Programmazione Java

Gianluca CANNATA

Febbraio 2016

Astrazioni e UML

L'**astrazione** è un *procedimento concettuale* per mezzo del quale si definisce un concetto più generale a partire da concetti più elementari, rimuovendo tutti gli aspetti di dettaglio e particolari, e mettendo in evidenza gli aspetti comuni e generali.

Astrazioni strutturali

Le **astrazioni strutturali** modellano oggetti e relazioni fra oggetti.

Classificazione ("instance of")

La **classificazione** lega oggetti e classi.

La classe definisce le proprietà comuni a tutti gli oggetti della classe e ogni oggetto possiede le proprietà definite dalla classe.

Esempio • Classificazione

Gianluca è un (esemplare di) Persona.

Generalizzazione (“is a”)

La **generalizzazione** lega una classe **genitore** (*superclasse*) a una o più classi **figlie** (*sottoclassi*).

Ogni oggetto della sottoclasse possiede tutte le proprietà definite dalla sottoclasse, ma anche quelle definite dalla superclasse.

Esempio • Generalizzazione

Aggregazione (“part of”)

L’**aggregazione** lega una classe **aggregato** con un insieme di classi **parti**.

Ogni oggetto della classe aggregato è costituito da oggetti delle classi parti.
L’aggregazione può essere di due tipi:

Aggregazione (debole)

Composizione (forte)

La **composizione** è una relazione più forte perché all’atto della distruzione dell’oggetto aggregato si ha la propagazione della distruzione agli oggetti parte.

Associazione (“has a”)

Una **associazione** definisce una connessione logica fra oggetti di una classe e oggetti di un’altra classe.

Esempio 1 • Associazione

UML: diagramma delle classi

Il **diagramma delle classi** consiste nell'identificazione delle classi e delle relazioni fra le stesse.
Attività fondamentale in fase di **analisi** e **progetto**.

Esempio • Diagramma delle classi

UML: diagramma degli oggetti

Il **diagramma degli oggetti** consiste nell'identificazione degli oggetti del sistema e dei loro stati durante il loro ciclo di vita.

Esempio • Diagramma degli oggetti

Associazioni (esempi)

Esempio 2 • Associazioni

Il **nome** di un'associazione esprime il **significato** dell'associazione (spesso è un **verbo**).

Il **ruolo** di un'associazione esprime il **ruolo** giocato dalle classi coinvolte nell'associazione (spesso è un **sostantivo** o un **aggettivo**).

La **cardinalità** di un'associazione esprime quante istanze di una classe possono essere associate a un'altra classe nell'associazione.

Esempio • Cardinalità

Associazioni multiple

Un'**associazione multipla** specifica più associazioni tra una stessa coppia di classi per arricchirne la descrizione.

Esempio • Associazioni multiple

Associazione a cappio

In un'**associazione a cappio** una classe è in relazione con sé stessa.

Esempio 1• Associazioni a cappio

Navigabilità

La **navigabilità** specifica chi vede cosa.

Esempio 1• Navigabilità

In Java, specifica la modalità con cui verrà realizzata l'associazione.

Esempio 2 • Navigabilità in Java

```
public class Frigorifero {
    Cibo c[] = new Cibo[20]; // l'associazione viene realizzata in questo modo
                             // la classe che vede l'altra classe conterrà
                             // i suoi oggetti come variabile di istanza
                             // di tipo classe.
    ...
    ...
    ...
}
```

Associazione (esempio studente universitario che possiede un libretto che contiene esami che si riferiscono a insegnamenti)

Introduzione ai computer e a Java

Un computer è un insieme di componenti hardware e software.
Il termine hardware indica la macchina fisica, mentre il termine software indica tutti quei programmi che forniscono istruzioni a un computer.

La **CPU** (*Central Processing Unit*, unità centrale di elaborazione), o semplicemente **processore**, è il dispositivo interno che esegue le istruzioni di un programma.

Definire classi e creare oggetti

Un **oggetto** è una rappresentazione di entità reali (persone, case, animali) oppure di entità astratte (colori, forme, parole).

Si dice che un oggetto è un'**istanza** di una classe perché la **classe** definisce le proprietà comuni a tutti gli oggetti di quella classe, chiamati **attributi** e **metodi**.

Tutti gli oggetti di una classe hanno gli stessi attributi e metodi che sono chiamati **membri** dell'oggetto.

Gli attributi vengono chiamati **variabili di istanza** perché ogni oggetto ha una propria copia delle variabili di istanza definite dalla classe.

La classe definisce solo il **tipo di dato** delle variabili, e non il valore che esse assumeranno una volta creato l'oggetto a cui appartengono.

Un **metodo di istanza** è un metodo che contiene istruzioni che possono fare riferimento alle variabili di istanza della classe in cui è definito.

Un metodo di istanza si differenzia da un metodo di classe per l'assenza del modificatore di visibilità `static` e per il fatto che i metodi di istanza possono essere invocati solo tramite gli oggetti della stessa classe in cui sono definiti.

Per creare un oggetto in Java si usa l'operatore `new`, in questo modo:

```
Persona p = new Persona();
```

L'operatore `new` crea un oggetto della classe `Persona`, restituisce il suo indirizzo di memoria e lo assegna alla variabile `p`.

All'atto della creazione dell'oggetto, tutte le sue variabili di istanza sono inizializzate ad un valore di default che varia in base al tipo di dato della variabile.

Per esempio, 0 per `int`, 0.0 per `float` e `double`, `False` per `boolean`, `null` per le variabili di tipo classe, tra cui anche `String` dato che in Java è definita come una classe.

È buona norma definire le variabili di istanza di una classe con il modificatore di visibilità `private`.

Questo permette al programmatore che usa la classe di poter modificare i valori delle variabili solo attraverso i metodi definiti nella classe dell'oggetto.
Così facendo la classe ha il pieno controllo di lettura e scrittura dei valori delle sue variabili, data l'impossibilità di accedervi direttamente.

In questo caso, l'unico modo per accedere al valore delle variabili di istanza è quello di definire dei metodi

particolari, detti metodi **`set`** e metodi **`get`**.

I metodi **`set`** accedono alla variabile di istanza modificando il suo valore.
Il vantaggio rispetto a un accesso diretto sta nel fatto che con i metodi set è possibile controllare il valore assegnato alle variabili.

Se per esempio una variabile di tipo `int` deve sempre avere un valore positivo, con un metodo set potremo implementare un'istruzione che controlli ogni qualvolta inseriamo un numero negativo, permettendo quindi a Java di individuare e segnalare subito l'errore a runtime.

Come è facile intuire, se non avessimo usato il metodo ***set*** per modificare il valore della variabile ma avessimo modificato la variabile direttamente tramite un assegnamento, non avremmo avuto a disposizione nessun modo per stabilire se il valore assegnato alla variabile fosse negativo o meno e quindi l'errore non sarebbe stato individuato e segnalato immediatamente.

La sintassi di un generico metodo di istanza è la seguente:

Sintassi

```
public tipo_valore_restituito nome_metodo (parametri) {
      istruzioni

}
```

Esempio

```
public void stampaPersona() {
    System.out.println("Nome: " + nome);
    System.out.println("Età: " + eta);

}
```

Mentre la sintassi per la definizione di una classe è la seguente:

Sintassi

```
public class nome_classe {
    dichiarazioni_variabili_di_istanza
    ...
    dichiarazioni_di_metodi

}
```

Esempio

```
public class Persona {

    private String nome;
    private int eta;

    public void setNome( String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return this.nome;
    }

    public void setEta(int eta) {
        this.eta = eta;
    }

    public int getEta() {
        return this.eta;
    }
```

Sebbene questa sia la forma più usata, è possibile anche mischiare definizioni di metodi e dichiarazioni di variabili di istanza.

Si sarà notato l'uso della parola chiave **this**.

La parola chiave **this** all'interno della definizione di un metodo può essere utilizzata come un nome per l'oggetto che invoca il metodo.

