UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Laboratorio di Programmazione ad Oggetti

Ph.D. Juri Di Rocco
juri.dirocco@univaq.it
http://jdirocco.github.io/

Sommario

- Ereditarietà
  - Introduzione e quando utilizzarla
  - Ereditarietà e costruttori
  - Hiding variabili
- Keyword `super`
- Keyword `abstract`
- Keyword `final`
- Invocazione metodi
- Polimorfismo
  - Overloading metodi
  - Argomenti variabili
  - Overriding metodi e annotazione `@Override`
- Casting & `instanceof`

Introduzione (1)

- Idea alla base: E' possibile creare una classe a partire da classi esistenti
- E' una relazione tra una cosa più generale (detta superclasse o padre) ed una più specifica (detta sottoclasse o figlia)
- Viene detta anche relazione *"IS-A"*
- Esempi
  - Cane **è** un animale, aereo **è** un veicolo, chitarra **è** uno strumento musicale
- Oggetti figlio possono essere utilizzati al posto di oggetti padre (principio di sostituibilità di **Liskov**) ma non il viceversa, cioè il padre non è un sostituto per il figlio
  - https://en.wikipedia.org/wiki/Liskov_substitution_principle
- Meccanismo di riuso *white-box*

Introduzione (2)

- Java usa `extends` per esprimere che una classe è una sottoclasse di un'altra
- E' possibile specificare **soltanto una** superclasse (ereditarietà singola)
- Sottoclasse eredita variabili e metodi dalla sua superclasse e da tutti i suoi predecessori ovvero
  - Eredita i membri che sono dichiarati `public` o `protected`
  - Eredita i membri dichiarati package se le classi appartengono allo stesso package
  - **NON** vengono ereditati i costruttori
- Sottoclasse può utilizzare tali membri così come sono oppure può
  - **Nascondere** le variabili
  - O può effettuare **l'override** dei metodi

Esempio 1 (1)

```
public class Libro {
    public String titolo;
    public String autore;
    public String editore;
    public int numeroPagine;
    public int prezzo;
    //. . .
}
public class LibroSuJava {
    public String titolo;
    public String autore;
    public String editore;
    public int numeroPagine;
    public int prezzo;
    public final String ARGOMENTO_TRATTATO = "Java";
    //. . .
}
```

Esempio 1 (2)

```
public class Libro {
    public String titolo;
    public String autore;
    public String editore;
    public int numeroPagine;
    public int prezzo;
    //. . .
}
public class LibroSuJava extends Libro {
    public final String ARGOMENTO_TRATTATO = "Java";
    //. . .
}
```

Esempio 2

```
public class Point {
	private int x,y;
	public Point(int x, int y) {
		this.x = x;
		this.y = y;
	}
}
public class Point3D extends Point {
	private int z;
	public Point3D(int x, int y, int z) {
		super(x, y);
		this.z = z;
	}
}
Point p = new Point3D();
```

⟸ Principio di sostituibilità di Liskov

Quando utilizzare l'ereditarietà (1)

- Come si risponde alla domanda
  - Tra classe A e B vi è una relazione «è un»?
- La risposta risiede nel fatto che vi deve essere **relazione** concettuale tra le due classi e non sui dati/operazioni contenuti nella classe

Quando utilizzare l'ereditarietà (2)

› Esempio errato (Trapezio non è un triangolo)

```
public class Triangolo {
    public final int NUMERO_LATI = 3;
    public float lunghezzaLatoUno;
    public float lunghezzaLatoDue;
    public float lunghezzaLatoTre;
    //. . .
}
public class Trapezio extends Triangolo {
    public final int NUMERO_LATI = 4;
    public float lunghezzaLatoQuattro;
    //. . .
}
```

Classe Object

› Classe Object appartiene al package `java.lang`

› Superclasse di ogni classe

› Esempio

```
public class Arte {
}
```

Compilatore traduce in

```
public class Arte extends Object {
}
```

Ereditarietà e costruttori (1)

- A cosa serve un costruttore?
  - Per creare oggetti (mediante keyword `new`) e soprattutto *inizializzare il suo stato* (tramite la sua invocazione) che rappresenta la manifestazione concreta di un concetto astratto descritto mediante una classe
  - Da un punto di vista implementativo la creazione di un oggetto comporta la creazione di un'area di memoria (nell'heap) che contiene i valori delle variabili di istanza
- Pertanto **non** possiamo ereditarli ed eventualmente ridefinirli (ovverride) in quanto devono essere **utilizzati, ovvero invocati,** all'atto della creazione *dell'oggetto figlio*
- Esempio

