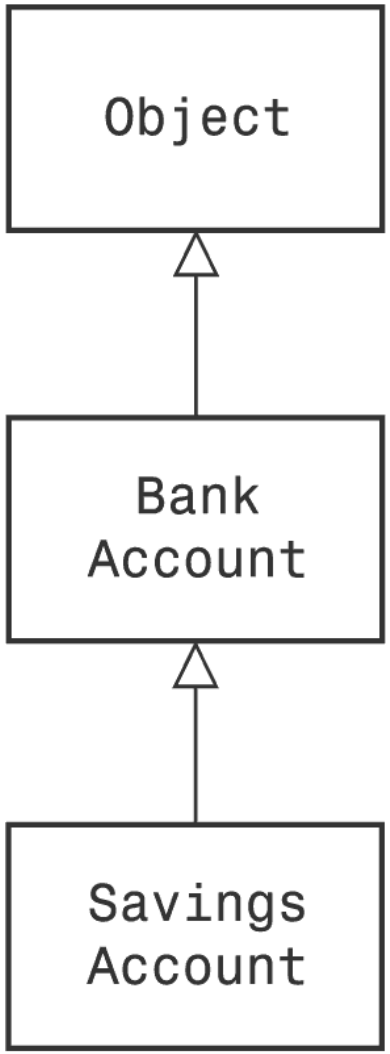
Classe Object

`Object`: La classe universale

- Ogni classe che non estende un'altra classe, estende per default la classe **`Object`**
- Metodi della classe **`Object`**
  - **`String toString()`**
    - Restituisce una rappresentazione dell'oggetto in forma di stringa
  - **`boolean equals(Object o)`**
    - Verifica se l'oggetto è uguale a un altro
  - **`Object clone()`**
    - Crea una copia dell'oggetto
- E' opportuno sovrascrivere questi metodi nelle nostre classi



