo il metodo:**

```
public String toString()
{
   return "BankAccount[balance=" + balance + "]";
}
```

```
new BankAccount(5000).toString();

// restituisce "BankAccount[balance=5000]"
```

`equals()`

- **nella classe `Object` coincide con il test `==`**
  - verifica se due riferimenti sono identici
- **sovrascritto in tutte le classi predefinite per implementare un test di *uguaglianza di stato***
- **Il medesimo effetto si ottiene per le classi *user-defined*, sempre mediante overriding**
- **La firma del metodo in `Object`:**

```
public boolean equals(Object otherObject)
```

Continua…

Esempio

- **Monete e monete da collezione**

```
class Coin
{
  private double value;
  private String name,
  . . .
}


class CollectibleCoin extends Coin
{
   private int year;
. . .
}
```

Monete uguali

```
public class Coin
{
  // sovrascrive il metodo di object
  public boolean equals(Object otherObject)
  {
    if (otherObject == null) return false;
    Coin other = (Coin) otherObject;
    return name.equals(other.name) && value == other.value;
  }
  . . .
}
```

NOTE

- *cast* per recuperare informazione sul parametro
- *equals* per confrontare campi riferimento

Continua …

Monete uguali

```
public class Coin
{
   . . .
   public boolean equals(Object otherObject)
   {
      if (otherObject == null) return false;
      Coin other = (Coin) otherObject;
      return name.equals(other.name) && value == other.value;
   }
   . . .
}
```

- Se `otherObject` non è un `Coin`, errore ...

Continua ...

Monete uguali

```
public class Coin
{
  // nuova versione del metodo equals
  public boolean equals(Coin other)
  {
   if (other == null) return false;
   return name.equals(other.name) && value == other.value;
  }

  // sovrascrive il metodo equals di object
  public boolean equals(Object other)
  {
    if (other == null) return false;
    if (other instanceof Coin) return equals((Coin)other);
    return false;
  }
  . . .
}
```

cast forza l'invocazione della versione corretta

Monete uguali

- **Nelle sottoclassi stessa logica ...**

```
class CollectibleCoin extends Coin {
{
 public boolean equals(CollectibleCoin other) { . . . }

 public boolean equals(Object other)
  {
    if (other == null) return false;
    if (other instanceof CollectibleCoin)
              return equals((CollectibleCoin)other);
    return false;
  }
  . . .
```

Monete uguali

- … ma attenzione alla struttura ereditata!

```
public boolean equals(CollectibleCoin other)
{
 if (!super.equals(other)) return false;
 return year == other.year;
}
```

Domanda

Quale versione di equals nella classe coin invoca questa chiamata?

```
  public boolean equals(CollectibleCoin other)
  {
   if (!super.equals(other)) return false;
   return year == other.year;
  }

 private int year;
}
```

Risposta

- **La versione con il parametro di tipo `Coin`**
  - **`other:CollectibleCoin`** è anche un **`Coin`**

Domanda

- Cosa dobbiamo aspettarci dalla chiamata `x.equals(x)`? Deve sempre restituire `true`?

Risposta

- **Si, a meno che, ovviamente, `x` non sia il riferimento `null`.**

clone()

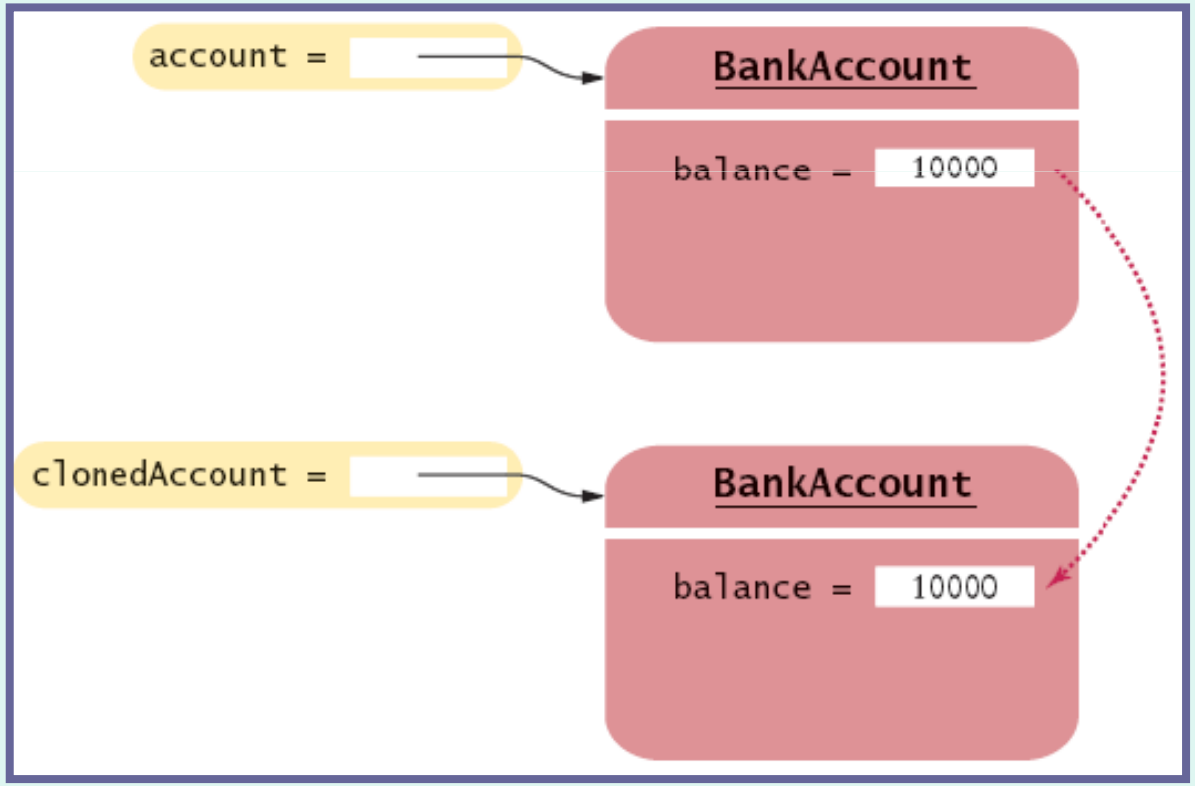
- **Come ben sappiamo, l'assegnamento tra due riferimenti crea due riferimenti che puntano allo stesso oggetto**

```
BankAccount account2 = account;
```