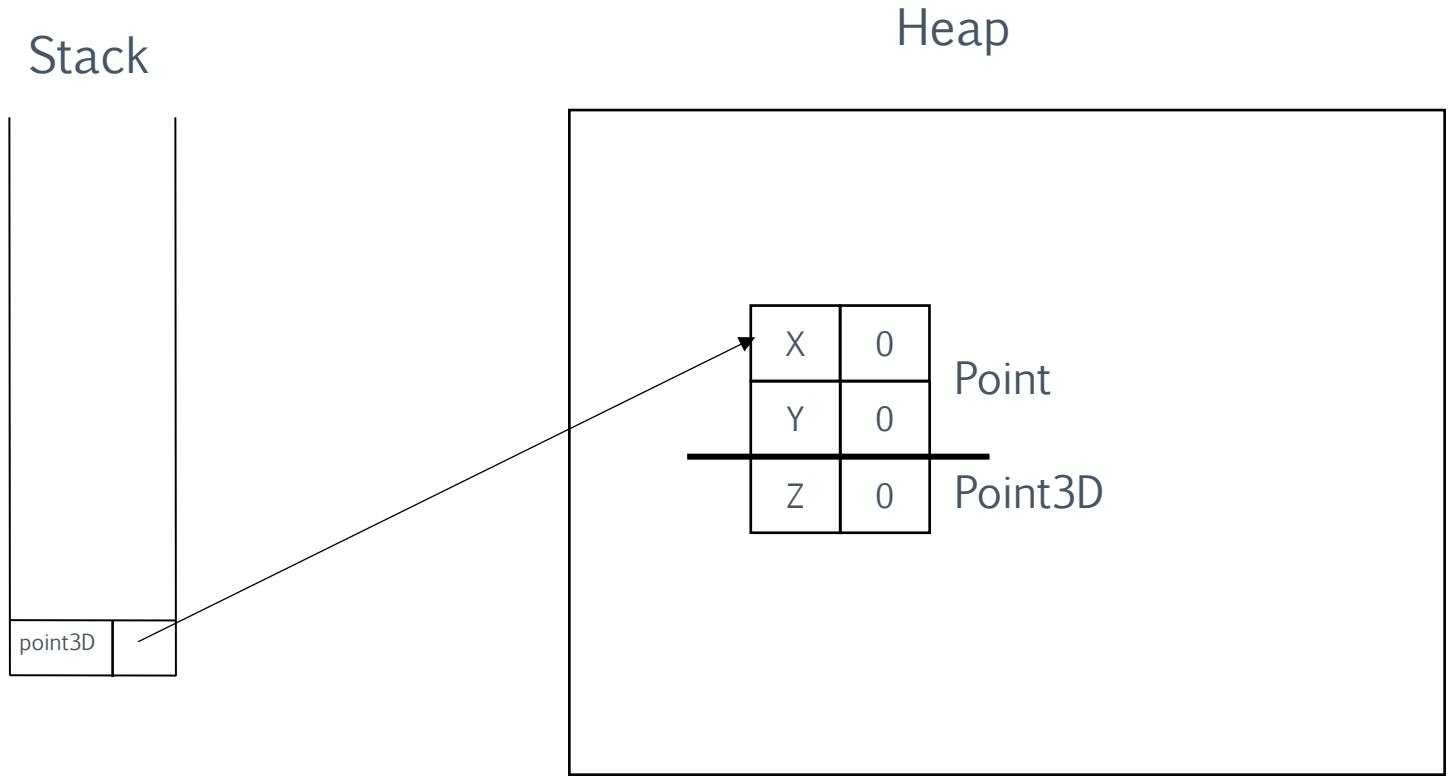
```
Point3D point3D = new Point3D();

Point3D point3D = new Point(); //Errore in compilazione
```

Ereditarietà e costruttori (2)

```
Point3D point3D = new Point3D();
```

L'oggetto deve memorizzare i valori delle variabili di istanza che rappresentano classe `Point3D` e chiaramente dei suoi antenati