Object: la classe universale

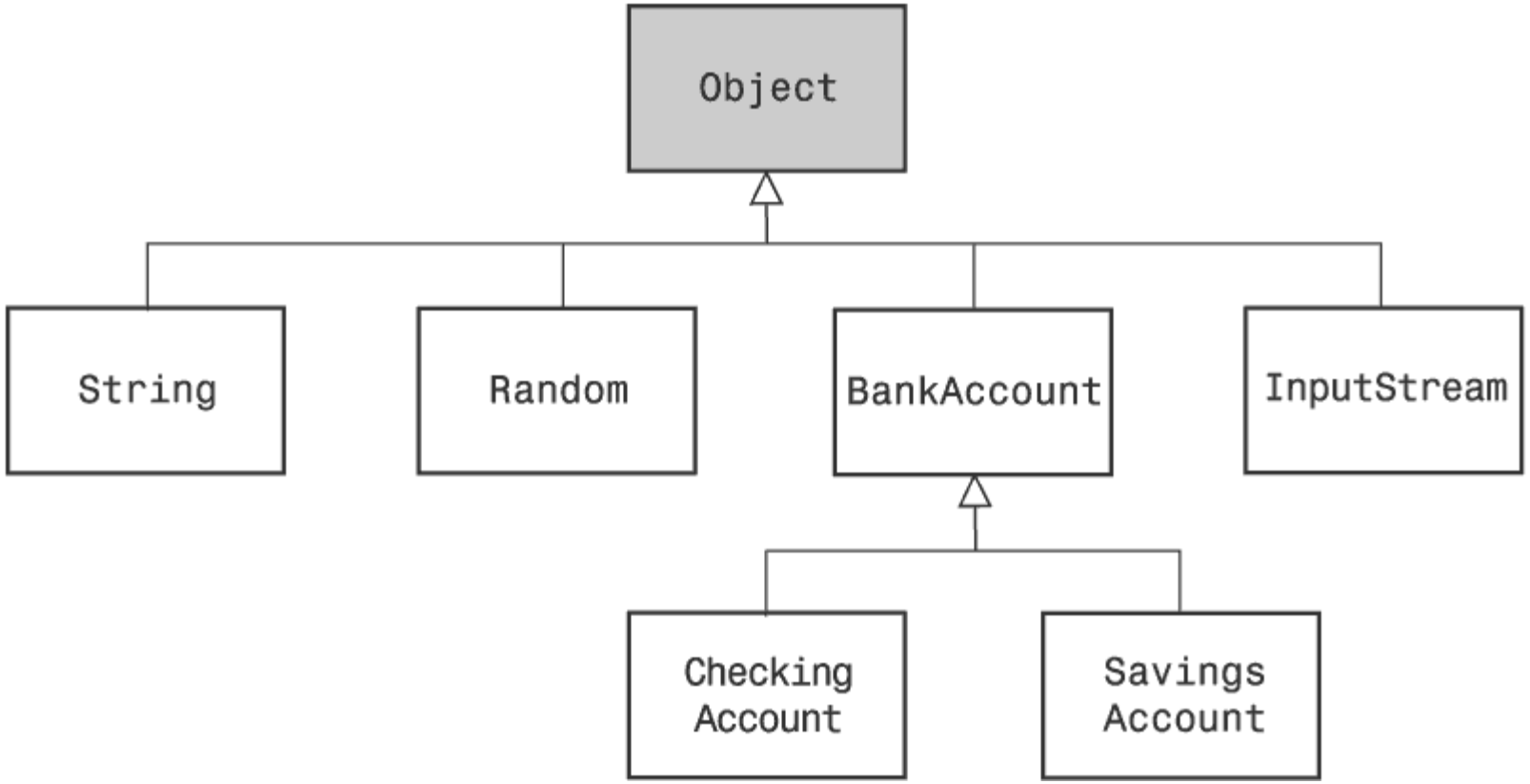


Figura 8 La classe Object è la superclasse di tutte le classi Java

Il metodo `toString`

- Restituisce una stringa contenente lo stato dell'oggetto in un formato specifico

```
Rectangle cerealBox =
                        new Rectangle(5,10,20,30);
String s = cerealBox.toString();
```

dove **s** si riferisce alla stringa

"java.awt.Rectangle[x=5,y=10,width=20,height=30]"

- Il formato da rispettare è:

nomeClasse lista_variabili_istanza_e_valori

dove la lista ha il formato

[instVarName$_1$=val$_1$, …,instVarName$_n$=val$_n$]

es. **[x=5,y=10,width=20,height=30]**

Il metodo `toString`

- per gli oggetti di una sottoclasse di una classe != da **Object**

```
SavingsAccount alexAccount =
                    new SavingsAccount(5,10,20,30);
String s = alexAccount.toString();
```

dove **s** si riferisce alla stringa

"SavingsAccount[balance=1000][interestRate=10]"

- Il formato da rispettare è:

nomeClasse listeSuperclassi listaClasse

Conversione di oggetti a String

- **`toString()`** può essere usato per convertire il parametro implicito in un oggetto String
- viene invocato automaticamente quando si concatena un oggetto di qualsiasi tipo con un oggetto di tipo String:

```
"cerealBox=" +cerealBox
```

viene valutata:

```
"cerealBox=java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

Conversione di oggetti a String

- **`toString()`** può essere invocato su qualsiasi oggetto
  - ogni classe estende la classe Object
  - **`toString()`** è nell'interfaccia pubblica di ogni classe
- la conversione automatica con operatore **+** avviene solo se uno dei due oggetti è già di tipo String
  - se nessuno dei due oggetti è di tipo String il compilatore genera un errore

Sovrascrivere `toString`

- Proviamo a usare il metodo **`toString()`** nella classe **`BankAccount`**:

```
BankAccount momsSavings =
                    new BankAccount(5000);
String s  = momsSavings.toString();
```

`s` si riferisce a **`"BankAccount@d24606bf"`**

- Viene stampato il nome della classe seguito dall'indirizzo in memoria dell'oggetto
- Siamo interessati al dato contenuto nell'oggetto!

Sovrascrivere `toString`

- E' importante fornire il metodo **`toString()`** in tutte le classi!
  - Ci consente di controllare lo stato di un oggetto
  - Se x è un oggetto e abbiamo sovrascritto **`toString()`**, possiamo invocare **`System.out.println(x)`**
  - Il metodo **`println`** della classe **`PrintStream`** invoca **`x.toString()`**

Sovrascrivere `toString`

- Dobbiamo sovrascrivere il metodo nella classe **`BankAccount`**:

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

- In tal modo:

```
BankAccount momsSavings = new BankAccount(500);
String s  = momsSavings.toString();
```

`s` si riferisce a **"BankAccount[balance=500]"**

Sovrascrivere `toString` in sottoclassi

- Nella sottoclasse dobbiamo sovrascrivere **`toString()`** e aggiungere i valori delle vbl di istanza della sottoclasse

