EREDITARIETÀ

INTRODUZIONE

- Uno dei principi basilari della programmazione orientata agli oggetti
- Obiettivo **riusabilità del codice**
- Creare una classe che rappresenta un concetto **generale**
  - creare classi **specializzate** che ereditano la classe + generale
- Terminologia
  - superclasse: la classe da cui si ereditata
  - sottoclasse: la classe derivata

> ⚠ **Attenzione**
>
> Non vengono **mai** ereditati variabili/metodi privati

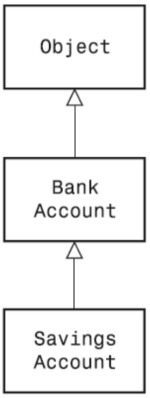
SINTASSI

- Si utilizza `extends`
  - si può fare extends solo di **una** classe

```
class Sottoclasse extends Superclasse {...}
```

TERMINOLOGIA E NOTAZIONE

- Termini **superclasse** e **sottoclasse** derivano dalla **teoria degli insiemi**
- Nel diagramma ereditarietà viene indicato con la freccia a punta triangolo vuoto



METODI DELLA SOTTOCLASSE

- Nella sottoclasse posso **sovrascrivere** dei metodi della superclasse
  - deve avere la **stessa firma**
- Invocare un metodo della superclasse
  - `super.nomeMetodo(parametri)`

VARIABILI DI INSTANZA NELLA SOTTOCLASSE

- **Non** può essere sovrascritta una variabile di istanza nella sottoclasse
  - se ne creo due uguali, è come se creassi due variabili diverse
- Anche le variabili `private` vengono ereditate nella sottoclasse, solo non accessibili

COSTRUTTORI NELLA SOTTOCLASSE

- Posso chiamare il costruttore della superclasse attraverso `super()`
- Viene invocato automaticamente se non scritto esplicitamente

CONVERSIONE FRA RIFERIMENTI

- Convertire da tipo **sottoclasse** a **superclasse*
  - la conversione avviene automaticamente
  - posso però solo usare i metodi del tipo che ho definito
- Conversione da tipo **superclasse** a tipo **sottoclasse**
  - NON avviene automaticamente
  - posso farla solo con un **cast**
    - se non si può fare: `ClassCastException`
- Posso accettare come parametro un oggetto di tipo **superclasse**
  - e posso passare in quel metodo qualunque oggetto di una sua **sottoclasse**

POLIMORFISMO

- **Polimorfismo** = "molte forme"
  - il tipo **non determina in modo completo** il tipo dell'oggetto a cui essa si riferisce
- Ricorda: i types sono noti solo all'interprete di java, **non** in fase di **esecuzione**
  - il tipo di una variabile oggetto non influenza il contenuto della variabile oggetto stessa
  - l'esecuzione di un metodo è sempre determinata dal tipo dell'oggetto e non dal tipo della variabile oggetto

- Esempio
  - creo metodo `deposit` nella superclasse e lo sovrascrivo nella sottoclasse
  - creo oggetto di tipo sottoclasse e poi lo converto nel tipo superclasse
  - chiamo metodo `deposit` nell'oggetto creato e verrà usato quello della sottoclasse

EARLY AND LATE BINDING

- Selezione anticipata
  - nel caso dell'overload, è il **compilatore** a scegliere quale metodo chiamare in base al **tip**
  - tra diversi metodi di overload, viene sempre scelto quello più specifico
    - dove il compilatore deve fare meno conversioni
- Selezione posticipata
  - avviene nel caso del polimorfismo
  - tra due metodi (superclasse, sottoclasse), a scegliere quale metodo chiamare è l'**interpr**
  - non può scegliere in base al tipo perché nel bytecode non sono presenti types

OPERATORE "INSTANCEOF"

- Verificare se una variabile di tipo superclasse è anche di tipo sottoclasse
  - buona pratica prima di effettuare un cast

```
if (varOggetto instanceof NomeClasse) {
    ...
}
```

SUPERCLASSE UNIVERSALE OBJECT

- Tutte le classi ereditano dalla superclasse universale `Object`
- se `extends` non viene indicato esplicitamente usa `java.lang.Object`

METODI PRINCIPALI DI OBJECT

Metodo	Obiettivo
`String toString()`	Restituisce una stringa che descrive l'oggetto
`boolean equals(Object otherObject)`	Verifica se l'oggetto è uguale a un altro
`Object clone()`	Crea una copia completa dell'oggetto

METODO "TO STRING"

- Metodo che viene invocato automaticamente quando si chiama `println()`

- Di default, `toString()` restituisce `NomeClasse@hashcode`
  - hashcode identifica indirizzo di memoria oggetto
- È utile sovrascrivere `toString()` per restituire **informazioni di stato** dell'oggetto
  - `return getClass().getName() + [informazioni]`

METODO "EQUALS"

- Di default, `equals(Object otherObject)` confronta indirizzi di memoria variabili oggetto
- È utile sovra-scriverlo per dare informazioni sulla coincidenza degli **stati degli oggetti**
  - es. confrontare "balance" di "BankAccount"
- Per sovrascriverlo, devo sempre usare come parametri nella firma `Object otherObject`

CONTROLLO DI ACCESSO

- 4 livelli
  - `public`
  - `package`
    - accesso all'interno del package
    - impostazione di default
  - `protected`
    - accessibili alle sottoclassi
  - `private`

CLASSI E METODI FINAL

- Un metodo `final` non può essere sovrascritto da sottoclassi
- Una classe `final` non può avere sottoclassi
  - esempio: classe `String`

CLASSI INTERNE

- Ci sono alcune classi che potrebbero venire usate solo internamente da un'altra classe
- Posso creare una classe all'interno di un'altra classe

```
public class ClasseEsterna {
    <tipoaccesso> class ClasseInterna {
        ...
    }
    ...
}
```

- Il compilatore traduce due file bytecode
  - `ClasseEsterna.class`
  - `ClasseEsterna$ClasseInterna.class`

VANTAGGI E LIMITAZIONI

- Ciascuna delle due classi ha accesso a tutti i metodi/variabili dell'altra, anche se privati
- Posso creare oggetti `ClasseInterna` solo dentro metodi non statici di `ClasseEsterna`
- La classe interna può essere resa **inaccessibile** al codice di altre classi

USO DI CLASSI INTERNE

- Se la classe interna è pubblica, si può accedere ad essa con:
  - `ClasseEsterna.ClasseInterna`
- Non è **MAI** possibile creare oggetti di tipo `ClasseInterna`
- Si può definire **variabili** oggetto di **TIPO** `ClasseInterna`

CLASSI ASTRATTE

- Classi che definiscono i metodi senza implementarli
  - non posso creare un'istanza di una classe astratta direttamente
  - devo per forza implementare una classe astratta attraverso una sottoclasse
- Sintassi:

```
public abstract class LaMiaClasseAstratta {
    public abstract ilMioMetodoAstratto area();
}
```