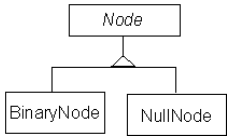
Design Pattern Singleton
Diagrammi di sequenza

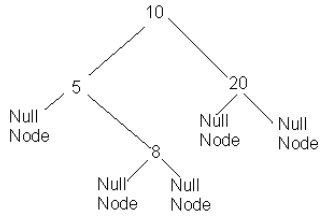
Prof. A.Calvagna

Binary Search Tree Example

Class Structure



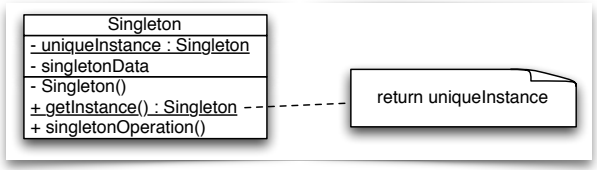
Object Structure



Design pattern Singleton

- Intento
  - –Assicurare che una classe abbia una sola istanza e fornire un punto di accesso globale all'istanza
- Motivazione
  - –Alcune classi dovrebbero avere esattamente una istanza in tutta l'applicazione, es. uno spooler di stampa, un file system, un window manager, una lista clienti, etc.
  - –Una variabile globale rende un oggetto accessibile ma non proibisce di avere più oggetti per una classe. Si vuole invece proibirlo
  - –La classe stessa dovrebbe essere responsabile di tener traccia del suo unico punto di accesso

Singleton



- Soluzione
  - –La classe che deve essere un Singleton dovrà implementare un'operazione `getInstance()` sulla classe (ovvero, in Java è un metodo `static`) che ritorna l'unica istanza creata
  - –La classe Singleton è responsabile per la creazione dell'istanza
  - –Il costruttore della classe Singleton è privato, così da non permettere la creazione tramite new ad altre classi

Esempio classe Singleton fib

```
// Classe Singleton che tiene una lista
// di interi
public class Fib {
  // l'unica istanza e' tenuta da obj
  private static Fib obj = new Fib();

  private int[] x = {1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144};
  private int i = 3;

  private Fib() {
  }
  public static Fib getInstance() {
     return obj; // restituisce l'istanza
  }
  public int getValue() {
        if (i < 11) i++;
        return x[i-1];
  }
  public void revert() {
     i = 0;
  }
}
```

```
public class MainFib {
  public static void main(String[] args) {
    // richiede una istanza di Fib
    Fib f = Fib.getInstance();
    System.out.print("f "+f.getValue());
    System.out.println(" "+f.getValue());

    // richiede una nuova istanza
    Fib f2 = Fib.getInstance();
    System.out.print("f2 "+f2.getValue());
    System.out.println(" "+f2.getValue());

    // Si ha un errore a compile-time con:
    // Fib f3 = (Fib) f2.clone();
    // Fib f4 = new Fib();
  }
}
```

Output dell'esecuzione
```
f 5 8
f2 13 21
```


esempio classe logs

```
public class Logs {                 // Classe Singleton
    private static Logs obj;        // obj tiene l'istanza
    private List<String> l;         // tiene i dati da registrare

    private Logs() {                // il costruttore è privato
        empty();
    }
    public static Logs getInstance() {    // restituisce l'unica istanza
          if (null == obj) obj = new Logs(); // crea l'istanza se non presente
          return obj;
    }
    public void record(String s) {    // accoda il dato
        l.add(s);
    }
    public String dumpLast() {        // restituisce l'ultimo dato
        return l.get(l.size()-1);
    }
    public String dumpAll() {         // restituisce tutti i dati
        String acc = "";
        for (String s : l)            // s tiene ciascun elemento in lista, ad ogni passata
            acc = acc.concat(s);
        return acc;
    }
     public void empty() {
        l = new ArrayList<>();
    }
}
```