```
public class SavingsAccount
                        extends BankAccount
{
  public String toString(){
    return super.toString() +
          "[interestRate=" + interestRate + "]";
  }
  ……
}
```

Sottoclassi: sovrascrivere `toString`

- Osserviamo un'invocazione di **`toString()`** su un oggetto di tipo **`SavingsAccount`**:

```
SavingsAccount sa = new SavingsAccount(10);
System.out.println(sa);
```

stampa

"BankAccount[balance=1000][interestRate=10]"

invece di

"SavingsAccount[balance=1000][interestRate=10]"

Sovrascrivere `toString`

- E' preferibile non scrivere il nome della classe come stringa, ma determinarlo dinamicamente con

  `getClass().getName()`

  - Il metodo `getClass()` (classe `Object`) restituisce un oggetto di tipo `Class`, da cui possiamo ottenere informazioni relative alla classe
    - `Class c = e.getClass();`
  - il metodo `getName()` della classe `Class` restituisce la stringa contenente il nome della classe

- In BankAccount l'implementazione corretta è:

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

Sovrascrivere `toString`

- Se scegliamo di ereditare **`toString()`** in **`SavingsAccount`**

```
SavingsAccount sa = new SavingsAccount(10);
System.out.println(sa);
```

stampa **"SavingsAccount[balance=1000]"**

bene il nome della classe ma manca interestRate

Sottoclassi: sovrascrivere `toString` riutilizzando codice superclasse

- Nella sottoclasse dobbiamo sovrascrivere **`toString()`** e aggiungere i valori delle vbl di istanza della sottoclasse

```
public class SavingsAccount extends BankAccount{
  public String toString(){
    return super.toString() +
           "[interestRate=" + interestRate + "]";
  }
  ……
}
```

Sottoclassi: sovrascrivere `toString`

- Vediamo la chiamata su un oggetto di tipo **`SavingsAccount`**:

