Appunti di Programmazione 2

Domande e Risposte

Capitolo 6: Ereditarietà

1) Cosa si intende per ereditarietà?

Rappresenta una delle tre principali caratteristiche della programmazione orientata agli oggetti e consente di creare nuove classi che riutilizzano, estendono e modificano il comportamento definito in altre classi (viene utilizzata per evitare la duplicazione del codice).

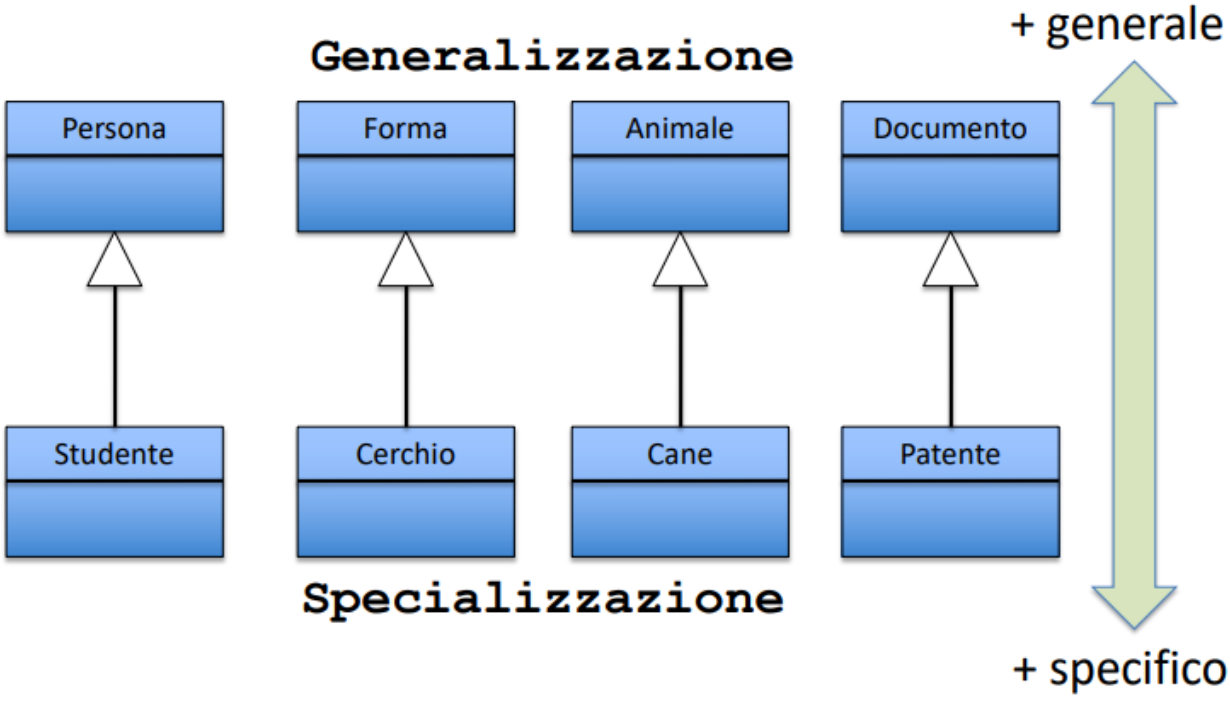
Esempio:

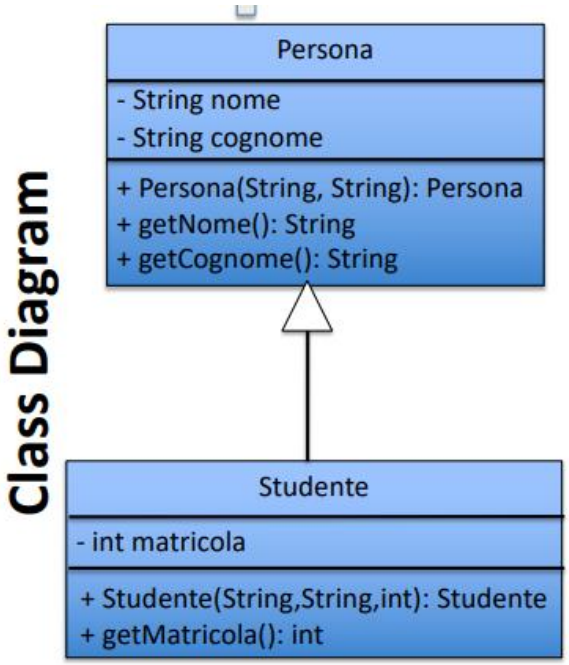
Cosa hanno in comune Persona e Studente?

Studente è (anche) una Persona,
meglio ancora Studente è un **tipo** di Persona

*relazione **is-a***

UML





Codice

```java
public class Persona {
	private String nome;
	private String cognome;

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

	public String getNome() {
		return nome;
	}
	public String getCognome(){
		return cognome;
	}
}
```

```java
public class Studente extends Persona {
	private int matricola;

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

	public int getMatricola() {
		return matricola;
	}
}
```

