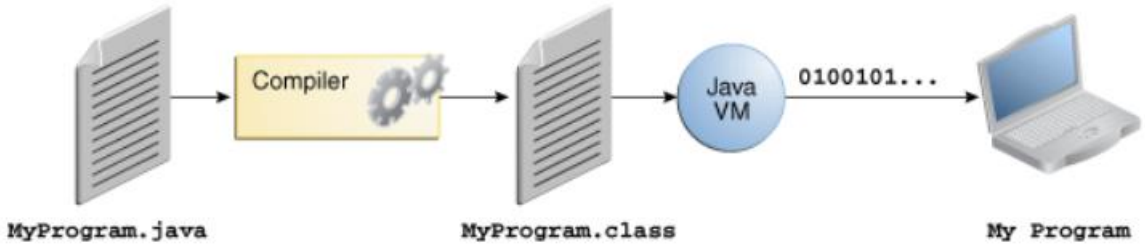
MODULO1

TEOREMA DI BOHM-JACOPINI

Il teorema di Böhm-Jacopini afferma che qualsiasi programma può essere riscritto utilizzando tre tipi di strutture di controllo: *sequenza*, *selezione* e *iterazione (ciclo).* Esso ha contribuito a limitare, se non proprio eliminare, l'uso sconsiderato delle istruzioni *go to* prevista in vari linguaggi di programmazione che consente di effettuare salti incondizionati da un punto all'altro del codice, generando un codice confuso e disordinato noto come spaghetti code, un termine dispregiativo per il codice sorgente con una struttura di controllo del flusso complessa e/o incomprensibile.

TECNOLOGIA JAVA



Nel linguaggio di programmazione Java, tutto il codice sorgente viene prima scritto in file di testo normale che terminano con l'estensione .java.

Tali file sorgente vengono poi compilati in .class file dal compilatore javac.

Un .class file non contiene codice nativo del processore; contiene invece *bytecode* ossia il linguaggio macchina della Java Virtual Machine (Java VM).

Il launcher di java esegue l'applicativo con un'istanza della JVM.

Attraverso Java VM, la stessa applicazione può essere eseguita su più piattaforme.

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

Paradigma di programmazione in cui un programma viene visto come un insieme di oggetti che interagiscono tra loro.
Alcuni principi fondamentali di Java sono:

ENCAPSULATION = Nascondere dettagli implementativi permettendo di aggiungere solo alcune informazioni(tramite accessors quali ad esempio il metodo get e tramite mutators quali ad esempio il metodo set).

ABSTRACTION = L'astrazione dei dati è il processo che nasconde determinati dettagli e mostra all'utente solo le informazioni essenziali. L'astrazione può essere ottenuta sia con classi astratte che con interfacce. La parola chiave abstract è un modificatore di non accesso, utilizzato per classi e metodi.

EREDITARIETA' = L'ereditarietà fornisce un meccanismo potente e naturale per organizzare e strutturare il software. Questa sezione spiega come le classi ereditano lo stato e il comportamento dalle loro superclassi e spiega come derivare una classe da un'altra utilizzando la semplice sintassi fornita dal linguaggio di programmazione Java.

POLIMORFISMO = Significa "avere molte forme". Il polimorfismo in Java ha due diversi tipi di comportamenti: polimorfismo dei metodi e polimorfismo dei dati. Il polimorfismo dei metodi si riferisce all'overloading e overriding dei metodi, mentre il polimorfismo dei dati permette ad una classe di assumere tutte le forme dei figli (per esempio se abbiamo la classe Person e la sottoclasse Professor allora Person p=new Professor() è ammesso.

OVERLOADING = Associare funzioni con stesso nome allo stesso oggetto, distinguibili per via dei parametri (Compile-time). I metodi Overload si differenziano per il numero e il tipo di argomenti passati al metodo.
NOTA: Non è possibile dichiarare più di un metodo con lo stesso nome e lo stesso numero e tipo di argomenti, perché il compilatore non può distinguerli.
Il compilatore non considera il tipo restituito quando differenzia i metodi, quindi non è possibile dichiarare due metodi con la stessa firma anche se hanno un tipo restituito diverso.

OVERRIDING = Sovrascrivere il funzionamento del metodo (Runtime). Esempio:
Sia Clock una classe con metodo setTime()
Sia WifiClock sottoclasse di Clock con un suo metodo setTime()
Overriding runtime sta nel fatto che dato un Clock c = new WifiClock() se chiamiamo c.setTime(), il compilatore vede che c è di tipo WifiClock e quindi utilizza il metodo setTime() di WifiClock e non quello di Clock.
Il caso dei due metodi setTime è un caso di polimorfismo run-time.

BINDING & LATE BINDING = Per "binding dinamico" (letteralmente, "bind" significa "legare") si intende il meccanismo per cui non è il compilatore, ma la JVM ad avere l'ultima parola su quale metodo invocare in corrispondenza di ciascuna chiamata a metodo.
In effetti, questo meccanismo è una diretta conseguenza del polimorfismo e dell'overriding. Ovvero, ciascun riferimento può puntare ad oggetti di tipo effettivo diverso (polimorfismo) e ciascuno di questi tipi effettivi può prevedere una versione diversa dello stesso metodo (overriding). Inoltre, il compilatore non può prevedere di che tipo effettivo sarà una variabile nel corso dell'esecuzione del programma (tecnicamente, questo problema è indecidibile).
Quindi, in Java il binding (collegare ciascuna chiamata ad un metodo vero e proprio) avviene in due fasi: la prima è chiamata Early binding (in questa fase il compilatore risolve l'overloading scegliendo la firma più appropriata alla chiamata), la seconda è Late binding (fase in cui la JVM risolve l'overriding scegliendo il metodo vero e proprio). Chiaramente, il late binding non è necessario per quei metodi che non ammettono overriding: i metodi privati, statici o final.

OGGETTO = Un oggetto è un entità software che mette insieme stato (campi) e comportamento (metodi). Gli oggetti software vengono spesso utilizzati per modellare gli oggetti del mondo reale che trovi nella vita di tutti i giorni. Gli attributi dell'oggetto si chiamano CAMPI e le funzioni degli oggetti si chiamano METODI.

CLASSE = Una classe è un progetto o un prototipo da cui vengono creati gli oggetti. Le classi quindi sono categorie e gli oggetti sono elementi all'interno di ciascuna categoria, tutti gli oggetti della classe dovrebbero avere le proprietà della classe di base (le proprietà principali includono gli attributi/valori effettivi e i metodi che possono essere utilizzati dall'oggetto).

INTERFACCIA = Un'interfaccia è un contratto tra una classe e il mondo esterno. Quando una classe implementa un'interfaccia, promette di fornire il comportamento pubblicato da quell'interfaccia.
Nota: l'interfaccia non ha stato e/o implementazioni.

PACKAGE = Un package è un namespace che organizza classi e interfacce in maniera logica. L'inserimento del codice nei pacchetti semplifica la gestione dei progetti software di grandi dimensioni. I package quindi possono essere visti come dei cassetti e hanno un puro scopo organizzativo.

METODO COSTRUTTORE = Una classe contiene costruttori che vengono richiamati per creare oggetti, le dichiarazioni del costruttore assomigliano alle dichiarazioni del metodo, tranne per il fatto che utilizzano il nome della classe e non hanno un tipo restituito

COMPILARE E ESEGUIRE DA LINEA DI COMANDO

javac HelloWorld.java	per compilare
java HelloWorld	per eseguire la classe HelloWorld.class

MEMORIA STATICA E MEMORIA DINAMICA

MEMORIA STATICA = Contiene il programma(software in esecuzione, istruzioni e informazioni).

MEMORIA DINAMICA = Si usa quando richiesta durante l'esecuzione del programma, quando si ha a che fare con lei si utilizza l'operatore new.

TIPI DI VARIABILI:

Esistono diversi tipi di variabili:

1. Variabili membro di una classe: sono chiamate *campi* (si dividono in statici e non statici).
2. Variabili in un metodo o in un blocco di codice: sono chiamate *variabili locali* .
3. Variabili nelle dichiarazioni di metodo: sono chiamate *parametri* .

Più nel dettaglio:

- **Variabili di istanza (Campi non statici)** Tecnicamente parlando, gli oggetti memorizzano i loro stati individuali in "campi non statici", cioè campi dichiarati senza la parola chiave static. I campi non statici sono noti anche come variabili di istanza perché i loro valori sono univoci per ciascuna istanza di una classe (per ciascun oggetto, in altre parole); il valore currentSpeed di una bicicletta è indipendente dal valore currentSpeed di un'altra.

- **Variabili di classe (Campi statici)** Una variabile di classe è qualsiasi campo dichiarato con il modificatore static; questo dice al compilatore che esiste esattamente una copia di questa variabile, indipendentemente da quante volte la classe è stata istanziata. Un campo che definisce il numero di marce per un particolare tipo di bicicletta potrebbe essere contrassegnato come static poiché concettualmente lo stesso numero di marce si applicherà a tutte le istanze. Il codice static int num = 6; creerebbe un campo statico. Inoltre, potrebbe essere aggiunta la parola chiave final per indicare che il numero di marce non cambierà mai ("costante").

