Nota: riuso di costruttori in una classe – this()

È possibile **riusare** costruttori gìa definiti anche nell'ambito di una stessa classe. Ciò è reso possibile dal costrutto `this()` che, analogamente a `super()`, deve comparire come *prima istruzione* nel costruttore della classe.

```
public class Persona {
  private String nome;
  private String residenza;

  public Persona(String n, String r) {
    nome = n;
    residenza = r;
  }

  public Persona(String n) {
    this(n, null);
  }
```

```
  public Persona() {
    this("Mario Rossi");
  }
  ...
}
```

NOTA: l'uso contemporaneo di `this()` e `super()` in uno stesso costruttore non è possibile (entrambi dovrebbero comparire come prima istruzione).

Gerarchie di classi

Una classe derivata può a sua volta fungere da classe base per una **successiva derivazione**.

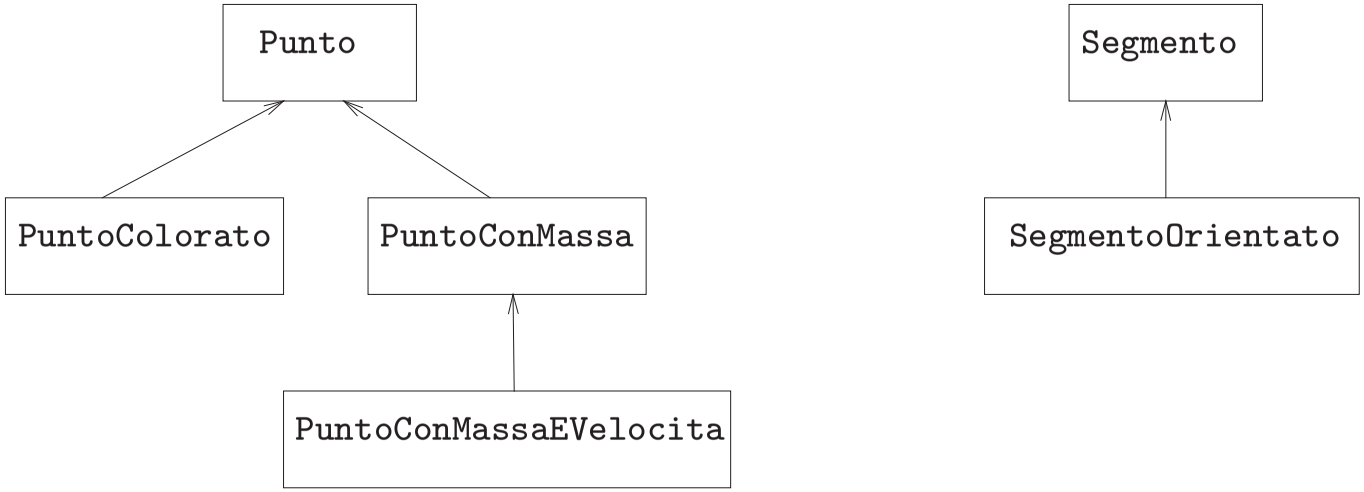
Ogni classe può avere **un numero qualsiasi** di classi derivate.

```
class B { ...
class D extends B { ...
class D2 extends B { ...
```

Una classe derivata può avere **una sola classe base**, (in Java non esiste la cosiddetta *ereditarietà multipla*).

Java supporta una sorta di ereditarietà multipla attraverso le *interfacce* – *maggiori dettagli in seguito*.

Gerarchie di classi: esempio



Esercizio 4: gerarchie di classi

Ignorando le funzioni e i livelli d'accesso dei campi, realizzare in Java la gerarchia di classi della figura precedente.

Significato dell'assegnazione

Abbiamo visto che, in base al principio 3 dell'ereditarietà, la seguente istruzione è lecita:

```
class B { /*...*/ }                 // CLASSE BASE
class D extends B { /*...*/ } // CLASSE DERIVATA
// ...
D d = new D();
B b = d;                            // OK: D al posto di B
```

- **Non viene creato** un nuovo oggetto.

- Esiste **un solo oggetto**, di classe `D`, che viene denotato:

  - **sia** con un riferimento `d` di classe `D`,

  - **sia** con un riferimento `b` di classe `B`.

Casting

Non è possibile accedere ai campi della classe `D` attraverso `b`. Per farlo dobbiamo prima fare un **casting**.

```
// File unita1/Esempio7.java
class B { }
class D extends B { int x_d; }

public class Esempio7 {
  public static void main(String[] args) {
    D d = new D();
    d.x_d = 10;
    B b = d;
    System.out.println(((D)b).x_d);  // CASTING
  }
}
```

Casting (cont.)