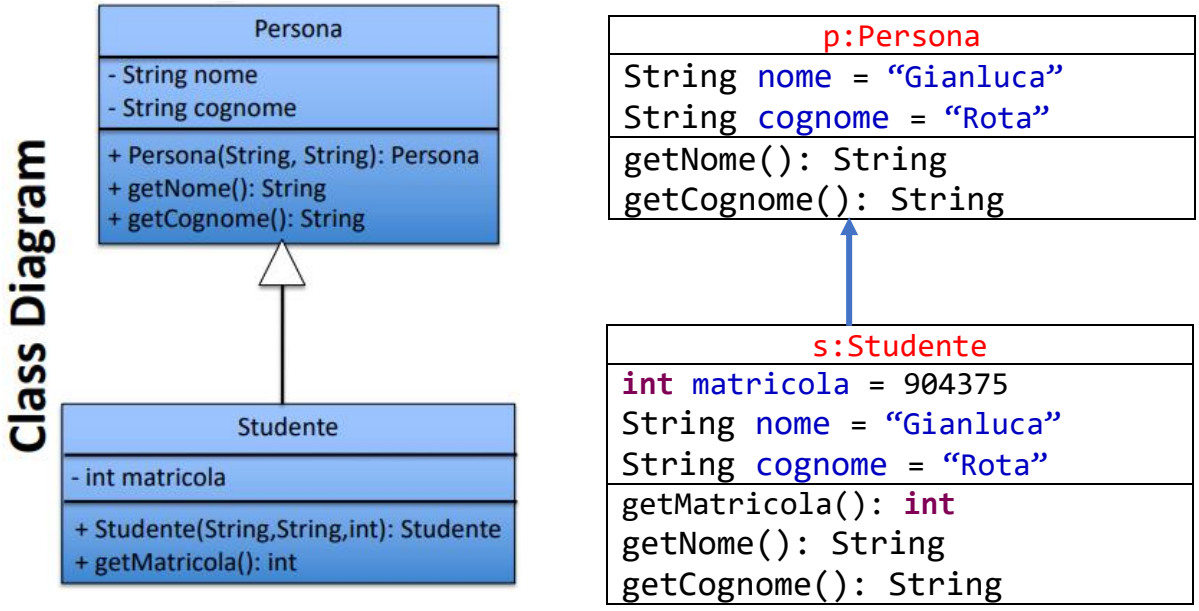
2) Qual è il costrutto che permette di implementare is - a?

Il costrutto Java che permette di implementare correttamente la relazione is - a è "Extend". Con extends la classe Studente "eredita" il codice della classe Persona.

Il seguente frammento di codice è corretto:

```
Studente s = new Studente("Gianluca","Rota",904375);
int matricola = s.getMatricola();
String nome = s.getNome();
String cognome = s.getCognome();
```

Così si ha:



→ Permette inoltre di dichiarare una classe come Classe Derivata di un'altra classe (Classe Base).

```
public class Studente extends Persona
```

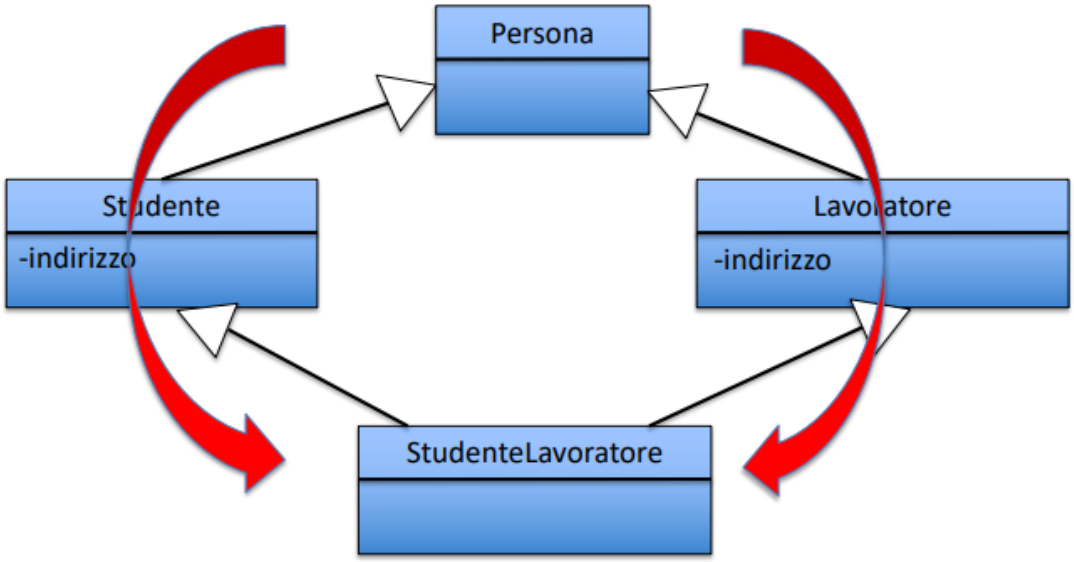
Nota: terminologie equivalenti

- ClasseDerivata `extends` ClasseBase…
- SottoClasse (o SubClass) `extends` SuperClasse (o SuperClass)
- ClasseFiglia (Classe Figlio) `extends` ClasseMadre (ClassePadre)

3) Definizione alternativa di Ereditarietà

È una relazione fra classi in cui una classe (superclasse o classe base) "raccoglie a fattor comune" le caratteristiche comuni di una o più altre classi (sottoclassi o specializzazioni). Le sottoclassi ereditano tutte le caratteristiche della superclasse.

4) Ereditarietà Multipla: è permessa?



L'ereditarietà multipla NON è permessa in Java.

```
class StudenteLavoratore extends Studente, Lavoratore {…
```

Non è una dichiarazione di classe legale.

5) Ereditarietà: motivi

Le quattro motivazioni dell'ereditarietà sono:

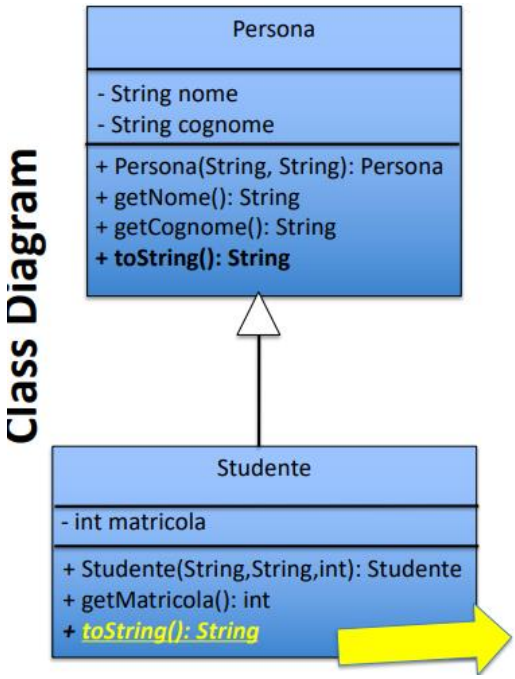
→ riuso: la superclasse può essere riusata in contesti diversi;

