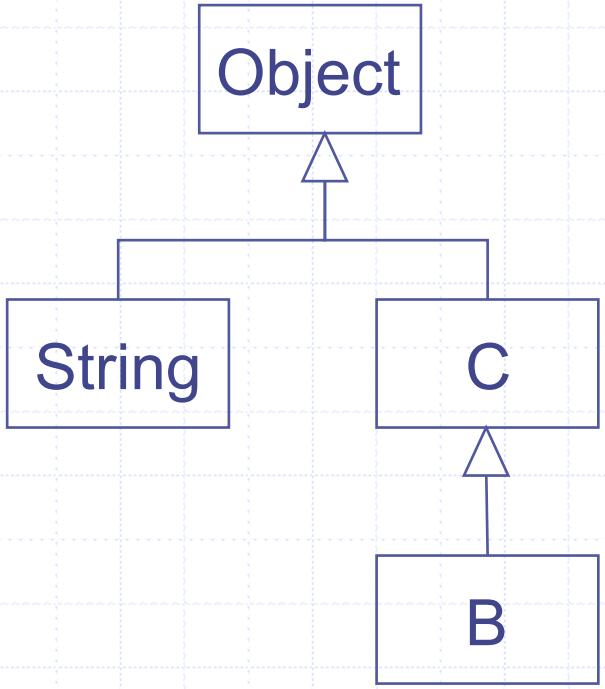
UML

Unified Modeling Language

Esiste una notazione grafica per mostrare le relazioni di ereditarietà.

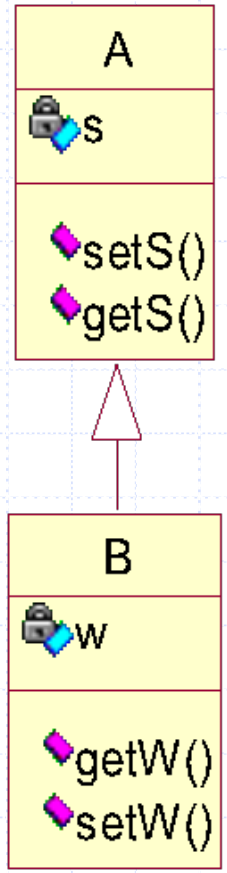



```
class C {…}
class B extends C
{…}
```

Tutte le classi ereditano da Object!

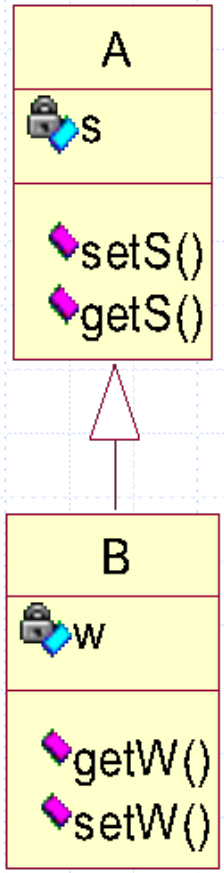
UML – Class Diagram

- rappresenta le classi e gli oggetti che compongono il sistema, ed i relativi attributi ed operazioni
- specifica, mediante le associazioni, i vincoli che legano tra loro le classi
- può essere definito in fasi diverse (analisi, disegno di dettaglio)



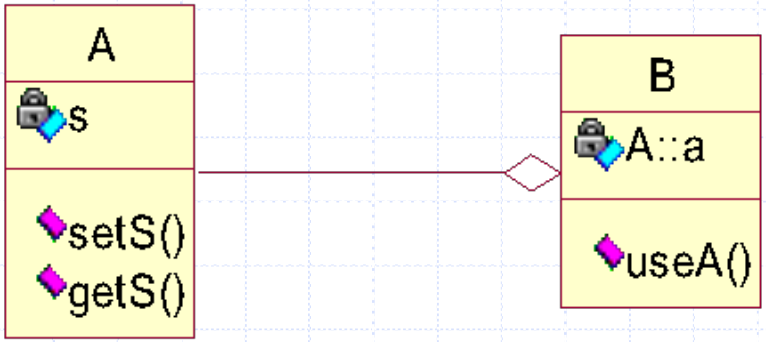
UML: Ereditarietà – “is-a”

```
class A {
      int s;
      public void setS(int) {…};
      public int getS() {…};
}
class B extends A {
      int w;
      public void setW(int) {…};
      public int getW() {…};
}
```



