OBJECT ORIENTATION

Paradigmi della PO

- Incapsulamento
- Ereditarietà
- Polimorfismo

INCAPSULAMENTO ED EREDITARIETÀ

Incapsulamento

- Una classe contiene dati e metodi
- Filosofia
  - Ai dati si accede solo attraverso i metodi
  - I metodi devono prevenire dati non corretti
- Realizzazione
  - Si rendono privati gli attributi
  - Si antepone il modificatore private
  - Dei metodi public garantiscono l'accesso ai dati

Esempio

- Classe non incapsulata

```
public class Data {
    public int giorno;
    public int mese;
    public int anno;
}
```

- Uso degli attributi

```
...
Data unaData = new Data();
unaData.giorno = interfaccia.dammiGiornoInserito();
unaData.mese = interfaccia.dammiMeseInserito();
unaData.anno = interfaccia.dammiAnnoInserito();...
```

Esempio di apsulamento

```
public class Data {
    private int giorno;
    private int mese;
    private int anno;

    public void setGiorno(int g) {
       if (g > 0 && g <= 31) {
           giorno = g;
        }
        else {
           System.out.println("Giorno non valido");
        }
    }

    public int getGiorno() {
        return giorno;
    }
    public void setMese(int m) {
        if (m > 0 && m <= 12) {
            mese = m;
        }
        else {
            System.out.println("Mese non valido");
        }
    }

    public int getMese() {
        return mese;
    }

    public void setAnno(int a) {
        anno = a;
    }

    public int getAnno() {
        return anno;
    }
}
```

```
...
Data unaData = new Data();
unaData.setGiorno(interfaccia.dammiGiornoInserito());
unaData.setMese(interfaccia.dammiMeseInserito());
unaData.setAnno(interfaccia.dammiAnnoInserito());
...
```

Vantaggi dell'Incapsulamento

- Robustezza
  - Controllo l'accesso ai dati
- Indipendenza e Riusabilità
  - Le altre classi conoscono solo l'interfaccia
  - Non devono conoscere i dettagli interni
- Manutenzione
  - Posso riscrivere il corpo dei metodi senza cambiare l'interfaccia

```
public void setGiorno(int g) {
    if (g > 0 && g <= 31 && mese != 2) {
        giorno = g;
    }
    else {
        System.out.println("Giorno non valido");
    }
}
```

Altro esempio

```
public class Dipendente  {
    private String nome;
    private int anni; //intendiamo età in anni
    . . .
    public String getNome() {
        return nome;
    }
    public void setNome(String n) {
        nome = n;
    }
    public int getAnni() {
        return anni;
    }
    public void setAnni(int n) {
        anni = n;
    }
    public int getDifferenzaAnni(Dipendente altro) {
        return (anni - altro.anni);
   }
}
```

Incapsulamento funzionale

- Dichiaro dei metodi con il modificatore private
  - Possono essere visti solo negli altri metodi della classe

```
public class ContoBancario {
    . . .
    public String getContoBancario(int codiceDaTestare)
    {
        return controllaCodice(codiceDaTestare);
    }

    private String controllaCodice(int codiceDaTestare) {
        if (codiceInserito == codice) {
            return contoBancario;
        }
        else  {
            return "codice errato!!!";
        }
    }
}
```

Oggetti ed incapsulamento

- Se dichiaro un membro privato questo non è accessibile da altre classi
  - Ne dagli oggetti delle altre classi
- Due oggetti della stessa classe possono accedere ai membri privati in "modo pubblico"
  - È comunque preferibile non farlo
  - Usiamo comunque i getter

```
public int getDifferenzaAnni(Dipendente altro){
    return (getAnni() - altro.getAnni());
}
```

Ereditarietà

- Partiamo da una classe “generale” e la estendiamo una o più volte particolareggiando le sue caratteristiche

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

La parola chiave extends

```
public class LibroSuJava extends Libro {
    public final String ARGOMENTO_TRATTATO = "Java";
    . . .
}
```