Hiding variabili (1)

```
class Point {
      protected int x,y;
      public int getX() {
            return x;
      }
      public int getY() {
            return y;
      }
}
class Point3D extends Point {
      protected float x, y;
      protected float z;
……
```

Hiding variabili (2)

```
.....
    public void move(int dx, int dy, int dz) {
            x = dx;
            y = dy;
            z = dz;
    }
}
class Test {
      public static void main(String[] args) {
            Point3D p = new Point3D();
            p.move(10, 20, 30);
            System.out.println("x=" + p.getX() + ", y=" + p.getY());
            System.out.println("x=" + p.x + ", y=" + p.y);
      }
}
```

Vediamo come si comporta JAVA in questo caso – Runner

Keyword super (1)

- Viene utilizzato all'interno di una classe per riferirsi alla superclasse
- Può essere utilizzato all'interno di un costruttore di una classe per invocarne uno della superclasse
  - Se non definito esplicitamente il compilatore invoca **automaticamente** quello di default
- Si può utilizzare per riferirsi ad attributi o a metodi
  - `super.x = 100;`
  - `super.getX();` ⟸ Lo vedremo dopo

Keyword super (2): costruttore

```
public class Libro {
    private String titolo;
    private String autore;
    private String editore;
    private int numeroPagine;
    private int prezzo;
    public Libro(String titolo, String autore) {
        this(titolo);
        setAutore(autore);
    }
    public Libro(String titolo) {
        this.titolo = titolo;
    }
……
}
```

Keyword super (3) : costruttore

```
public class LibroSuJava extends Libro {
    public LibroSuJava (String titolo) {
        super(titolo);
    }

    public LibroSuJava (String titolo, String autore){
        super(titolo, autore);
    }

    //   . . .
}
```

Cosa succede qui se lo rimuovo

NOTA: Invocazione di `super()` deve essere la prima istruzione nel costruttore
Anche invocazione di `this()` deve essere la prima istruzione nel costruttore → all'interno di un costruttore le due invocazioni NON sono ammesse

Keyword super (4): variabili

```
public class Point {
      protected int x,y;
}
public class Point3D extends Point {
      protected int z;

      public void move(int x, int y, int z) {
            super.x = x;
            super.y = y;
            this.z = z;
     }
}
```

Keyword abstract (1)

- In alcuni casi esistono classi che hanno senso soltanto da un punto di vista concettuale e non concretamente (ovvero oggetti di tale classe)
- E' necessario uno strumento che permetta di definire tali classi generiche e astratte
- Si usa la keyword `abstract` nella definizione di una classe
- In tal caso non è più possibile creare oggetti di tale classe
- E' possibile dichiarare `abstract` anche i metodi
  - Non viene fornita l'implementazione del metodo ma viene demandata a sottoclassi
- Ovviamente è sempre possibile **dichiarare** variabili all'interno di una classe astratta

Keyword abstract (2)

› Esempio

```
public abstract class Pittore {
   //. . .
   public abstract void dipingiQuadro();
   //. . .
}
public class PittoreImpressionista extends Pittore {
}
public class PittoreNeoRealista extends Pittore {
}

Pittore p = new Pittore(); //Errore in compilazione
```

Keyword abstract (3)

› Esempio 2

```
public abstract class Strumento { //Classe astratta
    public String nome;
    public String prezzo;
    public abstract void suonaFaDiesis(); //Ogni strumento suona in
                                            modo diverso! Impossibile
                                              definire questo metodo!
    //. . .
}

public class Chitarra extends Strumento { // Classe concreta
    public void suonaFaDiesis() { // Override (riscrittura) del metodo
      //Implementazione del metodo per la chitarra.
    }
   // . . .
}
```

Keyword abstract (4)

```
//Classe di nuovo astratta che estende Strumento
public abstract class StrumentoAFiato extends Strumento {
//metodo suonaFaDiesis ereditato ancora astratto e non riscritto perché troppo generico!
   // . . .
}

// Classe concreta che estende StrumentoAFiato
public class Flauto extends StrumentoAFiato {
    public void suonaFaDiesis() {
      //Implementazione del metodo per il flauto
    }
   //. . .
}
```

Keyword final (1)

› A volte si vuole evitare che si possa formare una sottoclasse

› Tali classi vengono dette *finali*

Si usa la keyword `final` nella definizione di una classe

› Esempio

```
public final class Flauto extends StrumentoAFiato {
}
```

› Classe `String` è dichiarata `final` per ragioni di sicurezza

Keyword final (2)