Prendendo come esempio la definizione della classe **Persona** appena sopra, usiamo la classe per creare un oggetto **p** nel programma **TestPersona**:

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

        Persona p = new Persona();

        p.setNome("Gianluca");
        p.setEta(21);

        System.out.println("Nome: " + p.getNome());
        System.out.println("Eta: " + p.getEta());
    }
}
```

Avendo utilizzato la parola chiave **this** come nome per l'oggetto invocante il metodo e avendo creato l'oggetto **p** di tipo **Persona**, la situazione che si presenterà una volta che il metodo sarà invocato tramite **p** è la seguente:

```
\\Per il metodo setNome()
    public void setNome("Gianluca") {
        p.nome = "Gianluca";
    }

\\Per il metodo setEta()
    public void setEta(21) {
        p.eta = 21;
    }

\\Per il metodo getNome()
    public String getNome() {
        return p.nome; \\p.nome = "Gianluca"
    }

\\Per il metodo getEta()
    public int getEta() {
        return p.eta; \\p.eta = 21;
    }
```

La parola chiave **this** è stata sostituita con il nome dell'oggetto invocante il metodo, cioè **p**.

Questo succede per ogni oggetto di tipo **Persona** che invoca i metodi definiti nella classe **Persona**.

Si può concludere quindi che la parola chiave **this** non è altro che uno spazio vuoto che aspetta di essere riempito con il nome dell'oggetto che invoca il metodo.

Bisogna precisare che l'uso della parola chiave `this` in una definizione di metodo non è obbligatoria, si può anche farne a meno; se non la si usa bisogna però prestare attenzione a non dare lo stesso nome della variabile di istanza della classe al parametro passato nella definizione del metodo.

Se per distrazione dovesse capitare una cosa di questo tipo

```
    public class Persona {

        private String nome;
        ...

        public void setNome(String nome) {
            nome = nome;
        }

        public int getNome() {
            return nome;
        }

        ...
        ...
    }
```

una volta creato un oggetto `p` di `Persoa` in `TestPersona` e invocato i suoi metodi tramite `p`

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

        Persona p = new Persona();

        p.setNome("Gianluca");
        p.setEta(21);


        System.out.println("Nome: " + p.getNome()); \\Stampa null
        System.out.println("Eta: " + p.getEta()); \\Stampa 0
    }

}
```

la situazione che si presenterà sarà la seguente

```
    \\Per il metodo setNome()
        public void setNome("Gianluca") {
            nome = "Gianluca";
        }

    \\Per il metodo setEta()
        public void setEta(21) {
            eta = 21;
        }
```

```
\\Per il metodo getNome()
    public String getNome() {
        return null; \\ritorna p.nome = null
    }

\\Per il metodo getEta()
    public int getEta() {
        return 0; \\ritorna p.eta = 0;
    }
```

Si può notare come in una definizione di metodo se non si usa la parola chiave `this` e la variabile di istanza e il parametro passato hanno lo stesso nome, allora Java non produrrà nessun errore in compilazione, ma i due nomi faranno riferimento sempre e solo al parametro a cui verrà riassegnato il suo stesso valore.

Questo perché Java pensa erroneamente che non ci sia nessuna ambiguità e che l'assegnamento che vogliamo fare sia corretto.

D'altro canto, il metodo `getNome` non ha parametri, quindi in questo caso non si presenta nemmeno l'ambiguità da risolvere, proprio perché nel metodo `setNome` il parametro era una variabile locale al metodo e quindi in `getNome` l'identificatore farà riferimento alla variabile di istanza, dato che per lui il parametro passato a `setNome` non esiste.
Purtroppo però in questo caso il metodo `getNome` ritornerà il valore di default della variabile di istanza dato che non gli è stato assegnato nessun valore in `setNome`.

In Java il corpo di un metodo può contenere l'invocazione di un altro metodo.

Grazie a questa possibilità, si possono definire dei metodi privati che svolgono dei compiti secondari e che rimangono visibili solo all'implementazione della classe.

Questi metodi vengono chiamati **metodi ausiliari** e si trovano all'interno di metodi pubblici, oppure si trovano all'interno di altri metodi privati che a loro volta solo all'interno di metodi pubblici.

Questi metodi vengono dichiarati privati proprio perché svolgono una funzione secondaria per conto di altri metodi e quindi sarebbe inutile utilizzarli al di fuori della classe in cui sono definiti.

Dichiarandoli privati, si impedisce la possibilità di utilizzarli al di fuori della classe in cui sono definiti in modo inappropriato.

Vediamo qualche esempio.

```
public class Persona {
    private String domanda;
    private String risposta;

    public void setDomanda(String domanda) {
     this.domanda = domanda;
     }

    public String getDomanda() {
     return this.domanda;
    }

    public void setRisposta(String risposta) {
     this.risposta = risposta;
    }

    public String getRisposta() {
     return this.risposta;
    }

    public void parla() {

        if ((domanda != null) && (risposta == null))
            domanda();

        else if ((domanda == null) && (risposta != null))
            risposta();
    }

    private void domanda() {
     System.out.println(getDomanda());
    }

    private void risposta() {
        System.out.println(getRisposta());
    }
}
```

Utilizzando la classe `Persona` nella classe `TestPersona`

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

     Persona p1 = new Persona();
     Persona p2 = new Persona();

     p1.setDomanda("Come ti chiami ?");
     p1.parla();

     p2.setRisposta("Io mio chiamo p2");
     p2.parla();
  }
}
```

In questo caso la parola chiave `this` non è necessaria all'interno della definizione del metodo `parla` per specificare l'oggetto che invoca i metodi privati `domanda` e `risposta`.
Questo perché quando nella classe `TestPersona` l'oggetto `p` invoca il metodo `parla()` le invocazioni successive ai metodi `domanda()` o `rispondi()` verranno risolte immediatamente da Java come `p.domanda()` oppure `p.rispondi()`.

Incapsulamento

L'**incapsulamento** è una forma di *information hiding* che consiste nel separare la definizione di una classe in due parti: l'interfaccia e l'implementazione.

L'**interfaccia di una classe** consiste nell'intestazione dei suoi metodi pubblici e delle sue costanti pubbliche, insieme ai commenti che indicano ai programmatori come usare i metodi e le costanti.

L'**implementazione di una classe** consiste principalmente delle variabile di istanza private e il corpo dei metodi pubblici e privati della classe.

Prendiamo come esempio la definizione della classe `Persona` a pagina 21.

Lo stile dei commenti `/** */` è usato per descrivere l'interfaccia.
Lo stile `//` è invece usato per i commenti sull'implementazione.

Si noti come l'interfaccia e l'implementazione di una classe in Java non sono separate, ma appartengono allo stesso file.

Le linee guida per definire una classe ben incapsulata sono le seguenti:

- Predisporre un commento prima della definizione della classe che descriva al programmatore cosa rappresenta la classe senza descrivere come lo fa;
- Le variabili di istanza della classe devono essere dichiarate private;
- Fornire metodi `set` per impostare i valori delle variabili di istanza dell'oggetto e metodi `get` per recuperare i valori delle variabili di istanza dell'oggetto.
  Fornire inoltre metodi pubblici che potrebbero servire al programmatore per gestire i dati della classe per altre necessita;
- Predisporre un commento prima di ogni intestazione di metodo pubblico che specifichi chiaramente come usare il metodo;
- Rendere privati tutti i metodi ausiliari;
- Scrivere commenti all'interno della classe per descrivere i dettagli implementativi.

Esempio • Incapsulamento

```
/** La classe Persona definisce una persona che può domandare e rispondere, simulando
una piccola conversazione.
*/
public class Persona {
    private String domanda; //La domanda
    private String risposta; //La risposta

    /** Imposta la domanda della persona */
    public void setDomanda(String domanda) {
     this.domanda = domanda;
     }

     /** Restituisce la domanda fatta dalla persona */
    public String getDomanda() {
     return this.domanda;
    }

    /** Imposta la risposta della persona */
    public void setRisposta(String risposta) {
     this.risposta = risposta;
    }

    /** Restituisce la risposta fatta dalla persona */
    public String getRisposta() {
     return this.risposta;
    }

    /** Mostra la domanda o la risposta fatta dall'utente. Almeno una delle due deve
    essere impostata a null quando l'altra non lo è. */
    public void parla() {