- **Variabili locali** In modo simile a come un oggetto memorizza il suo stato nei campi, un metodo spesso memorizza il suo stato temporaneo in variabili locali. La sintassi per dichiarare una variabile locale è simile alla dichiarazione di un campo (ad esempio, int count = 0;). Non esiste una parola chiave speciale che designi una variabile come locale; tale determinazione deriva interamente dalla posizione in cui viene dichiarata la variabile, ovvero tra le parentesi graffe di apertura e chiusura di un metodo. Pertanto, le variabili locali sono visibili solo ai metodi in cui sono dichiarate e non sono accessibili dal resto della classe.

- **Parametri** Ricordiamo che la firma del metodo main è public static void main(String[] args). Qui, la variabile args è il parametro di questo metodo. La cosa importante da ricordare è che i parametri sono sempre classificati come "variabili" e non come "campi". Questo vale anche per altri costrutti che accettano parametri (come costruttori e gestori di eccezioni).

DENOMINAZIONE:

Le regole e le convenzioni per denominare le variabili possono essere riepilogate come segue: i nomi delle variabili fanno distinzione tra maiuscole e minuscole. La convenzione è di iniziare sempre i nomi delle variabili con una lettera, se il nome che scegli è composto da una sola parola allora scrivi quella parola in tutte lettere minuscole. Se è composto da più parole allora scrivere in maiuscolo la prima lettera di ogni parola successiva.

TIPI DI DATO PRIMITIVI E VALORI DI DEFAULT ASSOCIATI:

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Una stringa in Java è un oggetto (e quindi un puntatore), infatti il suo valore di default è null.

ARRAY:

Un *array* è un oggetto contenitore che contiene un numero fisso di valori di un singolo tipo. La lunghezza di un array viene stabilita al momento della creazione dell'array. Dopo la creazione, la sua lunghezza viene fissata.

Dichiarazione di una variabile per fare riferimento a un array
Come le dichiarazioni per variabili di altri tipi, una dichiarazione di array ha due componenti: il tipo e il nome dell'array. La dimensione dell'array non fa parte del suo tipo (motivo per cui le parentesi sono vuote). Il nome di un array può essere qualsiasi cosa tu voglia, purché segua le regole e le convenzioni discusse in precedenza nella sezione sui nomi. Come con le variabili di altri tipi, la dichiarazione in realtà non crea un array; dice semplicemente al compilatore che questa variabile conterrà un array del tipo specificato. Puoi dichiarare array di vari tipi:

```
int[] anArrayOfInt;
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

Creazione, inizializzazione e accesso a un array
Un modo per creare un array è con l'operatore new, quest'ultimo è sempre associato all'allocazione di memoria di un oggetto; non serve quindi il suo utilizzo quando l'oggetto è gia memorizzato in memoria.

L'istruzione `int[] anArray = new int[10];` alloca un array con memoria sufficiente per 10 elementi interi e assegna l'array alla variabile anArray.

Per assegnare valori ad elementi dell'array si può usare:

```
anArray[0] = 100; // inizializza il primo elemento
anArray[1] = 200; // e così via
```

Si accede a ciascun elemento dell'array tramite il suo indice numerico:

```
System.out.println("Elemento 1 all'indice 0: " + anArray[0]);
```

Si può utilizzare la seguente scorciatoia per creare e inizializzare un array:

```
int[] anArray = {
100 , 200 , 300 ,
400 , 500 , 600 ,
700 , 800 , 900 , 1000
};
```

Qui la lunghezza dell'array è determinata dal numero di valori forniti tra parentesi graffe e separati da virgole.

È inoltre possibile dichiarare un array di array (noto anche come array multidimensionale) utilizzando due o più set di parentesi, ad esempio String[][] names. Ciascun elemento, pertanto, deve essere accessibile tramite un numero corrispondente di valori di indice.
Nel linguaggio di programmazione Java, un array multidimensionale è un array i cui componenti sono essi stessi array.

Puoi utilizzare la proprietà incorporata length per determinare la dimensione di qualsiasi array. Il codice seguente stampa la dimensione dell'array sull'output standard:

```
System.out.println(anArray.length);
```

FOR STATEMENT

```
for ( inizializzazione ; terminazione ; incremento ) {
    istruzione/i
}
```

```
for(int i=1; i<11; i++){
    System.out.println("Il conteggio è: " + i);
}
```

L' istruzione for ha anche un'altra forma progettata per l'iterazione attraverso raccolte e array . Questa forma viene talvolta definita istruzione *for migliorata* e può essere utilizzata per rendere i cicli più compatti e facili da leggere:

```
int[] numeri =
    {1,2,3,4,5,6,7,8,9,10};
for (int elemento: numeri) {
    System.out.println("Il conteggio è: " + elemento);
}
```

GLI STATEMENT BREAK , CONTINUE , RETURN

Puoi utilizzare break per terminare un ciclo for, while o do-while:

```
for (i = 0; i < arrayOfInts.length; i++) {
    if (arrayOfInts[i] == searchfor) {
        foundIt = true;
        break;
    }
}
```

L'istruzione continue salta l'iterazione corrente di un ciclo for, while o do-while saltando alla fine del corpo del ciclo più interno e valutando l'espressione booleana che controlla il ciclo:
Il return statement serve per uscire dal metodo corrente e il flusso di controllo ritorna al punto in cui è stato chiamato il metodo. L'istruzione return ha 2 forme: una che restituisce un valore (return ++count;) e l'altra che non lo fa(return;).
Se provi a restituire un valore da un metodo dichiarato void, riceverai un errore del compilatore.

```
for (int i = 0; i < max; i++) {
    // interessato solo ai p
    if (searchMe.charAt(i) != 'p')
        continue;

    // elabora
    il numPs++ di p;
}
```

Qualsiasi metodo che non è dichiarato void deve contenere un'istruzione return con un valore restituito corrispondente.
Quando un metodo utilizza un nome di classe come tipo restituito, la classe del tipo dell'oggetto restituito deve essere una sottoclasse o la classe esatta del tipo restituito.
Nota: puoi anche utilizzare i nomi delle interfacce come tipi restituiti, in questo caso l'oggetto restituito deve implementare l'interfaccia specificata.

ISTANCEOF

L'operatore instanceof ci consente di verificare se un oggetto appartiene ad una specifica classe. Anche l'ereditarietà viene presa in considerazione.
Esempio: obj instanceof Class // Ritorna true se obj è di tipo Class o è una sua sotto-classe.

CLASSI IN JAVA

```
public class Bicycle {
   // la classe Bicycle ha tre campi
   public int cadence;
   public int gear;
   public int speed;
   // la classe Bicycle ha un costruttore
   public Bicycle(int startCadence, int startSpeed, int startGear) {
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;
   }
   // la classe Bicycle ha quattro metodi
   public void setCadence(int newValue) {
      cadence = newValue;
   }
   public void setGear(int newValue) {
      gear = newValue;
   }
   public void applyBrake(int decrement) {
      speed -= decrement;
   }
   public void speedUp(int increment) {
      speed += increment;
   }
}
```

Una dichiarazione di classe per una classe MountainBike che è una **sottoclasse** di Bicycle potrebbe assomigliare a questa:

```
public class MountainBike extends Bicycle {
   // la sottoclasse MountainBike ha un campo
   public int seatHeight;
   // la sottoclasse MountainBike ha un costruttore
      public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
      super(startCadence, startSpeed, startGear);
      seatHeight = startHeight;
   }
   // la sottoclasse MountainBike ha un metodo
   public void setHeight(int newValue) {
      SeatHeight = newValue;
   }
}
```

MountainBike eredita tutti i campi e i metodi di Bicycle e aggiunge il campo seatHeight e un metodo per impostarlo (le mountain bike hanno sedili che possono essere spostati su e giù a seconda del terreno).

In generale una classe in Java si dichiara nel modo seguente (class declarations) :

```
class MyClass {
    // campi, costruttore, e
    // dichiarazione dei metodi
}
```

Il *corpo della classe* (l'area tra parentesi graffe) contiene tutto il codice che prevede il ciclo di vita degli oggetti creati dalla classe: costruttori per inizializzare nuovi oggetti, dichiarazioni per i campi che forniscono lo stato della classe e dei suoi oggetti, e metodi per implementare il comportamento della classe e dei suoi oggetti.

La dichiarazione di classe precedente è minima. Contiene solo i componenti di una dichiarazione di classe che sono richiesti. Puoi fornire ulteriori informazioni sulla classe, come il nome della sua superclasse, se implementa qualche interfaccia e così via, all'inizio della dichiarazione della classe. Per esempio:

```
class MyClass extends MySuperClass implements YourInterface {
    // dichiarazioni di campo, costruttore e
    // metodo
}
```

significa che MyClass è una sottoclasse di MySuperClass e che implementa l'interfaccia YourInterface.
Puoi anche aggiungere modificatori come *public* o *private* (il campo è accessibile solo all'interno della propria classe) all'inizio.

In generale, le dichiarazioni di classe possono includere questi componenti, nell'ordine:

1. Modificatori come public, private ecc.. (Tuttavia, tieni presente che il modificatore private può essere applicato solo alle classi nidificate).
2. Il nome della classe, con la lettera iniziale maiuscola per convenzione.
3. Il nome del genitore della classe (superclasse), se presente, preceduto dalla parola chiave extends. Una classe può estendere (sottoclasse) solo un genitore.
4. Un elenco separato da virgole di interfacce implementate dalla classe, se presenti, precedute dalla parola chiave implements . Una classe può implementare più di un'interfaccia.
5. Il corpo della classe, racchiuso tra parentesi graffe {}.

In generale, le dichiarazioni di metodo hanno sei componenti, nell'ordine:

1. Modificatori: come public, private e altri.
2. Il tipo restituito: il tipo di dati del valore restituito dal metodo o void se il metodo non restituisce un valore.
3. Il nome del metodo: le regole per i nomi dei campi si applicano anche ai nomi dei metodi, ma la convenzione è leggermente diversa.
4. L'elenco dei parametri tra parentesi: un elenco delimitato da virgole di parametri di input, preceduti dai relativi tipi di dati, racchiusi tra parentesi, (). Se non sono presenti parametri, è necessario utilizzare parentesi vuote.
5. Un elenco di eccezioni.
6. Il corpo del metodo, racchiuso tra parentesi graffe: il codice del metodo, inclusa la dichiarazione delle variabili locali, va qui.

Definizione: due dei componenti di una dichiarazione di metodo comprendono la *firma del metodo*: il nome del metodo e i tipi di parametro.
Un esempio di firma del metodo è calcolaRisposta(double, int, double, double)

DIFFERENZA TRA PARAMETRO E ARGOMENTO

public void setTime(LocalDataTime t){….} //t è un parametro (variabile generica rispetto al comportamento della funzione)

setTime(myTime); //myTime è un argomento (argomento specifico e non generico)

COSTRUTTORE

Il costruttore si scrive prima di get e set.

Bicycle ha il costruttore:

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
   gear = startGear;
   cadenza = startCadenza;
   velocità = velocità iniziale;
}
```

Per creare un nuovo oggetto Bicycle chiamato myBike, il costruttore viene chiamato con l'operatore new:

```
Bicicletta miaBike = new Bicycle(30, 0, 8);
```

new Bicycle(30, 0, 8) crea spazio in memoria per l'oggetto e inizializza i suoi campi.

Sebbene Bicycle abbia un solo costruttore, potrebbe averne altri(overloading), incluso un costruttore senza argomenti:

```
public Bicycle() {
   gear = 1;
   cadence = 10;
   speed = 0;
}
```

Bicycle yourBike = new Bicycle(); invoca il costruttore senza argomenti per creare un nuovo oggetto Bicycle chiamato yourBike.

Java mette a disposizione un costruttore di default senza parametri che viene usato se noi non definiamo alcun costruttore.

Nota: È possibile utilizzare i modificatori di accesso nella dichiarazione di un costruttore per controllare quali altre classi possono chiamare il costruttore. Se un'altra classe non può chiamare un costruttore MyClass, non può creare direttamente oggetti MyClass.

OGGETTI IN JAVA

Una classe fornisce la planimetria per gli oggetti, l'oggetto viene creato da una classe.
Esempio di istruzione per creare un oggetto assegnandolo ad una variabile:

```
Point originOne = new Point(23, 94); //creazione di un oggetto della classe Point
```

Questo statement è composto da tre parti:

1. **Dichiarazione** : ciò che è prima di = , associa un nome di variabile a un tipo di oggetto.
2. **Istanziazione** : la parola chiave new è un operatore Java che crea l'oggetto.
3. **Inizializzazione** : l' operatore new è seguito da una chiamata a un costruttore, che inizializza il nuovo oggetto.

Puoi anche dichiarare una variabile di riferimento in questo modo:

```
Point originOne;
```

Se dichiari originOne in questo modo, il suo valore sarà indeterminato finché un oggetto non verrà effettivamente creato e assegnato ad esso. La semplice dichiarazione di una variabile di riferimento non crea un oggetto. Per questo è necessario utilizzare l'operatore new.

ISTANZIARE UNA CLASSE

L' operatore new istanzia una classe allocando memoria per un nuovo oggetto e restituendo un riferimento a quella memoria. L' operatore new invoca anche il costruttore dell'oggetto.

Nota: la frase "istanziare una classe" significa la stessa cosa di "creare un oggetto". Quando crei un oggetto, stai creando un'"istanza" di una classe, quindi "istanziando" una classe.
L' operatore new richiede un singolo argomento suffisso: una chiamata a un costruttore.

NOTA: istanza e oggetto non sono sinonimi! Oggetto indica un nome mentre istanza indica la relazione che lega l'oggetto alla classe a cui appartiene!

GARBAGE COLLECTOR

L'ambiente runtime Java elimina gli oggetti quando determina che non vengono più utilizzati. Questo processo è chiamato *garbage collection*.
Un oggetto è idoneo per la Garbage Collection quando non sono più presenti riferimenti a quell'oggetto. In alternativa, puoi eliminare esplicitamente un riferimento a un oggetto impostando la variabile sul valore speciale null. Ricorda che un programma può avere più riferimenti allo stesso oggetto; tutti i riferimenti a un oggetto devono essere eliminati prima che l'oggetto sia idoneo per la garbage collection.
L'ambiente runtime Java dispone di un Garbage Collector che libera periodicamente la memoria utilizzata dagli oggetti a cui non viene più fatto riferimento. Il garbage collector fa il suo lavoro automaticamente quando determina che è il momento giusto.

PAROLA CHIAVE THIS

All'interno di un istance method o in un costruttore la parola chiave this serve a far riferimento all'oggetto corrente, cioè all'oggetto che chiama il metodo o il costruttore. È possibile fare riferimento a qualsiasi membro dell'oggetto corrente dall'interno di un metodo di istanza o di un costruttore utilizzando this.
Il motivo più comune per utilizzare la parola chiave this è perché un campo è "oscurato" da un metodo o da un parametro del costruttore:

```
public class Punto {
   public int x = 0;
   pubblico int y = 0;
   public Point(int x, int y) {
      this.x = x;
      questo.y = y;
   }
}
```

Dall'interno di un costruttore è anche possibile utilizzare la parola chiave this per chiamare un altro costruttore nella stessa classe. Questa operazione viene definita invocazione esplicita del costruttore e la vediamo in figura:

(Se presente, l'invocazione di un altro costruttore deve essere la prima riga del costruttore)

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 1, 1);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

MODIFICATORI DI ACCESSO

I modificatori di accesso determinano se altre classi possono utilizzare un campo particolare o invocare un metodo particolare. Esistono due livelli di controllo degli accessi:

1. Al livello più alto: public, o *package-private* (nessun modificatore esplicito).
2. A livello di membro(quindi a livello di campo o metodo): public , private, protected, o *package-private* (nessun modificatore esplicito).

Una classe può essere dichiarata con il modificatore public, in tal caso la classe è visibile a tutte le classi ovunque. Se una classe non ha alcun modificatore esplicito utilizza quello predefinito, noto anche come *package-private*, che la rende visibile solo all'interno del proprio package.
A livello di membro, puoi anche utilizzare il modificatore public o non utilizzarne alcuno (interviene così il *package-private*). Inoltre per i membri sono disponibili due modificatori di accesso aggiuntivi: private e protected.
Il modificatore private specifica che è possibile accedere al membro solo nella sua stessa classe, mentre il modificatore protected specifica che è possibile accedere al membro solo all'interno del proprio pacchetto (come con *package-private*) ma anche da una sottoclasse della sua classe che si trova in un altro pacchetto.
La tabella seguente mostra i livelli d'accesso ai membri consentito da ciascun modificatore:

Modificatore	Classe	Package	Sottoclasse	Mondo
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

La prima colonna di dati indica se la classe stessa ha accesso al membro definito dal livello di accesso. La seconda colonna indica se le classi nello stesso pacchetto della classe (indipendentemente dalla loro parentela) hanno accesso al membro. La terza colonna indica se le sottoclassi della classe dichiarata all'esterno di questo pacchetto hanno accesso al membro. La quarta colonna indica se tutte le classi hanno accesso al membro.

PAROLA CHIAVE STATIC

Utilizzata per creare campi e metodi che appartengono alla classe e non a un'istanza della classe. Quando vengono creati più oggetti dallo stesso progetto di classe, ciascuno di essi ha le proprie copie distinte delle variabili di istanza e ciascuno oggetto ha i propri valori per le variabili di istanza. A volte però è necessario avere variabili comuni a tutti gli oggetti. Ciò si ottiene con il modificatore static.
I campi che hanno il modificatore static nella loro dichiarazione sono chiamati campi statici o variabili di classe e sono associati alla classe, piuttosto che a qualsiasi oggetto. Ogni istanza della classe condivide una variabile di classe, che si trova in una posizione fissa nella memoria. Qualsiasi oggetto può modificare il valore di una variabile di classe, ma le variabili di classe possono anche essere manipolate senza creare un'istanza della classe.