Test per classe Logs

```
public class TestLogs {
    private Logs lg = Logs.getInstance();

    public void testSingl() {
        initLogs();
        Logs lg2 = Logs.getInstance();
        lg2.record("uno");
           lg2.record("due");
        if (lg.dumpLast().equals("due"))
            System.out.println("OK test logs singl");
        else
            System.out.println("FAILED test logs singl");
    }
    public void testLast() {
        initLogs();
        if (lg.dumpLast().equals("three "))
            System.out.println("OK test logs last");
        else
            System.out.println("FAILED test logs last");
    }
    public void testAll() {
        initLogs();
        if (lg.dumpAll().equals("one two three "))
            System.out.println("OK test logs all");
        else
            System.out.println("FAILED test logs all");
    }
```

```
    private void initLogs() {
        lg.empty();
           lg.record("one ");
           lg.record("two ");
           lg.record("three ");
    }
    public static void
        main(String[] args) {
        TestLogs tl = new TestLogs();
        tl.testSingl();
           tl.testAll();
           tl.testLast();
    }
}
```

Output dell'esecuzione
```
OK test logs singl
OK test logs all
OK test logs last
```


Conseguenze del Singleton

- La classe che è un Singleton ha pieno controllo di come e quando i client accedono al valore della sola istanza
- Evita che esistano variabili globali che tengono la sola istanza condivisa
- Permette di controllare il numero di istanze create in un programma, facilmente ed in un solo punto
- La soluzione è più flessibile rispetto a quella di usare `static` per tutte le operazioni e le variabili, poiché si può cambiare facilmente il numero di istanze consentite
- L'oggetto condiviso è comunque una normale istanza di classe, non l'ho resa statica.
- L'unico frammento di codice da modificare quando si vuol variare il numero di istanze create è quello della classe che è Singleton, mentre usando `static` si dovrebbero modificare tutte le invocazioni

Object Oriented Design

To design reusable object-oriented software you must:
- find pertinent objects,
- factor them into classes at the right granularity,
- define class interfaces and inheritance hierarchies,
- establish key relationships among them.

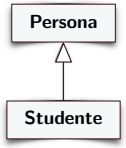
Your design should be specific to the problem at hand but also general enough to address future problems and requirements.

Riuso di classi

- Spesso si ha bisogno di classi simili
  - Si vuole cioè **riusare** classi esistenti per implementare attributi e metodi leggermente diversi
- Riuso implementazioni esistenti, **SENZA RICOPIARE LA CLASSE**
- Il riuso delle classi esistenti deve avvenire
  - Senza dover modificare codice esistente (e funzionante)
  - In modo semplice per il programmatore
- Due le strade possibili: statica (ereditarietà) e dinamica (composizione)

class inheritance and subtyping

- Class inheritance defines an object's implementation in terms of another object's implementation.
- Abstract feature inheritance (or subtyping) describes when an object can be used in place of another.
- No distinction in Java/C++ (use Abstract classes)

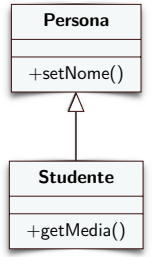


Ereditarietà classi

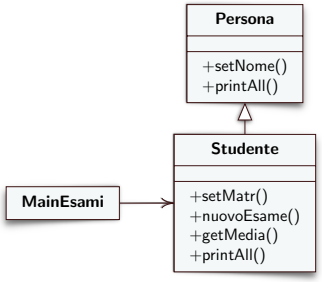
- Attraverso l'ereditarietà è possibile
  - Definire una nuova classe indicando solo cosa ha in più rispetto ad una classe esistente: ovvero attributi e metodi nuovi, e modificando i metodi esistenti
- Esempio: una classe `Persona` ha `nome` e `cognome` (più vari metodi)
- La classe `Studente` dovrebbe avere tutto ciò che `Persona` ha (attributi e metodi) e 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` e `Persona` è superclasse di `Studente`