› E' possibile dichiarare `final` anche un metodo

› Non è possibile effettuare l'override di tale metodo

› Esempio

```
class Employee {
  public final String getName() {return name;}
}

class Manager extends Employee {
  public String getName() { //Errore in compilazione
     return name;
  }
}
```

Invocazione metodi (1)

- Binding: attività di collegare l'invocazione del metodo con il corpo del metodo
- **Binding statico**: è il compilatore che determina il corpo del metodo da invocare
- **Binding dinamico** (o late binding): quale corpo invocare viene determinato a tempo di esecuzione
- Java binding dinamico per i metodi di istanza
- Java binding statico per metodi `static`, `private` e `final`
- Altri linguaggi hanno sia binding dinamico che statico da definire in modo esplicito (C++)

Invocazione metodi (2)

› Invocazione di un metodo di istanza (o statico) è composto di due fasi

1. Compilazione
   - Viene stabilito **se** un metodo può essere invocato
     - Metodi di istanza: dipende dal tipo della variabile e dai modificatori di accesso

```
Point p = …
p.move();
```

     - Metodi statici: dipende dalla classe e dai modificatori di accesso
   - Viene stabilita la segnatura da invocare (può essere multipla in caso di overloading)
   - Se il metodo è `static`, `private` o `final` viene determinato il corpo del metodo da invocare
2. Esecuzione (solo per metodi di istanza)
   - Viene stabilito **quale** corpo (del metodo) deve essere invocato
   - Viene risolto utilizzando il **tipo dell'oggetto** a cui la variabile punta

Esempio: metodi statici (1)

```
class Doubler {
      static int two() { return two(1); }
      private static int two(int i) { return 2*i; }
}
class Test extends Doubler {
      static long two(long j) { return j+1; }
      public static void main(String[] args) {
         System.out.println(two(3));
         System.out.println(Doubler.two(3)); //Errore in compilazione
      }
}
```

Esempio: metodi statici (2)

```
class Doubler {
      static int two() { return two(1); }
      static int two(int i) { return 2*i; }
}
class Test extends Doubler {
      static long two(long j) { return j+1; }
      public static void main(String[] args) {
            System.out.println(two(3));
      }
}
```

Polimorfismo

- Il termine polimorfismo, dal greco polymorfos “avere molte forme” si riferisce in generale alla possibilità data ad una determinata espressione di assumere valori diversi in relazione ai tipi di dato a cui viene applicata
- Esempio (+ è polimorfo): `3 + 4, 1 + "Ciao Mondo"`
- Polimorfismo in Java
  - Overloading metodi
  - Argomenti variabili
  - Overriding metodi
  - Generics (Vedremo dopo)
  - Lambda (Vedremo dopo)

Overloading (1)

- Overloading = sovraccarico (di significato)
- Si parla di overloading quando una classe può contenere due o più metodi (statici e non) con
  - Stesso nome
  - Diversa segnatura (nome metodo + lista parametri)
    - Tipo diverso in almeno un parametro se hanno lo stesso numero
    - Numero di parametri diversi
  - Il tipo di ritorno e le eccezioni **non** sono considerate

Overloading (2)

- Quando un metodo viene invocato, il **compilatore** determina la relativa segnatura del metodo utilizzando il numero attuale degli argomenti e i tipi degli argomenti
- Se il metodo è di istanza quale corpo del metodo da invocare sarà determinato durante l'esecuzione
- Se è statico il binding viene fatto durante la compilazione
- E' possibile effettuare l'overloading anche dei costruttori

Overloading (3)

```
public class Cliente {
    private String nome;
    private String indirizzo;
    private int numeroDiTelefono;

    public Cliente() {
        // costruttore inserito esplicitamente (non di default)
    }

    public Cliente(String nome) {
        this.nome = nome;
    }
```

……

Overloading (4)

```
……
    public Cliente(String nome, String indirizzo) {
        this(nome);
        this.indirizzo = indirizzo;
    }

    public Cliente(String nome, String indirizzo, int numeroDiTelefono) {
        this(nome, indirizzo);
        this.numeroDiTelefono = numeroDiTelefono;
    }
//    . . .
}
```