- Tutti i membri pubblici della classe Libro sono ereditati dalla classe LibroSuJava
  - Posso usare numeroPagine, prezzo, …
- Si dice che
  - Libro è la superclasse (o classe padre) di LibroSuJava
  - LibroSuJava è la sottoclasse (o classe figlia) di Libro

Ereditarietà Multipla

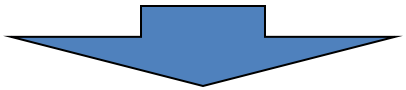
```
public class Idrovolante extends Nave, Aereo {
    . . .
}
```

- NON è permesso in Java estendere due classi!!
  - Potete estendere una sola classe
- Garantiamo la creazione di un albero
  - Questa scelta toglie della flessibilità al programmatore ma favorisce la robustezza
  - In C++ è permessa l'ereditarietà multipla

La Classe Object

- Nella libreria standard di java è definito il concetto di "oggetto generico"
  - java.lang.Object
- Ogni Classe java estende implicitamente la classe Object
  - Scopriremo di aver a disposizione dei metodi ereditati

```
public class Arte {
    . . .
}
```



In compilazione

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

Utilizzare l'ereditarietà

- Quando utilizzare l'ereditarietà?
  - Devo chiedermi se l'oggetto della candidata sottoclasse è un ("is a") oggetto della candidata superclasse
- Vediamo chi?

  Veicoli – Aerei

  Auto – Telaio

  Computer – Laptop

  Persona – Studente

  Quadrato – Rettangolo
- Generalizzazione
  - Se parto da alcune classi e raggruppo le caratteristiche comuni in una classe
- Specializzazione
  - Parto da una classe e derivo alcune sottoclassi

Ereditarietà ed Incapsulamento

- Se incapsulo una classe
  - Le variabili della classe non sono “visibili” della sottoclasse
  - Le variabili private non sono ereditate
- Se la superclasse espone i setter ed i getter
  - La sotto classe eredita questi metodi (se pubblici)
  - Accedo ai membri privati attraverso questi nella sottoclasse
- Quindi
  - La sottoclasse possiede degli attributi a cui può accedere attraverso dei metodi (ereditati)

Modificatore protected

- Un membro dichiarato protected
  - È accessibile dalle classi del package
  - È accessibile dalle sottoclassi
    - Viene ereditato dalle sottoclassi