```
SavingsAccount sa = new SavingsAccount(10);
System.out.println(sa);
```

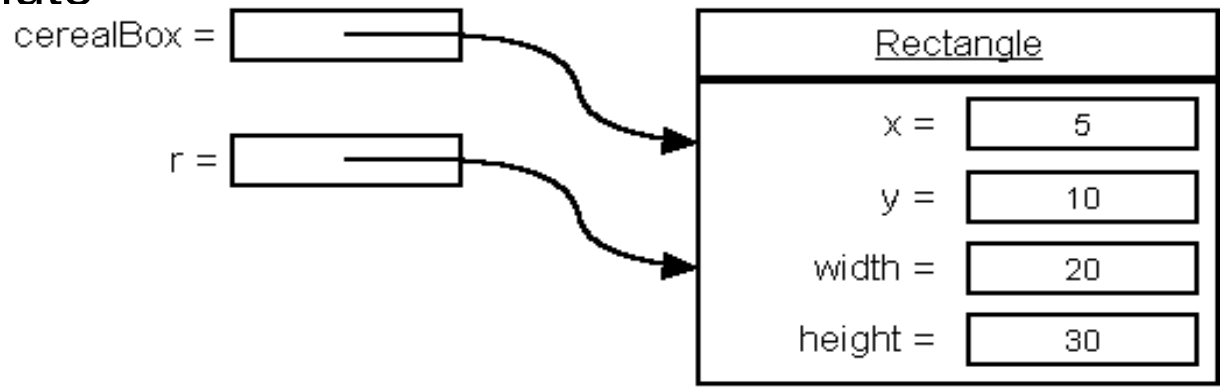
stampa

"SavingsAccount[balance=1000][interestRate=10]"

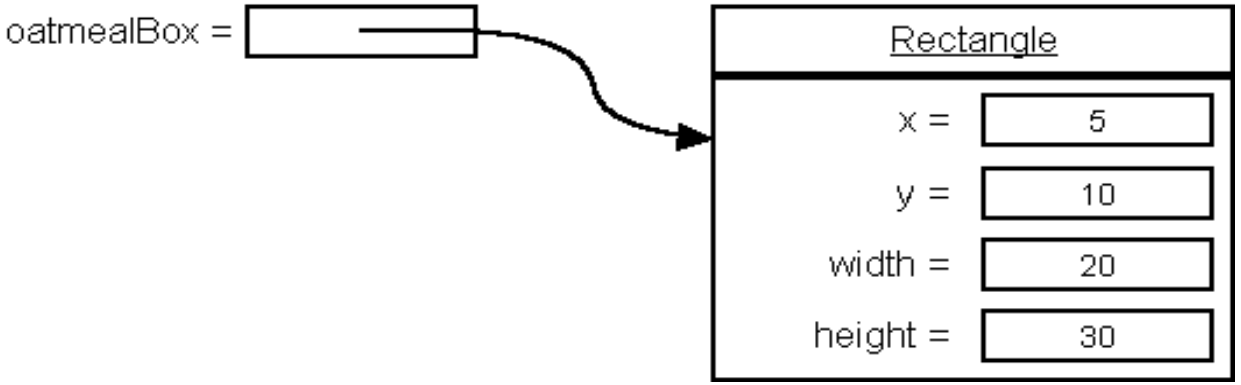
e quindi rispetta il formato standard!

Sovrascrivere `equals`

- Il metodo equals verifica se due oggetti hanno lo stesso contenuto

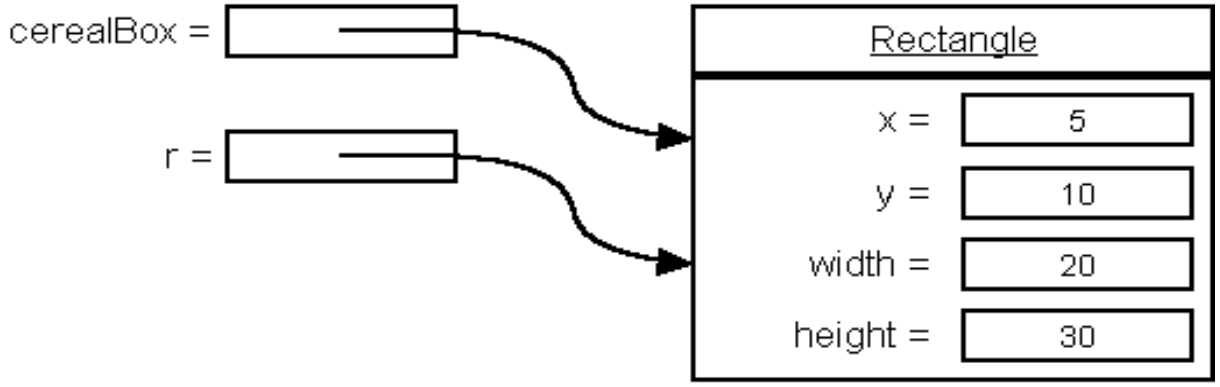



```
if (cerealBox.equals(oatmealBox))…
    restituisce true
```

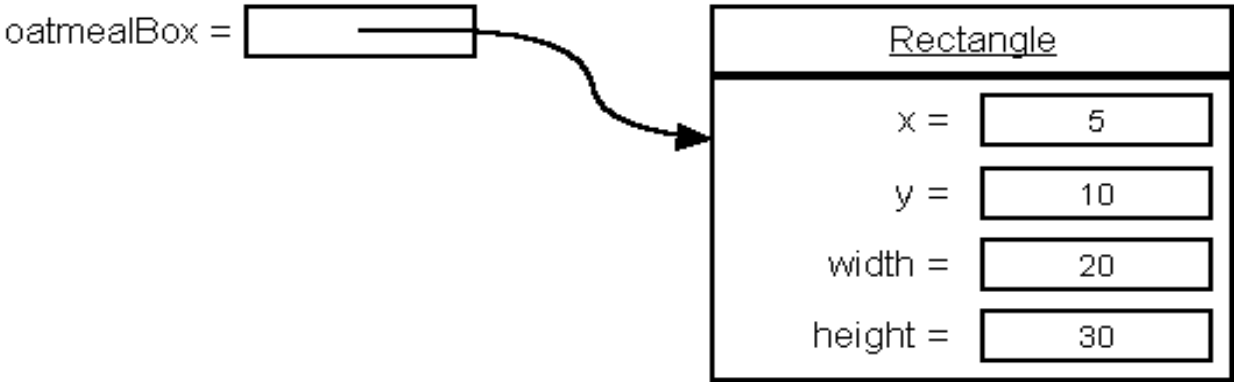


Sovrascrivere `equals`

- L'operatore == verifica se due riferimenti indicano lo stesso oggetto




