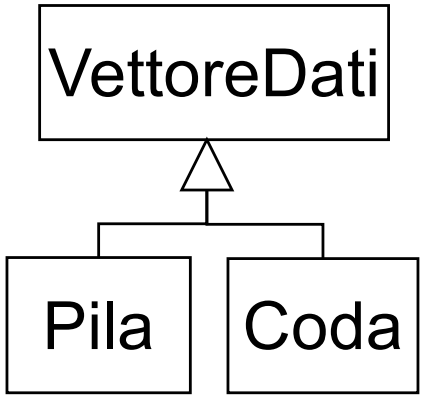
Classi e metodi astratti

- Un **metodo astratto** è un metodo per il quale non è specificata alcuna implementazione
- Una **classe astratta** è tale se contiene almeno un metodo astratto
- Classi e metodi astratti sono definiti tali mediante la parola chiave `abstract`
- Le classi astratte sono molto utili per introdurre astrazioni di alto livello
- Non è possibile creare istanze di una classe astratta: bisogna definire una loro sottoclasse, che ne implementa i metodi astratti

Pila, Coda, e classi astratte

```
abstract public class VettoreDati {
   int size;
   int defaultGrowthSize;
   int marker;
   int contenuto[];
   abstract public int estrai();
   … // implementazione altri metodi
}
public class Pila extends VettoreDati {
   public int estrai() { … }
}
public class Coda extends VettoreDati {
   public int estrai() { … }
}
```



Pila, Coda, VettoreDati e il loro Memory Layout

Polimorfismo

- Polimorfismo (in generale): la capacità di assumere forme diverse
- Polimorfismo (nei linguaggi di programmazione): la capacità di un elemento sintattico di riferirsi a elementi di diverso tipo

Polimorfismo

3 Tipi di Polimorfismo
https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

- Polimorfismo di Sottotipo
  questo che stiamo vedendo, tramite Ereditarieta`

- Polimorfismo ad Hoc
  anche detto overloading, visto in precedenza

- Polimorfismo Parametrico
  che vedremo con i Generici

Polimorfismo

Il tipo di un valore ci dice cosa possiamo fare con quel valore
- per gli oggetti: che metodi / campi ha

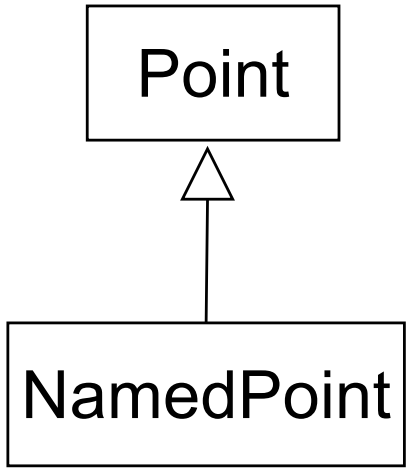
- **Principio di sostituzione di Liskov**:
  se X è un sottotipo di T, allora variabili di tipo T in un programma possono essere sostitituite da variabili di tipo X senza alterare alcuna proprietà desiderabile del programma

Polimorfismo in Java

- Una variabile di tipo riferimento **`T`** può riferirsi ad un qualsiasi oggetto il cui tipo sia **`T`** **o un suo sottotipo**
- Analogamente, un parametro formale di tipo riferimento **`T`** può riferirsi a parametri attuali il cui tipo sia **`T`** **o un suo sottotipo**

Polimorfismo in Java: esempi

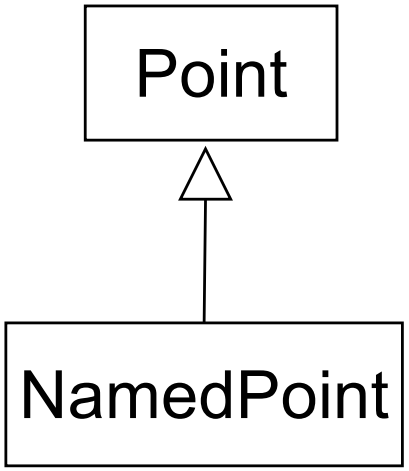
```
Point p = new Point(1,2);
p.move(3,4);
p = new NamedPoint(5,7,"A");
p.move(3,4);
```



- Ovunque c'è un **`Point`** posso mettere un **`NamedPoint`** …

Polimorfismo in Java: esempi

```
public class Line {
   Point p1, p2;
   Line(Point p1, Point p2){…}
}
```




```
Point p1 = new Point(1,2);
NamedPoint p2 = new NamedPoint(5,7,"A");
Line l = new Line(p1, p2);
```

