

08

## Polimorfismo (inclusivo, con le classi)

Mirko Viroli

`mirko.viroli@unibo.it`

C.D.L. Ingegneria e Scienze Informatiche  
ALMA MATER STUDIORUM—Università di Bologna, Cesena

a.a. 2022/2023

## Goal della lezione

- Illustrare la connessione fra polimorfismo inclusivo e ereditarietà
- Mostrare le interconnessioni con interfacce e classi astratte
- Mostrare le varie ripercussioni nel linguaggio

## Argomenti

- Polimorfismo inclusivo con le classi
- Layout oggetti in memoria
- Autoboxing dei tipi primitivi
- Tipi a run-time (cast, `instanceof`)
- Classi astratte

- 1 Polimorfismo inclusivo con le classi
- 2 Tipi a run-time
- 3 Classi astratte
- 4 Autoboxing dei tipi primitivi, e argomenti variabili

# Ereditarietà e polimorfismo

## Ricordando il principio di sostituibilità

Se B è un sottotipo di A allora ogni oggetto di B può essere utilizzato (e “deve poter” essere utilizzabile) dove ci si attende un oggetto di A

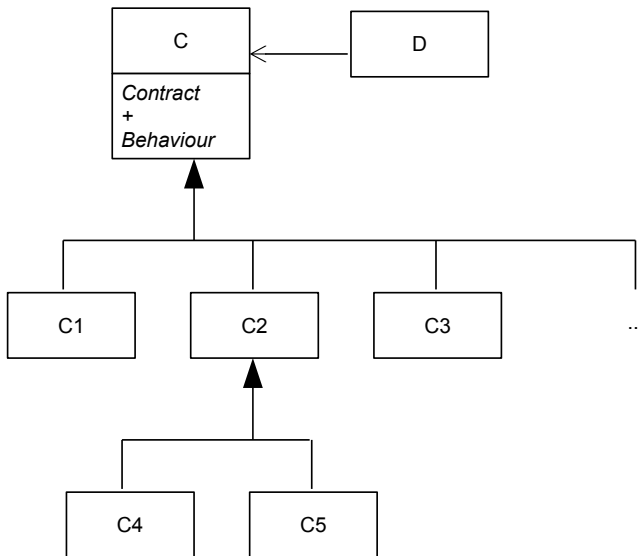
## Con l'ereditarietà

- Con la definizione: `class B extends A{..}`
- Gli oggetti della classe B rispondono a tutti i messaggi previsti dalla classe A, ed eventualmente a qualcuno in più
- Quindi un oggetto della classe B potrebbe essere passato dove se ne aspetta uno della classe A, senza comportare problemi (di “typing”)

## Conseguenza:

Visto che è possibile, corretto, ed utile, allora in Java si considera B come un sottotipo di A a tutti gli effetti!

# Polimorfismo con le classi



# Polimorfismo con le classi

## In una classe D che usi una classe C...

- ci saranno punti in D in cui ci si attende oggetti della classe C
- (come argomenti a metodi, o da depositare nei campi)
- si potranno effettivamente passare oggetti della classe C, ma anche delle classi C1, C2,...,C5, o di ogni altra classe successivamente creata che estende, direttamente o indirettamente C

## Le sottoclassi di C

A tutti gli effetti, gli oggetti delle sottoclassi di C sono compatibili con gli oggetti della classe C

- hanno medesimo contratto (in generale, qualche operazione in più)
- hanno tutti i campi definiti in C (in generale, qualcuno in più)
- hanno auspicabilmente un comportamento compatibile

# Layout oggetti in memoria

## Alcuni aspetti del layout degli oggetti in memoria...

Diamo alcune informazioni generali e astratte. Ogni JVM potrebbe realizzare gli oggetti in modo un po' diverso. Questi elementi sono tuttavia comuni.