        // Se domanda non è null ma risposta sì, invochiamo domanda.
        if ((this.domanda != null) && (this.risposta == null))
            domanda();
        // altrimenti se domanda è null e risposta no, invochiamo risposta.
        else if ((this.domanda == null) && (this.risposta != null))
            risposta();
    }

    private void domanda() {
     System.out.println(getDomanda());
    }

    private void risposta() {
        System.out.println(getRisposta());
    }
}
```

Oggetti e riferimenti

Una variabile di tipo classe contiene l'indirizzo di memoria dell'oggetto cui fa riferimento.
L'indirizzo di memoria dell'oggetto è detto **riferimento all'oggetto** (*reference*).
Per questo motivo, i tipi classe sono spesso chiamati **tipi riferimento** (*reference types*).

La differenza sostanziale tra una variabile di tipo primitivo e una di tipo reference sta nel fatto che una variabile primitiva necessita sempre della stessa quantità di memoria dato che i valori per un tipo primitivo sono prefissati, mentre per una variabile di tipo reference, come la classe `String`, non si saprà mai a priori quanta memoria verrà utilizzata dall'oggetto associato alla variabile.

Visto che le variabili di tipo reference non contengono valori ma bensì indirizzi di memoria, non è possibile quindi usare l'operatore `==` per stabilire se due oggetti sono uguali, ovvero hanno gli stessi valori, come per le variabili primitive.

Ad ogni definizione di classe si deve quindi definire un metodo `equals`, un particolare metodo che verifica se due oggetti sono uguali, ovvero hanno gli stessi valori nelle stesse variabili di istanza.

Esempio • Definizione di metodo `equals`

```
public class Persona {
    private String nome;
    private int eta;

    ...
    ...
    ...

    public boolean equals(Persona altraPersona) {
        return (this.nome.equalsIgnoreCase(altraPersona.nome))
            && (this.eta == altraPersona.eta);
    }
}
```

Se non si definisce un metodo `equals` per una classe, Java ne crea automaticamente uno con una definizione di default che però non potrebbe comportarsi nel modo voluto.
È meglio quindi definire un metodo `equals` per ogni nuova classe definita.

Anche l'istruzione di assegnamento = funziona in maniera differente per le variabili di tipo riferimento rispetto a quelle primitive.
Consideriamo l'esempio seguente

```
Persona p1 = new Persona(), p2 = new Persona();

p1.setNome("Gianluca");
p2.setNome("Zhou");

p2 = p1;

System.out.println("Il nome di p1 è: " + p1.getNome());
System.out.println("Il nome di p2 è: " + p2.getNome());
```

Come si potrebbe pensare, l'assegnamento dovrebbe copiare i valori delle variabili di istanza di `p1` in `p2` una sola volta, ma quello che è stato fatto è stato copiare l'indirizzo dell'oggetto cui fa riferimento `p1` in `p2`, ed ora in avanti la variabile `p2` sarà un altro nome per riferirsi all'oggetto cui fa riferimento la variabile `p1`.

Quindi le due invocazioni

```
p1.setNome("Gianluca");
p2.setNome("Zhou");
```

riferiranno allo stesso oggetto, cioè quello cui si riferisce `p1`.

Perciò le due istruzioni successive

```
System.out.println("Il nome di p1 è: " + p1.getNome());
System.out.println("Il nome di p2 è: " + p2.getNome());
```

Non stamperanno a video `Gianluca` e `Zhou`, bensì `Gianluca` e `Gianluca`, proprio perché la variabile `p2` si riferisce allo stesso oggetto cui si riferisce `p1`.

Abbiamo visto che il metodo `equals` è un metodo booleano, perciò può restituire un valore `true` o `false` che può essere assegnato a una variabile booleana che a sua volta può essere valutata nell'espressione di costrutti come `if` o `while`.

In alcuni casi si può anche fare a meno della variabile booleana e riportare l'invocazione del metodo direttamente all'interno delle parentesi del costrutto utilizzato che poi valuta il valore di ritorno del metodo booleano.

Per esempio, data la seguente linea di codice

```
Persona p1 = new Persona(), p2 = new Persona();

if (p1.isPiuGrande(p2))
    System.out.println(p1.getNome() + "è più grande di" + p2.getNome());
else
    System.out.println(p2.getNome() + "è più grande di" + p1.getNome());
```

è evidente che l'istruzione all'interno dell'`if` verrà intrapresa solo se `p1` è più grande di `p2`.

Se invece avessimo assegnato il valore di ritorno di `isPiuGrande` a una variabile booleana, per esempio in questo modo:

```
Persona p1 = new Persona(), p2 = new Persona();

boolean PpiuGrande = p1.isPiuGrande(p2);

if (piuGrande)
    System.out.println(p1.getNome() + "è più grande di" + p2.getNome());
else
    System.out.println(p2.getNome() + "è più grande di" + p1.getNome());
```

potrebbe non essere chiaro se `piuGrande` debba essere `true` o `false`.
Quindi è meglio riportare l'invocazione del metodo direttamente all'interno del costrutto per evitare certe ambiguità.

Costruttori

Un **costruttore** è un particolare metodo che viene invocato quando si utilizza l'operatore `new` per creare un nuovo oggetto di una classe.
Si consideri la seguente semplice definizione di classe:

```
/* public class Persona
   La classe Persona definisce un oggetto avente un nome:String e un età:int
 */

public class Persona {

    private String nome;
    private int eta;

    public Persona() {
        this.nome = null;
        this.eta = 0;
    }

    public Persona(String nome, int eta) {
        this.nome = nome;
        this.eta = eta;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return this.nome;
    }

    public void setEta(int Eta) {
        this.eta = eta;
    }

    public int getEta() {
        return this.eta;
    }

}
```

`Persona()` e `Persona(String nome, int eta)` sono due costruttori definiti nella classe `Persona`.

`Persona()` è detto **costruttore di default** perché non accetta nessun parametro come argomento e inizializza semplicemente tutte le variabili di istanza dell'oggetto appena creato con i valori di default dei loro tipi.

`Persona(String nome, int eta)` è invece un costruttore definito dall'utente che accetta due parametri come argomenti e inizializza le variabili di istanza dell'oggetto con i rispettivi valori assunti dagli argomenti.

Qualora una classe non contenga alcuna definizione di un costruttore, Java definisce automaticamente il costruttore di default. Tuttavia, se in una classe viene definito almeno un costruttore, non viene più aggiunto automaticamente nessun altro costruttore. Questo anche nel caso in cui sia definito un costruttore con parametri.

È importante quindi definire sempre in una classe almeno il costruttore di default.

Ricordarsi inoltre che non è possibile usare un oggetto già esistente per invocare un costruttore.
La seguente istruzione è sbagliata:

```
Persona p1 = new Persona();

p1.Persona("Marco", 23); //Non valido!
```

Una volta creato un oggetto, l'unico modo per modificare i valori delle sue variabili di istanza consiste nell'invocare i suoi metodi `set`.
Nel caso precedente, le seguenti istruzioni sono valide:

```
p1.setNome("Marco");
p1.setEta(23);
```

Infine, è inoltre possibile invocare un costruttore all'interno di un altro costruttore. Una volta definito il primo costruttore, è possibile definire un altro costruttore che invochi il primo costruttore. Per fare ciò, si utilizza la parola chiave `this` come se fosse il nome di un metodo in un'invocazione.
Per esempio, l'istruzione seguente:

```
this(null, 0);
```

invoca il costruttore con due parametri dal corpo di un altro costruttore della classe.
L'invocazione deve essere la prima azione eseguita all'interno del corpo del costruttore.

La definizione della classe seguente mostra come possono essere modificati i costruttori della classe precedente.