UML: Aggregazione

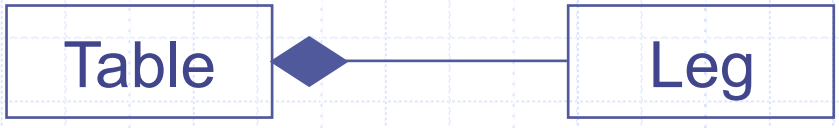
```
class A {
      int s;
      public void setS(int){…};
      public int getS() {…};
}
class B {A ob;
      public void useA() {…};
}
```



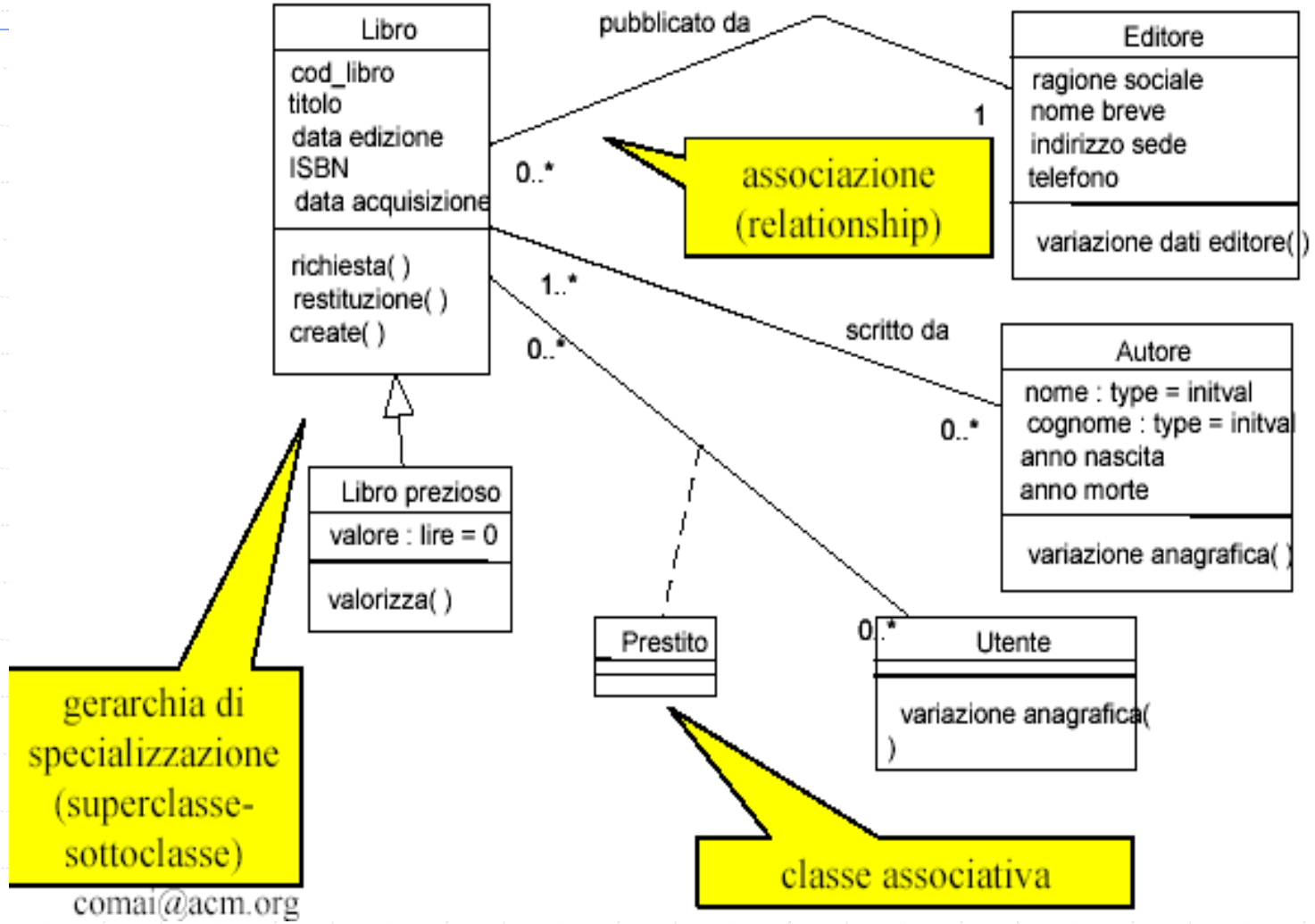
Aggregation - Composition

Use *aggregation (has-a)* when the lifecycle of the partecipating elements is different (one can exist without the other).

Use *composition (part-of)* when the *container* cannot be conceived without the *contained*.



UML – Class Diagram



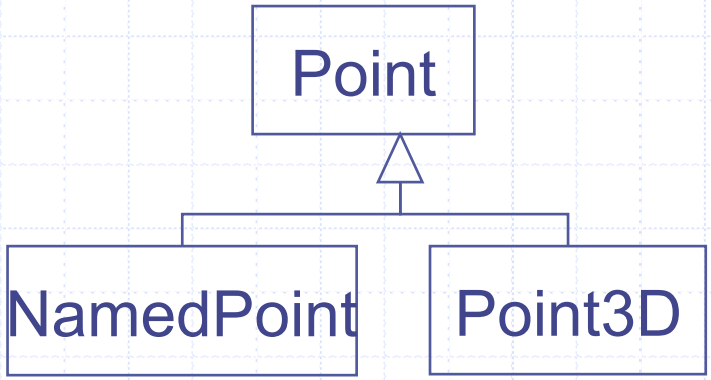
Disegno ripreso da: Adriano Comai
http://www.analisi-disegno.com/a_comai/corsi/sk_uml.htm

Overloading - Overriding

Overloading:
Funzioni con uguale nome e diversa firma possono coesistere.
move(int dx, int dy)
move(int dx, int dy, int dz)

Overriding:
Ridefinizione di una funzione in una sottoclasse (mantenendo immutata la firma)
Es. estrai() in Coda e Pila