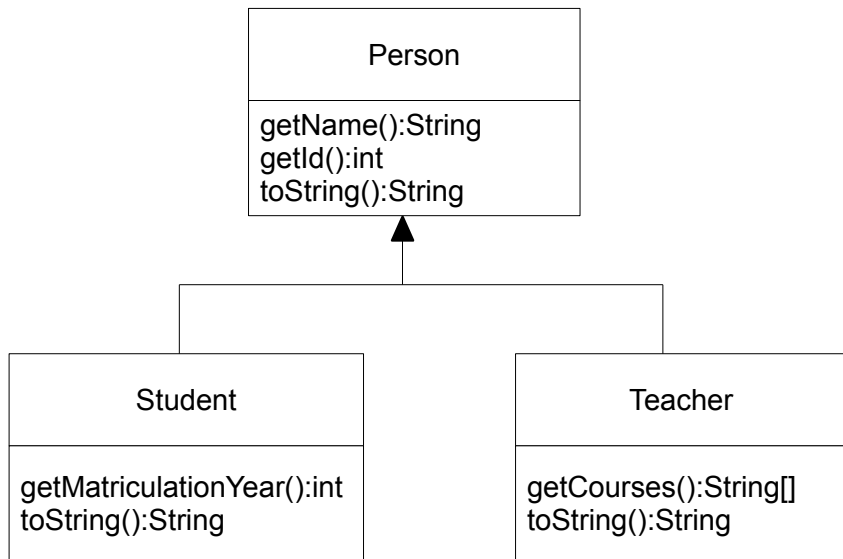
## Struttura di un oggetto in memoria

- Inizia con una intestazione ereditata da `Object` (16 byte circa), che include
  - ▶ Indicazione di quale sia la classe dell'oggetto (runtime type information)
  - ▶ Tabella dei puntatori ai metodi, per supportare il late-binding
  - ▶ I campi (privati) della classe `Object`
- Via via tutti i campi della classe, a partire da quelli delle superclassi

## Conseguenze: se la classe `C` è sottoclasse di `A`...

Allora un oggetto di `C` è simile ad un oggetto di `A`, ha solo informazioni aggiuntive in fondo, e questo semplifica la sostituibilità!

# Esempio applicazione polimorfismo fra classi – UML





# Person

```
1 public class Person {
2
3     private final String name;
4     private final int id;
5
6     public Person(final String name, final int id) {
7         super();
8         this.name = name;
9         this.id = id;
10    }
11
12    public String getName() {
13        return this.name;
14    }
15
16    public int getId() {
17        return this.id;
18    }
19
20    public String toString() {
21        return "P [name=" + this.name + ", id=" + this.id + "]";
22    }
23 }
```

# Student

```
1 public class Student extends Person {
2
3     final private int matriculationYear;
4
5     public Student(final String name, final int id,
6                   final int matriculationYear) {
7         super(name, id);
8         this.matriculationYear = matriculationYear;
9     }
10
11     public int getMatriculationYear() {
12         return matriculationYear;
13     }
14
15     public String toString() {
16         return "S [getName()=" + getName() +
17             ", getId()=" + getId() +
18             ", matriculationYear=" + matriculationYear + "]";
19     }
20
21 }
```

# Teacher

```
1 import java.util.Arrays;
2
3 public class Teacher extends Person {
4
5     final private String[] courses;
6
7     public Teacher(final String name, final int id,
8                     final String[] courses) {
9         super(name, id);
10        this.courses = courses;
11    }
12
13    public String[] getCourses() {
14        // copia difensiva necessaria a preservare incapsulamento
15        return Arrays.copyOf(courses, courses.length);
16    }
17
18    public String toString() {
19        return "T [getName()=" + getName() +
20            ", getId()=" + getId() +
21            ", courses=" + Arrays.toString(courses) + "]";
22    }
23 }
```

# UsePerson

```
1 public class UsePerson {
2
3     public static void main(String[] args) {
4
5         final var people = new Person[]{
6             new Student("Marco Rossi",334612,2013),
7             new Student("Gino Bianchi",352001,2013),
8             new Student("Carlo Verdi",354100,2012),
9             new Teacher("Mirko Viroli",34121,new String[]{
10                 "OOP","PPS","PC"
11             })
12         };
13
14         for (final var p: people){
15             System.out.println(p.getName()+" : "+p);
16         }
17     }
18 }
```

# La differenza col caso del polimorfismo con le interfacce

## Polimorfismo con le interfacce

- La classe D usa una interfaccia I, non un'altra classe C
- Si può assumere vi sia un certo contratto, ma non che vi sia uno specifico comportamento (quello di C) che sia stato eventualmente specializzato