Il casting fra classi che sono nello stesso cammino in una gerarchia di derivazione è sempre *sintatticamente* corretto, ma **è responsabilità del programmatore** garantire che lo sia anche *semanticamente*.

```
// File unita1/Esempio8.java
class B { }
class D extends B { int x_d; }

public class Esempio8 {
  public static void main(String[] args) {
    B b = new B();
    D d = (D)b;
    System.out.println(d.x_d); // ERRORE SEMANTICO: IL CAMPO x_d NON ESISTE
    // java.lang.ClassCastException: B
    //  at Esempio19.main(Compiled Code)
  }
}
```

Esercizio 5: casting

Con riferimento al seguente frammento di codice, scrivere una funzione `main()` che contiene un uso semanticamente corretto ed un uso semanticamente scorretto della funzione `f()`.

```
class B { }
class D extends B { int x_d; }
// ...
  static void f(B bb) {
    ((D)bb).x_d = 2000;
    System.out.println(((D)bb).x_d);
  }
```

Esercizio 6: costruttori e gerarchie di classi

Facendo riferimento alla gerarchia di classi vista in precedenza, riprogettare le classi `Punto`, `PuntoColorato`, `PuntoConMassa` e `PuntoConMassaEVelocita`, tenendo conto del livello d'accesso dei campi, con i seguenti costruttori:

`Punto`: con tre argomenti (le tre coordinate) e zero argomenti (nell'origine);

`PuntoColorato`: con quattro argomenti (coordinate e colore);

`PuntoConMassa`: con un argomento (massa); deve porre il punto nell'origine degli assi;

`PuntoConMassaEVelocita`: con due argomenti (massa e velocità); deve porre il punto nell'origine degli assi.

Derivazione e overloading

È possibile fare **overloading** di funzioni ereditate dalla classe base esattamente come lo si può fare per le altre funzioni.

```
public class B {
  public void f(int i) { ... }
}

public class D extends B {
  public void f(String s) { ... } // OVERLOADING DI f()
}
```

La funzione `B.f(int)` ereditata da `B` **è ancora accessibile** in `D`.

```
D d = new D();
d.f(1);        // invoca f(int) ereditata da B
d.f("prova"); // invoca f(String) definita in D
```

Overriding di funzioni

Nella classe derivata è possibile anche fare **overriding** (dall'inglese, *ridefinizione*, *sovrascrittura*) delle funzioni della classe base.

Fare overriding di una funzione `f()` della classe base `B` vuol dire definire nella classe derivata `D` una funzione con lo stesso nome, lo stesso numero e tipo di parametri della funzione `f()` definita in `B`. Si noti che **il tipo di ritorno delle due funzioni deve essere identico**.

```
public class B {
  public void f(int i) { ... }
}

public class D extends B {
  public void f(String s) { ... } // OVERLOADING DI f()
  public void f(int n) { ... }    // OVERRIDING DI f()
}
```

Overriding di funzioni: esempio

```
// File unita1/Esempio11.java
class B {
  public void f(int i) { System.out.println(i*i); } }
class D extends B {
  public void f(String s) { // OVERLOADING DI f()
    System.out.println(s); }
  public void f(int n) {    // OVERRIDING DI f()
    System.out.println(n*n*n); }
}
public class Esempio11 {
  public static void main(String[] args) {
    B b = new B();
    b.f(5);          // stampa 25
    D d = new D();
    d.f("ciao");     // stampa ciao
    d.f(10);         // stampa 1000     } }
```

Overriding e riscrittura

Su oggetti di tipo `D` **non è più possibile invocare** `B.f(int)`.

È ancora possibile invocare `B.f(int)` **solo dall'interno della classe** `D` attraverso un campo predefinito `super` (analogo a `this`).

Riassunto overloading e overriding

	OVERLOADING	OVERRIDING
nome della funzione	uguale	uguale
tipo restituito	qualunque	uguale
numero e/o tipo argomenti	diverso	uguale
relazione con la funzione della classe base	coesiste con la funzione della classe base	cancella la funzione della classe base

Esercizio 7: overriding e compatibilità

```
// File unita1/Esercizio7.java
class B {
  protected int c;
  void stampa() { System.out.println("c: " + c); }
}

class D extends B {
  protected int e;
  void stampa() {
    super.stampa();
    System.out.println("e: " + e);
  }
}

public class Esercizio7 {
  public static void main(String[] args) {
    B b  = new B();    b.stampa();
    B b2 = new D();    b2.stampa();
    D d  = new D();    d.stampa();
    D d2 = new B();    d2.stampa();
  }
}
```

Il programma contiene errori rilevabili dal compilatore?
Una volta eliminati tali errori, cosa stampa il programma?