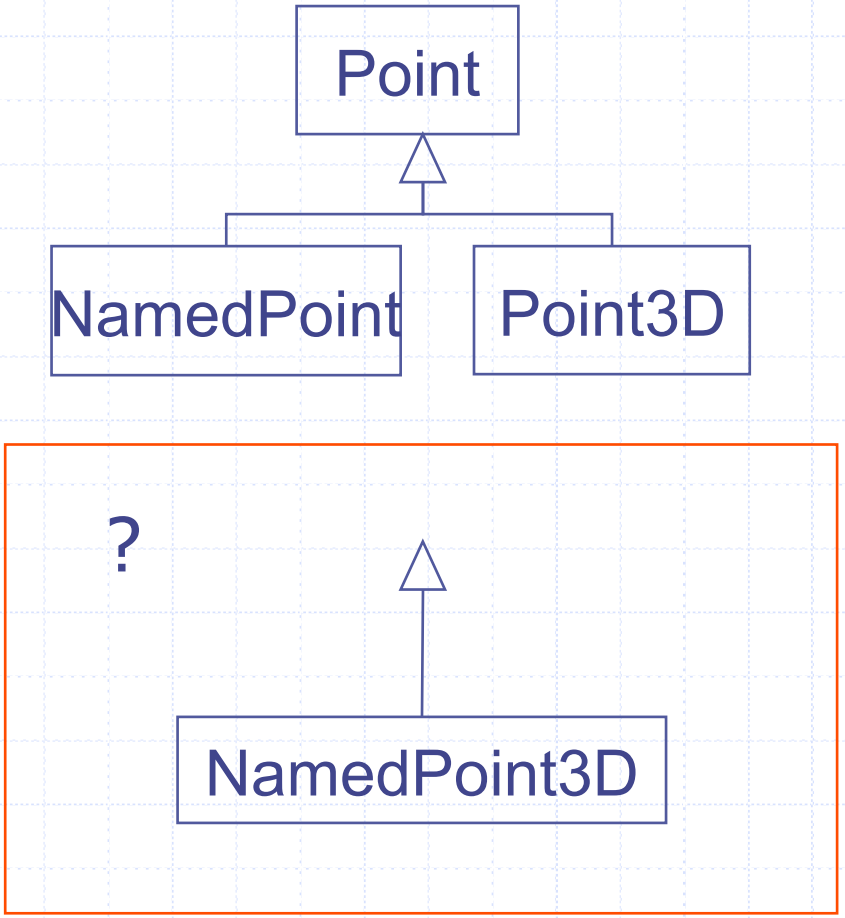
Esercizio



a) Scrivere un metodo move(int dx, int dy) in Point.

b) Estendere Point a Point3d aggiungendo una coordinata z, e fornendo un metodo move(int dx, int dy int dz) in Point3D.

Problemi con l'ereditarietà



Try – catch - finally

```
try {
    System.out.println("Dammi un intero");
    i=Integer.parseInt(s.readLine());
    System.out.println("Hai scritto "+i);
}
catch (Exception e) {e.printStackTrace();}
finally { doSomethingInEveryCase() }
```

Lettura di stringhe con GUI

```
import javax.swing.JOptionPane;
public A()  {
    ...
    String input =
JOptionPane.showInputDialog(
      "How are you?");
    System.out.println(input);
    System.exit(1);
}
```

Essenziale!
Altrimenti la thread che gestisce la GUI rimane viva, e il processo non termina

Polimorfismo

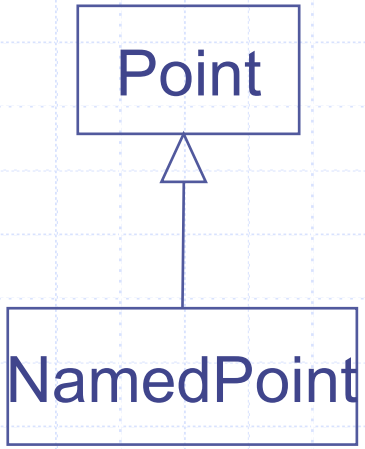
Liskov substitution principle

Se S è un sottotipo of T,
allora oggetti di tipo T in un programma
possono essere sostitituiti da oggetti di tipo S
senza alterare alcuna proprietà desiderabile del programma.

```
Point p=new
Point();
p.move(3,4);
```

Ovunque c'e' un Point posso
mettere un NamedPoint

```
Point p=new NamedPoint();
p.move(3,4);
```



Polimorphysm

```
Class X() {
   public static void main(String a[])
      Pila s; int type;
      do {
         try {
              type =Integer.parseInt(
                  JOptionPane.showInputDialog(