## Le classi non consentono “ereditarietà multipla” (in C++ si)

- \*NON\* è possibile in Java dichiarare: `class C extends D1, D2 ...`
  - ▶ si creerebbero problemi se D1 e D2 avessero proprietà comuni
  - ▶ diventerebbe complicato gestire la struttura in memoria dell'oggetto
- Con le interfacce non ci sono questi problemi, risultato:
  - ▶ è molto più semplice prendere una classe esistente e renderla compatibile con una interfaccia I, piuttosto che con una classe C

# Riassunto polimorfismo inclusivo

## Polimorfismo

- Fornisce sopratipi che raccolgono classi uniformi tra loro
- Usabili da funzionalità ad alta riusabilità
- Utile per costruire collezioni omogenee di oggetti

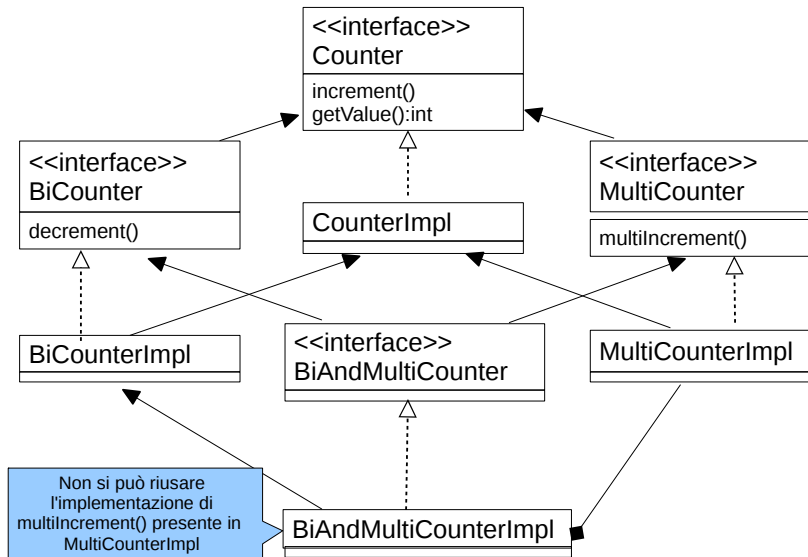
## Polimorfismo con le interfacce

- Solo relativo ad un contratto
- Facilità nel far aderire al contratto classi esistenti
- Spesso vi è la tendenza a creare un alto numero di interfacce

## Polimorfismo con le classi

- Relativo a contratto e comportamento
- In genere ci si aderisce per costruzione dall'inizio
- Vincolato dall'ereditarietà singola

# Come simulare ereditarietà multipla?



# Come simulare ereditarietà multipla?

## Definizioni

- `interface Counter{...}`
- `interface MultiCounter extends Counter{...}`
- `interface BiCounter extends Counter{...}`
- `interface BiAndMultiCounter extends MultiCounter,BiCounter{...}`
- `class CounterImpl implements Counter{...}`
- `class MultiCounterImpl extends CounterImpl implements MultiCounter{...}`
- `class BiCounterImpl extends CounterImpl implements BiCounter{...}`
- `class BiAndMultiCounterImpl extends BiCounterImpl implements BiAndMultiCounter{...}`

## Implementazione di BiAndMultiCounterImpl

- si estende da BiCounterImpl, si delega via composizione ad un oggetto di MultiCounterImpl

## Complessivamente

- si ha completa e corretta sostituibilità tramite le interfacce
  - p.e.: `BiCounter bc = new BiAndMultiCounterImpl(); // OK`
  - si ha ottimo riuso delle implementazioni
- ⇒ si esplori la possibilità di usare solo delegazione, non ereditarietà



- 1 Polimorfismo inclusivo con le classi
- 2 Tipi a run-time**
- 3 Classi astratte
- 4 Autoboxing dei tipi primitivi, e argomenti variabili

# Everything is an Object

## Perché avere una radice comune per tutte le classi?

- Consente di fattorizzare il comportamento comune
- Consente la costruzione di funzionalità che lavorano su qualunque oggetto

## Esempi di applicazione:

- Container polimorfici, ad esempio via array di tipo `Object[]`
  - ▶ permette di costruire un elenco di oggetti di natura anche diversa
  - ▶ `new Object[]{new SimpleLamp(), new Integer(10)}`
- Definizione di metodi con numero variabile di argomenti
  - ▶ argomenti codificati come `Object[]`