Overloading (5): tipi primitivi

```
class Aritmetica {
    public int somma(int a, int b) {
        System.out.println("somma(int a, int b)");
        return a + b;
    }
    public float somma(int a, float b) {
        System.out.println("somma(int a, float b)");
        return a + b;
    }
    public float somma(float a, int b) {
        System.out.println("somma(float a, int b)");
        return a + b;
    }
….
```

Overloading (6): tipi primitivi

…

```
    public int somma(int a, int b, int c) {
        System.out.println("somma(int a, int b, int c)");
        return a + b + c;
    }
    public double somma(int a, double b, int c) {
        System.out.println("somma(int a, double b, int c)");
        return a + b + c;
    }
}
```

Overloading (7): tipi primitivi

```
class TestAritmetica {

      public static void main(String[] args) {
            Aritmetica a = new Aritmetica();
            a.somma(3, 4);
            a.somma(3F, 4);
            a.somma(3, 4, 5);
            a.somma(3, 4.0, 5);
      }
}
```

Overloading (8): tipi primitivi

```
class ColoredPoint {
      int x, y;
      byte color;
      void setColor(byte color) { this.color = color; }
}
class Test {
      public static void main(String[] args) {
            ColoredPoint cp = new ColoredPoint();
            byte color = 37;
            cp.setColor(color);
            cp.setColor(37); //Errore in compilazione
      }
}
```

Overloading (9) tipi reference

```
public class Callee {

    public void foo(Object o) {
        System.out.println("foo(Object o)");
    }

    public void foo(String s) {
        System.out.println("foo(\"" + s + "\")");
    }

    public void foo(Integer i) {
        System.out.println("foo(" + i + ")");
    }
```

……

Overloading (10) tipi reference

…

```
        public static void main(String[] args) {

                Callee callee = new Callee();
                Object i = new Integer(12);
                Object s = "foobar";
                Object o = new Object();

                callee.foo(i);
                callee.foo(s);
                callee.foo(o);

        }
}
```

COSA SUCCEDE QUI?

Argomenti variabili (1)

- Introdotti in Java 1.5
- In precedenza veniva utilizzato un array
- Esempio

```
public static String format(String pattern,
                            Object... arguments);
```

- I tre punti alla fine indicano che ultimo argomento può essere passato con un array o una sequenza di argomenti
- Può essere soltanto ultimo argomento di un metodo

Argomenti variabili (2)

› Esempio

```
public class AritmeticaVarArgs {
    public double somma(double... doubles) {
        double risultato = 0.0D;
        for (double tmp : doubles) {
            risultato += tmp;
        }
        return risultato;
    }
}
```

Argomenti variabili (3)

```
public class TestAritmeticaVarArgs {
    public static void main(String args[]) {
        AritmeticaVarArgs ogg = new AritmeticaVarArgs();
        System.out.println(ogg.somma(1, 2, 3));
        System.out.println(ogg.somma());
        System.out.println(ogg.somma(1, 2));
        System.out.println(ogg.somma(1, 2, 3, 5, 6, 8, 2, 43, 4));

    }
}
```

Argomenti variabili (4)

› Esempio senza varargs

```
public class AritmeticaVarArgs {
    public double somma(double[] doubles) {
        double risultato = 0.0D;
        for (double tmp : doubles) {
            risultato += tmp;
        }
        return risultato;
    }
}
AritmeticaVarArgs ogg = new AritmeticaVarArgs();
double[] doubles = {1.2, 2, 3.14, 100.0};
System.out.println(ogg.somma(doubles));
```

Overriding (1)

- Meccanismo che permette di ridefinire nella sottoclasse l'implementazione di metodi definiti nella superclasse
- Regole
  - Il metodo di istanza della sottoclasse C deve avere la stessa segnatura (o firma) del metodo della superclasse
  - Tipo di ritorno deve essere lo stesso oppure può essere un sottotipo del tipo ritornato dal metodo sovrascritto (detto **tipo ritorno covariante**)
  - Modificatore di accesso deve essere almeno lo stesso accesso del metodo di cui viene effettuato l'override ovvero
    - Se è `public` deve rimanere tale
    - Se `protected` può essere `protected` o `public`
    - Se è `package` allora non può essere `private`

Overriding (2)

```
class Employee {
   private String name;
   private double salary;
   private Date hireDay;
   public Employee(String n, double s, int year, int month, int day) {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   public String getName() {
      return name;
   }
……..
```