                  "Pila (1) o Coda (2)?"));
           } catch (Exception e) {type=0;}
        } while (type<1 || type>2);
        switch (type) {
          case 2: s=new Coda(); break;
          case 1: s=new Pila(); break;
        }
      s.insert(3);s.insert(4);
      System.out.println(s.estrai());
   }
}
```

Una funzione può comportarsi in maniera diversa a seconda
- del tipo che le viene passato
- del tipo di dato su cui è chiamata

Concetti fondamentali

```
Pila s=new Coda();
s.insert(2); s.insert(2); s.estrai()
```

Quando si chiamano i metodi su s, il sistema fa riferimento alla dichiarazione di tipo (Pila) o all'istanziazione (Coda)?

STATIC BINDING -> Pila
DYNAMIC BINDING -> Coda

Sezione: Upcast - downcast

Upcast & downcast

```
public class Test {
   public static void main(String a[]) {
      new Test();
   }
```

cast

```
   Test() {
      A a;
      B b = new B();
      a=b;
        a.f1();
        a.f2();
  }
}
```

OK: upcast implicito

NO: "method f2 not found in class A" (compiler)

```
class A              { void f1()
  {System.out.println("f1");} }
class B extends A { void f2()
  {System.out.println("f2");} }
class C extends B { void f3()
  {System.out.println("f3");} }
```

```
public class Test {
   public static void main(String a[]) {
      new Test();
   }
```

cast

```
   Test() {
      A a;
      B b = new B();
      a=b;
        a.f1();
        ((B)a).f2();
  }
}
```