```java
/* public class Persona
   La classe Persona definisce un oggetto avente un nome:String e un età:int
 */

public class Persona {

    private String nome;
    private int eta;

    public Persona(String nome, int eta) {
        this.nome = nome;
        this.eta = eta;
    }

    public Persona() {
        this(null, 0);
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return this.nome;
    }

    public void setEta(int Eta) {
        this.eta = eta;
    }

    public int getEta() {
        return this.eta;
    }

}
```

La costante `null`

Se una variabile di tipo classe è inizializzata con la costante `null`, essa non fa riferimento a nessun oggetto in memoria. Per esempio:

```
Persona p1 = null;
```

È anche usuale utilizzare il valore `null` nei costruttori di una classe per inizializzare delle variabili di tipo classe quando non è ovvio quale oggetto utilizzare.

Variabili e metodi statici

Una **variabile statica** è una variabile unica e condivisa da tutti gli oggetti della classe in cui è definita.

La dichiarazione di una variabile statica contiene la parola chiave `static`:

```
private static int PI = 3,14159; //Pi greco e le prime 5 cifre decimali
```

È bene dichiarare la variabile statica come privata, in modo tale che sia accessibile solo attraverso i metodi della classe in cui è definita.

Una variabile statica può essere definita anche come costante aggiungendo la parola chiave `final`:

```
public static final int GIORNI_PER_SETTIMANA = 7;
```

In questo caso, la variabile statica è stata dichiarate pubblica perché la parola chiave `final` non permette la modifica del valore della variabile, nemmeno ai metodi della classe in cui è definita.

Un **metodo statico** è un metodo che può essere invocato senza usare alcun oggetto.

Normalmente si invoca un metodo statico utilizzando il nome della classe in cui è definito, anziché quello di un oggetto.
La dichiarazione avviene anteponendo la parola chiave `static` al nome del metodo:

```
public class Massimo {

    //Calcola il massimo tra due numeri interi
    public static int massimo(int a, int b);

}
```

In una classe, il metodo statico `massimo` sopra descritto verrà invocato come segue:

```
public class MassimoDemo
{
    public static void main (String args[])
    {
        int a = 23, b = 12, massimo;

        massimo = Massimo.massimo(a,b);

        System.out.println("Il massimo tra "+ a +" e "+ b +" è: "+ massimo);
    }
}
```

Nella definizione di un metodo statico, non è possibile riferirsi a una variabile di istanza di un oggetto.

Inoltre, un metodo statico può invocare un metodo di istanza di una classe solo se ha un oggetto della classe con cui invocarlo, che non sia un oggetto `this` implicito o esplicito.

```
\\Supponiamo che tutti gli impiegati abbiano le stesse ore di straordinari
\\Non sono riportati tutti i metodi get e set per abbreviare la stesura della classe, ma si
assume che almeno quelli base per modificare i valori delle variabili di istanza siano
presenti

public class Impiegato {
    private string nome;
    private int oreLavoro;

    private static int oreStraordinari = 0;


    public int getOreLavoro() {
        return this.oreLavoro;
    }

    public void aggiungiOreStraordinari(int ore) {
        oreStraordinari = ore;
    }

    public static void mostraOreTotali(Impiegato i) {
        System.out.println(oreStraordinari + i.getOreLavoro());
    }

}
```

D'altro canto, un metodo di istanza può accedere a una variabile static o invocare un metodo statico direttamente.

Se il metodo `main` di una classe è strutturato in più parti aventi lo stesso ciclo di istruzioni, è possibile suddividere il main in sotto-attività, ognuna identificata da un metodo statico privato definito all'interno della stessa classe in cui è definito il main.

Una strategia di questo tipo è utile per semplificare la logica del programma e renderlo più leggibile, eliminando quelle istruzioni che si ripetono più di una volta nello stesso programma.

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

        Impiegato i1 = new Impiegato(), i2 = new Impiegato();

        i1.setImpiegato("Gianluca", 35);
        i1.setImpiegato("Gianluca", 35);

        if (i1 == i2)
            System.out.println("Corrispondono secondo ==.");
        else
            System.out.println("Non corrispondono secondo ==.");

        if (i1.equals(i2))
            System.out.println("Corrispondono secondo il metodo equals.");
        else
            System.out.println("Non corrispondono secondo il metodo equals.");

        System.out.println("Cambiamo i valori delle variabili di istanza di i2...");
        i2.setImpiegato("Zhou", 40);

        if (i1.equals(i2))
            System.out.println("Corrispondono secondo il metodo equals.");
        else
            System.out.println("Non corrispondono secondo il metodo equals.");


    }
}
```

con dei sotto-metodi diventerebbe più compatto e leggibile

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

        Impiegato i1 = new Impiegato(), i2 = new Impiegato();

        i1.setImpiegato("Gianluca", 35);
        i1.setImpiegato("Gianluca", 35);

        testConOperatoreUguale(i1, i2);
        testConOperatoreEquals(i1, i2);

        System.out.println("Cambiamo i valori delle variabili di istanza di i2...");
        i2.setImpiegato("Zhou", 40);

        testConOperatoreUguale(i1, i2);
        testConOperatoreEquals(i1, i2);

    }

    private static void testConOperatoreUguale(Impiegato i1, Impiegato i2) {
        if (i1 == i2)
            System.out.println("Corrispondono secondo ==.");
        else
            System.out.println("Non corrispondono secondo ==.");
    }

    private static void testConOperatoreEquals(Impiegato i1, Impiegato i2) {
         if (i1.equals(i2))
            System.out.println("Corrispondono secondo il metodo equals.");
        else
            System.out.println("Non corrispondono secondo il metodo equals.");
    }
}
```

È utile sottolineare che un metodo `main` può anche essere aggiunto a una classe da cui creare oggetti per testare la classe stessa.
Quindi, riprendendo la definizione della classe `Persona`

```
public class Persona {

    private String nome;
    private int eta;

    public Persona(String nome, int eta) {
        this.nome = nome;
        this.eta = eta;
    }

    public Persona() {
        this(null, 0);
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return this.nome;
    }

    public void setEta(int Eta) {
        this.eta = eta;
    }

    public int getEta() {
        return this.eta;
    }

    public static void main (String args[]) {

        Persona p1 = new Persona("Gianluca", 0);

        p1.setEta(21);

        System.out.println("p1.nome: "+ p1.getNome() +", p1.età: "+ p1.getEta() +".");
    }

}
```

Abbiamo aggiunto un metodo main che testa la classe creando degli oggetti di tipo `Persona` e invocando i relativi metodi tramite gli oggetti. Questo è quello che abbiamo imparato poco fa, ovvero un metodo statico può invocare all'interno di sé stesso un metodo di istanza solo tramite l'oggetto della classe in cui il metodo è definito.

In questo caso, una volta terminato il collaudo della classe, è meglio rimuovere il metodo main onde evitare incomprensioni da parte di programmatori che non hanno definito di persona l'implementazione della classe, e che potrebbero scambiare il metodo main in essa definito come un metodo pubblico utile da invocare in altre classi.

Overloading

L'**overloading** del nome di un metodo consiste nell'avere più definizioni dello stesso metodo che differenziano per tipo e numero dei parametri.

Consideriamo la classe `Overload`

```
public class Overload {

    private static int calcolaMedia(int a, int b) {
        return (a + b) / 2;
    }

    private static int calcolaMedia(int a, int b, int c) {
        return (a + b + c) / 3;
    }

    private static double calcolaMedia(double a, double b) {
        return (a + b) / 2.0;
    }
}
```

Effettuando l'overloading, è necessario che le definizioni dei diversi metodi presentino differenze nell'elenco dei parametri che ricevono.

Java distingue i metodi sulla base del numero e del tipo di parametri che ricevono.

Se non esiste alcun metodo che corrisponde al metodo invocato, allora Java prova ad eseguire automaticamente delle conversioni di tipo agli argomenti forniti (per esempio da `int` a `double`) per verificare se trova una corrispondenza con i parametri nella definizione di un metodo.

Se non c'è corrispondenza nemmeno in questo caso, Java restituisce un errore durante la compilazione del programma.

Per brevità, il nome di un metodo, il numero e il tipo di parametri che utilizza sono detti **firma** (*signature*) del metodo.

È bene ricordare che Java cerca di usare l'overloading prima di usare la conversione automatica di tipo finché non si accerta che non esista una definizione di metodo che corrisponde al tipo degli argomenti passati al metodo.

Infine, non è possibile effettuare l'overloading di un metodo solo sulla base del tipo di ritorno.
Una situazione come questa

```
public class Persona {

    public int getEta() {
        ...
        ...
        ...


    public double getEta() {
        ...
        ...
        ...
```

non è possibile, proprio perché il tipo di ritorno di un metodo non fa parte della sua firma, e quindi Java non saprebbe quale definizione di metodo usare.

Array di riferimenti

Per realizzare le associazioni in Java, è possibile utilizzare degli array il cui tipo base è un tipo classe qualunque. Consideriamo come esempio una azienda che ha un certo numero di dipendenti

```
/** Classe Dipendete */

public class Dipendente {

    private String nome;

    Dipendente(String nome) {
        this.nome = nome;
    }

    Dipendente() {
        this(null);
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return this.nome;
    }

}
```

```java
import java.util.Scanner;