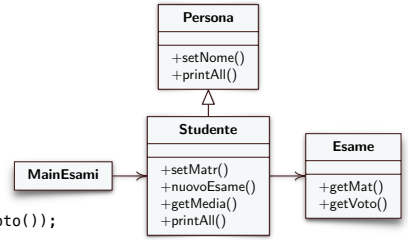


```java
public class Persona {
  public void setNome(String n, String c) { ... }
  public void printAll() { ... }
}

public class Studente extends Persona {
  public void setMatr(String m) { ... }
  public void nuovoEsame(String m, int v) { ... }
  public float getMedia() { ... }
  public void printAll() { ... }
}

public class MainEsami {
  public static void main(String[] args) {
    Studente s = new Studente();
    s.setNome("Alan", "Rossi");   // metodo della superclasse di s
    s.setMatr("M12345");
    s.nuovoEsame("Italiano", 8);  // metodo della classe di s
    s.printAll();                 // metodo della classe di s

    s.nuovoEsame("Fisica", 7);
    Persona p = s;   // p e' dichiarato di tipo Persona
    p.printAll();    // a runtime p punta all'istanza s
  //p.nuovoEsame("Geometria", 9); SEGNALATO IN COMPILAZIONE!
  }
}
```




```java
public class Persona {
  private String nome, co;
  public void setNome(String n, String c) {
    nome = n; co = c;
  }
  public void printAll() {
    System.out.println(nome + " " + co);
  }
}

public class Studente extends Persona {
  private String matr;
  private List<Esame> esami = new ArrayList<>();
  public void setMatr(String m) { matr = m; }
  public void nuovoEsame(String m, int v) {
    Esame e = new Esame(m, v);
    esami.add(e);
  }
  public float getMedia() {
    if (esami.isEmpty()) return 0;
    float sum = 0;
    for (Esame e : esami) sum += e.getVoto();
    return sum / esami.size();
  }
  public void printAll() {
    super.printAll();
    System.out.println("matr: " + matr);
    for (Esame e : esami)
      System.out.println(e.getMat() + ": " + e.getVoto());
    System.out.println("media: " + getMedia());
  }
}
```

```java
public class Esame {
  private String mat;
  private int voto;
  public Esame(String n,int v){
    mat = n;  voto = v;
  }
  public String getMat() {
    return mat;
  }
  public int getVoto() {
    return voto;
  }
}
```



Modificatori di accesso in Java

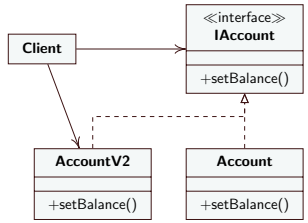
Access Modifiers

Modifier	Class	Package	Subclass	Global
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

Classi ed interfacce

- Un client che usa un'interfaccia rimane immutato quando l'implementazione dell'interfaccia cambia

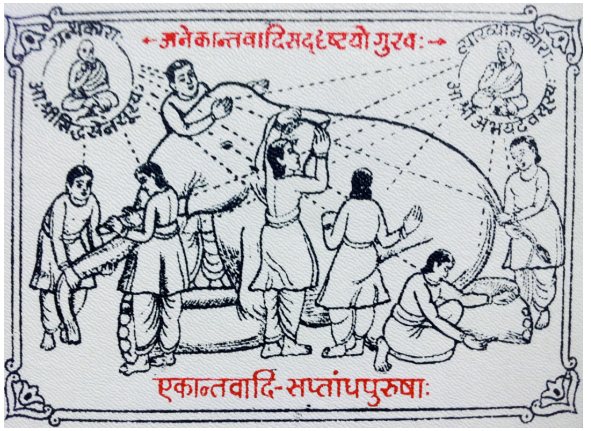
```java
public class AccountV2 implements IAccount {
  public void setBalance() {  }
}

public class Client {
  public void main(String[] args) {
    IAccount a = new AccountV2();
    a.setBalance();
  }
}
```



Classi Astratte in Java

```
public abstract class Libro {
  private String autore;
  public abstract void view();
  public String getAutore() {
    return autore;
  }
}
```


Interfaces:
Five blind philosophers touch different parts of an elephant



Interface inheritance

- Interfaces only partially describes an object's features
- An object can inherit (implement) multiple interfaces
- No relationship between interfaces

- Interface inheritance should not be used to represent structural subtype (is-a) relations between objects

Interface

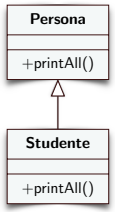
```
public interface IAccount {
  public void setBalance();
}
```

Da java 8:

- metodi static/default/private
- attributi static/final

Late binding e polimorfismo

```
public void m() {
  Persona p = new Persona();
  Studente s = new Studente();
  Persona px;
  if (i > 10)
    px = p;
  else
    px = s;
  px.printAll();
}
```

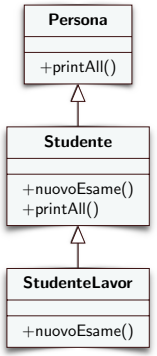


Polimorfismo

- 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 il comportamento è polimorfo

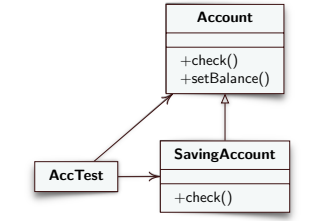
```
Studente s;
// ...
s.nuovoEsame("Maths", 8);
s.printAll();
```

- La variabile `s` potrebbe tenere il riferimento ad un'istanza di `Studente` o di `StudenteLavor`, quale metodo `nuovoEsame()` sarà chiamato è deciso a runtime, in base all'istanza. Lo stesso vale quando si ha una variabile di tipo `Persona` e per il metodo `printAll()`



Sottoclassi e dispatch

```
public class Account {
    protected float balance;
    public void setBalance(float amount) {
        System.out.println("in account set-balance");
        if (check(amount))
            balance += amount;
    }
    public boolean check(float amount) {
        System.out.println("in account check");
        return (balance + amount) >= 0;
    }
}

public class SavingAccount extends Account {
    public boolean check(float amount) {
        System.out.println("in saving-account check");
        return (balance + amount) >= 1000;
    }
}

public class AccTest {
    public static void main(String[] args) {
        Account acc = new SavingAccount();
        acc.setBalance(1234);
    }
}
```



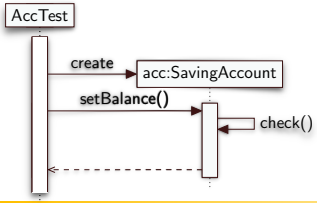


Diagramma UML di sequenza

- Esempio di creazione di un'istanza della classe `Student`

```
public class MainEsami {
  public static void main(String[] args)
{
    Studente s = new Studente();
  }
}
```

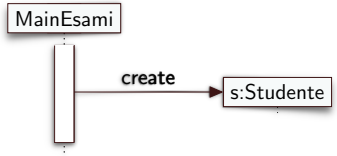


Diagramma UML di sequenza

- Esempio di chiamata di metodo su un'istanza di `Studente`

```
public class MainEsami {
  public static void main(String[] args)
{
    Studente s = new Studente();
    s.setNome("Alan", "Rossi");
  }
}
```

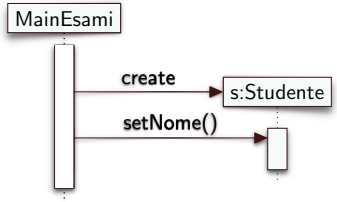


Diagramma UML di sequenza

- Esempio di chiamate di metodo a cascata

```
public class MainEsami {
  public static void main(String[] args)
{
    Studente s = new Studente();
    s.nuovoEsame("Italiano", 8);
  }
}
```

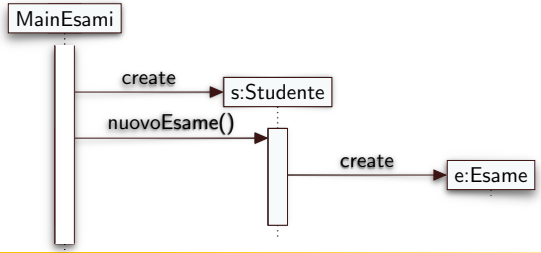


Diagramma UML di sequenza

- Mostra interazioni fra istanze di oggetti
- L'asse temporale è inteso in verticale verso il basso
- In alto in orizzontale ci sono vari oggetti
- In ciascuna colonna se l'oggetto esiste è indicato con una linea tratteggiata, detta linea della vita, e se è attivo con una barra di attivazione
- Una chiamata di metodo è indicata da una freccia piena che va dalla barra di attivazione di un oggetto ad un altro