```
if (cerealBox == oatmealBox)…
   viene valutato false
```



Sovrascrivere `equals`

```
boolean equals(Object otherObject){
        ..........
    }
```

- Sovrascriviamo il metodo equals nella classe **`Coin`**
  - il parametro otherObject è di tipo **`Object`** e non **`Coin`**
  - quando sovrascriviamo un metodo non possiamo variare la firma (altrimenti **sovraccarichiamo** il nome del metodo)
  - in questo caso quindi dobbiamo eseguire un cast sul parametro

```
Coin other = (Coin)otherObject;
```

Sovrascrivere `equals`

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

- Controlla se hanno lo stesso nome e lo stesso valore
  - Per confrontare **`name`** e **`other.name`** usiamo **`equals`** perché si tratta di riferimenti a stringhe
  - Per confrontare **`value`** e **`other.value`** usiamo == perché si tratta di variabili di tipo primitivo

Sovrascrivere `equals`

- Se invochiamo **`coin1.equals(x)`** e **`x`** non è di tipo **`Coin`**?
  - Viene generata un'eccezione
- Possiamo usare **`instanceof`** per controllare se **`x`** è di tipo **`Coin`**

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

Sovrascrivere `equals`

- Se si usa **`instanceOf`** per controllare se un oggetto è di un certo tipo,
  la risposta sarà **`true`** anche se l'oggetto appartiene a qualche sottoclasse…
- Dovrei verificare se i due oggetti appartengono alla stessa classe:

```
if (getClass() != otherObject.getClass())
     return false;
```

- Infine, **`equals`** dovrebbe restituire **`false`** se **`otherObject`** è **`null`**

Classe Coin: Sovrascrivere `equals` in maniera robusta e riutilizzabile

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

Sottoclassi: Sovrascrivere `equals`

- Creiamo una sottoclasse di **`Coin`**: **`CollectibleCoin`**
  - Una moneta da collezione è caratterizzata dall'anno di emissione (vbl. istanza aggiuntiva)

```
public CollectibleCoin extends Coin{
  …

    private int year;
}
```

- Due monete da collezione sono uguali se hanno uguali nomi, valori e anni di emissione
  - Ma **`name`** e **`value`** sono variabili private della superclasse!
  - Il metodo **`equals`** della sottoclasse non può accedervi

Sottoclassi: Sovrascrivere `equals`

- Soluzione:
  il metodo **`equals`** della sottoclasse invoca il metodo omonimo della superclasse
  - Se il confronto ha successo, procede confrontando le altre vbl aggiuntive

```
public boolean equals(Object otherObject){
    if (!super.equals(otherObject))
        return false;

  CollectibleCoin other =
                (CollectibleCoin)otherObject;
  return year == other.year;
}
```

Sovrascrivere `clone`

- Il metodo clone della classe **`Object`** crea un nuovo oggetto con lo stesso stato di un oggetto esistente (clone)