/** Classe Azienda */
public class Azienda {

	private Dipendente[] dipendenti;
	private int numeroDipendenti;

	public void setNumeroDipendenti(int numeroDipendenti) {
	    this.numeroDipendenti = numeroDipendenti;
	}

	public int getNumeroDipendenti() {
	    return this.numeroDipendenti;
	}

	public void setDipendenti(int numeroDipendenti) {
	    dipendenti = new Dipendente[numeroDipendenti];

	    for (int i = 0; i < numeroDipendenti; i++) {
	        dipendenti[i] = new Dipendente();
	    }
	}

	public void setNomeDipendenti() {

	    Scanner tastiera = new Scanner(System.in);

	    for (int i = 0; i < this.getNumeroDipendenti(); i++) {
	        System.out.print("Nome " + (i+1) +"o dipendente: ");
	        String nome = tastiera.nextLine();
	        dipendenti[i].setNome(nome);

	        if (dipendenti[i] == null)
	            break;
	    }
	}

	public void getNomeDipendenti() {
	    for (int i = 0; i < this.getNumeroDipendenti(); i++) {
	        System.out.println((i+1) + "o dipendente: " + dipendenti[i].getNome());
	    }
	}
```

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

            Azienda a = new Azienda();

            //Numero di dipendenti dell'azienda
            a.setNumeroDipendenti(2);

            //Inizializza i dipendenti nell'azienda
            a.setDipendenti(a.getNumeroDipendenti());

            //Inizializza il nome dei dipendenti dell'azienda
            a.setNomeDipendenti();

            //Visualizza i dipendenti dell'azienda
            a.getNomeDipendenti();
        }
}
```

Ereditarietà

L'**ereditarietà** (*inheritance*) permette di definire una classe generale e di definire in seguito classi più specializzate che aggiungono nuovi dettagli alla classe generale.

La classe **base**, o classe **genitore**, viene detta **superclasse**.

La classe **derivata**, o classe **figlia**, viene detta **sottoclasse**.

Una classe derivata eredita tutte le variabili di istanza (se private accessibili solo tramite metodi pubblici), le variabili statiche e tutti i metodi pubblici della classe base da cui è definita, e può aggiungere variabili proprie e metodi propri, oppure modificare i metodi ereditati dalla classe base.

In Java, ciò è specificato includendo l'espressione `extends` sulla prima riga della definizione della classe

```
public class Studente extends Persona {

    dichiarazione_di_variabili_aggiuntive
    dichiarazione_di_metodi_aggiuntivi_e_di_metodi_modificati

}
```

Una classe che è genitore di una classe che a sua volta è genitore di un'altra classe (ma la gerarchia si può estendere indefinitamente), è detta classe **antenato**.

Se la classe A è un antenato della classe B, allora la classe B è chiamata **discendente** della classe A.

Esempio • Classe derivata

```
public class Persona {

    private String nome;

    public Persona() {
        this.nome = null;
    }

    public Persona(String nome) {
        this.nome = nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return this.nome;
    }

    public void scriviOutput() {
        System.out.println("Nome: " + this.nome);
    }

    public boolean haLoStessoNome(Persona altraPersona) {
        this.nome.equalsIgnoreCase(altraPersona.nome);
    }

}
```

```java
public class Studente extends Persona {

    private int matricola;

    public Studente() {
        super();
        this.matricola = 0;
    }

    public Studente(String nome, int matricola) {
        super(nome);
        this.matricola = matricola;
    }

    public void reimposta(String nome, int matricola) {
        setNome(nome);
        this.matricola = matricola;
    }

    public int getMatricola() {
        return this.matricola;
    }

    public void setMatricola(int matricola) {
        this.matricola = matricola;
    }

    public void scriviOutput() {
        System.out.println("Nome: " + getNome());
        System.out.println("Matricola: " + getMatricola());
    }

    public boolean equals(Studente altroStudente) {
        return this.haLoStessoNome(altroStudente) &&
            (this.matricola == altroStudente.matricola);
    }
```

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

        Studente s = new Studente();

        s.setNome("Gianluca Cannata"); \\Ereditato dalla classe Persona
        s.setMatricola(1234);
        s.scriviOutput(); \\Ridefinito nella classe Studente

    }
}
```

Metodi ridefiniti (overriding)

L'**overriding** di un metodo consiste nella ridefinizione di un metodo della classe base nella classe derivata.

La classe derivata ridefinisce solo il corpo del metodo, mentre l'intestazione, ovvero il tipo di ritorno e i parametri che riceve (in termini di tipo, ordine e numero) rimangono invariati.

Il metodo `scriviOutput` presente nella classe `Studente` derivata da `Persona` è un esempio di overriding di metodo.
Si può notare come l'intestazione del metodo rimanga invariata, mentre il corpo del metodo è ridefinito in base alle esigenze richieste nella classe `Studente`.

L'unico caso in cui è possibile modificare il tipo di ritorno di un metodo ereditato è quando il tipo di ritorno del metodo della classe base è un tipo classe.

Si supponga che una classe includa le seguenti definizioni:

```
public class ClasseBase {
    ...
    public Persona getIndividuo(int identificatore) {
        ...
```

In questo caso, la classe derivata può lecitamente includere la seguente dichiarazione:

```
public class ClasseDerivata extends ClasseBase {
    ...
    public Studente getIndividuo(int identificatore) {
        ...
```

La ridefinizione del metodo `getIndividuo` in `ClasseDerivata` cambia il tipo di ritorno da `Persona` a `Studente`.

Questo è possibile perché ogni `Studente` è anche una `Persona`.

È possibile, infine, cambiare il modificatore d'accesso del metodo, a patto di non restringere la visibilità.

Quindi, mentre è possibile cambiare il modificatore da `private` a `public`, non è possibile il contrario.

Per distinguere *overloading* e *overriding*, si ricordi che *overloading* aggiunge un "carico" (*load*) sul nome di un metodo utilizzandolo per un'ulteriore attività, mentre l'*overriding* sostituisce una definizione del metodo.

Una classe derivata non può accedere direttamente alle variabili di istanza private della sua classe base, se non solamente attraverso l'utilizzo dei metodi `set` e `get` definiti nella classe base.

Inoltre, tutti i metodi privati definiti nella classe base non possono essere invocati dalla classe derivata.
L'unica eccezione alla regola è che la classe derivata invochi metodi pubblici che a loro volta invochino metodi privati, a patto però che entrambi i metodi siano definiti nella classe base.

Un metodo (o una variabile di istanza) dichiarato con modificatore d'accesso `protected` è accessibile per nome dalla classe cui appartiene, dalle classi derivate dalla classe cui appartiene e da qualsiasi classe nello stesso package della classe cui appartiene.

È però sconsigliabile usare il modificatore d'accesso `protected` per le variabili di istanza in quanto garantisce un basso livello di protezione rispetto al modificatore d'accesso `private`.

Solo in rare occasioni si potrebbe voler dichiarare un metodo come `protected`.

Costruttori nelle classi derivate

Una classe derivata ha i suoi costruttori e non eredita alcun costruttore dalla classe base.
La tipica definizione di un costruttore per la classe derivata, consiste nell'invocare un costruttore della classe base.

Ciò avviene utilizzando la parola chiave `super` come nome di metodo per invocare il costruttore della classe base.

La parola chiave `super` deve essere la prima istruzione da eseguire all'interno del costruttore della classe derivata.
Se in ogni costruttore della classe derivata non si include un'invocazione esplicita al costruttore della classe base, Java includerà automaticamente un'invocazione al costruttore di default della classe base.

Inoltre, se la classe base non definisce nessun costruttore di default e se si omette l'invocazione a uno dei costruttori della classe base nella definizione di un costruttore della classe derivata, si avrà un errore in compilazione.
Infatti, `super()` non esiste nella classe base.

Esempio • Chiamare un costruttore della classe base

```
public Studente(String nome, int matricola) {
    super(nome);
    this.matricola = matricola;
}
```

Nelle classi derivate è sempre possibile invocare costruttori definiti nella stessa classe attraverso la parola chiave `this`.
Anche in questo caso, `this` deve essere la prima istruzione da eseguire dal costruttore della classe derivata, quindi non è possibile riportare `this` e `super` nella stessa definizione di un costruttore.

L'unico modo è quello di utilizzare `this` per invocare un costruttore che appartiene alla classe derivata che a sua volta invoca un costruttore della classe base attraverso `super`.

Invocare un metodo ridefinito

Nella definizione di un metodo di una classe derivata si può invocare un metodo ridefinito della classe base facendolo precedere da `super` e un punto.

Sintassi