OK: upcast implicito

OK: downcast corretto

```
class A              { void f1()
  {System.out.println("f1");} }
class B extends A { void f2()
  {System.out.println("f2");} }
class C extends B { void f3()
  {System.out.println("f3");} }
```

```
public class Test {
   public static void main(String a[]) {
      new Test();
   }
```

cast

```
   Test() {
      A a;
      B b = new B();
      a=b;
        a.f1();
        ((C)a).f3();
  }
}
```

OK: upcast implicito

NO: downcast illecito (runtime)
java.lang.ClassCastException

```
class A              { void f1()
  {System.out.println("f1");} }
class B extends A { void f2()
  {System.out.println("f2");} }
class C extends B { void f3()
  {System.out.println("f3");} }
```

Type conversion - cast

Si può applicare cast SOLO all'interno di una gerarchia di ereditarietà

È consigliabile usare l'operatore instanceof per verificare prima effettuare un downcast

```
if (staff[1] instanceof Manager) {
        Manager n = (Manager)staff[1];
        ...
}
```

Coercion

Una funzione può essere polimorfa senza essere stata disegnata tale intenzionalmente.

Sia *f* una funzione che prende un argomento di tipo *T*, e *S* sia un tipo che può essere *automaticamente convertito* in *T*. Allora *f* può essere detta polimorfa rispetto a *S* e *T*.

float somma(float x, float y)
accetta anche
somma (3, 3.14)
somma(2,3)
(coercion di int a float)

```
public static void main(String args[])
    try {
        Stack s=null;
        int type=0;
        do {
            try {
                type =Integer.parseInt(
                  JOptionPane.showInputDialog(
                    "Pila (1) o Coda (2)?"));
              } catch (Exception e) {type=0;}
            } while (type<1 || type>2);
            switch (type) {
              case 1: s=new Pila(); break;
              case 2: s=new Coda(); break;
            }
…
}
```

Usare Pile e Code

Modificatori: abstract

Classi dichiarate abstract non possono essere istanziate, e devono essere subclassate.

Metodi dichiarati abstract devono essere sovrascritti

Una class non abstract non può contenere abstract metods

```
public abstract class Stack{
    // qui i soliti campi: marker, contenuto…
    public void inserisci(int) {…}
    public void cresci() {…}
    public abstract int estrai();
}
public class Pila extends Stack{
   public int estrai{…}
}
public class Coda extends Stack{
    public int estrai{…}
}
```