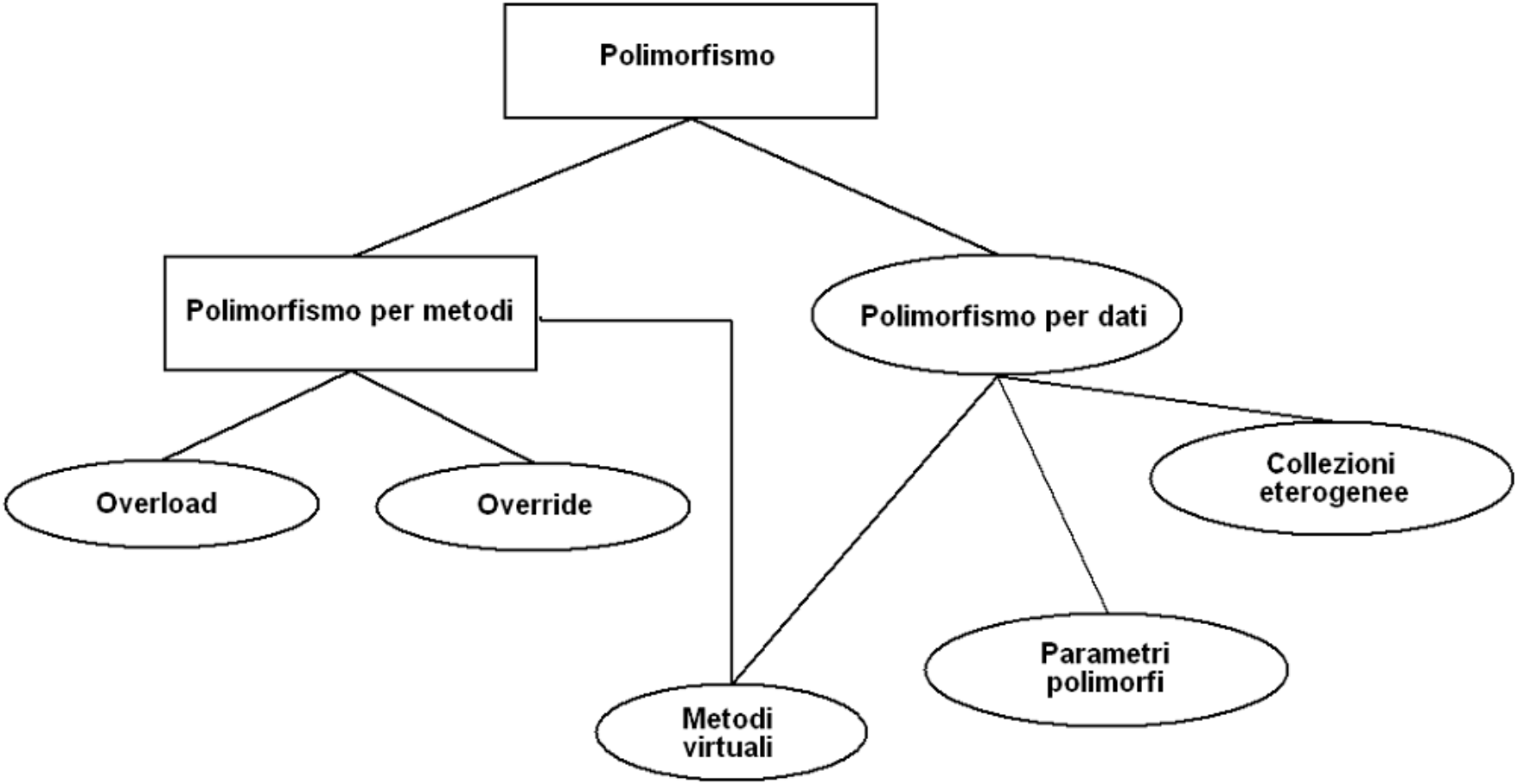
```
package package1;

public class Superclasse {
  protected void metodo(){

  }
}
```