```
super.nome_del_metodo_ridefinito(elenco_argomenti)
```

Esempio

```
public void scriviOutput() {
    super.scriviOutput(); \\Inoca scriviOutput nella classe base
    System.out.println("Matricola: " + this.matricola);
}
```

Infine, si ricordi che una situazione di questo tipo non è possibile

```
super.super.scriviOutput();
```

In Java è errato utilizzare più volte in sequenza la parola chiave `super`.

La classe `Object`

La classe `Object` è la classe antenata di ogni classe, sia delle classi derivate da altre classi sia delle classi che non sono classi derivate.

Quindi, si può dire che ogni classe è anche di tipo `Object` e di conseguenza eredita i metodi definiti dalla classe `Object`.

Un esempio di questi metodi ereditati sono il metodo `equals`, il metodo `toString` e il metodo `clone`.

Tutti e questi tre metodi svolgono una funzione ben specifica, ma non sono definiti correttamente per funzionare anche per gli oggetti delle classi derivate.
Perciò bisogna ridefinire questi metodi e adattarli in base alle esigenze della classe derivata.

Come esempio, ridefiniamo il metodo `equals` con una definizione più appropriata:

```
public boolean equals(Object altroOggetto) {

    boolean uguale = false;

    if ((altroOggetto != null) && (altroOggetto instanceof Studente)) {
        Studente altroStudente = (Studente)altroOggetto;
        uguale = this.haLoStessoNome(altroStudente) &&
            (this.matricola == altroStudente.matricola);
    }

    return uguale;
}
```

La parola chiave `instanceof` garantisce se l'oggetto passato come parametro al metodo `equals` è di tipo `Studente`.
Se non è di tipo `Studente`, il metodo `equals` ritorna direttamente il valore `false`, evitando così possibili ambiguità grazie alla presenza della parola chiave `instanceof`.

Polimorfismo, classi astratte e interfacce

Il **polimorfismo** permette di associare più significati a un nome di metodo per mezzo di un meccanismo conosciuto come *binding* dinamico (o *dynamic binding* o *late binding*).

Il *binding* dinamico fa sì che l'invocazione di un metodo venga associata alla sua definizione solamente a run-time, quindi nel momento in cui l'invocazione viene effettivamente eseguita.

Java utilizza il *binding* dinamico per tutti i metodi, tranne per quelli discussi più avanti.
Consideriamo l'esempio seguente:

```
public class Figura {

        public void centra() {
            // Istruzioni per spostare la figura
            visualizza();
        }

        public void visualizza() {
            // Implementazione vuota: in Figura non si sa come implementarlo
            // poiché dipende dalla specifica figura
        }
}
```

mentre quella della classe `Rettangolo` potrebbe essere

```
public class Rettangolo extends Figura {
    private double altezza;
    private double larghezza;
    private double puntoCentraleX;
    private double puntoCentraleY;

    public void visualizza() {
        // Istruzioni per visualizzare il rettangolo
    }
}
```

Grazie al **binding dinamico**, se un oggetto `r` della classe `Rettangolo` invocasse il metodo `centra` definito nella classe `Figura`, che a sua volta invocherebbe il metodo `visualizza` definito sia in `Figura` che in `Rettangolo`, Java risolverebbe l'invocazione automaticamente a run-time grazie proprio al binding dinamico e invocando quindi la definizione del metodo `visualizza` presente nella classe `Rettangolo` e non quello della classe `Figura`.

Al contrario, se si fosse utilizzato il **binding statico** allora l'associazione del nome del metodo alla sua definizione appropriata sarebbe avvenuta in fase di compilazione, e, nel caso precedente, il metodo `centra` invocato dall'oggetto `r` avrebbe invocato il metodo `visualizza` definito in `Figura`.

Fortunatamente, Java utilizza il binding dinamico.

Se si aggiunge il modificatore `final` alla definizione di un metodo, si impedisce che il metodo venga ridefinito nelle classi derivate.

Se si aggiunge il modificatore `final` alla definizione di una classe , si impedisce che la classe possa essere utilizzata come classe base per la definizione di classi derivate.

Infine, Java non utilizza il binding dinamico per i metodi privati, i metodi `final` e i metodi statici.

Downcast e upcast

Il tipo di una variabile determina quali metodi possono essere invocati usando la variabile, ma l'oggetto referenziato dalla variabile determina l'esatta definizione di metodo da eseguire.

Il tipo di un parametro determina quali metodi possono essere invocati utilizzando il parametro, ma l'argomento determina l'esatta definizione del metodo da eseguire.

Assegnare un oggetto di una classe derivata a una variabile del tipo della classe base (o una qualsiasi superclasse) è spesso chiamato **upcast**.

```
\\ Upcast
Cane c = new Cane("Fido");
Animale a;
a = (Animale)c;
```

La conversione da una classe base a una classe derivata (o da una superclasse a una sottoclasse) viene chiamata **downcast**.

```
\\ Downcast
Animale a = new Animale();
Cane c;
c = (Cane)a;
```

Se con un upcast non ci sono problemi, non è sempre sensato fare un downcast di un oggetto.
Infatti, il problema più comune è quello che la classe derivata abbia una variabile di istanza che una classe base non ha, quindi la classe base non può essere del tipo della classe derivata.

Uno dei casi in cui è necessario fare il downcast è quando viene definito il metodo `equals` di una classe:

```
Object oggetto = new Object();
Cane c = (Cane)oggetto;
```

Senza questo downcast le variabili di istanza della classe `Cane` sarebbero utilizzate illegalmente nell'istruzione di `return` del metodo `equals`, proprio perché la classe `Object` non ha le variaibli di istanza della classe `Cane`.

L'operatore `instanceof` controlla se un oggetto è del tipo specificato come secondo argomento dell'operatore. La sintassi è:

```
oggetto instanceof nome_di_una_classe
```

L'espressione restituisce `true` se *oggetto* è di tipo *nome_di_una_classe*, altrimenti restituisce `false`.

Classi astratte

Una classe è definita **astratta** se almeno uno dei suoi metodi non è completamente definito.

Non è possibile creare oggetti utilizzando i costruttori di una classe astratta, ma è possibile utilizzare una classe astratta come classe base per la definizione di classi derivate.

La classe derivata deve definire tutti i metodi ereditati dalla classe base astratta, altrimenti anche la classe derivata dovrà essere dichiarata astratta.

Se si definiscono tutti i metodi in una classe, ma non si vuole che si creino oggetti da quella classe, basta definire la classe come astratta.
È sempre possibile però dichiarare una variabile del tipo di una classe astratta e assegnarli un oggetto di una qualsiasi delle sue classi discendenti.

Quando viene invocato un metodo ridefinito, il metodo che verrà eseguito è quello definito nella classe usata per creare l'oggetto usando l'operatore `new`.
Questo perché Java utilizza il binding dinamico.

La sintassi per classi e metodi astratti è la seguente:

```
public abstract class nome_classe {

public abstract tipo_di_ritorno nome_metodo (parametri_formali);

}
```

Esempio di programmazione.
La classe `Leone` non viene presentata, poiché è sostanzialmente simile alla classe `Gatto`.

```
/**
Una generica classe Animale. Un animale ha un nome.
I comportamenti che un qualsiasi animale ha sono: dormi e parla.
*/

public abstract class Animale {
    private String nome;

    public Animale() {
        this(null);
    }

    public Animale(String nome) {
        this.nome = nome;
    }

    public abstract void parla();

    public abstract void dormi();

    public String getNome() {
        return this.nome;
    }

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

```java
/**
Un felino è un particolare tipo di animale.
La classe definisce il metodo dormi.
*/

public abstract class Felino extends Animale {

    public Felino() {
        super();
    }

    public Felino(String nome);
        super(nome);
    }

    public void dormi() {
        System.out.println("Ronf...");
    }
}



/**
Un gatto è un particolare tipo di felino.
La classe definisce il metodo  parla  e un ulteriore metodo faiLeFusa.
*/
public class Gatto extends Felino {

    public Gatto() {
        super();
    }

    public Gatto(String nome) {
        super(nome);
    }

    public void parla() {
        System.out.println("Miao");
    }

    public void faiLeFusa() {
        System.out.println("Prrrr");
    }
}
```

Interface Java

Un'interfaccia Java inizia come una definizione di classe, tranne per il fatto che utilizza la parola riservata `interface` al posto di `class`.