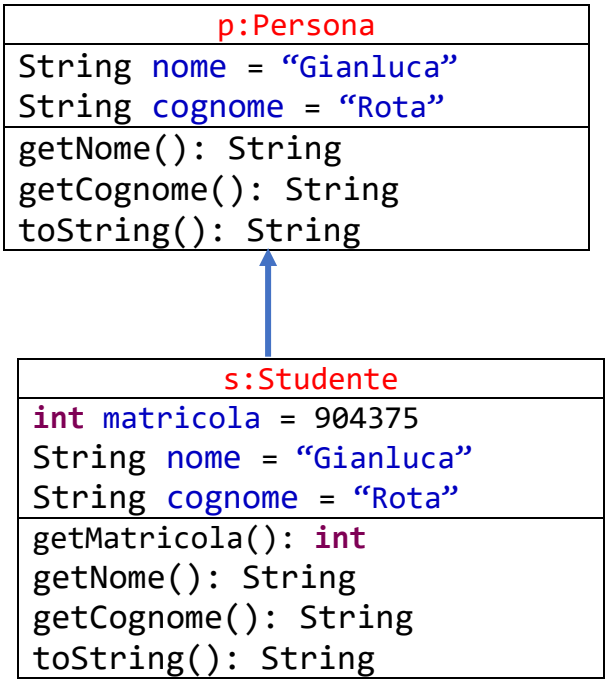
→ economia: tutti gli attributi, operazioni e associazioni comuni a un certo insieme di classi vengono scritte una sola volta;

→ consistenza: se un attributo, operazione o associazione della superclasse viene cambiato, la modifica si ripercuote automaticamente su tutte le sottoclassi;

→ estendibilità: è sempre possibile, anche in un secondo momento, aggiungere nuove sottoclassi senza dover riscrivere la parte comune contenuta nella superclasse.

6) Overriding dei Metodi





Codice

```java
public class Persona {
    private String nome;
    private String cognome;

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

    public String getNome() {
        return nome;
    }
    public String getCognome(){
        return cognome;
    }

    @Override
    public String toString() {
        return "Persona [nome = " + nome + ", cognome = " + cognome + "]";
    }
}
```

```java
public class Studente extends Persona {
    private int matricola;

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

    public int getMatricola() {
        return matricola;
    }

    @Override
    public String toString() {
        return "Studente [nome = " + super.getNome() +
                ", cognome = " + super.getCognome() +
                ", matricola=" + this.matricola + "]";
    }
}
```

L'utilizzo di `@Override` non è strettamente necessario ma permette al compilatore di verificare se quello che si ha scritto è effettivamente una ridefinizione di un metodo e migliora la documentazione del codice.

Con le seguenti istruzioni:

```java
Studente s = new Studente("Gianluca","Rota",904375);
System.out.println(s.toString());
```

Si ha in output

```
Studente [nome = Gianluca, cognome = Rota, matricola=904375]
```

Esempio di errore di compilazione

```
public class Studente extends Persona {
    private int matricola;

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

    public int getMatricola() {
        return matricola;
    }

    @Override
    public String toString() {
        return "Studente [nome = " + super.getNome() +
                ", cognome = " + super.getCognome() +
                ", matricola=" + this.matricola + "]";
    }

    @Override
    public String getNomeCompleto(){
        return "stud." + getNome() + " " + getCognome();
    }

}
```

Il metodo in oggetto non è stato ancora implementato, per cui viene rilevato l'errore di compilazione.

```java
public class Studente extends Persona {
    private int matricola;

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

    public int getMatricola() {
        return matricola;
    }

    @Override
    public String toString() {
        return "Studente [nome = " + super.getNome() +
                ", cognome = " + super.getCognome() +
                ", matricola=" + this.matricola + "]";
    }

    public String getNomeCompleto(){
        return "stud." + getNome() + " " + getCognome();
    }
}
```

6) Keyword super

La keyword super viene utilizzata per poter accedere ai metodi della classe base dalla classe derivata.

Esempio

```java
public class Studente extends Persona {
    private int matricola;

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

    public int getMatricola() {
        return matricola;
    }

    @Override
    public String toString() {
        return "Studente [nome = " + super.getNome() +
                ", cognome = " + super.getCognome() +
                ", matricola=" + this.matricola + "]";
    }

    public String getNomeCompleto(){
        return super.toString() + ", matricola = " + getMatricola();
    }
}
```

7) Differenza tra overloading e overriding

L'overloading è il meccanismo che permette di dotare più metodi aventi lo stesso nome ma parametri differenti, mentre l'overriding è il meccanismo che permette di ridefinire il comportamento di un metodo.

8) Keyword final

La keyword final permette di non ridefinire il metodo nelle classi derivate e se essa precede la definizione di una classe, essa non potrà essere estesa ad altre classi.

9) Costruttori: regole fondamentali

Le regole fondamentali per i costruttori della classe derivata sono:
1. I costruttori devono invocare un costruttore della classe base come prima istruzione;
2. La keyword per invocare un costruttore della classe base è super(…);
3. Se omessa si considera una invocazione implicita a super(), cioè al costruttore senza parametri della classe base;
4. Se nella classe base non esiste il costruttore senza parametri bisogna invocare un costruttore esplicitamente oppure sarà prodotto un errore di compilazione.