POLIMORFISMO

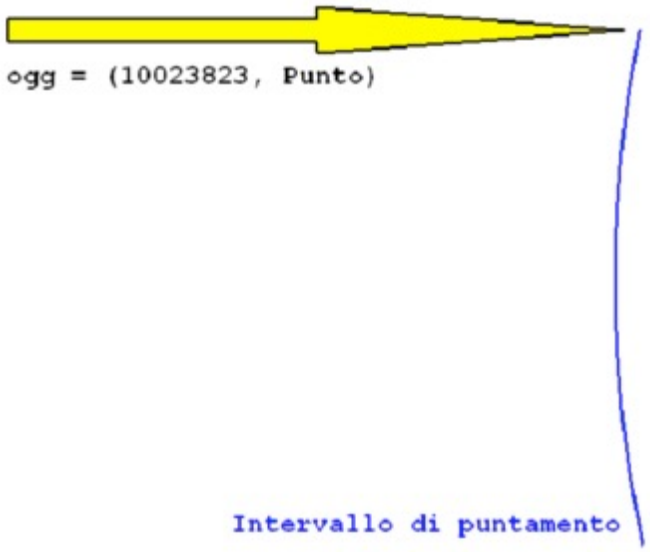
Polimorfismo

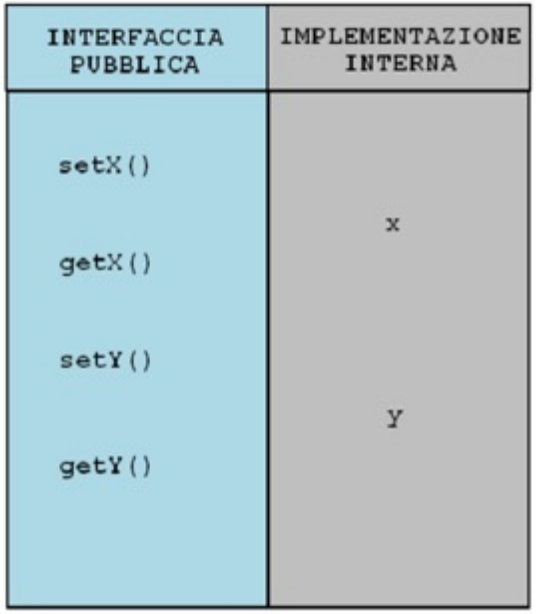


Ancora su i reference

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

```
Punto ogg = new Punto();
```





Overload

- Firma del metodo
  - La coppia identificatore – lista di parametri
  - Es. “public int somma(int a, int b)”
- Overload:
  - In una classe posso dichiarare metodi con lo stesso identificatore ma firma differente
- Tipicamente:
  - Sono metodi con la stessa funzionalità

Esempio di Overload

```
public class Aritmetica
{
    public int somma(int a, int b)
    {
        return a + b;
    }
    public float somma(int a, float b)
    {
        return a + b;
    }
    public float somma(float a, int b)
    {
        return a + b;
    }
    public int somma(int a, int b, int c)
    {
        return a + b + c;
    }
    public double somma(int a, double b, int c)
    {
        return a + b + c;
    }
}
```

Override

- Override:
  - In una sottoclasse posso dichiarare metodi con la stessa firma della superclasse

- Regole:
  - Utilizzare la stessa identica firma
    - altrimenti è un overload e non un override.
  - Il tipo di ritorno del metodo deve coincidere con quello del metodo che si sta riscrivendo.
  - Il metodo ridefinito non deve essere meno accessibile del metodo che ridefinisce

Classe Punto

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

    public void setX()
    {
        this.x = x;
    }
    public int getX()
    {
        return x;
    }
    public void setY()
    {
        this.y = y;
    }
    public int getY()
    {
        return y;
    }
    public double distanzaDallOrigine()
    {
        int tmp = (x*x) + (y*y);
        return Math.sqrt(tmp);
    }
}
```

Esempio di Override

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

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

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

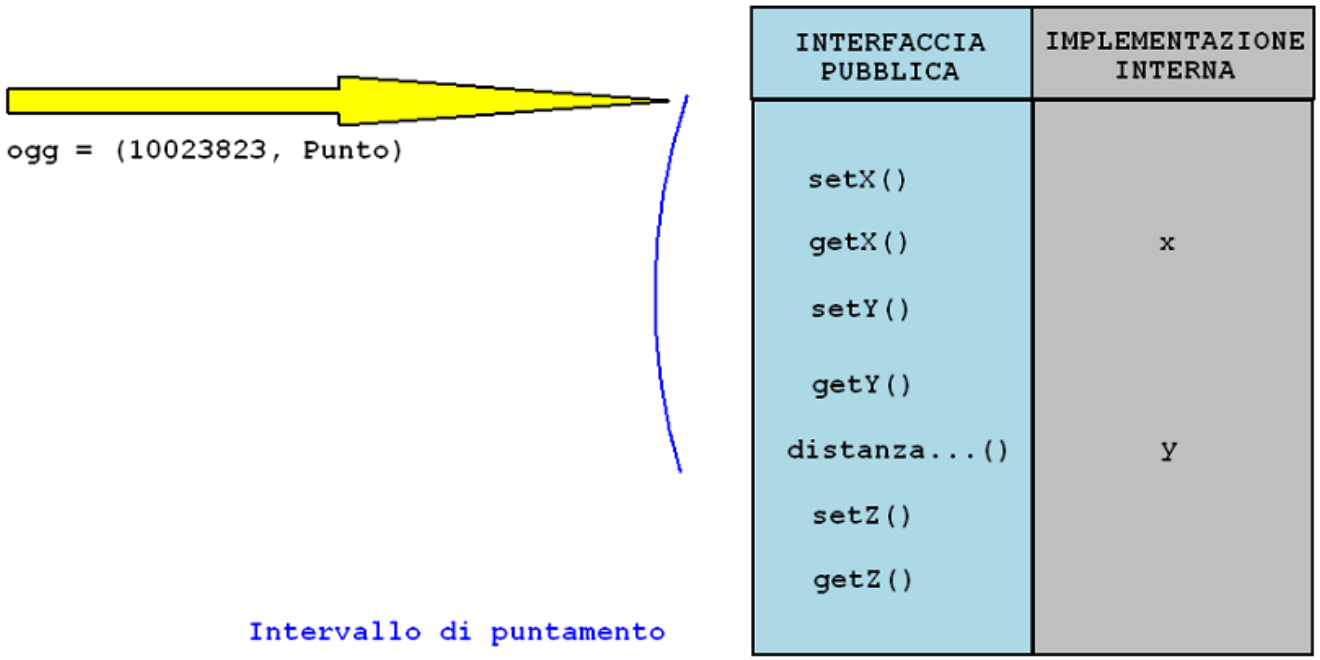
Override di Object

- toString()
  - Restituisce la stringa così formata: NomeClasse@IndirizzoInEsadecimale
- clone()
  - Duplica l'oggetto
- equals()
  - Controlla l'uguaglianza tra due oggetti
- hashCode()
  - Restituisce un int unico per ogni oggetto

Polimorfismo per Dati

- Possiamo assegnare ad un reference della superclasse una istanza della sottoclasse