- Ovunque c'è un **`Point`** posso mettere un **`NamedPoint`** …

Polimorfismo in Java: esempi

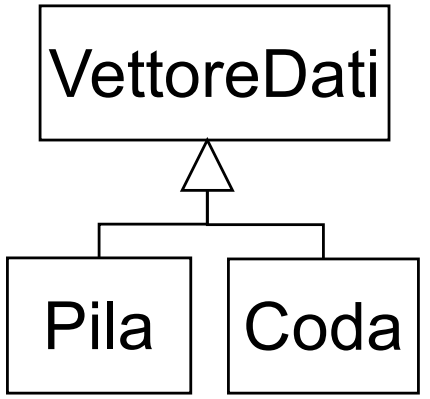
```
class Automobile {...}
class AutomobileElettrica extends Automobile {...}
class Parcheggio {
  private Automobile buf[];
  private int nAuto;
  public Parcheggio(int dim) {buf=new Automobile[dim];}
  public void aggiungi(Automobile a) {buf[nAuto++]=a;}
}
...
AutomobileElettrica b = new AutomobileElettrica();
Automobile a = new Automobile();
Parcheggio p = new Parcheggio(100);
p.aggiungi(a);
p.aggiungi(b);
```

A cosa serve?

- Ereditarietà e polimorfismo sono fondamentali in object-orientation
- Insieme, consentono di:
  - scegliere di volta in volta la "vista" più appropriata per gestire un oggetto (generale vs. specifica)
  - mantenere la definizione di una classe "aperta" a estensioni (specializzazioni) successive, senza dover modificare il codice che la utilizza
- Forte impatto positivo su leggibilità e manutenibilità del codice!

Decisioni al volo…

```
public static void main(String a[]){
    VettoreDati p;
    // leggi k
    if (k==1) p = new Pila();
    else p = new Coda();
    p.inserisci(1);
    p.inserisci(2);
    p.estrai();
}
```



- Il tipo di **p** viene deciso a **runtime**!
- Il legame tra un oggetto e il suo tipo è **dinamico** (*dynamic binding, late binding,*)

Binding dinamico: esempio

```
class Persona {
  private String nome;
  public Persona(String nome) { this.nome = nome;}
  public void chiSei() {
    System.out.print("Ciao, io sono " + nome);
  }
}
class Studente extends Persona {
  public Studente(String nome) { super(nome); }
  public void chiSei() {
    super.chiSei();
    System.out.print(" e sono uno studente");}
}
...
Persona p = new Studente("Giovanni");
p.chiSei();
```

Output:

```
Ciao, io sono Giovanni e sono uno studente
```

Tipo statico e tipo dinamico

- In presenza di polimorfismo si distingue tra il **tipo statico** (dichiarato a *compilation time*) ed il **tipo dinamico** (assunto a *runtime*) di una variabile o parametro formale
  - La regola di sostituzione enunciata obbliga il tipo dinamico ad essere un sottotipo del tipo statico
  - In Java, in un'invocazione **`x.f(x1,...,xn)`**, l'implementazione scelta per il metodo **`f`** dipende anche dal tipo dinamico di **`x`** e non solo dal suo tipo statico

Regola per il binding dinamico

- Si assuma
  ```
  C o = …;
  o.m(…);
  ```
  il metodo scelto dipende dal tipo dinamico di **o**, e viene deciso (a *runtime*) con questa logica:

Il tipo statico determina quali metodi possono essere invocati; il tipo dinamico determina quale (ri)definizione eseguire

1. Si cerca all'interno della classe **C** (tipo statico di **o**) il metodo **m** con la firma più simile all'invocazione
   - Le firme da considerare sono quelle fissate a *compile time*, guardando cioè il tipo statico dei parametri attuali
2. Si guarda al tipo dinamico **D** di **o**; se è un sottotipo di **C**, si deve verificare se ridefinisce (*override*) **m**. Se sì, si usa l'implementazione di **D**, altrimenti quella di **C**

Esercizio: qual è l'output?

```
public class A {
  public void stampa(A p) {
    System.out.println("A");
  }
}
public class B extends A {
  public void stampa(B p) {
    System.out.println("B");
  }
  public void stampa(A p) {
    System.out.println("A-B");
  }
}
public class C extends A {
  public void stampa(C p) {
    System.out.println("C");
  }
  public void stampa(A p) {
    System.out.println("A-C");
  }
}
```

```
(nel main)

A a1, a2;
B b1;
C c1;
a1 = new B();
b1 = new B();
c1 = new C();
a2 = new C();
b1.stampa(b1);
a1.stampa(b1);
b1.stampa(c1);
c1.stampa(c1);
c1.stampa(a1);
a2.stampa(c1);
```

Esercizio: qual è l'output?

tipo dinamico = tipo statico:
non c'è scelta

tipo dinamico ≠ tipo statico: uso la ridefinizione del metodo di **A** in **B**

tipo dinamico = tipo statico:
i parametri hanno tipi diversi ma è irrilevante, conta solo il tipo statico di **c1**

tipo dinamico = tipo statico:
non c'è scelta

tipo dinamico = tipo statico:
i parametri hanno tipi diversi ma è irrilevante, conta solo il tipo statico di **a1**

tipo dinamico ≠ tipo statico: uso la ridefinizione del metodo di **A** in **C**

```
(nel main)

A a1, a2;
B b1;
C c1;
a1 = new B();
b1 = new B();
c1 = new C();
a2 = new C();
b1.stampa(b1); // B
a1.stampa(b1); // A-B
b1.stampa(c1); // A-B
c1.stampa(c1); // C
c1.stampa(a1); // A-C
a2.stampa(c1); // A-C
```