10) Classe Object

La classe Object è la superclasse, diretta o indiretta, di ciascuna classe in Java. Grazie al meccanismo dell'ereditarietà, i suoi metodi possono essere invocati su tutti gli oggetti. Restituisce una rappresentazione testuale dell'oggetto in forma di stringa: è molto utile ad esempio per le stampe.
Package:

- `java.lang.Object`
- E' il tipo più generale:
  - ogni classe "is-a" `Object`
  - ogni classe (implicitamente) estende `Object`

11) Alcuni metodi della classe Object

I metodi della classe Object sono:

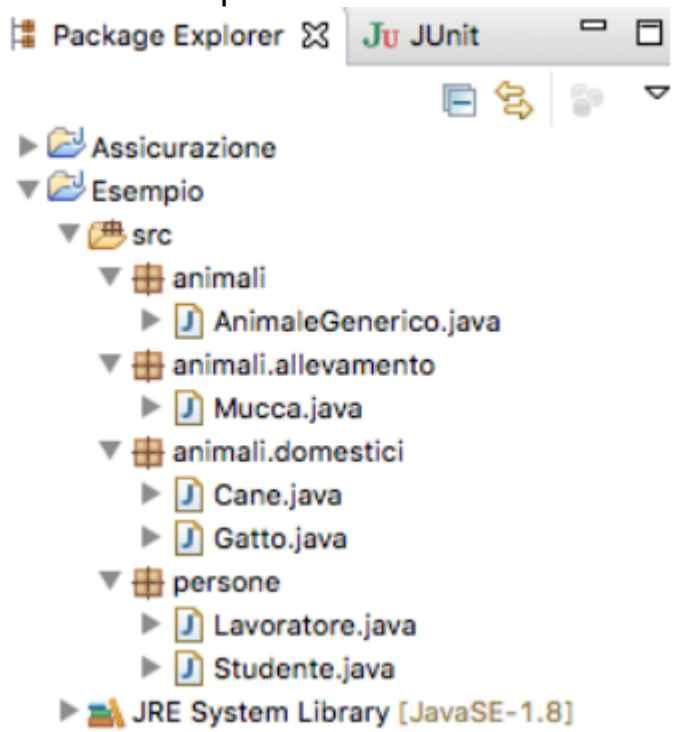
`public boolean equals(Object o)`
- Ritorna `true` se `this==o`

`public String toString()`
- Ritorna "nomeClasse@hashcode"

L'implementazione è troppo generica e questi metodi vengono tipicamente ridefiniti (override) nelle sottoclassi.

12) Java Package

Un package è una collezione di classi correlate a cui viene assegnato un nome.
Le caratteristiche del Java package sono:
→ astrazione similare alle cartelle del file system;
→ permette di organizzare le classi in insiemi logicamente correlati tra loro;
→ evita i conflitti tra i nomi delle classi;
→ una classe ha visibilità di tutte le classi che si trovano nello stesso package;
→ la visibilità può essere estesa tramite il comando di import.



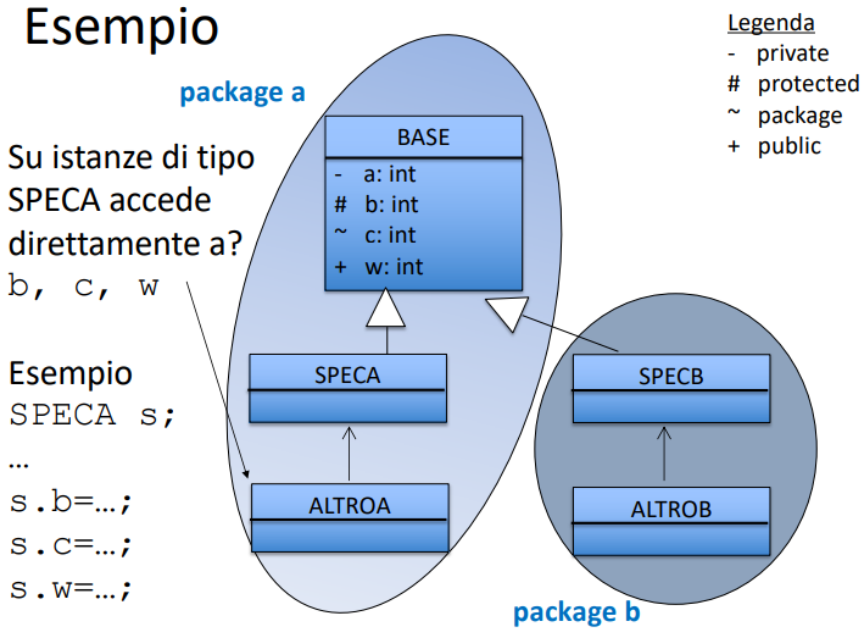
Nome completo di una classe
(fully qualified name):
`package.classe`

`animali.Animale`
`animali.domestici.Cane`

13) Visibilità attributi

Le visibilità degli attributi sono:
→ public: nessuna restrizione;
→ private: accessibili solo dalla classe di definizione;
→ protected: non accessibili dall'esterno fatta eccezione per le classi derivate e le classi nello stesso package;

→ package-wide: accessibili da tutte le classi nello stesso package.



Capitolo 7 – Polimorfismo e Binding Dinamico

1) Definizione e caratteristiche

Proprietà della OOP di comportarsi in modo diverso in base al contesto dell'esecuzione.