```
Punto ogg = new PuntoTridimensionale();
```



- Non è lecito:

```
ogg.setZ(5);
```

Parametri polimorfici

- Quando chiamo un metodo il valore del riferimento viene passato al metodo
  - All'indirizzo passato può corrispondere un oggetto della sottoclasse

- Il metodo println accetta parametri della classe Object

```
Punto p1 = new Punto();
System.out.println(p1);
```

- Il corpo del metodo println invoca il metodo toString()
  - Se lo ho riscritto viene chiamato quello di Punto

Collezioni Eterogenee

- Insiemi di oggetti diversi
  - Es Array di Object

```
Object arr[] = new Object[3];
arr[0] = new Punto();     //arr[0], arr[1], arr[2]
arr[1] = "Hello World!"; //sono reference ad Object
arr[2] = new Date();      //che puntano ad oggetti
                          //istanziati da sottoclassi
```

  - oppure

```
Object arr[]={new Punto(),"Hello World!",new Date()};
```

Esempio di Collezione

```
public class Dipendente {
    public String nome;
    public int stipendio;
    public int matricola;
    public String dataDiNascita;
    public String dataDiAssunzione;
}

public class Programmatore extends Dipendente {
    public String linguaggiConosciuti;
    public int anniDiEsperienza;
}

public class Dirigente extends Dipendente {
    public String orarioDiLavoro;
}

public class AgenteDiVendita extends Dipendente {
    public String [] portafoglioClienti;
    public int provvigioni;
}
. . .
```

```
Dipendente [] arr = new Dipendente [180];
arr[0] = new Dirigente();
arr[1] = new Programmatore();
arr[2] = new AgenteDiVendita();
. . .
```

instanceof

- Fornisce la Classe dell'istanza