Un'interfaccia non dichiara alcun costruttore, i suoi metodi devono essere pubblici e può anche definire un numero qualsiasi di costanti pubbliche.

Essa non contiene, tuttavia, variabili di istanza o definizioni complete di metodo.
Una definizione di interfaccia inizia quindi con:

Sintassi

```
public interface nome_interfaccia {

    definizioni_di_costanti_pubbliche
    ...
    intestazione_del_metodo_pubblico_1;
    ...
    intestazione_del_metodo_pubblico_n;

}
```

Esempio

```
/**
    Un'interfaccia di metodi statici per convertire misurazioni da piedi a pollici e
    viceversa.
*/

public interface Convertibile {

    public static final int POLLICI_PER_PIEDE = 12;

    public static double convertiInPollici(double piedi);
    public static double convertiInPiedi(double pollici);

}
```

Una classe che implementa un'interfaccia deve definire un corpo per ogni metodo specificato nell'interfaccia.
Se non definisce il corpo di tutti i metodi, deve essere dichiarata astratta.
La classe potrebbe anche definire metodi non dichiarati nell'interfaccia.
Inoltre, una classe può implementare più di un'interfaccia.
Per implementare un'interfaccia, si devono fare due cose.

1. Includere l'espressione

   ```
   implements nome_interfaccia
   ```

   all'inizio della definizione di classe. Per implementare più di un'interfaccia, è sufficiente elencare il nome di tutte le interfacce, separati da una virgola, come segue:

   ```
   implements MiaInterfaccia, TuaInterfaccia
   ```

2. Definire ogni metodo dichiarato nell'interfaccia (o nelle interfacce) per creare una classe concreta. In caso contrario, sarà una classe astratta.

Esempio di programmazione

```
/**
   Un'interfaccia che contiene metodi che calcolano
   e restituiscono il perimetro e l'area di un oggetto
*/

public interface Misurabile {

   /**
   Restituisce il perimetro.
   */
   public double getPerimetro();

   /**
   Restituisce l'area.
   */
   public double getArea();

}
```

```java
/** Una classe che rappresenta quadrati */

public class Quadrato implements Misurabile {

    private double lato;

    public Quadrato(double lato) {
        this.lato = lato;
    }

    public double getArea() {
        return lato * lato;
    }

    public double getPerimetro() {
        return lato * 4;
    }

}


/** Una classe che rappresenta cerchi */

public class Cerchio implements Misurabile {

    private double raggio;

    public Cerchio(double raggio) {
        this.raggio = raggio;
    }

    public double getPerimetro() {
        return 2 * Math.PI * raggio;
    }

    public double getCirconferenza() {
        return getPerimetro();
    }

    public double getArea() {
        return Math.PI * raggio * raggio;
    }

}
```

Un'interfaccia è un tipo riferimento, quindi è possibile scrivere un metodo che ha un parametro di un tipo d'interfaccia, per esempio di tipo `Misurabile`

```
public static void visualizza(Misurabile figura) {
    double perimetro = figura.getPerimetro();
    double area = figura.getArea();
    System.out.printf("Perimetro: %f, Area: %f\n", perimetro, area);
}
```

Il programma può richiamare questo metodo passandogli un oggetto di una qualsiasi classe che implementi l'interfaccia `Misurabile`, per esempio

```
Misurabile scatola = new Quadrato(5.0);
Misurabile disco = new Cerchio(5.0);
```

Bisogna però prestare attenzione: è il tipo di una variabile che determina i nomi dei metodi che si possono utilizzare, mentre il tipo dell'oggetto referenziato dalla variabile determina l'esatta definizione del metodo.

Quindi se la variabile `disco` richiama il metodo `getCirconferenza` della classe `Cerchio`, il compilatore non sa che esiste, perché l'interfaccia `Misurabile` non conosce nessun metodo `getCirconferenza`.

È possibile definire una nuova interfaccia che **estende** (*extends*) un'interfaccia già esistente, creando un'interfaccia formata dall'insieme di metodi da lei definiti e quelli "ereditati".
Per esempio, si considerino le classi di animali presentate all'inizio di questo paragrafo e la seguente interfaccia

```
public interface Nominabile {
    public void setNome(String nomeAnimale);
    public String getNome();
}
```

Si può estendere `Nominabile` per creare l'interfaccia `Chiamabile`

```
public interface Chiamabile extends Nominabile {
    public void vieni(String nomeAnimale);
}
```

Una classe concreta che implementa `Chiamabile` deve implementare i metodi `vieni`, `setNome` e `getNome`.
Si possono anche combinare più interfacce a formare una nuova interfaccia. Per esempio, si supponga che, in aggiunta alle precedenti due interfacce, si definiscano le seguenti interfacce

```
public interface Capace {
    public void ascolta();
    public String rispondi();
}

public interface Ammaestrabile extends Chiamabile, Capace {
    public void siedi();
    public String parla();
    public void sdraiati();
}
```

Una classe concreta che implementa `Ammaestrabile` deve implementare i metodi `setNome`, `getNome`, `vieni`, `ascolta` e `rispondi` e anche i metodi `siedi`, `parla` e `sdraiati`.

ArrayList e generici

ArrayList

Un `ArrayList` è una **struttura dati dinamica**, ciò vuol dire che le sue dimensioni aumentano o diminuiscono dinamicamente durante l'esecuzione del programma.

Dinamicamente significa senza l'intervento diretto del programmatore.

La classe `ArrayList` si trova nel package `java.util` e amplia le capacità di un array.

Per importare la classe `ArrayList` bisogna includere la seguente istruzione all'inizio di ogni file:

```
import java.util.ArrayList
```

e si crea un istanza di `ArrayList` come un oggetto di una qualsiasi altra classe, ad eccezione del fatto che bisogna specificare il tipo base

```
ArrayList<String> lista = new ArrayList<String>();
```

In questo caso il tipo base è `String` e la lista potrà memorizzare istanze della sola classe `String`.

Il tipo base di un `ArrayList` deve essere sempre una classe. Non si possono utilizzare tipi primitivi come `int` o `double`.

L'unica soluzione è quella di utilizzare le classi wrapper per i tipi primitivi, come `Integer`, che alla fine non sono altro che classi.

La **capacità iniziale** di un `ArrayList` indica quanti elementi può inizialmente contenere la lista.

Il costruttore di `ArrayList` specifica il numero totale degli elementi all'atto della creazione dell'oggetto di tipo `ArrayList`.

```
ArrayList<String> lista = new ArrayList<String>(20);
```

crea un'istanza della classe `ArrayList` avente capacità iniziale di 20 elementi.

Ogni volta che si riempirà tutta la capacità iniziale, la lista aumenterà la sua dimensione per far posto a un elemento, quindi, se si riempiranno tutti e i venti elementi della lista, la sua dimensione aumenterà dinamicamente a 21.

Se il costruttore di `ArrayList` non specifica nessun intero, la capacità iniziale della lista sarà fissata a 10 elementi.

Gli indici di una lista sono come gli indici degli array, quindi vanno da 0 a $n-1$, dove n è la capacità iniziale fornita come argomento al costruttore della lista.

Per aggiungere un elemento alla lista si usa il metodo `add` di `ArrayList`

```
lista.add("Pasta");
```

Il metodo `add` aggiunge l'elemento `"Pasta"` alla fine della lista e ne incrementa la dimensione di una unità.

Se si vuole aggiungere l'elemento a una specifica posizione, si utilizza il metodo `add` in questo modo

```
lista.add(1, "Pasta");
```

In questo caso l'elemento `"Pasta"` verrà aggiunto all'indice `1` della lista e tutti gli elementi che seguono l'indice `1` verranno spostati di una unità e se necessario la dimensione della lista verrà anch'essa aumentata di una unità.

Altri metodi della classe `ArrayList` sono

- `get(indice)` restituisce l'elemento all'indice indicato come argomento al metodo
- `set(indice, tipo_base nuovoElemento)` sostituisce un elemento **esistente** con `nuovoElemento` alla posizione indicata da `indice` e restituisce l'elemento sostituito
- `remove(indice)` rimuove l'elemento alla posizione specificata da `indice`
- `size()` restituisce il numero di elementi presenti nella lista
- `clear()` rimuove tutti gli elementi presenti nella lista

I metodi elencati sopra sono alcuni dei metodi della classe `ArrayList`.