```
protected Object clone()
```

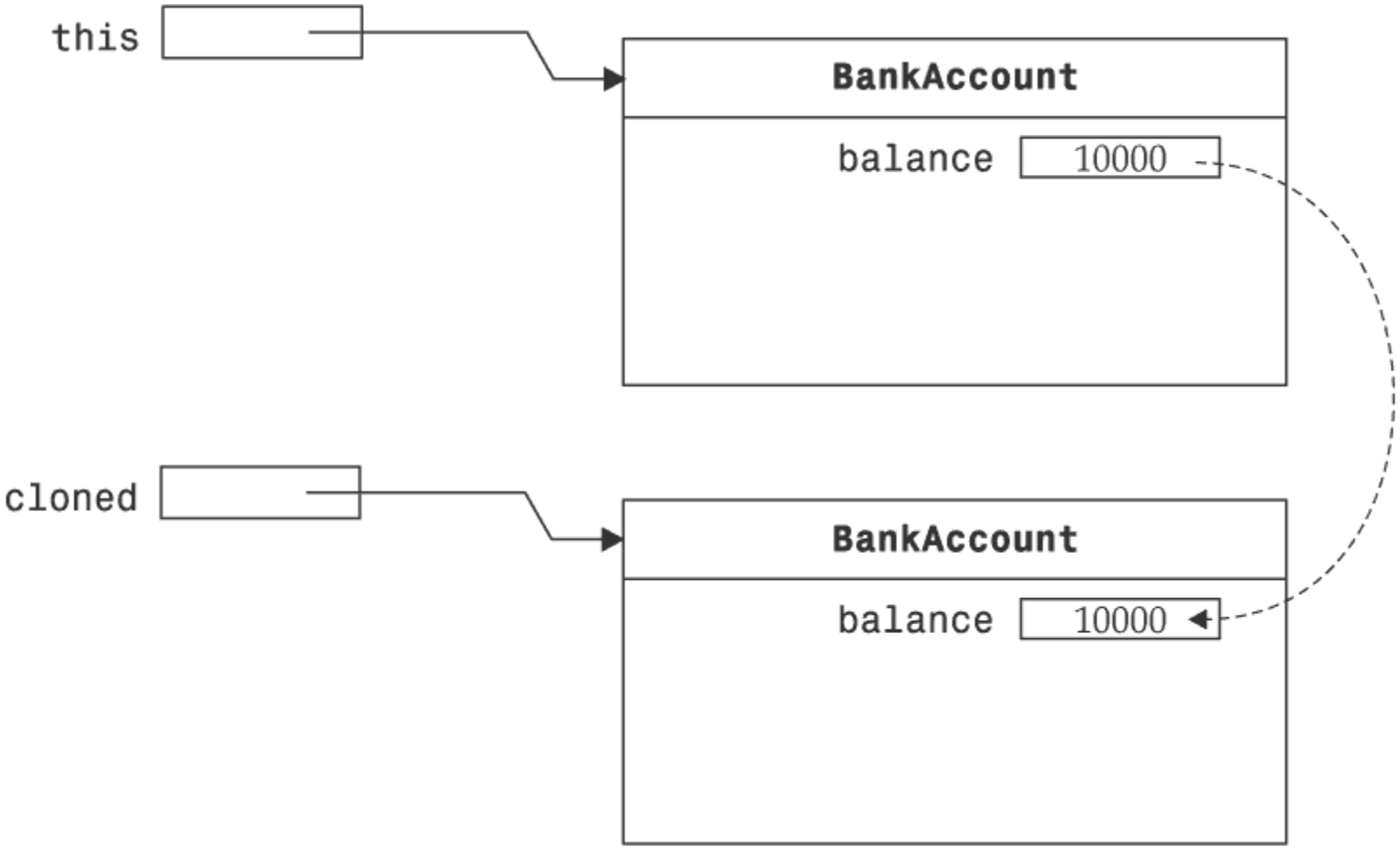
- Se x è una variabile che contiene un riferimento ad un oggetto, allora
  - **`x.clone()`** e **`x`** si riferiscono a due oggetti
  - con diversa identità
  - con stesso contenuto
  - della stessa classe

Sovrascrivere `clone`

- Clonare un conto corrente

```
public BankAccount clone()
{
  BankAccount cloned = new BankAccount();
  cloned.balance = balance;
  return cloned;
}
```

Nota: **`BankAccount`** è un sottotipo di **`Object`** quindi questa riscrittura è consentita


this
BankAccount
balance
10000
cloned
BankAccount
balance
10000

L'ereditarietà e il metodo clone

- Abbiamo visto come clonare un oggetto **BankAccount**

```
public BankAccount clone(){
   BankAccount cloned= new BankAccount();
   cloned.balance = balance;
   return cloned; }
```

- Problema: questo metodo non funziona per le sottoclassi!

```
SavingsAccount s = new SavingsAccount(0.5);
SavingsAccount clonedAccount = s.clone();
      //NON VA BENE
```
istanzia un **BankAccount**

L'ereditarietà e il metodo clone

- Possiamo invocare il metodo clone della classe **Object**
  - Crea un nuovo oggetto dello stesso tipo dell'oggetto originario
  - Copia il contenuto delle variabili di istanza dall'oggetto originario a quello clonato