# Uso di Object []

```
1  /* Tutti gli oggetti possono formare un elenco Object[] */
2  public class AObject {
3      public static void main(String[] s) {
4          final Object[] os = new Object[5];
5          os[0] = new Object();
6          os[1] = "string";
7          os[2] = Integer.valueOf(10);
8          os[3] = new int[] { 10, 20, 30 };
9          os[4] = new java.util.Date();
10         printAll(os);
11         System.out.println(Arrays.toString(os));
12         System.out.println(Arrays.deepToString(os));
13     }
14
15
16     public static void printAll(final Object[] array) {
17         for (final Object o : array) {
18             System.out.println("Object:" + o.toString());
19         }
20     }
21 }
```

# Tipo statico e tipo a run-time

## Una dualità introdotta dal subtyping (polimorfismo inclusivo)

- Tipo statico: il tipo di dato di una espressione desumibile dal compilatore
- Tipo run-time: il tipo di dato del valore(/oggetto) effettivamente presente (potrebbe essere un sottotipo di quello statico)
  - ▶ in questo caso le chiamate di metodo sono fatte per late-binding

## Esempio nel codice di `printAll()`, dentro al `for`

- Tipo statico di `o` è `Object`
- Tipo run-time di `o` varia di volta in volta: `Object, String, Integer, ...`

## Ispezione tipo a run-time

- In alcuni casi è necessario ispezionare il tipo a run-time
- Lo si fa con l'operatore `instanceof`

## instanceof e conversione di tipo

```
1 /* Everything is an Object, ma quale?? */
2 public class AObject2 {
3     public static void main(String[] s) {
4         final Object[] os = new Object[5];
5         os[0] = new Object();
6         os[1] = Integer.valueOf(10);
7         os[2] = Integer.valueOf(20);
8         os[3] = new int[] { 10, 20, 30 };
9         os[4] = Integer.valueOf(30);
10        printAllAndSum(os);
11    }
12
13    /* Voglio anche stampare la somma degli Integer presenti */
14    public static void printAllAndSum(final Object[] array) {
15        int sum = 0;
16        for (final Object o : array) {
17            System.out.println("Object:" + o.toString());
18            if (o instanceof Integer) { // test a runtime
19                final Integer i = (Integer) o; // (down)cast
20                sum = sum + i.intValue();
21            }
22        }
23        System.out.println("Sum of Integers: " + sum);
24    }
```

## instanceof e conversione di tipo

### Ispezione ed uso della sottoclasse effettiva

Data una variabile (o espressione) del tipo statico C può essere necessario capire se sia della sottoclasse D, in tal caso, su tale oggetto si vuole richiamare un metodo specifico della classe D.

### Coppia instanceof + conversione

- con l'operatore instanceof si verifica se effettivamente sia di tipo D
- con la conversione si deposita il riferimento in una espressione con tipo statico D
- a questo punto si può invocare il metodo

### Solo due tipi di conversione fra classi consentite

- Upcast: da sottoclasse a superclasse (spesso automatica)
- Downcast: da superclasse a sottoclasse (potrebbe fallire)

# Errori possibili connessi alle conversioni

## Errori semantici (a tempo di compilazione, quindi innocui)

- Tentativo di conversione che non sia né upcast né downcast
- Chiamata ad un metodo non definito dalla classe (statica) del receiver

## Errori d'esecuzione (molto pericolosi, evitabili con l'`instanceof`)

- Downcast verso una classe incompatibile col tipo dinamico, riportato come `ClassCastException`

```
1  /* Showing ClassCastException */
2  public class ShowCCE {
3      public static void main(String[] as) {
4          Object o = Integer.valueOf(10);
5          Integer i = (Integer) o; // OK
6          String s = (String) o; // ClassCastException
7          // int i = o.intValue(); // No, intValue() non def.
8          // String s = (String)i; // No, tipi incompatibili
9      }
10 }
```

## instanceof, conversioni e Person

```
1 public class UsePerson2 {
2
3     public static void main(String[] args) {
4
5         final Person[] people = new Person[] {
6             new Student("Marco Rossi", 334612, 2013),
7             new Student("Gino Bianchi", 352001, 2013),
8             new Student("Carlo Verdi", 354100, 2012),
9             new Teacher("Mirko Viroli", 34121,
10                 new String[] { "PO", "FINF-A", "ISAC" }) };
11
12         for (final Person p : people) {
13             if (p instanceof Student) {
14                 final Student s = (Student) p; // Qui non fallisce
15                 System.out.println(s.getName() + " " +
16                     s.getMatriculationYear());
17             }
18         }
19     }
20 }
```



- 1 Polimorfismo inclusivo con le classi
- 2 Tipi a run-time
- 3 Classi astratte**
- 4 Autoboxing dei tipi primitivi, e argomenti variabili

# Motivazioni

## Fra interfacce e classi

- Le interfacce descrivono solo un contratto
- Le classi definiscono un comportamento completo
- ...c'è margine per costrutti intermedi?