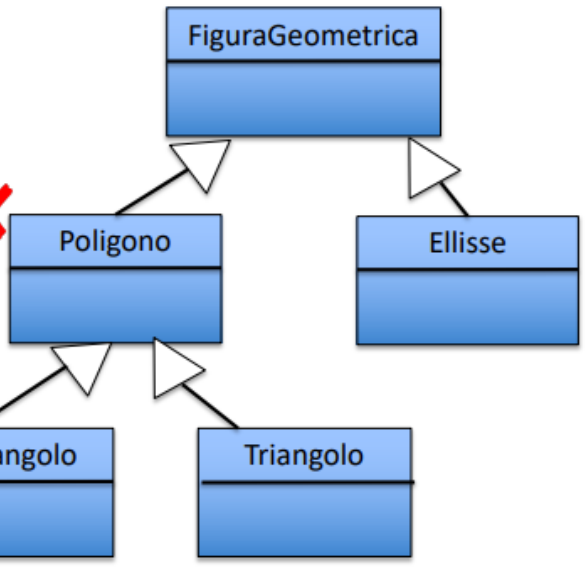
→ Polimorfismo tra reference: uso di reference per riferirsi a oggetti di tipo (dinamico) diverso da quello dichiarato nel codice (tipo statico, noto a compile-time);

→ Polimorfismo per inclusione: il tipo dinamico è limitato dalla corrispondenza tra tipi definita dalle gerarchie di ereditarietà.

I tipi dinamici per un reference sono SOLO i sotto-tipi del tipo statico.

Esempio:

TIPO STATICO		TIPO DINAMICO	
Poligono	p = new	Poligono();	✓
Poligono	p = new	Rettangolo();	✓
Poligono	p = new	FiguraGeometrica();	✗
FiguraGeometrica	f= new	Poligono();	✓
Poligono	p = new	Ellisse();	✗
FiguraGeometrica	f= new	Rettangolo();	✓
FiguraGeometrica	f= new	Ellisse();	✓
Rettangolo	r= new	Triangolo();	✗



Vantaggio polimorfismo: il codice scritto per un elemento della gerarchia può essere riutilizzato per tutti i sotto-tipi.

2) Binding dinamico

Il binding dinamico permette di definire e ridefinire un qualsiasi oggetto runtime. Il legame tra definizione e invocazione di un metodo non viene definito staticamente ma dinamicamente (durate l'esecuzione). Il compilatore conosce il tipo dinamico di una variabile solamente a run time mediante la generazione del codice. In Java il binding è dinamico.

3) Per quali metodi non esiste il binding dinamico?

Non si ha binding dinamico per metodi static (associati a classi e non ad oggetti) e metodi final (che non possono essere ridefiniti nelle sottoclassi).

4) Come risolvere gli overloading e overriding?

```
class Top {
	public String f(Object o) {return "Top";}
}

class Sub extends Top {
	public String f(String s) {return "Sub";}
	public String f(Object o) {return "SubObj";}
}


public class Esempio {
	public static void main(String[] args) {
		Sub sub = new Sub();
		Top top = sub;
		String str = "Something";
		Object obj = str;
		System.out.println(top.f(str));
	}
}
```

Risoluzione overloading (compile – time)

→ invocazione di un metodo con firma f(String) su un oggetto di tipo Top.
→ il tipo Top implementa un solo metodo con firma compatibile: f(Object);
→ viene decisa la firma del metodo (f(Object)).

Risoluzione overriding (run – time)

→ la firma è decisa, f(Object), bisogna decidere l'implementazione da eseguire;
→ il tipo dinamico di top è Sub;
→ l'implementazione di f(Object) è in sub e quindi viene stampato "SubObj".

4) Casting: UpCasting e DownCasting

In Java esistono due tipologie di casting: up – casting (type – safe) e down casting (type unsafe).

UpCasting: coercizione di un tipo specializzato in un tipo più generico (up, più in alto nella gerarchia) ed è sempre corretto e garantito dal compilatore. Dopo l'up-casting si possono invocare solo i metodi del tipo statico, ma i metodi che vengono eseguiti sono quelli del tipo dinamico.

DownCasting: coercizione di un tipo generico in un tipo più specializzato e deve essere dichiarato esplicitamante dal programmatore. Il compilatore "si fida" e nel caso di downcasting espliciti errati, potrenno verificarsi errori runtime.

L'uso di tipi incompatibili, all'interno o all'esterno della gerarchia, sono riconosciuti e segnalati dal compilatore.

```
public static void main(String [] args){
    Libro libro = new Libro(12.0," Dan Brown","Il Codice da Vinci"));
    CD cd = (CD) libro;
    Rettangolo r = (Rettangolo) libro;
}
```

Generano un errore a tempo di compilazione

5) InstanceOf e classe Object

→ L'operatore instanceOf permette di controllare il tipo dinamico associato ad un reference e ritorna true se il tipo dinamico è compatibile, altrimenti false.

```
<reference> instanceof <NomeClasse>
```

→ È possibile effettuare sia l'UpCasting esplicito e il DownCasting esplicito. Questi sono sempre accettati dal compilatore.

5) Metodo: getClass()

Il metodo restituisce una rappresentazione del tipo dinamico dell'oggetto e può essere confrontato con != e ==.

5) Equals e toString()

Tutte le classi ereditano equals solo che è necessario ridefinirle opportunamente.

```
public boolean equals(Object obj)
```

E' smpre inoltre possibile invocare il metodo toString(), dato che ogni oggetto è un Object.

```
public boolean equals(Object o){
  if(this == o) return true;
  if(o == null)return false;
  if(getClass() != o.getClass()) return false;
  Persona p = (Persona)o;
  if (nome.equals(p.getNome()) &&
  cognome.equals(p.getCognome())) {
      return true;
  } else {
      return false;
  }
 }
}
```

- stampa di un oggetto con `println()`
  - `Persona p = …`

    `System.out.println(p);`

    Equivalente a System.out.println(p.toString());
- concatenazione di stringhe
  - `Persona p = …`