Esempio: supponiamo di voler creare una serie di oggetti Bicycle e di assegnare a ciascuno un numero di serie, iniziando con 1 per il primo oggetto. Questo numero ID è univoco per ciascun oggetto ed è quindi una variabile di istanza. Allo stesso tempo, c'è bisogno di un campo in cui tenere traccia di quanti oggetti Bicycle sono stati creati in modo da sapere quale ID assegnare a quello successivo. Tale campo non è correlato ad alcun oggetto individuale, ma alla classe nel suo insieme. Per questo è necessaria una variabile di classe, numberOfBicycles come segue:

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
    private int id;              //aggiunta di una variabile di istanza per l'ID dell'oggetto
```

```
    private static int numberOfBicycles = 0; // aggiunta una variabile di classe per il
                                             numero di oggetti Bicycle istanziati
    ...
}
```

NOTA: Le variabili di classe fanno riferimento al nome della classe stessa: Bicycle.numberOfBicycles
Ciò rende chiaro che si tratta di variabili di classe.

I metodi statici, che hanno il modificatore static nelle loro dichiarazioni, dovrebbero essere invocati con il nome della classe, senza la necessità di creare un'istanza della classe:

ClassName.methodName(args)

Un uso comune dei metodi statici è accedere ai campi statici. Ad esempio, potremmo utilizzare un metodo statico nella classe Bicycle per accedere al campo statico numberOfBicycles:

```
public static int getNumeroBiciclette() {
    return numeroBiciclette;
}
```

NOTA: I metodi di classe *non possono* accedere direttamente alle variabili di istanza o ai metodi di istanza: devono utilizzare un riferimento a un oggetto. Inoltre, i metodi della classe non possono utilizzare la parola chiave this poiché non esiste alcuna istanza a cui fare riferimento.

Il modificatore static inoltre, in combinazione col modificatore final, viene utilizzato anche per definire le costanti.
Il modificatore final indica che il valore di questo campo non può cambiare.
Esempio: static final double PI = 3.141592653589793;
Le costanti definite in questo modo non possono essere riassegnate e si verifica un errore in fase di compilazione se il programma tenta di farlo. Per convenzione, i nomi dei valori costanti vengono scritti in lettere maiuscole. Se il nome è composto da più parole, le parole vengono separate da un carattere di sottolineatura (_).

Un altro utilizzo di static lo si vede nei blocchi di inizializzazione statici, ossia normali blocchi di codice racchiuso tra parentesi graffe e preceduto dalla parola chiave static. Esempio:

```
static {
    // qualunque codice sia necessario per l'inizializzazione va qui
}
```

Una classe può avere un numero qualsiasi di blocchi di inizializzazione statici e questi possono essere visualizzati in qualsiasi punto del corpo della classe. Il sistema runtime garantisce che i blocchi di inizializzazione statici vengano richiamati nell'ordine in cui appaiono nel codice sorgente.
Un'alternativa ai blocchi statici è scrivere un metodo statico privato:

```
class Whatever {
    public static varType myVar = initializeClassVariable();
    private static varType initializeClassVariable() {
        // il codice di inizializzazione va qui
    }
}
```

Il vantaggio dei metodi statici privati è che possono essere riutilizzati in seguito se è necessario reinizializzare la variabile della classe.

CLASSI NIDIFICATE

Il linguaggio di programmazione Java consente di definire una classe all'interno di un'altra classe. Tale classe è chiamata *classe nidificata* ed è illustrata qui:

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
    static class StaticNestedClass {
        ...
    }
}
```

Terminologia: le classi nidificate sono divise in due categorie: non statiche e statiche. Le classi nidificate non statiche sono chiamate *classi interne (inner classes)* mentre le classi nidificate dichiarate static sono chiamate *classi nidificate statiche* (*static nested classes)*.
Utilizza una classe nidificata non statica (inner class) se hai bisogno dell'accesso ai campi e ai metodi non pubblici di un'istanza di inclusione. Utilizza una classe nidificata statica(*static nested classe)* se non richiedi questo accesso.

Una classe nidificata è un membro della classe che la racchiude. Le classi nidificate non statiche *(inner classes)* hanno accesso agli altri membri della classe che le racchiude, anche se sono dichiarate private. Le classi nidificate statiche (*static nested classes)* non hanno accesso ad altri membri della classe che le racchiude.

Perché utilizzare classi nidificate?

1. È un modo di raggruppare logicamente le classi che vengono utilizzate solo in un posto: se una classe è utile solo a un'altra classe, allora è logico incorporarla in quella classe e tenere insieme le due. Nidificare tali "classi helper" rende il loro pacchetto più snello.
2. Aumenta l'incapsulamento: considera due classi di primo livello, A e B, dove B necessita dell'accesso ai membri di A che altrimenti verrebbero dichiarati private. Nascondendo la classe B all'interno della classe A, i membri di A possono essere dichiarati privati e B può accedervi. Inoltre, B stesso può essere nascosto al mondo esterno.
3. Può portare a un codice più leggibile e gestibile: annidare piccole classi all'interno di classi di livello superiore posiziona il codice più vicino a dove viene utilizzato.

NOTA: Puoi utilizzare per le classi interne gli stessi modificatori che utilizzi per gli altri membri della classe esterna. Aesempio, puoi utilizzare gli specificatori di accesso private, public e protected per limitare l'accesso alle classi interne proprio come li usi per limitare l'accesso ad altri membri della classe.

INNER CLASS: Come con i metodi e le variabili di istanza, una inner class è associata a un'istanza della classe che la racchiude e ha accesso diretto ai metodi e ai campi di quell'oggetto. Inoltre, poiché una classe interna è associata a un'istanza, non può definire alcun membro statico.
Gli oggetti che sono istanze di una inner class esistono all'interno di un'istanza della outer class.
Consideriamo le seguenti classi:

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Un'istanza di InnerClass può esistere solo all'interno di un'istanza di OuterClass e ha accesso diretto ai metodi e ai campi dell'istanza che la racchiude.
Per istanziare una classe interna, è necessario prima istanziare la classe esterna.
Quindi, crea l'oggetto interno all'interno dell'oggetto esterno con questa sintassi:

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

Esistono due tipi speciali di inner class: classi locali e classi anonime.

STATIC NESTED CLASS: Come con i metodi e le variabili della classe, una classe nidificata statica è associata alla sua classe esterna. E come i metodi di classe statica, una classe nidificata statica non può fare riferimento direttamente a variabili di istanza o metodi definiti nella classe che la racchiude: può usarli solo attraverso un riferimento a un oggetto.
Nota: una classe nidificata statica interagisce con i membri dell'istanza della sua classe esterna (e di altre classi) proprio come qualsiasi altra classe di livello superiore. In effetti, una classe nidificata statica è, dal punto di vista comportamentale, una classe di primo livello che è stata nidificata in un'altra classe di primo livello per comodità di confezionamento.

CLASSI LOCALI E CLASSI ANONIME: Sono due tipi di inner class.

- Una classe locale va usata se devi creare più di un'istanza di una classe, accedere al suo costruttore o introdurre un nuovo tipo con nome.
  Si parla di classi locali quando si dichiara una inner class all'interno del corpo di un metodo.
  E' possibile definire una classe locale all'interno di un qualsiasi blocco: nel corpo di un metodo, in un ciclo for o in una clausola if.
- Una classe anonima va usata se devi dichiarare campi o metodi aggiuntivi.
  Le classi anonime sono come le classi locali, tranne per il fatto che non hanno un nome.
  Vengono utilizzate quando c'è il bisogno di usare una classe locale solo una volta e servono a rendere il codice più conciso. Consentono di dichiarare e istanziare una classe allo stesso tempo.
  Mentre le classi locali sono dichiarazioni di classe, le classi anonime sono espressioni, il che significa che definisci la classe in un'altra espressione. La sintassi di un'espressione di classe anonima è come l'invocazione di un costruttore, tranne per il fatto che esiste una definizione di classe contenuta in un blocco di codice.
  Esempio: consideriamo l'istanziazione dell'oggetto frenchGreeting nel seguente codice

```
HelloWorld frenchGreeting = new HelloWorld() {
        String name = "tutto il mondo";
        public void greet() {
            greetSomeone("tutto il mondo");
        }
        public void greetSomeone(String someone) {
            name = someone;
            System.out.println("Un saluto a " + name);
        }
};
```

Poiché una definizione di classe anonima è un'espressione, deve far parte di un'istruzione. In questo esempio, l'espressione della classe anonima fa parte dell'istruzione che crea un'istanza dell'oggetto frenchGreeting (Questo spiega perché c'è un punto e virgola dopo la parentesi graffa di chiusura).