```
public void pagaDipendente(Dipendente dip) {
    if (dip instanceof Programmatore) {
        dip.stipendio = 1200;
    }
    else if (dip instanceof Dirigente){
        dip.stipendio = 3000;
    }
    else if (dip instanceof AgenteDiVendita) {
        dip.stipendio = 1000;
    }
    . . .
}
```

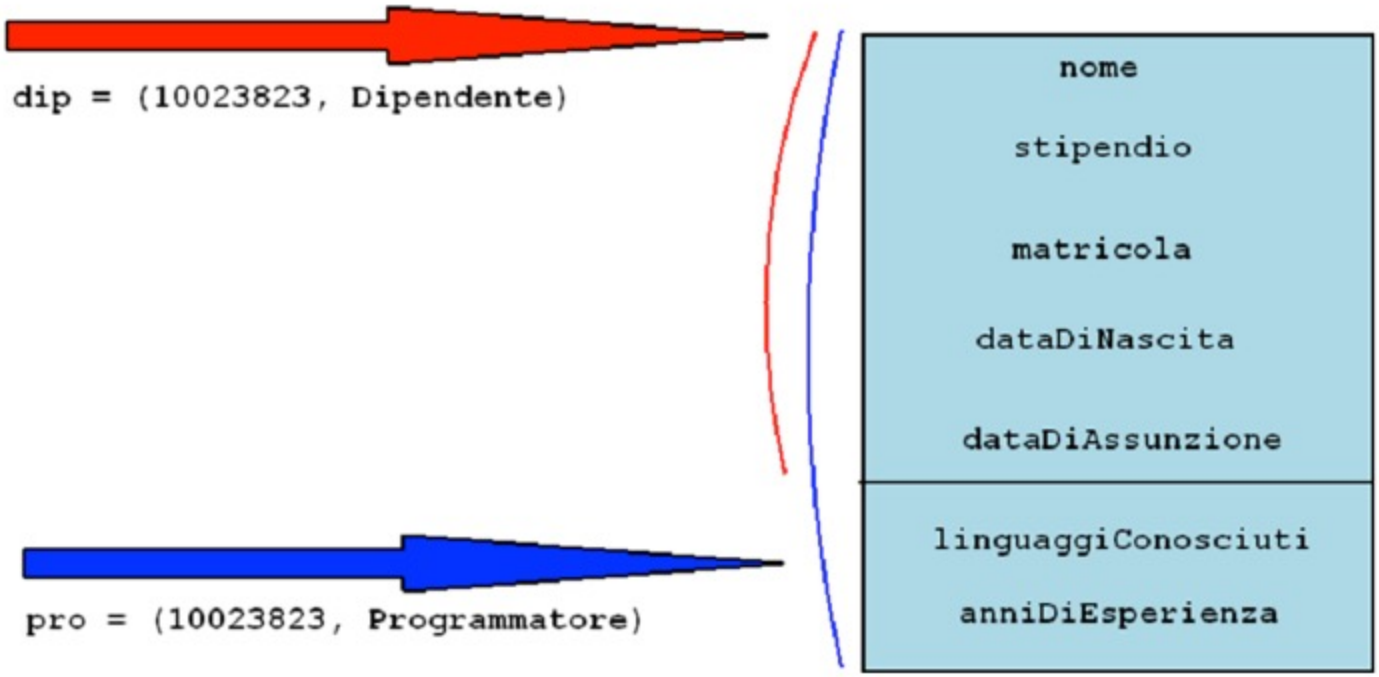
```
. . .
for (Dipendente dipendente : arr) {
    pagaDipendente(dipendente);
    . . .
}
```

Casting di Oggetti

- È lecito questo costrutto??

```
if (dip instanceof Programmatore) {
    Programmatore pro = (Programmatore) dip;
    . . .
```

- Associo un reference della sottoclasse all'oggetto



Metodi Virtuali

- Situazione
  - B è una sottoclasse di A
  - B ridefinisce (override) un metodo *m* di A
  - Ho creato una istanza di B
  - Invoco *m* con un reference di classe A
- Penso di invocare un metodo di A ma in realtà sto invocando il metodo di B
- Esempio

```
. . .
Object obj = new Date();
String s1 = obj.toString();
. . .
```

COSTRUTTORI

Inizializzare oggetti