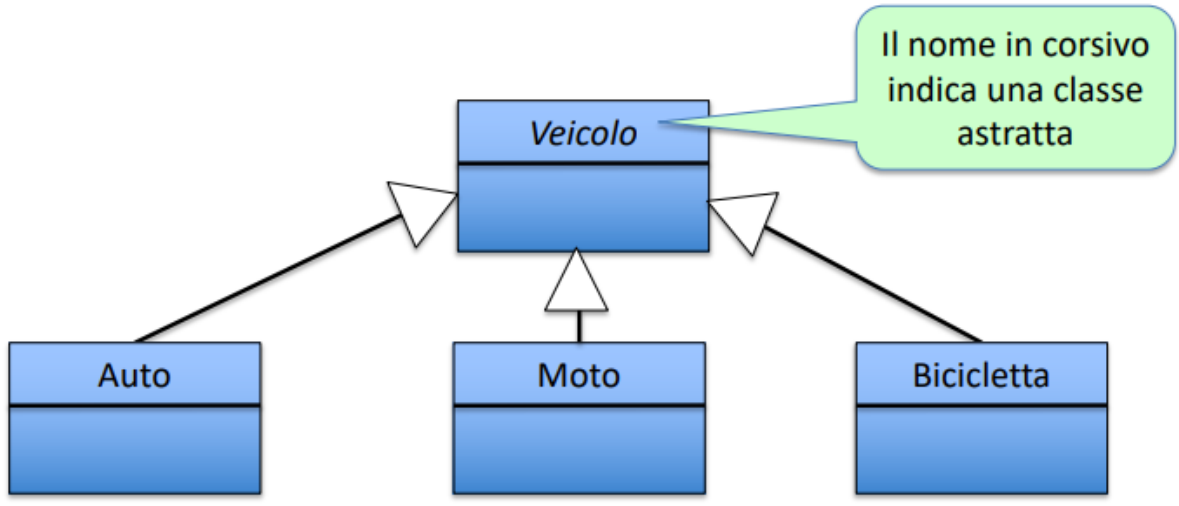
    `String str = "Persona=" + p;`

    Equivalente a "Persona=" + p.toString();

Capitolo 8 - Classi Astratte e Interfacce

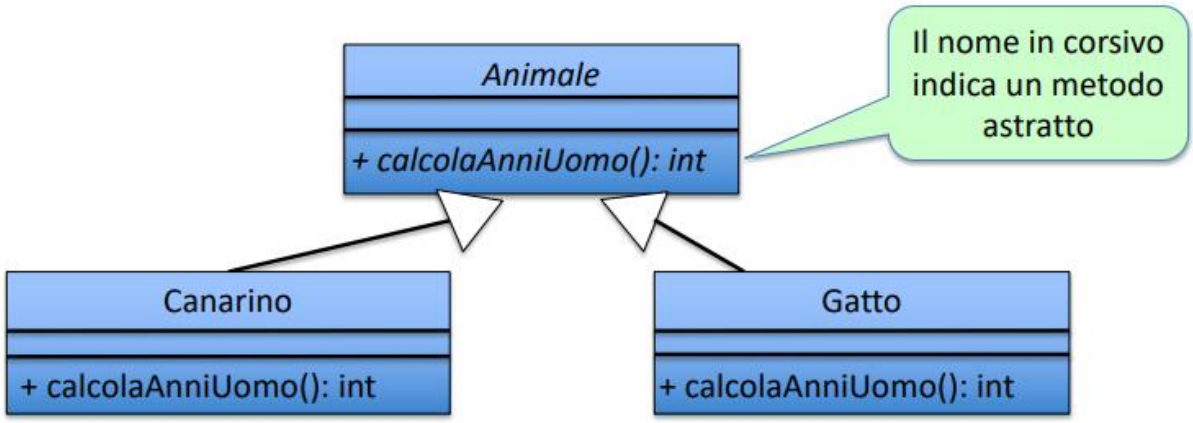
1) Classe Astratta

→ La classe astratta è una classe che non rappresenta un concetto concreto ma un dominio applicativo e non può essere istanziata.



2) Metodi Astratti

Il metodo astratto è un metodo privo di implementazione, non invocabile che può essere definito solamente nella classe astratta. E' utile per imporre l'implementazione comune a tutte le sottoclassi di cui non si può dare l'implementazione nella super classe.



3) Interfaccia

L'interfaccia è l'insieme delle operazioni e dei dati pubblici di una classe che specifica "COSA FA": definisce i comportamenti utilizzabili dalle altre parti del programma. In Java viene utilizzata la keyword interface e informalmente è una classe completamente astratta.

```
public interface Ordina{
   public void ordina();
}
```

```
public interface IStack {
    public int size();
    public boolean isEmpty();
    public Object top();
    public void push(Object elt);
    public Object pop();
}
```

Definisce una interfaccia e non una classe

L'implementazione dei metodi non è presente

```
public class ArrayStack implements IStack {
    private Object items[];
    private int top = -1;

    public ArrayStack(int maxSize) {
        items = new Object[maxSize];
    }
    public int size(){ return top + 1; }
    public boolean isEmpty(){ return top == -1; }
    public void push(Object o){ items[++top] = o; }
    public Object top(){ return items[top]; }
    public Object pop(){ return items[top--]; }
    public void printStack {…}
}
```

```
public interface IStack {
    public int size();
    public boolean isEmpty();
    public Object top();
    public void push(Object elt);
    public Object pop();
}
```

Nelle interfacce è possibile dichiarare attributi che sono:
→ public: pubblicamente accessibili;
→ static: sono attributi dell'interfaccia e non delle istanze;
→ final: costanti.

```
public interface IOrdinamentoArray {
  int maxElem=100;

  public void setArray(int[] numeri);
  public int[] getArray();
  public void ordina();
}
```

```
public interface IOrdinamentoArray {
    public final static int maxElem=100;

    public void setArray(int[] numeri);
    public int[] getArray();
    public void ordina();
}
```

Capitolo 9 – Gestione delle eccezioni

1) Definizione: eccezioni

Le eccezioni sono lo strumento messo a disposizione da Java per gestire in modo ordinato le situazioni anomale.

2) Caratteristiche delle eccezioni

Le eccezioni sono costituite da un tipo e dei dati associati che danno indicazione sul problema incontrato. Possono essere definite dall'utente (chiamate eccezioni personalizzate).