Overriding di funzioni: late binding

Invocando `f(int)` su un oggetto di `D` viene invocata **sempre** `D.f(int)`, **indipendentemente** dal fatto che esso sia denotato attraverso un riferimento `d` di tipo `D` o un riferimento `b` di tipo `B`.

```
public class B {
  public void f(int i) { ... }
}

public class D extends B {
  public void f(int n) { ... }
}
// ...
D d = new D();
d.f(1);  // invoca D.f(int)
B b = d; // OK: classe derivata usata al posto di classe base
b.f(1);  // invoca di nuovo D.f(int)
```

Late binding (cont.)

Secondo il meccanismo del **late binding** la scelta di quale funzione invocare non viene effettuata durante la compilazione del programma, **ma durante l'esecuzione**.

```
public static void h (B b) { b.f(1); }
// ...
B bb = new B();
D d = new D();
h(d);    // INVOCA D.f(int)
h(bb);   // INVOCA B.f(int)
```

Il gestore run-time riconosce **automaticamente** il tipo dell'oggetto di invocazione:

`h(d)`: `d` denota un oggetto della classe `D` – viene invocata la funzione `f(int)` definita in `D`;

`h(bb)`: `bb` denota un oggettp della classe `B` – viene invocata la funzione `f(int)` definita in `B`.

Esercizio 8: cosa fa questo programma?

```
// File unita1/Esercizio8.java
class B {
  protected int id;
  public B(int i) { id = i; }
  public boolean get() { return id < 0; }
}

class D extends B {
  protected char ch;
  public D(int i, char c) {
    super(i);
    ch = c;
  }
  public boolean get() { return ch != 'a'; }
}

public class Esercizio8 {
  public static void main(String[] args) {
    D d = new D(1,'b');
    B b = d;
    System.out.println(b.get());
    System.out.println(d.get());
  }
}
```

Overriding e livello d'accesso

Nel fare overriding di una funzione della classe base è possibile cambiare il livello di accesso alla funzione, **ma solo rendendolo meno restrittivo**.

```
// File unita1/Esempio13.java
class B {
  protected void f(int i) { System.out.println(i*i); }
  protected void g(int i) { System.out.println(i*i*i); }
}

class D extends B {
  public void f(int n) { System.out.println(n*n*n*n); }
  //  private void g(int n) {
  //               ^
  // Methods can't be overridden to be more private.
  // Method void g(int) is protected in class B.
  //    System.out.println(n*n*n*n*n); }
}
```

Impedire l'overriding: final

Qualora si voglia **bloccare l'overriding** di una funzione la si deve dichiarare `final`.

Anche una classe può essere dichiarata `final`, impedendo di derivare classi dalla stessa e rendendo implicitamente `final` tutte le funzioni della stessa.

```
// File unita1/Esempio14.java
class B {
  public final void f(int i) { System.out.println(i*i); }
}
class D extends B {
// public void f(int n) { System.out.println(n*n*n*n); }
// Final methods can't be overriden. Method void f(int) is final in class B.
}
final class BB {}
// class DD extends BB {}
// Can't subclass final classes: class BB
```

Sovrascrittura dei campi dati

Se definiamo nella classe derivata una variabile con lo stesso nome e di diverso tipo di una variabile della classe base, allora:

- la variabile della classe base **esiste ancora** nella classe derivata, ma non può essere acceduta utilizzandone semplicemente il nome;
- si dice che la variabile della classe derivata **nasconde** la variabile della classe base;
- per accedere alla variabile della classe base è necessario utilizzare **un riferimento ad oggetto della classe base**.

Sovrascrittura dei campi dati: esempio

```
// File unita1/Esempio15.java
class B { int i; }
class D extends B {
  char i;
  void stampa() {
    System.out.println(i); System.out.println(super.i);
  }
}
public class Esempio15 {
  public static void main(String[] args) {
    D d = new D();
    d.i = 'f';
    ((B)d).i = 9;
    d.stampa();
  }
}
```

Classi astratte

Le classi astratte sono classi particolari, nelle quali una o più funzioni possono essere solo **dichiarate** (cioè si descrive la segnatura), ma non **definite** (cioè non si specificano le istruzioni).

Esempio:

Ha certamente senso associare alla classe `Persona` una funzione che calcola la sua aliquota fiscale, ma il vero e proprio calcolo per una istanza della classe `Persona` **dipende** dalla sottoclasse di `Persona` (ad esempio: straniero, pensionato, studente, impiegato, ecc.) a cui l'istanza appartiene.