Esempio di programmazione

```
import java.util.ArrayList;
import java.util.Scanner;

/* Inserisce elementi in una lista e ne stampa il contenuto */

public class ArrayListDemo {

    public static void main(String[] args) {

        Scanner s = new Scanner(System.in);

        ArrayList<String> lista = new ArrayList<String>();

        int i = 0;
        while (i < 10) {

            System.out.print("\nElemento da aggiungere alla lista: ");
            String elemento = s.nextLine();

            lista.add(elemento);

            i++;
        }

        for (String elemento : lista)
            System.out.println(elemento);

    }
}
```

Generici

Classi parametriche e tipi di dato generico

Una **classe parametrica** (*parameterized class*) è una classe avente un *tipo_base*, che può essere sostituito con qualsiasi tipo classe per ottenere una classe parametrica che memorizza oggetti di tipo *tipo_base*.

tipo_base è anche detto **tipo di dato generico** (*generic data type*).

`ArrayList` è un esempio di classe parametrica.

I parametri per i tipi di dato *tipo_base* sono detti **generici** (*generics*).

Il **tipo parametrico** (*type parameter*) specifica il tipo di classe usata dal programmatore per le sue esigenze nel trattare uno specifico tipo di dato.

I tipi parametrici possono essere applicati sia alle classi che ai metodi.

Esempio classe parametrica

```
public class Esempio<T> {

    private T dati;

    public Esempio() {
        dati = null;
    }

    public Esempio(T dati) {
        this.dati = dati;
    }

    public void setDati(T dati) {
        this.dati = dati;
    }

    public T getDati() {
        return dati;
    }
}
```

Come si nota dall'esempio, le dichiarazione dei costruttori di una classe parametrica non includono il tipo parametrico tra parentesi angolari.

Il tipo `T` è il tipo parametrico usato per la variabili d'istanza `dati`, il parametro `dati` del metodi `setDati` e come valore di ritorno del metodo `getDati` della classe `Esempio`.

Con un tipo parametrico non è possibile fare la seguente

```
dati = new T();
```

oppure

```
T[] unArray; // Legale

unArray = new T[20]; // Illegale
```

A partire da Java 7 è possibile non specificare il tipo parametrico in questo modo

```
Classe<TipoBase> oggetto = new Classe<>();
```

ma siccome il vecchio formato che indica in entrambi le parentesi angolari il tipo parametrico è usato da molto tempo, molti definizione di classi usano ancora il vecchio formato.

Ricapitolando, istanziare un oggetto di classe `Esempio` può avvenire nel seguente modo

```
Esempio<String> oggetto = new Esempio<String>();
```

Il metodo `setDati` viene invocato come qualsiasi altro metodo di istanza

```
oggetto.setDati("Ciao");
```

È possibile inoltre specificare più di un tipo parametrico in una definizione di classe

`Coppia<T>` ha un solo tipo parametrico

`Coppia<T1, T2>` ha due tipi parametrici

È possibile imporre dei vincoli sui tipi parametrici di una classe.

Per esempio, se si vuole che il tipo parametrico di una classe implementi solo l'interfaccia `Comparable`, si inizia la definizione della classe come segue:

```
public class Coppia<T extends Comparable>
```

La parte `extends Comparable` è chiamata **vincolo** sul tipo parametrico `T`.

Se si cerca di sostituire a `T` un tipo che non implementa la l'interfaccia `Comparable`, si otterrà un errore in fase di compilazione.

È necessario utilizzare la parola chiave `extends` e non `implements` come ci si potrebbe aspettare.

In linea generale, è possibile specificare in un vincolo al massimo una classe e una o più interfacce con la seguente sintassi

```
public class NomeClasse <Tipo extends ClasseBase & Interfaccia1 & Interfaccia2 & ...
& UltimaInterfaccia>
```

La classe base deve essere sempre specificata per prima e se si specificano una o più interfacce, quest'ultime sono elencate in successione con un ampersand (&).

Se ci sono più tipi parametrici, vanno separati da virgole

```
public class Coppia<T1 extends Docente & Comparable, T2 Studente & Comparable>
```

Anche un'interfaccia può avere uno o più tipi parametrici.
I dettagli e la notazione da utilizzare sono gli stessi visti per le classi con tipi parametrici.

Metodi generici

È possibile definire all'interno di una classe generica dei metodi generici i cui tipi parametrici differiscono da quello specificato nella classe

```
public class Esempio<T> {

    private T dati;

    public Esempio(T dati) {
        this.dati = dati;
    }

    public <TipoVisualizzatore> void mostraA(TipoVisualizzatore visualizzatore) {
        System.out.println("Ciao " + visualizzatore);
        System.out.println("I dati sono " + dati);
    }

    ...
}
```

È anche possibile definire dei metodi generici all'interno di classi non generiche

```
public class MetodiUtili {
    ...

    public static <T> T getPuntoMedio(T[] a) {
        return a[a.length / 2];
    }

    public static <T> T getPrimo(T[] a) {
        return a[0];
    }

    ...
}
```

Si noti che il tipo parametrico tra parenetesi angolari, <T>, è posto dopo tutti i modificatori (in questo caso, `public` e `static`) e prima del tipo di ritorno.

L'invocazione di un metodo generico deve precisare il tipo parametrico tra parentesi angolari in questo modo

```
String puntoMedio = MetodiUtili.<String>getPuntoMedio(b);
double primoNumero = MetodiUtili.<Double>getPrimo(c);
```

Si noti che il punto è prima della specifica del tipo tra parentesi angolari: il tipo fa parte del nome del metodo, non di quello della classe.

Di seguito si mostra il metodo della classe `Esempio`

```
Esempio<Integer> oggetto = new Esempio<Integer>(42);
oggetto.<String>mostraA("Amico");
```

il risultato prodotto

```
Ciao Amico
I dati sono 42
```

Ereditarietà con classi generiche

Una classe generica può estendere una classe ordinaria o un'altra classe generica.

Il seguente esempio mostra una classe generica `CoppiaNonOrdinata` che estende la classe generica `Coppia`

```
public class Coppia<T> {
    private T primo;
    private T secondo;

    public Coppia() {
        primo = null;
        secondo = null;
    }

    public Coppia(T primo, T secondo) {
        this.primo = primo;
        this.secondo = secondo;
    }

    public void setPrimo(T primo) {
        this.primo = primo;
    }

    public void setSecondo(T secondo) {
        this.secondo = secondo;
    }

    public T getPrimo() {
        return primo;
    }

    public T getSecondo() {
        return secondo;
    }

    public String toString() {
        return (String.format("primo: %s, secondo: %s", primo.toString(),
        secondo.toString()));
}
```

```
public class CoppiaNonOrdinata<T> extends Coppia<T> {

   public CoppiaNonOrdinata() {
      setPrimo(null);
      setSecondo(null);
   }

   public CoppiaNonOrdinata(T primo, T secondo) {
      setPrimo(primo);
      setSecondo(secondo);
   }

   public boolean equals(Object altroOggetto) {
      if (altroOggetto == null)
         return false;
      else if (getClass() != altroOggetto.getClass())
         return false;
      else {
         CoppiaNonOrdinata<T> altraCoppia = (CoppiaNonOrdinata <T>)altroOggetto;
         return (getPrimo().equals(altraCoppia.getPrimo())
            && getSecondo().equals(altraCoppia.getSecondo()))
            ||
            (getPrimo().equals(altraCoppia.getSecondo()) &&
            getSecondo().equals(altraCoppia.getPrimo()));
      }
   }
}
```

```
public class CoppiaNonOrdinataDemo {

   public static void main(String[] args) {

      CoppiaNonOrdinata<String> p1 = CoppiaNonOrdinata<String>("noccioline", "birra");
      CoppiaNonOrdinata<String> p2 = CoppiaNonOrdinata<String>("birra", "noccioline");

      if (p1.equals(p2)) {
         System.out.println(String.format("%s e %s è lo stesso di %s e %s",
         p1.getPrimo(), p1.getSecondo(), p2.getPrimo(), p2.getSecondo()));
      }
   }
}
```