Usare Pile e Code

Sezione: Costruttori

Costruttori

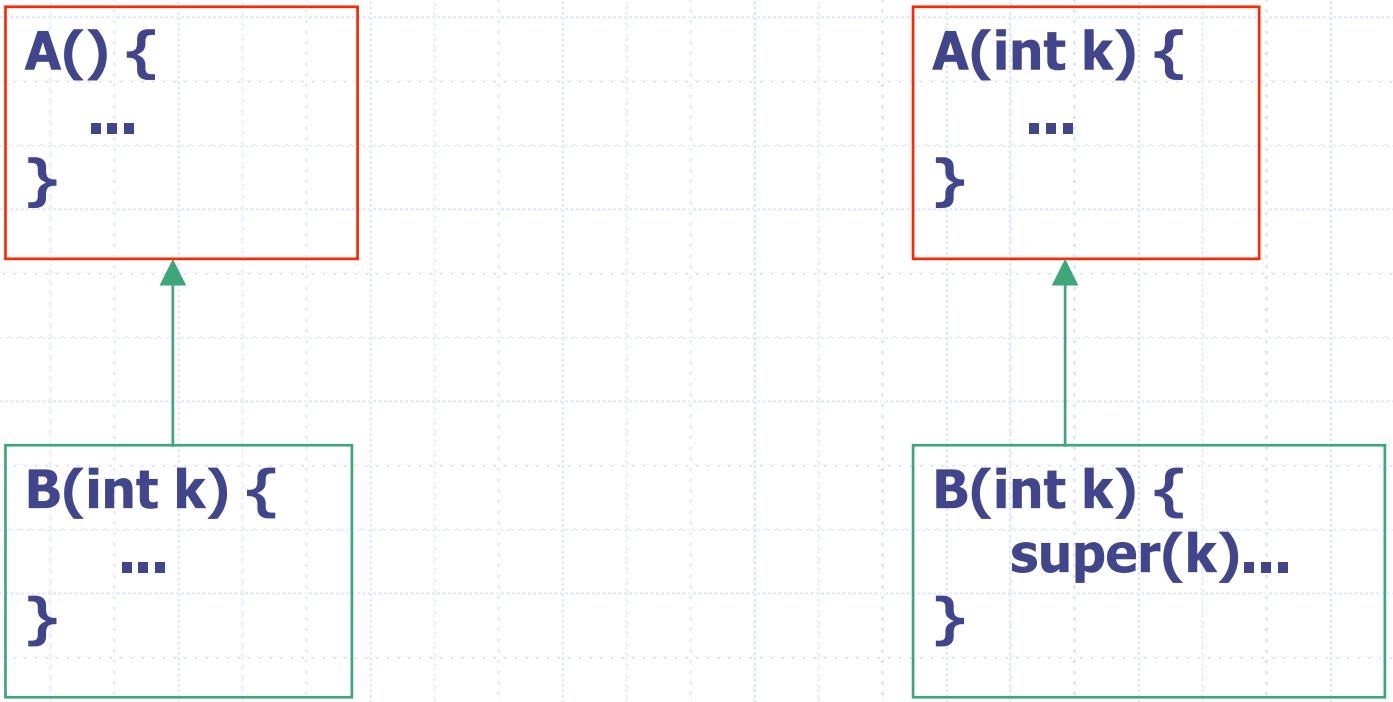
Definizione dei costruttori

Se per una classe A non scrivo nessun costruttore, il sistema automaticamente crea il costruttore A();

Se invece definisco almeno un costruttore non void, ad es. A(int s), il sistema non crea il costruttore A();

Definizione dei costruttori

Se B è figlia di A, il costruttore di B come prima cosa invoca A(), a meno che la prima istruzione non sia una super.



Invocazione dei costruttori

```
public class A {
  public A() {
    System.out.println("Creo A");
  }
}
public class B extends A {
  public B() {
    System.out.println("Creo B");
  }
  public B(int k) {
    System.out.println("Creo B_int”);
  }
}
```

```
public static void main(String [] a) {
   B b=new B(1);
 }
```

