

Riuso di classi

- Spesso si ha bisogno di classi simili
 - Si vuole cioè riusare classi esistenti per gestire attributi e metodi leggermente diversi
- Copiare la classe originaria e modificarne attributi o metodi non è pratico
 - Proliferazione di classi
 - Il programmatore deve fare tante attività
- Il riuso delle classi esistenti deve avvenire
 - Senza dover modificare codice esistente (e funzionante)
 - In modo semplice per il programmatore

E. Tramontana Eredità Object-Orientation - 5 mag 10 1

Ereditarietà

- Attraverso l'ereditarietà è possibile
 - Definire una nuova classe indicando solo cosa ha in più rispetto ad una classe esistente
 - E' possibile aggiungere attributi e metodi nuovi
 - E' possibile modificare metodi esistenti
- Esempio
 - Una classe Persona ha nome e cognome (più vari metodi)
 - La classe Studente dovrebbe avere tutto ciò che Persona fornisce (attributi e metodi) ed inoltre nuovi attributi e metodi
 - Studente aggiunge esami, voti, etc.
 - La classe Studente eredita da Persona

```
public class Studente extends Persona { ... }
```

 - Studente è sottoclasse di Persona
 - Persona è superclasse di Studente

E. Tramontana Eredità Object-Orientation - 5 mag 10 2

Ereditarietà

- La sottoclasse
 - Eredita tutti i metodi e gli attributi della superclasse e può usarli come se fossero definiti localmente
 - Aggiunge altri metodi
 - Può ridefinire i metodi della superclasse
 - Non può eliminare metodi o attributi della superclasse
- Esempio
 - La classe Studente
 - Può usare tutti i metodi della classe Persona, es. setName()
 - Può aggiungere metodi, es. media()

E. Tramontana Eredità Object-Orientation - 5 mag 10 3

Ereditarietà

- Visibilità
 - Ciò che è `private` è visibile solo alla classe, non alla sottoclasse
 - Ciò che è `public` è visibile a tutti, anche alla sottoclasse
- Se voglio far vedere qualche metodo o attributo alle sottoclassi ma non a tutti
 - Uso `protected`
- Il nome della sottoclasse deve comunicare a quale classe è simile e come è diversa
 - Sottoclassi che servono come radici di una gerarchia devono avere nomi concisi, altrimenti si può dare un nome più lungo

E. Tramontana Eredità Object-Orientation - 5 mag 10 4

Classi astratte

- Una classe astratta è una classe parzialmente implementata
 - Alcuni metodi sono implementati, altri no (sono dichiarati `abstract`)
 - E' utile avere un metodo `abstract` (senza implementazione)
 - I client si aspettano di poterlo invocare
 - Forza le sottoclassi (concrete) ad implementare il metodo `abstract`
 - La classe astratta non può essere istanziata
 - Le sottoclassi ereditano implementazioni e attributi
- Es.

```
public abstract class Libro {
    public abstract void insert();
    public String getAutore() { ... }
}
```

[E. Tramontana Eredità Object-Orientation - 5 mag 10](#) 5

Interfacce

- In Java una interfaccia riprende il concetto di interfaccia di sistemi orientati ad oggetti
 - Non fornisce una implementazione per i metodi
 - Permette di definire un tipo
 - Elenca le signature dei metodi `public` (senza corpo dei metodi)
 - Posso solo dichiarare i metodi
 - Niente attributi non inizializzati, niente costruttori

```
public interface IAccount {
    public boolean deposito(int amount);
    public void setInitial();
    public boolean check();
}
```

[E. Tramontana Eredità Object-Orientation - 5 mag 10](#) 6

Classi e Interfacce

- Una classe può implementare una interfaccia
 - Ovvero, la classe fornisce un'implementazione dei metodi def. dall'interfaccia
- Non è possibile istanziare interfacce
- Tramite l'interfaccia i client sanno cosa possono invocare
 - Posso usare (per istanziazioni di oggetto e invocazioni di metodo) una qualsiasi delle implementazioni disponibili per l'interfaccia
- Un client che usa una interfaccia rimane immutato quando l'implementazione dell'interfaccia cambia

```
public class Account implements IAccount { ... }
public class AccountV2 implements IAccount { ... }
```

...

```
IAccount a = new AccountV2();
a.setBalance();
```

[E. Tramontana Eredità Object-Orientation - 5 mag 10](#) 7

Compatibilità tra classi

- L'ereditarietà permette una classificazione di tipi
- Una sottoclasse è un *sottotipo* compatibile con la superclasse, ovvero
 - Una sottoclasse è anche ciò che è la superclasse
 - Es. il tipo `Studente` è compatibile con il tipo `Persona`
 - La classe `Studente` fa tutto ciò che fa `Persona`
 - La classe `Studente` fa altre cose oltre quelle che fa `Persona`
- Una sottoclasse può prendere il posto della superclasse
 - Es. posso usare un'istanza di `Studente` al posto di una di `Persona`
 - Dove compare ad es. `p.setName()` con `p` di tipo `Persona` posso sostituire `s.setName()` con `s` di tipo `Studente`
- Attenzione: non vale il contrario, non posso usare la superclasse dove usavo la sottoclasse

[E. Tramontana Eredità Object-Orientation - 5 mag 10](#) 8

```

public class Persona {
    protected String nome, cognome;
    public void setName(String nom,
        String cog) {
        nome = nom;
        cognome = cog;
    }
    public void printAll() {
        System.out.println("Nome: "+
            nome+" "+cognome);
    }
}