L'espressione della classe anonima è composta da:

1. L' operatore new
2. Il nome di un'interfaccia da implementare o di una classe da estendere. In questo esempio, la classe anonima sta implementando l'interfaccia HelloWorld.
3. Parentesi che contengono gli argomenti di un costruttore, proprio come una normale espressione di creazione di un'istanza di classe.
   NOTA: quando implementi un'interfaccia, non esiste un costruttore, quindi usi una coppia di parentesi vuote, come in questo esempio.
4. Un corpo, che è un corpo di dichiarazione di classe. Più specificamente, nel corpo sono consentite le dichiarazioni di metodo ma le istruzioni no.

Nelle classi anonime si possono dichiarare: campi, metodi aggiuntivi, inizializzatori di istanze, classi locali; tuttavia non è possibile dichiarare costruttori in una classe anonima

Accesso alle variabili locali nello scope da parte di classi anonime e classi locali:

i. Una classe anonima ha accesso ai membri della classe che la racchiude.
ii. Una classe anonima non può accedere alle variabili locali nel suo ambito di inclusione che non sono dichiarate come final o effettivamente finali.
iii. Come una classe nidificata, una dichiarazione di un tipo (come una variabile) in una classe anonima nasconde qualsiasi altra dichiarazione nell'ambito di inclusione che abbia lo stesso nome.

Restrizioni rispetto ai loro membri di classi anonime e classi locali:

i. Non è possibile dichiarare inizializzatori statici o interfacce membro in una classe anonima.
ii. Una classe anonima può avere membri statici purché siano variabili costanti.

LAMBDA ESPRESSIONI

Per le classi con un solo metodo, anche una classe anonima sembra un po' eccessiva e macchinosa, in questi casi di solito si tenta di passare la funzionalità come argomento a un altro metodo. Le espressioni Lambda ti consentono di farlo, di trattare la funzionalità come argomento del metodo o il codice come dati.
Tieni presente che un'espressione lambda assomiglia molto a una dichiarazione di metodo; puoi considerare le espressioni lambda come metodi anonimi, ovvero metodi senza nome.
Per quanto riguarda la sintassi, una espressione lambda è costituita da quanto segue:

1. Un elenco separato da virgole di parametri formali racchiusi tra parentesi.
   Nota: è possibile omettere il tipo di dati dei parametri in un'espressione lambda. Inoltre, puoi omettere le parentesi se è presente un solo parametro.
2. La freccia ->
3. Un corpo, costituito da una singola espressione o da un blocco di istruzioni.

Esempio:

```
p -> p.getGender() == Person.Sex.MALE
     && p.getAge() >= 18
     && p.getAge() <= 25
```

ENUM TYPE

Un tipo *enum* è un tipo di dati speciale che consente a una variabile di essere un insieme di costanti predefinite, esempio comune di utilizzo è il suo uso per i giorni della settimana.
La sintassi è la seguente (si deve trovare in un file CorsoDiStudi.java):

```
public enum CorsoDiStudi {
        INFORMATICA("INF"),
        INGEGNERIA("ING"),
        PSICOLOGIA("PSI");
        private String codice;
        CorsoDiStudi(String codice){
                this.codice = codice;
        }
        public String getCode(){
                return codice;
        }
}
```

La dichiarazione enum definisce una classe. Il corpo della classe enum può includere metodi e altri campi. Il compilatore aggiunge automaticamente alcuni metodi speciali quando crea un'enum. Ad esempio, hanno un metodo statico .values() che restituisce un array contenente tutti i valori dell'enumerazione nell'ordine in cui sono dichiarati.

ANNOTAZIONI

Sono una forma di metadati che non hanno alcun effetto diretto sul funzionamento del codice che annotano.
Hanno diversi usi tra cui: fungono da informazione per il compilatore, utili per elaborazione in fase di compilazione e per elaborazione in fase di esecuzione.
Nella sua forma più semplice, un'annotazione è simile alla seguente:

```
@Override
void mySuperMethod() { ... }
```

l carattere chiocciola (@) indica al compilatore che quanto segue è un'annotazione.
L'annotazione può includere elementi , che possono essere denominati o senza nome, e sono presenti valori per tali elementi:

```
@Author(
   name = "Benjamin Franklin",
   date = "3/27/2003"
)
class MyClass { ... }
```

oppure:

```
@SuppressWarnings(value = "unchecked")
void myMethod() { ... }
```

Dove possono essere utilizzate le annotazioni? Le annotazioni possono essere applicate alle dichiarazioni: dichiarazioni di classi, campi, metodi e altri elementi del programma. Quando utilizzata in una dichiarazione, ciascuna annotazione appare spesso, per convenzione, su una propria riga.
Le annotazioni possono essere applicate anche all'uso dei tipi. Esempio:

```
myString = (@NonNull String) str;
```

Annotazioni predefinite principali utilizzate da Java: @Deprecated (indica che l'elemento contrassegnato non deve più essere utilizzato), @Override (informa il compilatore che l'elemento ha lo scopo di sovrascrivere un elemento dichiarato in una superclasse), e @SuppressWarnings (indica al compilatore di sopprimere avvisi specifici che altrimenti genererebbe) .

INTERFACCE

Nel linguaggio di programmazione Java, un'interfaccia *è* un tipo di riferimento, simile a una classe, che può contenere *solo* costanti, firme di metodo, metodi predefiniti, metodi statici e tipi annidati. I corpi dei metodi esistono solo per i metodi predefiniti e per i metodi statici. Le interfacce non possono essere istanziate: possono solo essere *implementate* da classi o *estese* da altre interfacce. Una differenza significativa tra classi e interfacce è che le classi possono avere campi mentre le interfacce no. Inoltre, puoi istanziare una classe per creare un oggetto, cosa che non puoi fare con le interfacce.
La definizione di un'interfaccia è simile alla creazione di una nuova classe:

```
public interface OperateCar {
   // constant declarations, if any
   // method signatures
   // An enum with values RIGHT, LEFT
   int turn(Direction direction, double radius, double startSpeed, double endSpeed);
   int changeLanes(Direction direction, double startSpeed, double endSpeed);
        ......
   // more method signatures
}
```

Si noti che le firme dei metodi non hanno parentesi graffe e terminano con un punto e virgola.
Per utilizzare un'interfaccia, scrivi una classe che *implements* l'interfaccia. Quando una classe istanziabile implementa un'interfaccia, fornisce un corpo del metodo per ciascuno dei metodi dichiarati nell'interfaccia.