Esercizio: qual è l'output?

```
public class A {
  public void stampa(A p) {
    System.out.println("A");
  }
}
public class B extends A {
  public void stampa(B p) {
    System.out.println("B");
  }
  public void stampa(A p) {
    System.out.println("A-B");
  }
}
public class C extends B {
  public void stampa(C p) {
    System.out.println("C");
  }
  public void stampa(A p) {
    System.out.println("A-C");
  }
}
```

```
(nel main)

A a1, a2;
B b1;
C c1;
a1 = new B();
b1 = new B();
c1 = new C();
a2 = new C();
b1.stampa(b1);
a1.stampa(b1);
b1.stampa(c1);
c1.stampa(c1);
c1.stampa(a1);
a2.stampa(c1);
```

Esercizio: qual è l'output?

tipo dinamico = tipo statico:
non c'è scelta

tipo dinamico ≠ tipo statico: uso la ridefinizione del metodo di **A** in **B**

tipo dinamico = tipo statico:
i parametri hanno tipi diversi ma è irrilevante, conta solo il tipo statico di **c1**; il metodo più specifico è quello che accetta **B**

tipo dinamico = tipo statico:
non c'è scelta

tipo dinamico = tipo statico:
i parametri hanno tipi diversi ma è irrilevante, conta solo il tipo statico di **a1**;
stampa(B) in **B** non può essere usato

tipo dinamico ≠ tipo statico: uso la ridefinizione del metodo di **A** in **C**

```
(nel main)

A a1, a2;
B b1;
C c1;
a1 = new B();
b1 = new B();
c1 = new C();
a2 = new C();
b1.stampa(b1); // B
a1.stampa(b1); // A-B
b1.stampa(c1); // B
c1.stampa(c1); // C
c1.stampa(a1); // A-C
a2.stampa(c1); // A-C
```

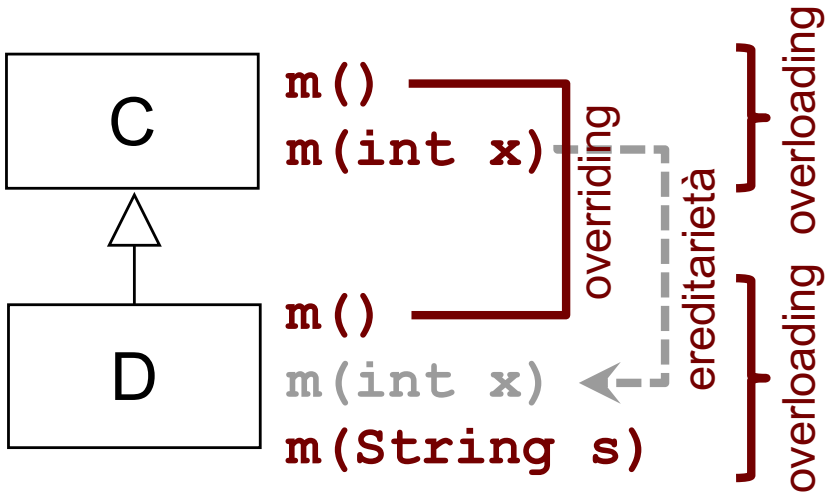
Esercizio: qual è l'output?

```
class A {
  private int val;
  public A(int v) { val = v; }
  public int valore() { return val; }
  public int somma(A o) { return valore() + o.valore(); }
}
class B extends A {
  B(int v) { super(v); }
  public int somma(A o) {
    return valore() +
      o.valore() + 2;
  }
  public int somma(B o) {
    return valore() +
      o.valore() + 1;
  }
}
```

```
public class Prova {
  public static void main(String[] args) {
    A a1, a2;
    B b;
    a1 = new A(4);
    a2 = new B(5);
    b = new B(6);
    System.out.println(a1.somma(a2));
    System.out.println(a2.somma(b));
    System.out.println(b.somma(a1));
    System.out.println(b.somma(b));}
}
```

Quali metodi sono visibili?

- Ereditarietà, overriding, overloading e polimorfismo determinano i metodi visibili a partire da un dato riferimento
- Con la gerarchia a destra e le istruzioni qui sotto …




```
C c = new C();   c.m();   c.m(5);   c.m("hello");
C cd = new D();  cd.m();  cd.m(5);  cd.m("hello");
D d = new D();   d.m();   d.m(5);   d.m("hello");
```

overridden ereditato

Static e dynamic binding

- Il C++ offre al programmatore complessi meccanismi per decidere se usare **dynamic binding** (tipo deciso a runtime) o **static binding** (tipo deciso a compile time)
- In Java le decisioni sono sempre a runtime
  - … salvo quando sia possibile decidere *automaticamente* a compile time, e cioè per:
    - costruttori
    - metodi **`static`**, **`private`**, e **`final`**