Continua…

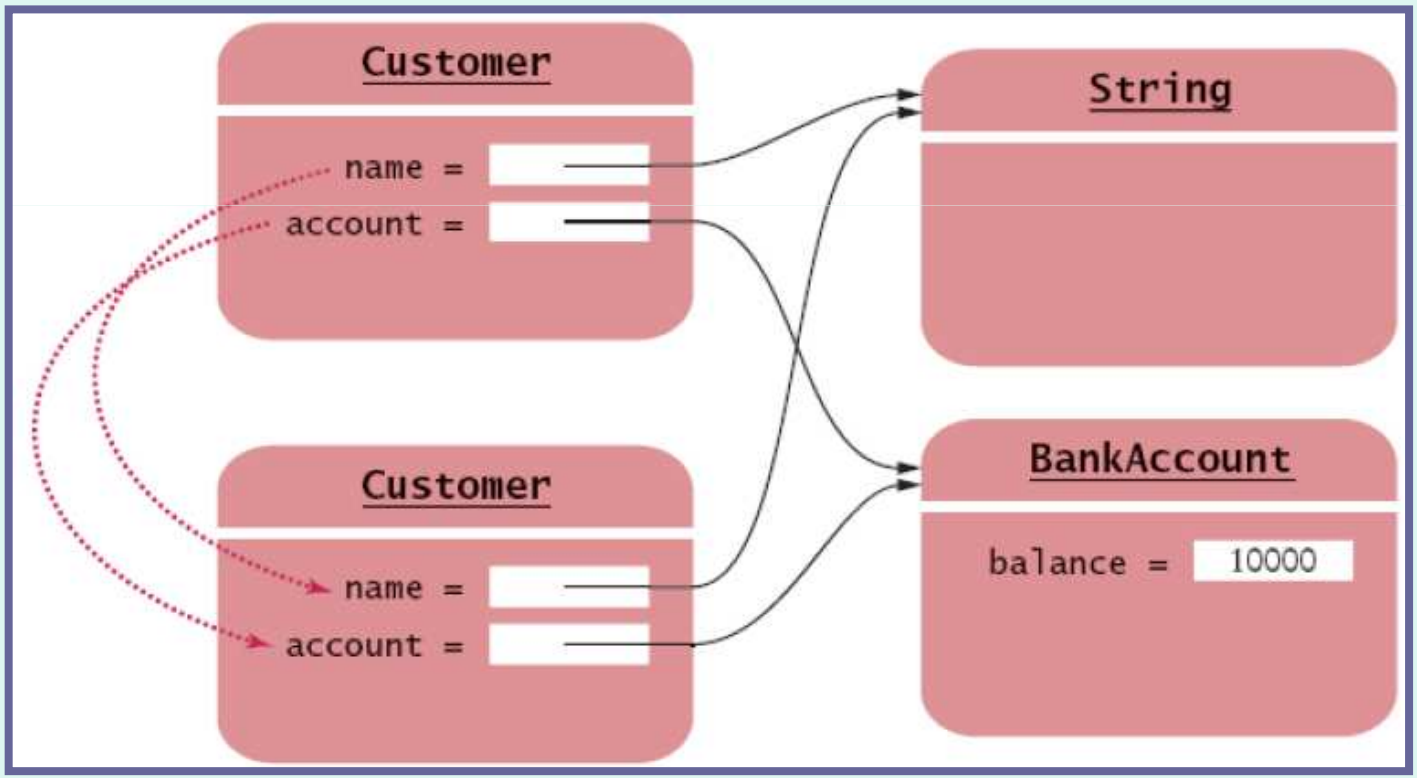
clone()

- **Talvolta è utile/necessario creare una copia dell'intero oggetto, non del solo riferimento**



Object.clone()

- **Crea *shallow copies* (copie superficiali)**



Continua…

`Object.clone()`

- Non ripete il cloning sistematicamente sui campi di tipo riferimento
- Dichiarato `protected` per proibirne l'invocazioni su oggetti la cui classe non abbia ridefinito esplicitamente `clone()` con livello di accesso `public`
- Controlla che l'oggetto da clonare implementi l'interfaccia `Cloneable`
- La ridefinizione del metodo nelle sottoclassi richiede attenzione (vedi API)

`clone()` – overriding

- Una classe che ridefinisce `clone()` deve attenersi alla firma del metodo nella classe `Object`

```
Object clone()
```

- L'uso del metodo che ne deriva è:

```
BankAccount cloned = (BankAccount) account.clone();
```

- Necessario il cast perchè il metodo ha `Object` come tipo risultato