```
public class BankAccount{
…
   public BankAccount clone(){
   //invoca il metodo clone() di Object
       BankAccount cloned =
                        (BankAccount) super.clone();
       return cloned; }
}
```

L'ereditarietà e il metodo clone

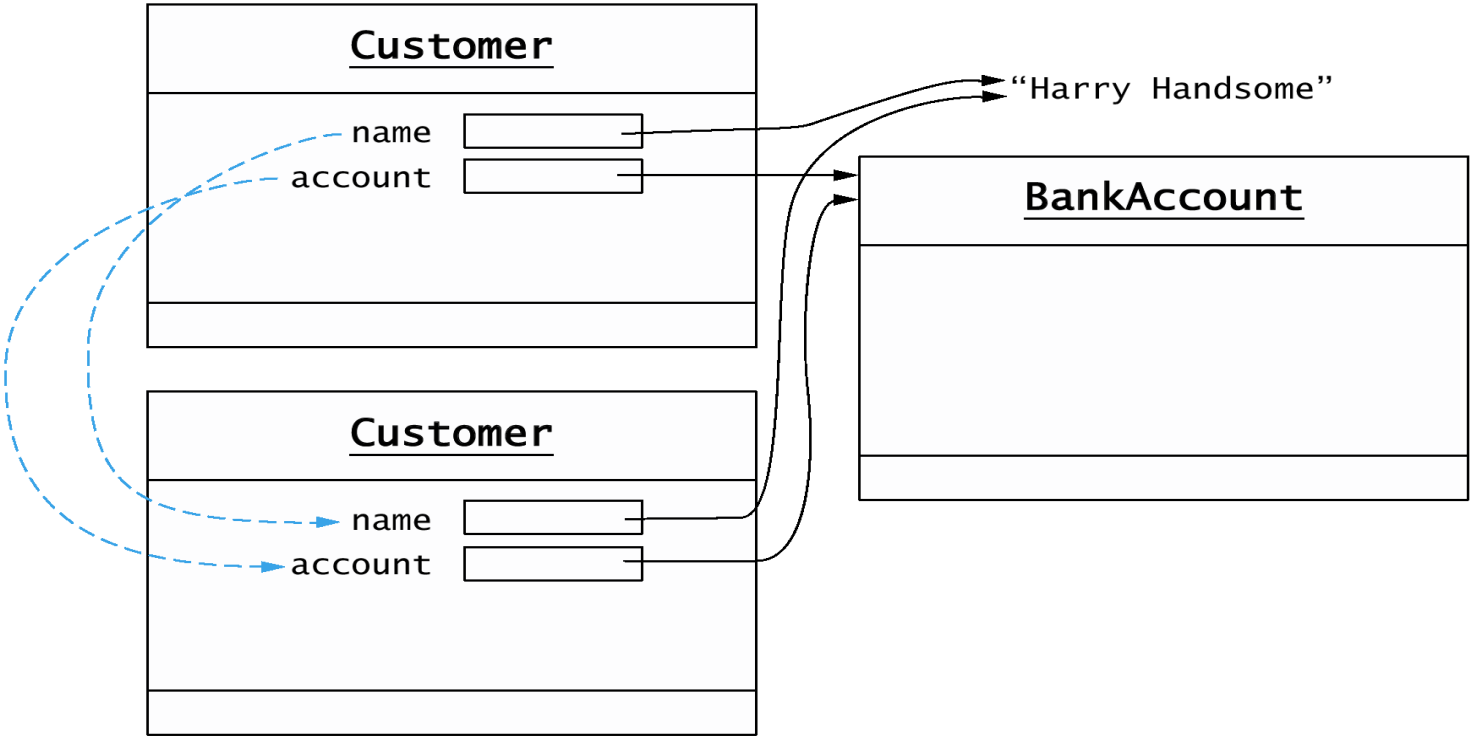
- Consideriamo una classe **`Customer`**
  - Un cliente è caratterizzato da un nome e un conto bancario
  - L'oggetto originale e il clone condividono un oggetto di tipo **`String`** e uno di tipo **`BankAccount`**
    - Nessun problema per il tipo **`String`** (oggetto immutabile)
    - Ma l'oggetto di tipo **`BankAccount`** potrebbe essere modificato da qualche metodo di **`Customer`**!
    - Andrebbe clonato anch'esso

L'ereditarietà e il metodo clone

- Problema con il metodo **`clone`** di **`Object`**:

viene creata una copia superficiale

Se un oggetto contiene un riferimento ad un altro oggetto, viene creata una copia di riferimento all'oggetto, NON un clone!



L'ereditarietà e il metodo clone

- Il metodo `clone` di `Object` esegue una clonazione corretta se un oggetto contiene
  - numeri, valori booleani, stringhe (e in generale oggetti immutabili)

- Bisogna però integrare il suo operato se l'oggetto contiene riferimenti ad altri oggetti
  - quindi non è sufficiente per la maggior parte delle classi!

L'ereditarietà e il metodo clone

- Precauzioni dei progettisti di Java:
  - Il metodo `clone` di `Object` è dichiarato protected
    - Non possiamo invocare `x.clone()` se non all'interno della classe dell'oggetto `x`, di una sua sottoclasse o del pachetto che la contiene
  - Una classe che utilizza questo metodo deve implementare l'interfaccia `Cloneable`
    - In caso contrario viene lanciata un'eccezione di tipo `CloneNotSupportedException`
    - Questa eccezione va catturata anche se la classe implementa `Cloneable`
- In genere, quando sovrascriviamo clone lo ridefiniamo public così è possibile usarlo dovunque.

Interfacce di contrassegno