DEFINIRE UN'INTERFACCIA:
Una dichiarazione di interfaccia è composta da modificatori, la parola chiave interface, il nome dell'interfaccia, un elenco separato da virgole di interfacce principali (se presenti) e il corpo dell'interfaccia. Per esempio:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {
   // constant declarations
   // base of natural logarithms
   double E = 2.718282;
   // method signatures
   void doSomething (int i, double x);
   int doSomethingElse(String s);
}
```

Il modificatore di accesso public indica che l'interfaccia può essere utilizzata da qualsiasi classe in qualsiasi pacchetto. Se non specifichi che l'interfaccia è pubblica, l'interfaccia sarà accessibile solo alle classi definite nello stesso pacchetto dell'interfaccia.
Un'interfaccia può estendere altre interfacce, proprio come una sottoclasse di classe o estendere un'altra classe. Tuttavia, mentre una classe può estendere solo un'altra classe, un'interfaccia può estendere un numero qualsiasi di interfacce. La dichiarazione di interfaccia include un elenco separato da virgole di tutte le interfacce che estende.

Il corpo dell'interfaccia può contenere metodi astratti , metodi predefiniti e metodi statici . Un metodo astratto all'interno di un'interfaccia è seguito da un punto e virgola, ma senza parentesi graffe (un metodo astratto non contiene un'implementazione). I metodi predefiniti sono definiti con il modificatore default e i metodi statici con la parola chiave static. Tutti i metodi astratti, predefiniti e statici in un'interfaccia sono implicitamente public, quindi puoi omettere il modificatore public. Inoltre, un'interfaccia può contenere dichiarazioni costanti. Tutti i valori costanti definiti in un'interfaccia sono implicitamente public, static e final. Ancora una volta, puoi omettere questi modificatori.

IMPLEMENTARE UN'INTERFACCIA:
Per dichiarare una classe che implementa un'interfaccia, includi una clausola *implements* nella dichiarazione della classe.
Una classe può implementare più di un'interfaccia, quindi la parola chiave implements è seguita da un elenco separato da virgole delle interfacce implementate dalla classe. Per convenzione, la clausola implements segue la clausola extends, se ce n'è una.

UTILIZZO DI UN'INTERFACCIA COME TIPO:
Quando definisci una nuova interfaccia, stai definendo un nuovo tipo di dati di riferimento. È possibile utilizzare i nomi di interfaccia ovunque sia possibile utilizzare qualsiasi altro nome di tipo di dati. Se definisci una variabile di riferimento il cui tipo è un'interfaccia, qualsiasi oggetto che le assegni *deve* essere un'istanza di una classe che implementa l'interfaccia.

EREDITARIETA' (INHERITANCE)

Nel linguaggio Java, le classi possono essere *derivate* da altre classi, *ereditando* così campi e metodi da quelle classi.

DEFINIZIONI: Una classe derivata da un'altra classe è chiamata sottoclasse (classe figlia). La classe da cui deriva la sottoclasse è chiamata superclasse (classe genitore).
Ad eccezione di Object che non ha superlcasse, ogni classe ha una ed una sola superclasse diretta (ereditarietà singola). In assenza di qualsiasi altra superclasse esplicita, ogni classe è implicitamente una sottoclasse di Object.
La classe Object, nel pacchetto java.lang, si trova in cima all'albero gerarchico delle classi: ogni classe è una discendente diretta o indiretta di essa. La classe Object ha alcuni metodi che le sottoclassi ereditano tra i quali troviamo boolean equals(Object obj), Class getClass(),String toString().
Nota:Il metodo toString() quindi è presente in tutti gli oggetti.

Una sottoclasse eredita tutti i *membri* (campi, metodi e classi nidificate) dalla sua superclasse. I costruttori non sono membri, quindi non vengono ereditati dalle sottoclassi, ma il costruttore della superclasse può essere invocato dalla sottoclasse.
Una sottoclasse eredita tutti i membri *public* e *protected* del suo genitore, indipendentemente dal pacchetto in cui si trova la sottoclasse. Se la sottoclasse si trova nello stesso pacchetto del genitore, eredita anche i membri *private* del pacchetto del genitore. Puoi utilizzare i membri ereditati così come sono, sostituirli, nasconderli o integrarli con nuovi membri:

- I campi ereditati possono essere utilizzati direttamente, proprio come qualsiasi altro campo.
- Puoi dichiarare un campo nella sottoclasse con lo stesso nome di quello nella superclasse, *nascondendolo* così (non consigliato).
- Puoi dichiarare nuovi campi nella sottoclasse che non si trovano nella superclasse.
- I metodi ereditati possono essere utilizzati direttamente così come sono.
- Puoi scrivere un nuovo metodo *di istanza* nella sottoclasse che abbia la stessa firma di quello nella superclasse, *sovrascrivendolo* così .
- Puoi scrivere un nuovo metodo *statico* nella sottoclasse che abbia la stessa firma di quello nella superclasse, *nascondendolo* così .
- Puoi dichiarare nuovi metodi nella sottoclasse che non si trovano nella superclasse.
- È possibile scrivere un costruttore di sottoclasse che invochi il costruttore della superclasse, implicitamente o utilizzando la parola chiave super.

MEMBRI PRIVATI IN UNA SUPERCLASSE:
Una sottoclasse non eredita i membri private della sua classe genitore. Tuttavia, se la superclasse dispone di metodi pubblici o protetti per accedere ai propri campi privati, questi possono essere utilizzati anche dalla sottoclasse.

CASTING DI OGGETTI:
Abbiamo visto che un oggetto è del tipo dati della classe da cui è stato istanziato. Se ad esempio scriviamo public MountainBike myBike = new MountainBike(); allora myBike è di tipo MountainBike.
MountainBike discende da Bicycle e da Object, pertanto MountainBike è una Bicycle e anche un Object, e quindi può essere usato ovunque siano richiesti oggetti di tale tipo. Non è necessariamente vero il contrario in quanto una Bicycle può essere una MountainBike ma può anche non esserlo.

Il *casting* mostra l'utilizzo di un oggetto di un tipo al posto di un altro tipo, tra gli oggetti consentiti dall'ereditarietà e dalle implementazioni. Se ad esempio scriviamo:

```
Object obj = new MountainBike();
```

allora obj è sia un Object che una MountainBike, questo di chiama casting implicito.
Se invece scriviamo:

```
MountainBike myBike = obj;
```

allora otterremmo un errore in fase di compilazione perché il compilatore non sa che obj è di tipo MountainBike. Tuttavia, per fare ciò senza ottenere un errore è possibile utilizzare il casting esplicito: `MountainBike myBike = (MountainBike)obj;`

Questo cast inserisce un controllo di runtime in cui a obj viene assegnato un MountainBike in modo che il compilatore possa presumere con sicurezza che obj sia un MountainBike. Se obj non è MountainBike in fase di esecuzione, verrà generata un'eccezione.

NOTA**:** è possibile effettuare un test logico sul tipo di un particolare oggetto utilizzando l' operatore instanceof. Questo può salvarti da un errore di runtime dovuto a un cast improprio. Per esempio:

```
if (obj instanceof MountainBike) {
   MountainBike myBike = (MountainBike)obj;
}
```

SOVRASCRITTURA E OCCULTAMENTO:

La capacità di una sottoclasse di sovrascrivere un metodo consente a una classe di ereditare da una superclasse il cui comportamento è "abbastanza vicino" e quindi di modificare il comportamento secondo necessità.
Se una sottoclasse definisce un metodo statico con la stessa firma di un metodo statico nella superclasse, allora il metodo nella sottoclasse *nasconde* quello nella superclasse.
La distinzione tra nascondere un metodo statico e sovrascrivere un metodo di istanza ha importanti implicazioni: la versione del metodo dell'istanza sovrascritta che viene richiamata è quella nella sottoclasse; mentre la versione del metodo statico nascosto che viene richiamato dipende dal fatto che venga richiamato dalla superclasse o dalla sottoclasse.

La tabella seguente riassume cosa succede quando definisci un metodo con la stessa firma di un metodo in una superclasse:

	Metodo dell'istanza della superclasse	Metodo statico delle superclassi
Metodo dell'istanza della sottoclasse	Sostituisce	Genera un errore in fase di compilazione
Metodo statico della sottoclasse	Genera un errore in fase di compilazione	Nasconde

NOTA: in una sottoclasse è possibile eseguire l'overload dei metodi ereditati dalla superclasse. Tali metodi sovraccaricati non nascondono né sovrascrivono i metodi dell'istanza della superclasse: sono metodi nuovi, unici per la sottoclasse.

POLIMORFISMO

La definizione del dizionario di *polimorfismo* si riferisce a un principio in biologia in cui un organismo o una specie può avere molte forme o stadi diversi. Questo principio può essere applicato anche alla programmazione orientata agli oggetti e ai linguaggi come il linguaggio Java. Le sottoclassi di una classe possono definire i propri comportamenti unici e tuttavia condividere alcune delle stesse funzionalità della classe genitore.
Il polimorfismo in Java ha due diversi tipi di comportamenti: polimorfismo dei metodi e polimorfismo dei dati. Il polimorfismo dei metodi si riferisce all'overloading e overriding dei metodi, mentre il polimorfismo dei dati permette ad una classe di assumere tutte le forme dei figli (per esempio se abbiamo la classe Person e la sottoclasse Professor allora Person p=new Professor() è ammesso.

CAMPI NASCOSTI & PAROLA CHIAVE SUPER

All'interno di una classe, un campo che ha lo stesso nome di un campo della superclasse nasconde il campo della superclasse, anche se i suoi tipi sono diversi. All'interno della sottoclasse, il campo della superclasse non può essere referenziato con il suo nome semplice. È invece necessario accedere al campo tramite la parola chiave *super.*
Se il tuo metodo sovrascrive uno dei metodi della sua superclasse, puoi richiamare il metodo sovrascritto tramite l'uso della parola chiave super. Puoi anche usare super per fare riferimento a un campo nascosto (anche se nascondere i campi è sconsigliato).
Esempio:

```
public class Superclass {
    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}
```

Ecco una sottoclasse, chiamata Subclass, che sovrascrive printMethod():

```
public class Subclass extends Superclass {

    // overrides printMethod in Superclass
    public void printMethod() {
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

La compilazione e l'esecuzione di Subclass stampa quanto segue:

```
Printed in Superclass.
Printed in Subclass
```

COME UTILIZZARE LA PAROLA CHIAVE SUPER PER RICHIAMARE IL COSTRUTTORE DI UNA SUPERCLASSE?
La sintassi per chiamare un costruttore di superclasse è:

```
super();                          oppure
super(lista parametri);
```

con super() viene chiamato il costruttore senza argomenti della superclasse; mentre con super(lista parametri) viene chiamato il costruttore della superclasse con un elenco di parametri corrispondente.

Ecco il costruttore di MountainBike (sottoclasse) che chiama il costruttore della superclasse e quindi aggiunge il proprio codice di inizializzazione:

```
public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
    super(startCadence, startSpeed, startGear);
    altezza sedile = altezza inizio;
}
```

L'invocazione di un costruttore della superclasse deve essere la prima riga nel costruttore della sottoclasse.

SCRITTURA DI CLASSI E METODI FINAL

Puoi dichiarare i metodi di una classe *final*. Si utilizza la parola chiave final in una dichiarazione di metodo per indicare che il metodo non può essere sovrascritto dalle sottoclassi. Utile quando si vuole rendere definitivo un metodo se ha un'implementazione che non deve essere modificata ed è fondamentale per lo stato coerente dell'oggetto.
I metodi chiamati dai costruttori dovrebbero generalmente essere dichiarati final. Se un costruttore chiama un metodo non final, una sottoclasse può ridefinire quel metodo con risultati sorprendenti o indesiderati.
Tieni presente che puoi anche dichiarare finale un'intera classe. Una classe dichiarata finale non può essere sottoclassata. Ciò è particolarmente utile, ad esempio, quando si crea una classe immutabile come la classe String.