3) Come vengono sollevate le eccezioni?

Mediante la keyword throw con un reference a un oggetto eccezione di seguito. L'oggetto eccezione ha metodi e dati che determina il tipo dell'eccezione e includono informazioni sul problema riscontrato. L'effetto della sollevazione

dell'eccezione è la terminazione del blocco di codice e propagazione dell'eccezione al chiamante.

4) Come vengono gestite le eccezioni?

Le eccezioni vengono gestite mediante il blocco try – catch.

```
try {
    <blocco di codice che potrebbe sollevare eccezione>
} catch(TipoEccezione e) {
    <codice di gestione: da eseguire quando si verifica l'eccezione>
}
```

Il ramo catch può gestire un'eccezione di tipo T se T è di tipo Ex o se T è un sottotipo di Ex. Se l'eccezione viene gestita da un blocco try catch, successivamente all'esecuzione del codice del blocco, il programma continua dal comando successivo al blocco stesso.

5) Tipi di eccezioni

Le eccezioni controllate sono dovute a circostanze esterne che anche il più attento dei programmatori non può escludere semplicemente scrivendo codice robusto. Le eccezioni non controllate sono dovute a errori fatali che non ha senso prevedere nel codice.

6) E' possibile definire una nuova eccezioni?

Si, mediante l'ereditarietà in cui attraverso l'ereditarietà la classe estende un elemento della gerarchia delle eccezioni, chiamata Exception.

Esempio: `NonPositiveBaseException` invece di `NonPositiveBase`

7) Ramo finally

Ramo finale del blocco try catch che viene comunque eseguito a seconda se viene sollevata o meno l'eccezione.

Il blocco ha la seguente struttura:

```
try {
    …
    //blocca accesso alla stampante
    //stampa
} catch(Exception1 e) {
    //gestione dell'eccezione Exception1
} catch(Exception2 e) {
    //gestione dell'eccezione Exception2
} finally {
    …//libera accesso alla stampante
}
```

8) Quali sono le possibili strategie di gestione delle eccezioni?

Esse sono:

→ Masking: viene gestita l'eccezione e l'esecuzione prosegue normalmente

→ Forwarding: l'eccezione non può essere gestita, si prosegue quindi ritornando la stessa eccezione

→ Re – throwing: l'eccezione viene catturata ma non può essere gestita (oppure lo è solo parzialmente), si prosegue quindi con la generazione di una eccezione di tipo differente.

Capitolo 10 – Collection Frameworks

1) Wrapper: definizione

Un wrapper di tipo primitivo è un oggetto che incapsula un attributo di tipo primitivo, che ha le seguenti caratteristiche:

→ stesso comportamento del tipo primitivo;

→ in aggiunta può essere usato come un Object;

→ il compilatore trasforma in automatico i tipi primitivi in tipi wrapper e viceversa.

Tipo Primitivo	Tipo Wrapper
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

2) Collection framework

Una collezione (o container) è un elemento "contenitore" di oggetti denominati elementi. La Java Collection Framework è un insieme di interfacce e di classi per l'implementazione di collezioni.

Vi sono diverse interfacce disponibili:

→ Collection (Collection<E>): collezioni di elementi di tipo E;

→ Set: Collection che non ammette duplicati;
→ List: è una Collection con elementi ordinati (gli elementi hanno una posizione individuata da un indice intero);
→ Map: la cui dichiarazione usa Map<K,V>, tale che K è la chiave e V rappresenta il valore memorizzata nella chiave. Una chiave può occorrere al più una volta.
→ SortedSet: è un Set totalmente ordinato e viene individuata la duplicazione eseguendo il metodo equals.
→ SortedMap: è una Map con chiavi totalmente ordinate.

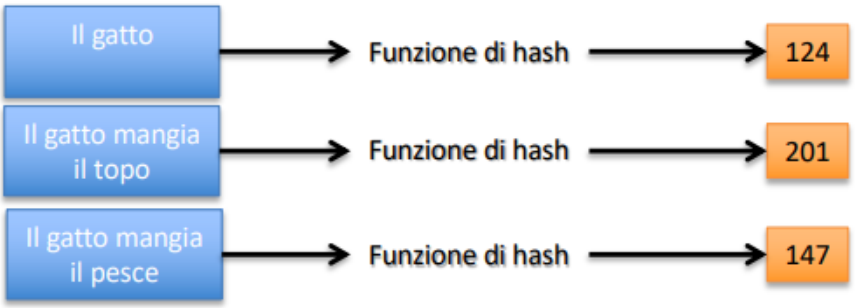
3) Come viene effettuato l'ordinamento

Gli oggetti di una classe possono essere ordinati se la classe implementa l'interfaccia Comparable, la quale predispone di compareTo.
Il metodo è un intero che ritorna:

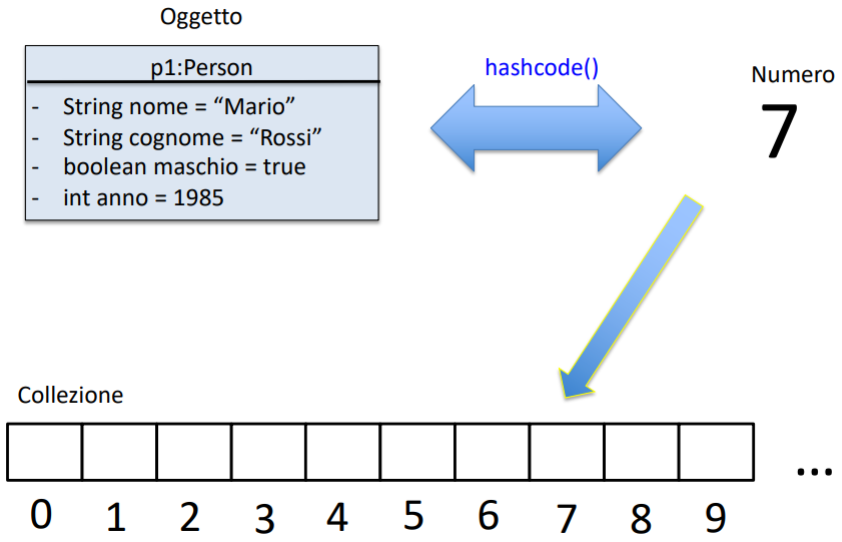
0	Se **o1** e **o2** sono 'uguali'
Intero negativo	Se **o1** è più piccolo di **o2**
Intero positivo	Se **o1** è più grande di **o2**