```
public interface Cloneable{
}
```

- Interfaccia di contrassegno
  - Non ha metodi
  - Usata solo per controlli come nel caso di Cloneable con clone

Clonare un BankAccount

```
public class BankAccount implements Cloneable
{
   …
   public BankAccount clone()
   {

        try
        {
           return (BankAccount) super.clone();
        }
        catch (CloneNotSupportedException e)
        {
           //non succede mai perchè implementiamo Cloneable
           return null;
        }
    }
}
```

Clonare CheckingAccount e SavingsAccount

- Basta ereditare metodo clone di **`BankAccount`**
  - **`interestRate`** e **`transactionCount`** sono di tipo primitivo
  - eventualmente si sovrascrive solo per fare il casting
- Non c'è bisogno di usare **`implements Cloneable`**

… **`class SavingsAccount extends BankAccount{…}`**

… **`class CheckingAccount extends BankAccount{…}`**

  - **`BankAccount`** già implementa **`Cloneable`**, e quindi anche le sue sottoclassi

Clonare un Customer

```
public class Customer implements Cloneable
{
   …
   public Customer clone()
   {
        try
        {
           Customer cloned = (Customer) super.clone();
           cloned.account = account.clone();
           return cloned;
        }
        catch (CloneNotSupportedException e)
        {
          //non succede mai perchè implementiamo Cloneable
           return null;
        }
    }
   private String name;
   private BankAccount account;
}
```

Clonare un PremiumCustomer

```
public class PremiumCustomer Extends Customer
{

//Customer che ha diritto a consegne a domicilio gratis


   ……
   public PremiumCustomer clone()
   {
          PremiumCustomer cloned =
                              (PremiumCustomer)super.clone();
          cloned.indirizzoConsegne =
                              indirizzoConsegne.clone();
          return cloned;
   }

   private Indirizzo indirizzoConsegne;
}
```

Violazione incapsulamento

- se si può modificare lo stato di un oggetto (in un'altra classe) senza utilizzare i metodi dell'interfaccia pubblica l'incapsulamento è violato
- per i dati nelle variabili di istanza non primitive questo può avvenire anche se dichiarate private
  - assegnando direttamente una variabile di istanza con un parametro esplicito
  - restituendo il riferimento contenuto in una variabile di istanza
- soluzione:
  - usare oggetti immutabili oppure
  - la clonazione

Clonazione e incapsulamento

Nella classe **`Customer`**:

```
public void setAccount(BankAccount anAccount){
    account = anAccount; //incapsulamento violato
}
```

si può incapsulare il dato assegnando un clone di **`anAccount`**

```
account = anAccount.clone();
```

```
public BankAccount getAccount(){
      return account; //violazione incapsulamento
}
```

si può incapsulare il dato memorizzato in **`account`** restituendo un suo clone

```
return account.clone();
```

Clonazione e oggetti immutabili

- le variabili di istanza contenenti riferimenti ad oggetti immutabili NON necessitano la clonazione

- rispetto alla clonazione e ai problemi di violazione dell'incapsulamento visti

  gli oggetti immutabili sono assimilabili ai tipi primitivi

Nota su ArrayList

- **`ArrayList`** sovrascrive i metodi **`toString`**, **`equals`** e **`clone`**

- Il metodo clone esegue una copia superficiale
  - per clonare occorre forzare clonazione di ogni elemento dell'**`ArrayList`**

- **`toString`** e **`equals`** funzionano come ci si aspetta