## Classi astratte

- Sono usate per descrivere classi dal comportamento parziale, ossia in cui alcuni metodi sono dichiarati ma non implementati
- Tali classi non sono istanziabili (l'operatore `new` non può essere usato)
- Possono essere estese e ivi completate, da cui la generazione di oggetti

## Tipica applicazione: pattern Template Method

Serve a dichiarare uno schema di strategia con un metodo che definisce un comportamento comune (spesso `final`), ma che usa metodi da concretizzare in sottoclassi

# Classi astratte

## Una classe astratta:

- è dichiarata tale: `abstract class C ... {...}`
- non può essere usata per generare oggetti
- può opzionalmente dichiarare metodi astratti:
  - ▶ hanno forma ad esempio: `abstract int m(int a, String s);`
  - ▶ ossia senza body, come nelle dichiarazioni delle interfacce

## Altri aspetti

- può definire campi, costruttori, metodi, concreti e non
- ...deve definire con cura il loro livello d'accesso
- può estendere da una classe astratta o non astratta
- può implementare interfacce, senza essere tenuta ad ottemperarne il contratto
- chi estende una classe astratta può essere non-astratto solo se implementa tutti i metodi definiti

# Esempio: LimitedLamp come classe astratta

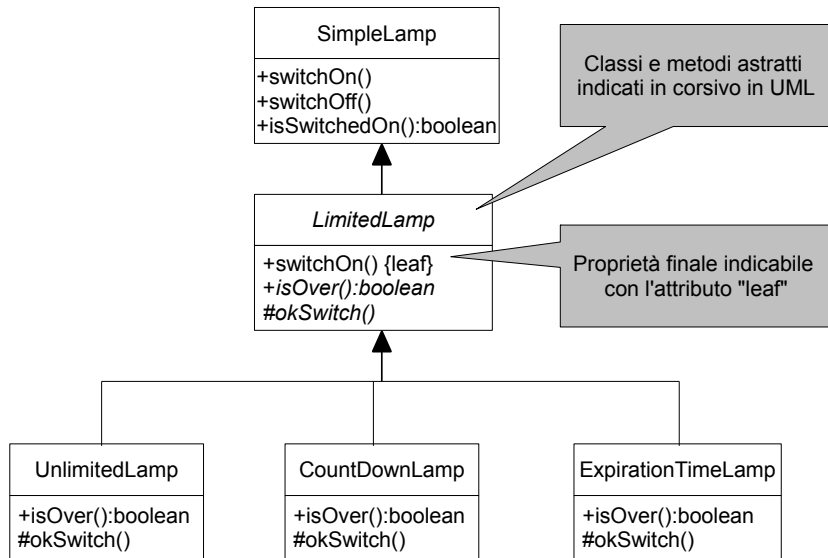
## Obbiettivo

- Vogliamo progettare una estensione di SimpleLamp col concetto di esaurimento
- La strategia con la quale gestire tale esaurimento può essere varia
- Ma bisogna far sì che qualunque strategia si specifichi, sia garantito che:
  - ▶ la lampadina si accenda solo se non esaurita
  - ▶ in caso di effettiva accensione sia possibile tenerne traccia ai fini della strategia

## Soluzione

- Un uso accurato di `abstract`, `final`, e `protected`
- Daremo tre possibili specializzazioni per una LimitedLamp
  - ▶ che non si esaurisce mai
  - ▶ che si esaurisce all'n-esima accensione
  - ▶ che si esaurisce dopo un certo tempo dalla prima accensione