Vogliamo poter definire la classe `Persona`, magari con un insieme di campi e funzioni normali, anche se non possiamo scrivere il codice della funzione `Aliquota()`.

Classi astratte: esempio

La soluzione è definire la classe `Persona` come **classe astratta**, con la funzione `Aliquota()` astratta, e definire poi le sottoclassi di `Persona` come classi non astratte, in cui definire la funzione `Aliquota()` con il relativo codice:

```
abstract class Persona {
  abstract public int Aliquota(); // Questa e' una DICHIARAZIONE
                                  // (senza codice)
  private int eta;
  public int Eta() { return eta; }
}

class Studente extends Persona {
  public int Aliquota() { ... } // Questa e' una DEFINIZIONE
}

public class Professore extends Persona {
  public int Aliquota() { ... } // Questa e' una DEFINIZIONE
}
```

Quando una classe va definita astratta

Una classe `A` si definirà come astratta quando:

- non ha senso pensare a oggetti che siano istanze di `A` **senza essere istanze anche di una sottoclasse (eventualmente indiretta) di** `A`;

- esiste una funzione che ha senso associare ad essa, ma il cui codice non può essere specificato a livello di `A`, mentre può essere specificato a livello delle sottoclassi di `A`; si dice che tale funzione è astratta in `A`.

Anche se spesso si dice che una classe astratta `A` non ha istanze, ciò non è propriamente corretto: la classe astratta `A` non ha istanze dirette, ma ha come istanze tutti gli oggetti che sono istanze di sottoclassi di `A` non astratte.

Si noti che la classe astratta può avere funzioni non astratte e campi dati.

Uso di classi astratte

Se `A` è una classe astratta, allora:

- **Non possiamo** creare direttamente oggetti che sono istanze di `A`. Non esistono istanze dirette di `A`: gli oggetti che sono istanze di `A` lo sono indirettamente.

- **Possiamo:**

  – definire variabili o campi di altri classi (ovvero, riferimenti) di tipo `A` (durante l'esecuzione, conterranno indirizzi di oggetti di classi non astratte che sono sottoclassi di `A`),

  – usare normalmente i riferimenti (tranne che per creare nuovi oggetti), ad esempio: definire funzioni che prendono come argomento un riferimento di tipo `A`, restituire riferimenti di tipo `A`, ecc.

Vantaggi delle classi astratte

Se non ci fosse la possibilità di definire la classe `Persona` come classe astratta, dovremmo prevedere un meccanismo (per esempio un campo di tipo `String`) per distinguere istanze di `Studente` da istanze di `Professore`, e definire nella classe `Persona` la funzione `Aliquota()` così

```
class Persona {
  private String tipo;
  public int Aliquota() {
      if (tipo.equals("Studente"))
           // codice per il calcolo dell'aliquota per Studente
      else if (tipo.equals("Professore"))
           // codice per il calcolo dell'aliquota per Professore
   }
  // ....
}
```

All'aggiunta di una sottoclasse di `Persona`, si dovrebbe riscrivere e ricompilare la classe `Persona` stessa. **Riuso ed estendibilità sarebbero compromessi!**

Vantaggi delle classi astratte (cont.)

Supponiamo di dovere scrivere una funzione esterna alla classe `Persona` che, data una persona (sia essa uno studente, un professore, o altro), verifica se è tartassata dal fisco (cioè se la sua aliquota è maggiore del 50 per cento).

Se ho definito `Persona` come classe astratta posso semplicemente fare così:

```
// ....
static public boolean Tartassata(Persona p) {
     return p.Aliquota() > 50;
}
```

È importante notare che, quando la funzione verrà attivata, verrà passato come parametro attuale un riferimento ad un oggetto di una classe non astratta, in cui quindi la funzione `Aliquota()` è definita con il codice. Il late binding farà il suo gioco, e chiamerà la **funzione giusta**, cioè la funzione definita nella classe più specifica non astratta di cui l'oggetto passato è istanza.

Esercizio 9

Si definisca una classe per rappresentare soggetti fiscali. Ogni soggetto fiscale ha un nome, e di ogni soggetto fiscale deve essere possibile calcolare l'anzianità, tenendo però presente che l'anzianità si calcola in modo diverso a seconda della categoria (impiegato, pensionato o straniero) a cui appartiene il soggetto fiscale. In particolare:

- se il soggetto è un impiegato, allora l'anzianità si calcola sottraendo all'anno corrente l'anno di assunzione;
- se il soggetto è un pensionato, allora l'anzianità si calcola sottraendo all'anno corrente l'anno di pensionamento;
- se il soggetto è uno straniero, allora l'anzianità si calcola sottraendo all'anno corrente l'anno di ingresso nel paese.