```
public class Cliente
{
    private String nome;
    private String indirizzo;
    private String numeroDiTelefono;
    public void setNome(String n)
    {
        nome = n;
    }
    public void setIndirizzo(String ind)
    {
        indirizzo = ind;
    }
    public void setNumeroDiTelefono(String num)
    {
        numeroDiTelefono = num;
    }
    public String getNome()
    {
        return nome;
    }
    public String getIndirizzo()
    {
        return indirizzo;
    }
    public String getNumeroDiTelefono()
    {
        return numeroDiTelefono;
    }
}
```

Uso della Classe

```
Cliente cliente1 = new Cliente();
cliente1.setNome("James Gosling");
cliente1.setIndirizzo("Palo Alto, California");
cliente1.setNumeroDiTelefono("0088993344556677L");
```

Costruttori e polimorfismo

```
public class Cliente
{
    private String nome;
    private String indirizzo;
    private String numeroDiTelefono;
    //il seguente è un costruttore
    public Cliente(String n, String ind, String num)
    {
        nome = n;
        indirizzo = ind;
        numeroDiTelefono = num;
    }
    public void setNome(String n)
    {
        . . . . . .
```

Uso della Classe

```
Cliente cliente1 = new Cliente("James Gosling", "Palo
Alto, California",
  "008899334455667");
```

Un codice migliore

```
public Cliente(String nome, String indirizzo, String
numeroDiTelefono)
{
    this.setNome(nome);
    this.setIndirizzo(indirizzo);
    this.setNumeroDiTelefono(numeroDiTelefono);
}
```

Costruttori ed Ereditarietà

```
public class Punto
{
    private int x,y;
    public Punto()
    {
        System.out.println("Costruito punto
bidimensionale");
    }
    . . .
    // inutile riscrivere l'intera classe
}

public class Punto3D extends Punto
{
    private int z;
    public Punto3D()
    {
        System.out.println("Costruito punto " +
          "tridimensionale");
    }
    . . .
    // inutile riscrivere l'ir
}
```

```
new Punto3D(); /* N.B. :L'assegnazione di un reference
                  non è richiesta per istanziare un
                  oggetto */
```

- Output:

  Costruito punto bidimensionale

  Costruito punto tridimensionale

Tutte le proprietà dei Costruttori

1. Hanno lo stesso nome della classe
2. Non hanno tipo di ritorno
3. Sono chiamati automaticamente (e solamente) ogni volta che viene istanziato un oggetto della classe cui appartengono, relativamente a quell'oggetto.
4. Sono presenti in ogni classe.
5. non sono ereditati dalle sottoclassi
6. un qualsiasi costruttore (anche quello di default), come prima istruzione, invoca sempre un costruttore della superclasse.

Il riferimento super

```
public class Persona
{
    private String nome, cognome;
    public String toString()
    {
        return nome + " " + cognome;
    }
    . . .
    //accessor e mutator methods (set e get)
}

public class Cliente extends Persona
{
    private String indirizzo, telefono;
    public String toString()
    {
        return super.toString()+ "\n"+
        indirizzo + "\nTel:" + telefono;
    }
    . . .
    //accessor e mutator methods (set e get)
}
```

- super è un reference implicito all'intersezione tra l'oggetto corrente e la sua superclasse
- Permette di accedere ai membri della superclasse anche se riscritti

Costruttori e super

```
public class Punto
{
    private int x, y;
    public Punto()
    {
        super(); //inserito dal compilatore
    }
    public Punto(int x, int y)
    {
        super(); //inserito dal compilatore
        setX(x); //riuso di codice già scritto
        setY(y);
    }
    . . .
}
public class Punto3D extends Punto
{
     private int z;
     public Punto3D()
     {
         super(); //inserito dal compilatore
     }
     public Punto3D(int x, int y, int z)
     {
         super(x,y); //Chiamata esplicita al
         //costruttore con due parametri interi
         setZ(z);
     }
     . . .
}
```

- La chiamata al costruttore della superclasse è operata con super()
- Posso inserire esplicitamente la chiamata con super(parametri) se devo selezionare il costruttore da chiamare
  - Deve essere la prima istruzione nel costruttore della sottoclasse
  - Se non esiste un costruttore di default sono obbligato a esplicitare la chiamata