```

```

public class Test {
    static void main(String[] args) {
        Studente s = new Studente();
        s.setName("Jeff", "Riddle");
        s.nuovoEsame("Italiano", 8);
        s.nuovoEsame("Fisica", 7);
        s.printAll();
        Persona p = s;
        p.printAll();
    }
}

```

```

public class Studente extends Persona {
    private int numEsami = 0;
    private String[] esami = new String[10];
    private int[] voti = new int[10];
    public void nuovoEsame(String e, int v) {
        esami[numEsami] = e;
        voti[numEsami] = v;
        numEsami++;
    }
    public float media() {
        if (numEsami == 0) return 0;
        float sum = 0;
        for (int i=0; i<numEsami; i++)
            sum = sum + voti[i];
        return sum/numEsami;
    }
    public void printAll() {
        super.printAll();
        for (int i=0; i<numEsami; i++)
            System.out.println(esami[i]+" "+
                voti[i]);
        System.out.println("media = "+media());
    }
}

```

Considerazioni sul codice

- La classe Studente
 - Eredita tutto ciò che fornisce Persona
 - Ridefinisce il metodo printAll() (ovvero fa *override*), quindi modifica il comportamento del metodo printAll() ereditato
 - super.printAll() per invocare printAll() di Persona da Studente
 - super permette di accedere metodi della superclasse
- Per la classe Test
 - La variabile p è di tipo Persona, ma punta una istanza di Studente
 - Posso invocare su p il metodo printAll()
 - Quale sarà il risultato? Viene invocato printAll() di Studente
 - Non posso invocare su p il metodo nuovoEsame()
 - Il tipo di p a compile time (Persona) non ha il metodo nuovoEsame()
 - quindi, il compilatore non può far invocare nuovoEsame() su p (nonostante p punterà a runtime ad una istanza di Studente)

E. Tramontana - Eredità Object-Orientation - 5 mag 10 10

Late binding e polimorfismo

```

public class Test {
    public static void main(String[] args) {
        Persona p = new Persona();
        Studente s = new Studente();
        Persona px;
        ...
        if ( ... ) px = p;
        else px = s;
        px.printAll();
    }
}

```

- printAll() invocato su px può assumere il comportamento definito in Persona o quello definito in Studente
- Il compilatore riconosce che printAll() è definito per px (qualunque sia l'istanza puntata)
- A runtime si decide quale printAll() eseguire, ovvero si ha late binding
 - Il comportamento di printAll() è polimorfo

E. Tramontana - Eredità Object-Orientation - 5 mag 10 11

Polimorfismo

- Nei sistemi ad oggetti possono esistere metodi con lo stesso nome e la stessa signature (in classi diverse)
- Quando si usa l'ereditarietà e sono stati definiti metodi con lo stesso nome, la chiamata ad un metodo può avere effetti diversi, ovvero si ha un comportamento polimorfo
 - Es. ciascun elemento di s può indicare una qualsiasi istanza la cui classe ha come superclasse Shape

```

public class TestShape {
    public static void main(String[] args) {
        Shape[] s = new Shape[3];
        s[0] = new Box(2,3);
        s[1] = new Circle(4);
        s[2] = new TriangleRect(3, 4);
        for (int i=0; i<3; i++) s[i].show();
    }
}

```

E. Tramontana - Eredità Object-Orientation - 5 mag 10 12

Polimorfismo

- Il polimorfismo è una caratteristica fondamentale dei sistemi ad oggetti
- Il late binding è tipico dei sistemi in cui esiste il polimorfismo
- Senza polimorfismo
 - Dovremmo inserire uno switch sul chiamante per valutare la classe di ciascuna istanza e chiamare il metodo corrispondente a tale classe

E. Tramontana Eredità Object-Orientation - 5 mag 10 13

Sottoclassi e Dispatch

```
public class Account {
    public void setBalance(float amount) {
        if (check(amount)) balance = amount;
    }
    public boolean check(float amount) {
        return (balance+amount) >= 0
    }
}

public class SavingAccount extends Account {
    public boolean check(float amount) {
        return (balance+amount) >= 1000
    }
}
```

```
Account acc = new SavingAccount();
acc.setBalance(1234);
```

E. Tramontana Eredità Object-Orientation - 5 mag 10 14

Dispatch

- Quale versione di check() è chiamata da setBalance()?
 - Quando un metodo (setBalance()) è chiamato su un oggetto, viene controllato il suo tipo (a runtime): SavingAccount
 - Viene cercato il metodo: non trovato
 - Se non trovato si cerca la superclasse: Account
 - Se trovato si esegue: Account.setBalance()
 - Questo chiama check(), come prima
 - Viene cercato prima su SavingAccount: trovato
 - Si esegue SavingAccount.check()

E. Tramontana Eredità Object-Orientation - 5 mag 10 15

Tipi

- Tipi dichiarati e tipi a runtime
- A compile time, un tipo dichiarato
 - Determina su quale specifica chi lo usa può contare
 - Può essere una classe o una interfaccia
 - Un oggetto può essere di uno o più tipi differenti (a compile-time)
- A runtime
 - Un oggetto ha una sola implementazione che è sempre una classe, non una interfaccia
 - Il tipo a runtime di un oggetto non cambia mai

E. Tramontana Eredità Object-Orientation - 5 mag 10 16