Overriding (3)

```
.....

    public double getSalary() {
        return salary;
    }
    public Date getHireDay() {
        return hireDay;
    }
    public void raiseSalary(double byPercent) {
        double raise = salary * byPercent / 100;
        salary += raise;
    }

}
```

Overriding (4)

```
class Manager extends Employee {
  private double bonus;
  public Manager(String n, double s, int year, int month, int day) {
      super(n, s, year, month, day);
      bonus = 0;
   }
   public double getSalary() {
      double baseSalary = super.getSalary();
      return baseSalary + bonus;
   }
   public void setBonus(double b) {
      bonus = b;
   }
}
```

Overriding (5)

```
public class ManagerTest {
   public static void main(String[] args) {
      Manager boss = new Manager("Carl Cracker",  80000, 1987, 12, 15);
      boss.setBonus(5000);
      Employee[] staff = new Employee[3];
      staff[0] = boss;
      staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      staff[2] = new Employee("Tommy Tester", 40000, 1990, 3, 15);

      for (int i=0; i < staff.length; i++ ) {
         System.out.println("name=" + staff[i].getName() +
                            ",salary=" + staff[i].getSalary());
      }
   }
}
```

Overriding (6)

› Da notare

```
boss.setBonus(5000)          //OK
staff[ 0 ].setBonus(5000)    //ERRORE
Manager m = staff[ 0 ];      //ERRORE
```

Ancora un esempio

Overriding (7): abstract

› Esempio

```
abstract class Person {
   private String name;

   public Person(String n) {
      name = n;
   }

   public abstract String getDescription();

   public String getName() {
      return name;
   }
}
```

Overriding (8): abstract

```
class Employee extends Person {
   private double salary;
   private Date hireDay;
   public Employee(String n, double s, int year, int month, int day) {
      super(n);
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      // Month value is 0-based. e.g., 0 for January.
      hireDay = calendar.getTime();
   }
   public String getDescription() {
      return "an employee with a salary of " + salary;
   }
…
}
```

GregorianCalandar API
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/GregorianCalendar.html

Overriding (9): abstract

```
class Student extends Person {
   private String major;
   public Student(String n, String m) {
      super(n);
      major = m;
   }

   public String getDescription() {
      return "a student majoring in " + major;
   }

}
```

Overriding (10): abstract

```
public class PersonTest {


  public static void main(String[] args) {
      Person[] people = new Person[2];
      people[0] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
      people[1] = new Student("Maria Morris", "computer science");
      for (int i=0; i < people.length; i++) {
         System.out.println(people[ i ].getName() + ", " +
                            people[ i ].getDescription());
      }
   }
}
```

Overriding (11): metodi statici

- Nel caso di metodi statici non si parla di ovverride di metodi ma di **hide** di metodi
- Inoltre, un metodo statico non può effettuare l'hide di metodi di istanza
- Esempio

```
class Base {
  static String greeting() { return "Goodnight";}
  String name() { return "Richard";}
}
class Sub extends Base {
  static String greeting() { return "Hello";}
  String name() { return "Dick";}
}
class Test {
   public static void main(String[] args) {
            Base b = new Sub();
            System.out.println(b.greeting() + ", " + b.name());
   }
}
```

Overriding (12): tipo ritorno covariante

› Esempio

```
class Punto {
      public Punto elaboraPunto() {
        //…
      }
}

class PuntoTridimensionale extends Punto {
      public PuntoTridimensionale elaboraPunto() {
        //…
      }
}
```

Overriding (13): meno accessibile

› Esempio

```
class Punto {
      public Punto elaboraPunto() {
        //…
      }
}

class PuntoTridimensionale extends Punto {
      //Non puo' essere private, protected o di package
      public Punto elaboraPunto() {
        //…
      }
}
```

Overriding (14)

- Binding dinamico di metodi avviene sempre in Java anche all'interno di costruttori (No C++)
- Metodi utilizzati all'interno di un costruttore è conveniente dichiararli `private`
- Esempio

```
class Point {
      int x, y;
      Point(int x, int y) {
            this.x = x;
            this.y = y;
            init();
      }
      protected void init() {
            System.out.println("Point.init()");
      }
}
```

Overriding (15)

```
class ExtendedPoint extends Point {

  ExtendedPoint(int x, int y ){super(x,y);}
  protected void init() {
      System.out.println("ExtendedPoint.init()");
  }
}

class Test {
   public static void main(String[] args) {
       Point p = new ExtendedPoint(10, 20);
   }
}
```