# UML complessivo



# SimpleLamp

```
1 public class SimpleLamp {  
2  
3     private boolean switchedOn;  
4  
5     public SimpleLamp() {  
6         this.switchedOn = false;  
7     }  
8  
9     public void switchOn() {  
10        this.switchedOn = true;  
11    }  
12  
13    public void switchOff() {  
14        this.switchedOn = false;  
15    }  
16  
17    public boolean isSwitchedOn() {  
18        return this.switchedOn;  
19    }  
20 }
```

# LimitedLamp

```
1 public abstract class LimitedLamp extends SimpleLamp {
2
3     public LimitedLamp() {
4         super();
5     }
6
7     /* Questo metodo è finale:
8      * - regola la coerenza con okSwitch() e isOver() */
9     public final void switchOn() {
10         if (!this.isSwitchedOn() && !this.isOver()) {
11             super.switchOn();
12             this.okSwitch();
13         }
14     }
15
16     // Cosa facciamo se abbiamo effettivamente acceso? Dipende dalla strategia
17     protected abstract void okSwitch();
18
19     /* Strategia per riconoscere se la lamp è esaurita */
20     public abstract boolean isOver();
21
22     public String toString() {
23         return "Over: " + this.isOver() +
24             ", switchedOn: " + this.isSwitchedOn();
25     }
26 }
```

# UnlimitedLamp

```
1 /* Non si esaurisce mai */
2 public class UnlimitedLamp extends LimitedLamp {
3
4     /* Nessuna informazione extra da tenere */
5     public UnlimitedLamp() {
6         super();
7     }
8
9     /* Allo switchOn.. non faccio nulla */
10    protected void okSwitch() {
11    }
12
13    /* Non è mai esaurita */
14    public boolean isOver() {
15        return false;
16    }
17 }
```



# CountdownLamp

```
1  /* Si esaurisce all'n-esima accensione */
2  public class CountdownLamp extends LimitedLamp {
3
4      /* Quanti switch mancano */
5      private int countdown;
6
7      public CountdownLamp(final int countdown) {
8          super();
9          this.countdown = countdown;
10     }
11
12     /* Allo switchOn.. decremento il count */
13     protected void okSwitch() {
14         this.countdown--;
15     }
16
17     /* Finito il count.. lamp esaurita */
18     public boolean isOver() {
19         return this.countdown == 0;
20     }
21 }
```

# ExpirationTimeLamp

```
1  /* Si esaurisce dopo un certo tempo (reale) dopo la prima accensione */
2  public class ExpirationTimeLamp extends LimitedLamp {
3
4      /* Tengo il momento dell'accensione e la durata */
5      private Date firstSwitchDate;
6      private long duration;
7
8      public ExpirationTimeLamp(final long duration) {
9          super();
10         this.duration = duration;
11         this.firstSwitchDate = null;
12     }
13
14     /* Alla prima accensione, registro la data */
15     protected void okSwitch() {
16         if (this.firstSwitchDate == null) {
17             this.firstSwitchDate = new java.util.Date();
18         }
19     }
20
21     /* Esaurita se è passato troppo tempo */
22     public boolean isOver() {
23         return this.firstSwitchDate != null &&
24             (new Date().getTime() - this.firstSwitchDate.getTime())
25             >= this.duration;
26     }
27 }
```

# UseLamps

```
1 public class UseLamps {
2     // clausola throws Exception qui sotto necessaria!!
3     public static void main(String[] s) throws Exception {
4         LimitedLamp lamp = new UnlimitedLamp();
5         lamp.switchOn();
6         System.out.println("ul| " + lamp);
7         for (int i = 0; i < 1000; i++) {
8             lamp.switchOff();
9             lamp.switchOn();
10        }
11        System.out.println("ul| " + lamp); // non si è esaurita
12
13        lamp = new CountdownLamp(5);
14        for (int i = 0; i < 4; i++) {
15            lamp.switchOn();
16            lamp.switchOff();
17        }
18        System.out.println("cl| " + lamp);
19        lamp.switchOn();
20        System.out.println("cl| " + lamp); // al quinto switch si esaurisce
21
22        lamp = new ExpirationTimeLamp(1000); // 1 sec
23        lamp.switchOn();
24        System.out.println("el| " + lamp);
25        Thread.sleep(10000); // attendo 1.1 secs
26        System.out.println("el| " + lamp); // dopo 1.1 secs si è esaurita
27    }
28 }
```