METODI E CLASSI ASTRATTE

Una *classe astratta* è una classe dichiarata abstract: può includere o meno metodi astratti. Non è possibile istanziare le classi astratte, ma è possibile creare sottoclassi.
Un *metodo astratto* è un metodo dichiarato senza implementazione (senza parentesi graffe e seguito da un punto e virgola), in questo modo:

```
abstract void moveTo(double deltaX, double deltaY);
```

Se una classe include metodi astratti, allora la classe stessa *deve* essere dichiarata abstract, come in:

```
public abstract class GraphicObject {
  // declare fields
  // declare nonabstract methods
  abstract void draw();
}
```

Quando una classe astratta viene sottoclassata, la sottoclasse solitamente fornisce implementazioni per tutti i metodi astratti nella sua classe genitore. Tuttavia, in caso contrario, è necessario dichiarare anche la sottoclasse abstract.

NOTA: i metodi in *un'interfaccia* che non sono dichiarati come predefiniti o statici sono *implicitamente* astratti, quindi il modificatore abstract non viene utilizzato con i metodi dell'interfaccia (può essere utilizzato, ma non è necessario).

COMPARAZIONE TRA CLASSI ASTRATTE E INTERFACCE

Le classi astratte sono simili alle interfacce. Non è possibile istanziarli e potrebbero contenere un mix di metodi dichiarati con o senza un'implementazione. Tuttavia, con le classi astratte, è possibile dichiarare campi che non sono statici e definitivi e definire metodi concreti pubblici, protetti e privati. Con le interfacce, tutti i campi sono automaticamente pubblici, statici e finali e tutti i metodi dichiarati o definiti (come metodi predefiniti) sono pubblici. Inoltre, è possibile estendere solo una classe, astratta o meno, mentre è possibile implementare un numero qualsiasi di interfacce.

Classi astratte o Interfacce? Quale scegliere?
considera l'utilizzo di classi astratte se una qualsiasi di queste affermazioni si applica alla tua situazione:

- Desideri condividere il codice tra diverse classi strettamente correlate.
- Ti aspetti che le classi che estendono la tua classe astratta abbiano molti metodi o campi comuni o richiedano modificatori di accesso diversi da public (come protected e private).
- Vuoi dichiarare campi non statici o non finali. Ciò consente di definire metodi che possono accedere e modificare lo stato dell'oggetto a cui appartengono.

prendi in considerazione l'utilizzo delle interfacce se una qualsiasi di queste affermazioni si applica alla tua situazione:

- Ti aspetti che le classi non correlate implementino la tua interfaccia.
- Desideri specificare il comportamento di un particolare tipo di dati, ma non preoccuparti di chi ne implementa il comportamento.
- Si desidera sfruttare l'ereditarietà multipla del tipo.

PACKAGES (PACCHETTI)

Per rendere i tipi più facili da trovare e utilizzare, per evitare conflitti di denominazione e per controllare l'accesso, i programmatori raggruppano gruppi di tipi correlati in pacchetti.

DEFINIZIONE: un package è un raggruppamento di tipi correlati che forniscono protezione dell'accesso e gestione dei name space. Tieni presente che i tipi si riferiscono a classi, interfacce, enumerazioni e tipi di annotazione. Le enumerazioni e i tipi di annotazione sono tipi speciali di classi e interfacce, rispettivamente, quindi ci si può riferire ai tipi semplicemente come classi e interfacce.

I tipi che fanno parte della piattaforma Java sono membri di vari pacchetti che raggruppano classi per funzione: le classi fondamentali sono in java.lang, le classi per la lettura e la scrittura (input e output) sono in java.io, e così via. Puoi anche inserire i tuoi tipi nei pacchetti.

Supponiamo di scrivere un gruppo di classi che rappresentano oggetti grafici, come cerchi, rettangoli, linee e punti. Scrivi anche un'interfaccia, Draggable, che le classi implementano se possono essere trascinate con il mouse.
Dovresti raggruppare queste classi e l'interfaccia in un pacchetto per diversi motivi, inclusi i seguenti:

1. Tu e altri programmatori potete facilmente determinare che questi tipi sono correlati.
2. Tu e altri programmatori sapete dove trovare tipi che possono fornire funzioni relative alla grafica.
3. I nomi dei tipi non saranno in conflitto con i nomi dei tipi in altri pacchetti perché il pacchetto crea un nuovo spazio dei nomi.
4. È possibile consentire ai tipi all'interno del pacchetto di avere accesso illimitato tra loro, ma limitare comunque l'accesso per i tipi esterni al pacchetto.

CREAZIONE DI UN PACKAGE: Per creare un pacchetto, scegli un nome per esso(I nomi dei pacchetti sono scritti tutti in minuscolo per evitare conflitti con i nomi delle classi o delle interfacce) e inserisci un'istruzione package con quel nome all'inizio di ogni file sorgente che contiene i tipi (classi, interfacce, enumerazioni e tipi di annotazione) che desideri includere nel pacchetto. L'istruzione package (ad esempio package graphics;) deve essere la prima riga nel file di origine. Può essere presente una sola istruzione package in ciascun file di origine e si applica a tutti i tipi nel file.

Se non usi un'istruzione package, il tuo tipo finisce in un pacchetto senza nome. In generale, un pacchetto senza nome è solo per applicazioni piccole o temporanee o quando si sta appena iniziando il processo di sviluppo. Altrimenti, le classi e le interfacce appartengono ai pacchetti denominati.

UTILIZZO DEI MEMBRI DEL PACCHETTO: I tipi che compongono un pacchetto sono noti come *membri del pacchetto* (*package members*).
Per utilizzare un membro public del pacchetto esternamente al suo pacchetto, è necessario effettuare una delle seguenti operazioni:

1) Fare riferimento al membro con il suo nome completo.
   Questo è indicato per un uso troppo frequente, tuttavia quando un nome viene utilizzato ripetutamente digitarlo ripetutamente diventa noioso e il codice diventa difficile da leggere. Puoi utilizzare il nome semplice di un membro del pacchetto se il codice che stai scrivendo si trova nello stesso pacchetto di quel membro o se quel membro è stato importato. Tuttavia, se stai tentando di utilizzare un membro di un pacchetto diverso e il pacchetto non è stato importato, devi utilizzare il nome completo del membro, che include il nome del pacchetto.
   Di seguito è riportato il nome completo per la classe Rectangle dichiarata nel pacchetto graphics: graphics.Rectangle
   Potresti utilizzare questo nome completo per creare un'istanza di graphics.Rectangle:
   graphics.Rectangle myRect = nuova grafica.Rectangle();

2) Importa il membro del pacchetto.
   Per importare un membro specifico nel file corrente, inserire un'istruzione import all'inizio del file prima di qualsiasi definizione di tipo ma dopo l'istruzione package, se presente.
   Ecco come importerai la classe Rectangle dal pacchetto graphics:
   import graphics.Rectangle;
   Ora puoi fare riferimento alla classe Rectangle con il suo nome semplice:
   Rectangle myRectangle = new Rectangle();
   Questo approccio funziona bene se si utilizzano solo alcuni membri del pacchetto graphics. Ma se usi molti tipi da un pacchetto, dovresti importare l'intero pacchetto.

3) Importa l'intero pacchetto del membro

Per importare tutti i tipi contenuti in un particolare pacchetto, utilizzare l'istruzione import con il carattere jolly asterisco (*):

```
import graphics.*;
```