Annotazione @Override (1)

```
public class Punto {
    private int x, y;

    public void setX(int x) {this.x = x;}
    public int getX() {return x;}
    public void setY(int y) {this.y = y;}
    public int getY() {return y;}

    public double distanzaDallOrigine() {
        int tmp = (x*x) + (y*y);
        return Math.sqrt(tmp);
    }
}
```

Annotazione @Override (2)

- @Override è una annotazione aggiunta a partire da Java 5 ed utilizzata per indicare al compilatore che il metodo su cui è posta sta effettuando un override (sovrascrittura) di un metodo della sua superclasse o, da Java 6, anche di un'interfaccia.

- La domanda nasce spontanea: "cosa cambia se non lo utilizzo?". Proviamo a fare un override sul metodo equals() della superclasse Object:

Annotazione @Override (3)

```
public class Animale {
     private String nome;
     public boolean equals(Animale animale){
          return nome.equals(animale.nome);
     }
}
```

- Come si nota non abbiamo utilizzato @Override... e infatti non stiamo effettuando un override, bensì un **overloading**!
- Sicuramente la maggior parte di voi non avrà fatto caso all'errore. Il metodo equals() ereditato della classe Object accetta come parametro un Object (non un Animale), di conseguenza il compilatore riconoscerà il metodo equals() della classe Animale() come un metodo aggiuntivo e non come uno che sovrascrive l'originale.

Annotazione @Override (4)

› La risposta è quindi facile: @Override nasce per soddisfare esigenze di pratiche di buona programmazione e facilitare il programmatore nel prevenire errori di difficile individuazione (come nel caso dell'esempio).

› Inserendo @Override nella classe Animale otterremmo infatti un errore:

```
public class Animale{
	private String nome;
	@Override
	public boolean equals(Animale animale){
		return nome.equals(animale.nome);
	}
}
```

› Il compilatore non ci permetterà di creare il .class fornendo a video il seguente messaggio:

› The method equals(Animale) of type Animale must override or implement a supertype method

› Invece, se utilizzassimo correttamente @Override potremmo compilare la classe Animale correttamente:

›

Annotazione @Override (5)

```
package override;
public class Animale{
      private String nome;
      @Override
      public boolean equals(Object animale){
      return nome.equals(((Animale) animale).nome);     }
}
```

Annotazione @Override (6)

```
public class PuntoTridimensionale extends Punto {

      //@Override
      public double distanzadallOrigine() {
      }
}
```

› Metodo non è override «d» minuscola

› Non c'è errore in fase di compilazione

› Aggiungendo annotazione compilatore da errore

Annotazione @Override (7)

```
public class PuntoTridimensionale extends Punto {
    private int z;

    public void setZ(int z) {this.z = z;}

    public int getZ() {return z;}

    @Override
    public double distanzaDallOrigine() {
        int tmp = (getX()*getX()) + (getY()*getY())
           + (z*z); // N.B. : x e y non sono ereditate
        return Math.sqrt(tmp);
    }

}
```

Casting & instanceof (1)

- Operatore di cast `()` può essere applicato anche ai tipi reference
- Se si assegna un riferimento a una sottoclasse in una variabile della superclasse no problem (Liskov)
- Esempio

```
class Person{}

class Employee extends Person {}


Employee employee = new Employee(……);

Person p = employee; //OK
```

Casting & instanceof (2)

› Il contrario non è possibile farlo a meno che non si utilizzi il cast

```
Person p = new Employee();

Employee employee = (Employee) p;
```

› Non è possibile effettuare il casting tra fratelli o più in generale tra elementi non facente parte della gerarchia

```
class Student extends Person {}

Student student = new Student(……);

Employee emp = (Employee) student;  //ERRORE in compilazione
```

Casting & instanceof (3)

```
Person p = new Person();
Employee employee = (Employee) p;
```

› Se `p` contiene un oggetto che non è di tipo `Employee` a run-time viene lanciata un'eccezione `ClassCastException`

› Per ovviare a questo problema si utilizza l'operatore `instanceof`

```
Person p = null;
if ("Employee".equalsIgnoreCase(args[0])) {
        p = new Employee();
} else {
        p = new Person();
}
if (p instanceof Employee) {
        Employee e = (Employee) p;
}
```