Output:
Creo A
Creo B_int

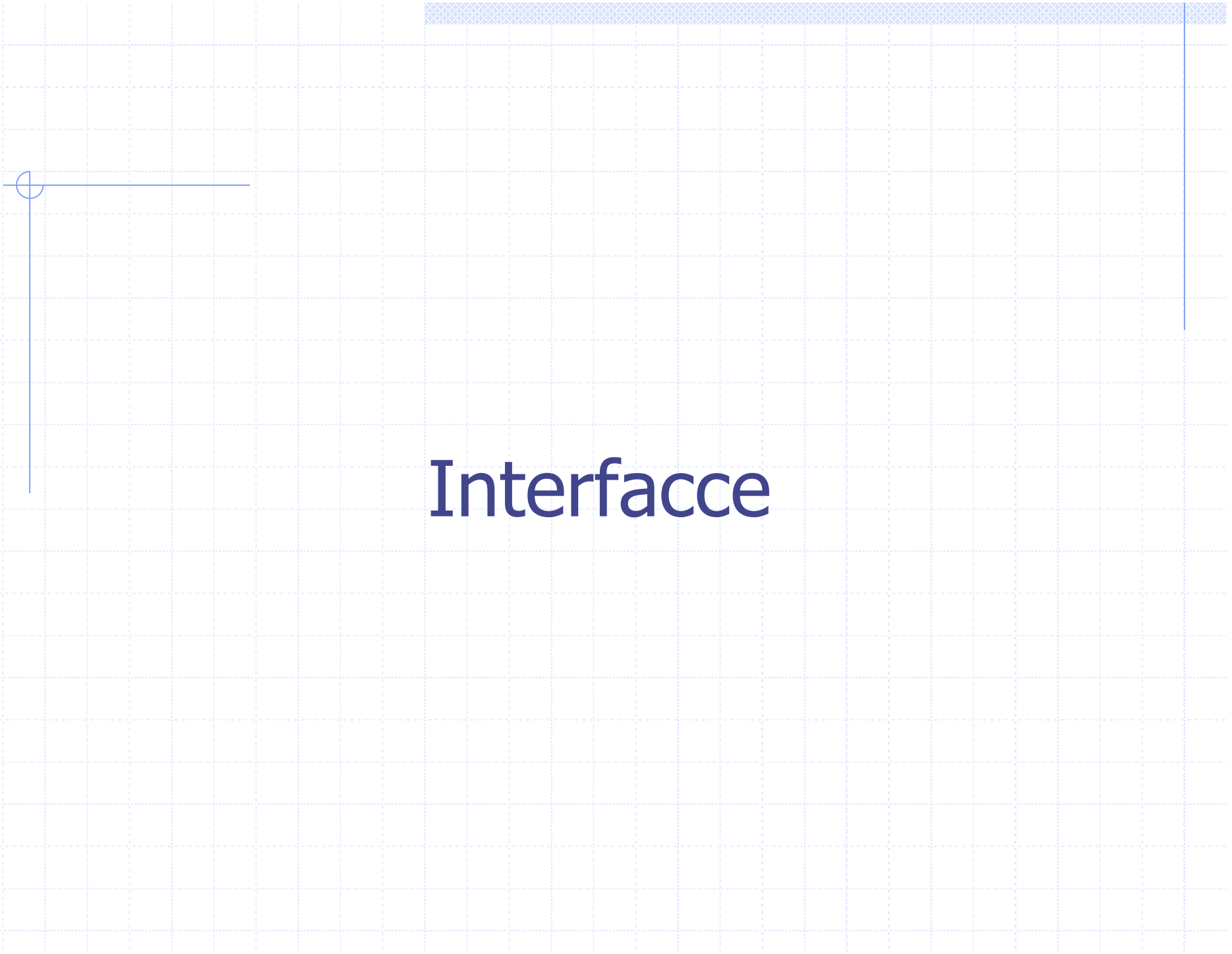
Invocazione dei costruttori

```
public class A {
  public A(int k) {
    System.out.println("Creo A");
  }
}
public class B extends A {
  public B() {
    System.out.println("Creo B");
  }
  public B(int k) {
    System.out.println("Creo B_int”);
  }
}
```

Output:
ERRORE !

Perchè ?

```
public static void main(String [] a) {
   B b=new B(1);
 }
```

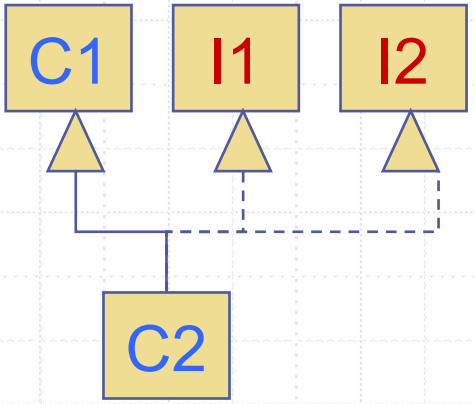

Interfacce

Interfacce

Un *interface* è una collezione di firme di metodi (senza implementazione).

Una interfaccia può dichiarare costanti.