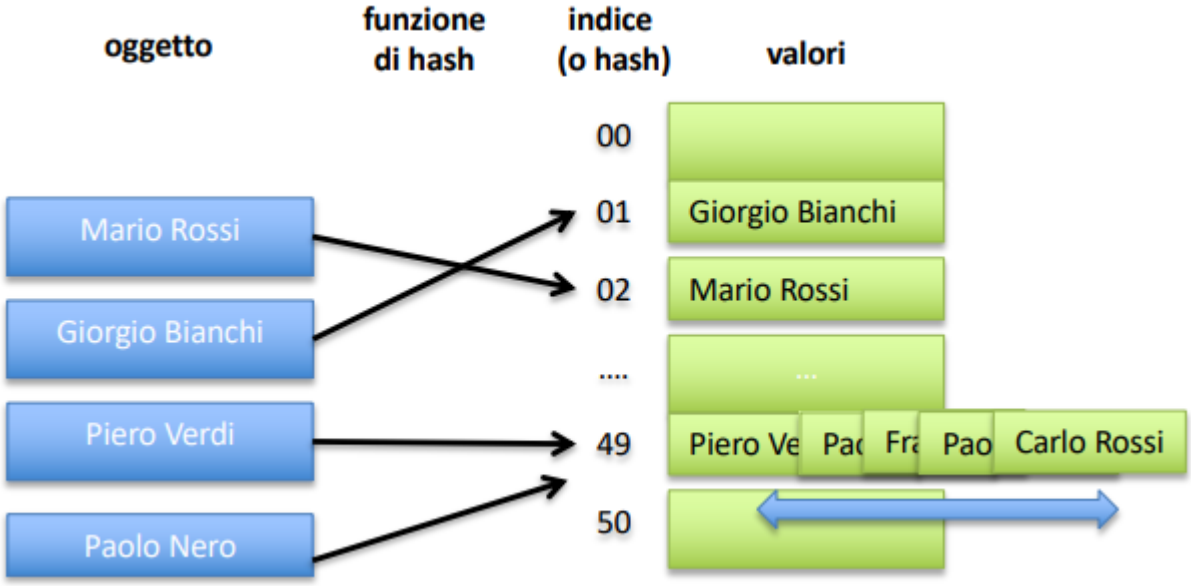
4) Funzione hash

La funzione hash è una funzione che trasforma dei dati di lunghezza arbitraria in dati di lunghezza fissa e permette di semplificare il lavoro di inidicizzazione e ricerca.



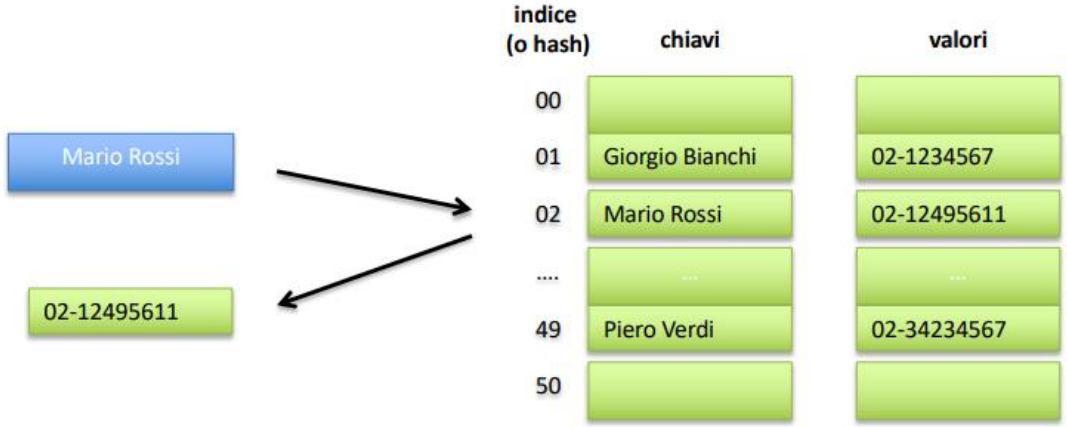
Il metodo hashcode(), implmentato da Object, ritorna il codice hash di un oggetto. Viene utilizzato per recuperare i dati in modo efficiente gli oggetti del Set. La funzione di hashcode() può provocare conflitti se due oggetti diversi hanno lo stesso hashcode().





Con l'HashMap, data la chiave, il valore può essere recuperato in tempo costante.

```
String telefono = get(new Persona("Mario", "Rossi"))
```



5) Iteratori

L'iteratore è un oggetto che permette di effettuare la scansione di una Collection. Tutti gli iteratori implementano una interfaccia con i metodi next(), che ritorna il prossimo elemento della collezione, spostandosi in avanti di una posizione, ed hasNext() che ritorna true se c'è almeno un altro elemento nella collezione. Gli iteratori possono essere ottenuti da una collezione nel seguente modo:

→ iterator(): ritorna un iteratore;

→ keySet() o values() (nel caso delle Set e Collection): che permettono di invocare iterator. In quel caso è possibile utilizzare l'istruzione foreach che permette di ciclare sugli ellementi della collection e nascondere la presenza di un iteratore.

Sintassi

- `for(tipo_var nome_var:collezione o array)`

- `for(Animale a:animali) {a.verso()}`