# Classi astratte vs interfacce

- Due versioni quasi equivalenti
- Unica differenza, ereditarietà multipla per le interfacce

```
1 /* Versione interfaccia */
2 public interface Counter{
3     void increment();
4     int getValue();
5 }
6
7 /* Versione classe astratta */
8 public abstract class Counter{
9     public abstract void increment();
10    public abstract int getValue();
11 }
```

## Wrap-up su ereditarietà

Il caso più generale:

```
class C extends D implements I,J,K,L {...}
```

Cosa deve/può fare la classe C

- deve implementare tutti i metodi dichiarati in I,J,K,L e super-interfacce
- può fare overriding dei metodi (non finali) definiti in D e superclassi

Classe astratta:

```
abstract class CA extends D implements I,J,K,L {...}
```

Cosa deve/può fare la classe CA

- non è tenuta a implementare alcun metodo
- può implementare qualche metodo per definire un comportamento parziale

- 1 Polimorfismo inclusivo con le classi
- 2 Tipi a run-time
- 3 Classi astratte
- 4 Autoboxing dei tipi primitivi, e argomenti variabili**

# Autoboxing dei tipi primitivi

## Ancora sui Wrapper dei tipi primitivi

- `Integer i = Integer.valueOf(10);`
- `Double d = Double.valueOf(10.5);`
- ... ossia, ogni valore primitivo può essere “boxed” in un oggetto

## Autoboxing

- Una tecnica recente di Java
- Si simula l'equivalenza fra tipi primitivi e loro Wrapper
- Due meccanismi:
  - ▶ Se si trova un primitivo dove ci si attende un oggetto, se ne fa il boxing
  - ▶ Se si trova un wrapper dove ci si attende un primitivo, si fa il de-boxing

## Risultato

- Si simula meglio l'idea “Everything is an Object”

# Uso dell'autoboxing

```
1  /* Showcase dell'autoboxing */
2  public class Boxing {
3      public static void main(String[] s) {
4          final Object[] os = new Object[5];
5          os[0] = new Object();
6          os[1] = 5; // equivale a os[1]=Integer.valueOf(5);
7          os[2] = 10; // equivale a os[2]=Integer.valueOf(10);
8          os[3] = true; // equivale a os[3]=Boolean.valueOf(true);
9          os[4] = 20.5; // equivale a os[4]=Double.valueOf(20.5);
10         final Integer[] is = new Integer[] { 10, 20, 30, 40 };
11         final int i = is[0] + is[1] + is[2] + is[3];
12         // equivale a: is[0].intValue() + is[1].intValue() + ..
13         // non funzionerebbe se 'is' avesse tipo Object[]..
14         System.out.println("Sum: " + i); // 100
15     }
16 }
```



# Variable arguments

A volte è utile che i metodi abbiano un numero variabile di argomenti

- `int i=sum(10,20,30,40,50,60,70);`
- `printAll(10,20,3.5,new Object());`
- prima di Java 5 si simulava passando un unico array

## Variable arguments

- L'ultimo (o unico) argomento di un metodo può essere del tipo "Type... argname"
  - ▶ p.e. `void m(int a,float b,Object... args){...}`
- Nel body del metodo, argname è trattato come un `Type[]`
- Chi chiama il metodo, invece che passare un array, passa una lista di argomenti di tipo Type
- Funziona automaticamente con polimorfismo, autoboxing, `instanceof`,...

# Uso dei variable arguments

```
1 public class VarArgs {
2     // somma un numero variabile di Integer
3     public static int sum(final Integer... args) {
4         int sum = 0;
5         for (int i : args) {
6             sum = sum + i;
7         }
8         return sum;
9     }
10
11     // stampa il contenuto degli argomenti, se meno di 10
12     public static void printAll(final String start, final Object... args) {
13         System.out.println(start);
14         if (args.length < 10) {
15             for (final Object o : args) {
16                 System.out.println(o);
17             }
18         }
19     }
20
21     public static void main(String[] s) {
22         System.out.println(sum(10, 20, 30, 40, 50, 60, 70, 80));
23         printAll("inizio", 1, 2, 3.2, true, new int[] { 10 }, new Object());
24
25         System.out.format("%d %d\n", 10, 20); // C-like printf
26     }
27 }
```

# Preview del prossimo laboratorio

## Obbiettivi

Familiarizzare con:

- Estensione delle classi e corrispondente polimorfismo
- Classi astratte
- Tipi a run-time e boxing