Interfacce

Una interfaccia è un'astrazione per un insieme di funzioni pubbliche delle quali si definisce solo la segnatura, e non le istruzioni. Un'interfaccia viene poi implementata da una o più classi (anche astratte). Una classe che implementa un'interfaccia deve definire o dichiarare tutte le funzioni della interfaccia.

Dal punto di vista sintattico, un'interfaccia è costituita da un insieme di dichiarazioni di funzioni pubbliche (**no campi dati**, a meno che non sia `final`), la cui definizione **è necessariamente lasciata alle classi che la implementano**. Possiamo quindi pensare ad una interfaccia come ad una dichiarazione di un tipo di dato (inteso come un insieme di operatori) di cui non vogliamo specificare l'implementazione, ma che comunque può essere utilizzato da moduli software, indipendentemente appunto dall'implementazione.

Esempio: interfaccia `I` con una sola funzione `g()`

```
public interface  I {
  void g(); // implicitamente public;  e' una DICHIARAZIONE: notare ';'
}
```

Cosa si fa con un'interfaccia

Se `I` è un'interfaccia, allora **possiamo**:

- definire una o più classi che **implementano** `I`, cioè che definiscono tutte le funzioni dichiarate in `I`

- definire variabili e campi di tipo `I` (durante l'esecuzione, conterranno indirizzi di oggetti di classi che implementano `I`),

- usare i riferimenti di tipo `I`, sapendo che in esecuzione essi conterranno indirizzi di oggetti (quindi possiamo definire funzioni che prendono come argomento un riferimento di tipo `I`, restituire riferimenti di tipo `I`, ecc.);

mentre **non possiamo**:

- creare oggetti di tipo `I`, cioè non possiamo eseguire `new I()`, perchè non esistono oggetti di tipo `I`, ma esistono solo riferimenti di tipo `I`.

Utilità delle interfacce

Le funzioni di un'interfaccia costituiscono un modulo software `S` che:

- può essere utilizzato da un modulo esterno T (ad esempio una funzione `t()` che si aspetta come parametro un riferimento di tipo `S`), **indipendentemente** da come le funzioni di `S` sono implementate; in altre parole, non è necessario avere deciso l'implementazione delle funzioni di `S` per progettare e scrivere altri moduli che usano `S`;

- può essere implementato in modi alternativi e diversi tra loro (nel senso che più classi possono implementare le funzioni di `S`, anche in modo molto diverso tra loro);

- ovviamente, però, al momento di attivare un modulo `t()` che ha un argomento tipo `S`, occorre passare a `t()`, in corrispondenza di `S`, un oggetto di una classe che implementa `S`.

Tutto ciò aumenta la possibilità di **riuso**.

Esempio di interfaccia e di funzione cliente

Vogliamo definire una interfaccia `Confrontabile` che offra una operazione che verifica se un oggetto è **maggiore** di un altro, ed una operazione che verifica se un oggetto è **paritetico** ad un altro. Si noti che **nulla si dice** rispetto al criterio che stabilisce se un oggetto è maggiore di o paritetico ad un altro.

Si vuole scrivere poi una funzione che, dati tre riferimenti a `Confrontabile`, restituisca il maggiore tra i tre (o più precisamente un *massimale*, ovvero uno qualunque che non abbia tra gli altri due uno maggiore di esso).

Notiamo che, denotando con gli operatori binari infissi '$>$' e '$=$' le relazioni "maggiore" e "paritetico" (rispettivamente), x_1 è massimale in $\{x_1, x_2, x_3\}$ se e solo se:

$$(x_1 > x_2 \ \vee \ x_1 = x_2) \quad \wedge \quad (x_1 > x_3 \ \vee \ x_1 = x_3)$$

Esempio di interfaccia e di f. cliente (cont.)

```
// File unita1/Esempio16.java
interface  Confrontabile {
  boolean Maggiore(Confrontabile x);
  boolean Paritetico(Confrontabile x);
}

class Utilita {
  static public Confrontabile MaggioreTraTre(Confrontabile x1,
                                             Confrontabile x2,
                                             Confrontabile x3) {
    if ((x1.Maggiore(x2) || x1.Paritetico(x2)) &&
        (x1.Maggiore(x3) || x1.Paritetico(x3)))
      return x1;
    else if ((x2.Maggiore(x1) || x2.Paritetico(x1)) &&
             (x2.Maggiore(x3) || x1.Paritetico(x3)))
      return x2;
    else return x3;  }  }
```