Interfacce



Esempio di interface

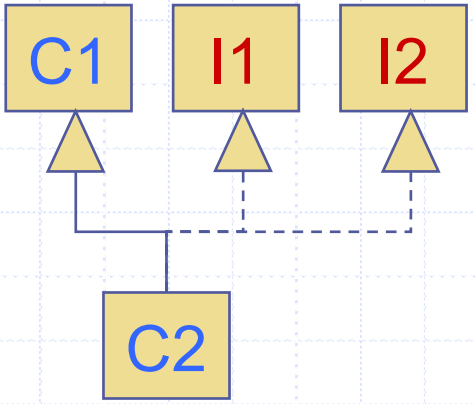
```
package strutture;
public interface Stack{
  public int estrai();
   public void insert(int z);
}
```

```
package strutture;
public class Pila implements Stack{
   …
}
```

```
package strutture;
public class Coda extends Pila{
   …
}
```

Interfacce

Le interfacce possono essere usate come "tipi"




```
I1 x = new C2();
// I1 x = new I1();  NO!!
```

Javadoc

Input

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The
name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url
argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
   try { return getImage(new URL(url, name));
   } catch (MalformedURLException e) { return null; }
}
```

Output

getImage

public Image **getImage**(URL url, String name)

Returns an Image object that can then be painted on the screen. The url argument must specify an absolute URL. The name argument is a specifier that is relative to the url argument. This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:

url - an absolute URL giving the base location of the image

name - the location of the image, relative to the url argument

Returns:

the image at the specified URL

See Also:

Image

Tags

Include tags in the following order:

@author (classes and interfaces only, required)
@version (classes and interfaces only, required.)
@param (methods and constructors only)
@return (methods only)
@exception (@throws is a synonym added in Javadoc 1.2)
@see (additional references)
@since (since what version/ since when is it available?)
@serial (or @serialField or @serialData)
@deprecated (why is deprecated, since when, what to use)

Documentation generation

To generate the html documentation, run javadoc followed by the list of source files, which the documentation is to be generated for, in the command prompt (i.e. *javadoc [files]*).

javadoc also provides additional options which can be entered as switches following the javadoc command (i.e. *javadoc [options] [files]*).

javadoc options

Here are some basic javadoc options:

- ***-author*** **- generated documentation will include a author section**
- ***-classpath [path]*** **- specifies path to search for referenced .class files.**
- ***-classpathlist [path];[path];...;[path]*** **- specifies a list locations (separated by ";") to search for referenced .class files.**
- ***-d [path]*** **- specifies where generated documentation will be saved.**
- ***-private*** **- generated documentation will include private fields and methods (only public and protected ones are included by default).**
- ***-sourcepath [path]*** **- specifies path to search for .java files to generate documentation form.**
- ***-sourcepathlist [path];[path];...;[path]*** **- specifies a list locations (separated by ";") to search for .java files to generate documentation form.**
- ***-version*** **- generated documentation will include a version section**

Examples

Basic example that generates and saves documentation to the current directory (c:\MyWork) from A.java and B.java in current directory and all .java files in c:\OtherWork\.

c:\MyWork> javadoc A.java B.java c:\OtherWork*.java

More complex example with the generated documentation showing version information and private members from all .java files in c:\MySource\ and c:\YourSource\ which references files in c:\MyLib and saves it to c:\MyDoc.

c:\> javadoc -version -private -d c:\MyDoc
-sourcepathlist c:\MySource;c:\YourSource
-classpath c:\MyLib

More info

http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

The javadoc tool does not directly document anonymous classes -- that is, their declarations and doc comments are ignored. If you want to document an anonymous class, the proper way to do so is in a doc comment of its outer class

Modificatori: visibilità

public	visibile da tutti
(non def.)	visibile da tutti nello stesso package
protected	visibile dalle sottoclassi
private	nascosta da tutti

Uso di metodi "di accesso":

```
public class  ACorrectClass {
    private String  aUsefulString;
    public String  getAUsefulString() {
        // "get" the value
        return aUsefulString;
    }
    private protected void setAUsefulString(String s)
{
        // "set" the value
        aUsefulString = s;
   }
}
```

Matrice degli accessi

Access Levels				
Specifier	**Class**	**Package**	**Subclass**	**World**
private	Y	N	N	N
no specifier	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

Vedi anche
http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Modificatori: final

Variabili dichiarate final sono costanti.

Metodi dichiarati final non possono essere sovrascritti

Classi dichiarate final non possono essere subclassate.

Convenzioni

I nomi delle Classi iniziano con la MAIUSCOLA

I nomi degli Oggetti iniziano con la MINUSCOLA

```
Pila p=new Pila();
```