Ora puoi fare riferimento a qualsiasi classe o interfaccia nel pacchetto graphics con il suo nome semplice:

```
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

COLLECTIONS (RACCOLTE), LORO INTERFACCE E STRUTTURE DATI

Una collection, a volte chiamata contenitore, è semplicemente un oggetto che raggruppa più elementi in una singola unità. Le collections vengono utilizzate per archiviare, recuperare, manipolare e comunicare dati aggregati. In genere, rappresentano elementi di dati che formano un gruppo naturale, come una mano di poker (una raccolta di carte), una cartella di posta (una raccolta di lettere) o un elenco telefonico (una mappatura di nomi in numeri di telefono).

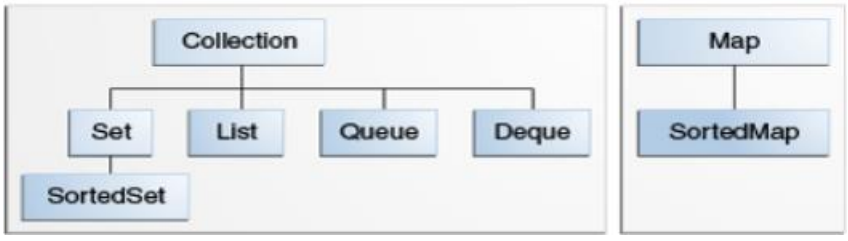
Per rappresentare e manipolare le collections si utilizza un'architettura unificata chiamata Collection Framework. Tutti i framework delle collection contengono interfacce, implementazioni e algoritmi:

A. Le implementazioni sono le implementazioni concrete delle interfacce di raccolta. In sostanza, sono strutture dati riutilizzabili.
   Le implementazioni di scopo generale sono riepilogate nella tabella seguente:

Interfacce	Implementazioni della tabella hash	Implementazioni di array ridimensionabili	Implementazioni di alberi	Implementazioni di liste collegate	Tabella hash + Implementazioni liste collegate
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

B. Gli algoritmi sono i metodi che eseguono calcoli utili, come la ricerca e l'ordinamento, sugli oggetti che implementano le interfacce di raccolta. Gli algoritmi sono funzionalità riutilizzabili.
C. Le interfacce sono tipi di dati astratti che rappresentano raccolte. Le interfacce consentono di manipolare le raccolte indipendentemente dai dettagli della loro rappresentazione. Nei linguaggi orientati agli oggetti, le interfacce generalmente formano una gerarchia.
   Di norma, dovresti pensare alle interfacce e non alle implementazioni.

La gerarchia della Java Collections Framework è costituita da due alberi di interfaccia distinti:



Un Set è un tipo speciale di Collection, un SortedSet è un tipo speciale di Set e così via. Si noti che la gerarchia è composta da due alberi distinti: Map non è un vero Collection.

L'elenco seguente descrive le interfacce delle collection principali:

1. Collection=la radice della gerarchia. Il primo albero inizia con l'interfaccia Collection, che fornisce le funzionalità di base utilizzate da tutte le raccolte, come i metodi add e remove.

L'interfaccia collection viene utilizzata per passare raccolte di oggetti in cui si desidera la massima generalità. Le sue sottointerfacce forniscono raccolte più specializzate.

2. Set=l'interfaccia Set non consente elementi duplicati. Questa interfaccia modella l'astrazione matematica degli insiemi e viene utilizzata per rappresentare insiemi.
   L'interfaccia Set ha la sottointerfaccia SortedSet che provvede all'ordinamento degli elementi nell'insieme.
   Operazioni di base dell'interfaccia Set sono: size, isEmpty, add, remove, iterator.

3. List= l'interfaccia List prevede una collection ordinata che può contenere elementi duplicati. Tramite List si ha generalmente un controllo preciso su dove nell'elenco è inserito ciascun elemento e si può accedere agli elementi tramite il loro indice intero (posizione).
   Operazioni di base dell'interfaccia List sono: add, addAll, remove, indexOf, lastIndexOf, listIterator, sublist.

4. Queue = questa interfaccia ricorda la struttura dati coda(le code in genere ordinano gli elementi in modo FIFO), oltre alle operazioni di base ereditate da collection fornisce operazioni di inserimento, estrazione e ispezione. Gli elementi in Queue sono generalmente ordinati su base FIFO.
   Per quanto riguarda la struttura dell'interfaccia Queue, ciascun metodo queue esiste in due forme: la prima forma genera un'eccezione se l'operazione fallisce, la seconda forma restituisce un valore speciale se l'operazione fallisce(anche null).

Tipo di operazione	Genera un'eccezione	Restituisce un valore speciale
Inserire	`add(e)`	`offer(e)`
Rimuovere	`remove()`	`poll()`
Esaminare	`element()`	`peek()`

5. Deque = questa interfaccia consente operazioni di inserimento, cancellazione e ispezione su entrambe le estremità. Gli elementi in un Deque possono essere utilizzati sia in LIFO che in FIFO, pertanto tutti i nuovi elementi possono essere inseriti, recuperati e rimossi ad entrambe le estremità (una Deque può essere vista come una coda che lavora su entrambi i lati).
   Metodi di Deque:

Tipo di operazione	Primo elemento (inizio dell'istanza `Deque`)	Ultimo Elemento (Fine dell'istanza `Deque`)
Inserire	`addFirst(e)` `offerFirst(e)`	`addLast(e)` `offerLast(e)`
Rimuovere	`removeFirst()` `pollFirst()`	`removeLast()` `pollLast()`
Esaminare	`getFirst()` `peekFirst()`	`getLast()` `peekLast()`

6. Map = mappa le chiavi sui valori. Map non può contenere chiavi duplicate e ciascuna chiave può essere associata al massimo a un valore. La sottointerfaccia denominata SortedMap è una versione ordinata di Map.
   Operazioni di base dell'interfaccia Map: put, get, containsKey, containsValue, size, isEmpty.

   Esempio: `Map<String,Integer> m=new HashMap<String,Integer>();`
   `Map<K,V> copy=new HashMap<K,V>(m);`

EXCEPTIONS (ECCEZIONI)

Il linguaggio di programmazione Java utilizza *le eccezioni* per gestire errori e altri eventi eccezionali.
Un'eccezione è un evento che si verifica durante l'esecuzione di un programma che interrompe il normale flusso delle istruzioni, un exception Object viene creato dal metodo in cui è avvenuta l'eccezione e contiene informazioni quali ad esempio il tipo e lo stato del programma. La creazione di un oggetto eccezione e il suo passaggio al sistema runtime viene chiamata generazione di un'eccezione.

Ci sono tre tipi di eccezione:

1. Checked exception (eccezione controllata), queste sono condizioni eccezionali che una domanda ben scritta dovrebbe anticipare e da cui riprendersi.
   Sono soggette al Catch or Specify Requirement.
2. Error, si tratta di condizioni eccezionali esterne all'applicazione e che in generale le applicazioni non possono prevedere. Sono generati dalla JVM e sono diversi da exception (sono molto meno frequenti). Un esempio di errore è quando non c'è abbastanza spazio in memoria.
   NOTA: error si verifica a run-time.
   Non sono soggette al Catch or Specify Requirement.
3. Runtime exception (eccezioni runtime), queste di solito non sono prevedibili e provengono da bug di programmazione, come errori logici o uso improprio di un'API.
   Non sono soggette al Catch or Specify Requirement.

Gli errori e le eccezioni di runtime sono noti collettivamente come *eccezioni non controllate*.

Costruzione di un gestore di eccezioni:

```
try {
    codice che potrebbe generare un'eccezione
} catch (ExceptionType nomeEccezione){
    //codice eseguito se viene richiamato il gestore delle eccezioni
}
```

Si può utilizzare anche il blocco *finally* se si vuole che il codice al suo interno venga *sempre* eseguito quando si esce dal blocco try, ciò garantisce che il blocco finally venga eseguito anche se si verifica un'eccezione imprevista.
L' istruzione try deve contenere almeno un blocco catch o un blocco finally e può avere più blocchi catch.

THROWS E THROW: throws e throw non sono la stessa cosa.
throws utilizzata nella signature della funzione è utilizzata quando la funzione ha alcune istruzioni che possono portare ad eccezioni. throw utilizzata all'interno di una funzione è usata quando è necessario lanciare un'eccezione. Esempio:

```
public void disiscriviStudente(String matricola) throws StudentNotFoundException {
        if(iscritti.remove(matricola) == null) throw new
StudentNotFoundException(matricola);
}
```

ESEMPIO CREAZIONE DI UNA ECCEZIONE (nel file StudentNotFoundException.java) :

```
public class StudentNotFoundException extends Exception {
        public StudentNotFoundException (String matricola) {
                super("Lo studente con matricola: " + matricola + " non esiste");
        }
}
```

VANTAGGI DELLE ECCEZIONI:

1. Separare il codice di gestione degli errori dal codice "normale"; le eccezioni forniscono i mezzi per separare i dettagli di cosa fare quando accade qualcosa fuori dall'ordinario dalla logica principale di un programma.
2. Propagazione degli errori nello stack di chiamate dei metodi
3. Raggruppamento e differenziazione dei tipi di errore. Poiché tutte le eccezioni generate all'interno di un programma sono oggetti, il raggruppamento o la categorizzazione delle eccezioni è un risultato naturale della gerarchia delle classi. Un esempio di un gruppo di classi di eccezioni correlate nella piattaforma Java sono quelle definite in java.io — IOException e i suoi discendenti. IOException è il più generale e rappresenta qualsiasi tipo di errore che può verificarsi durante l'esecuzione dell'I/O. I suoi discendenti rappresentano errori più specifici. Ad esempio, FileNotFoundException significa che non è stato possibile individuare un file sul disco.

STRING

Le stringhe, ampiamente utilizzate nella programmazione Java, sono una sequenza di caratteri. Nel linguaggio di programmazione Java, le stringhe sono oggetti, infatti la piattaforma Java fornisce la classe String per creare e manipolare stringhe.
l modo più diretto per creare una stringa è scrivere:

```
String saluto = "Ciao mondo!";
```

Un metodo molto utile della classe String è il metodo lenght() che restituisce il numero di caratteri:

```
int len = parola.lenght();
```

L'operatore + è molto utilizzato nelle print per concatenare due o più stringhe; tale concatenazione può essere utilizzata per qualsiasi oggetto e in tal caso per ogni oggetto che non è String verrà chiamato il metodo toString() per convertirlo in stringa.

CONVERSIONE DA STRING A INTERO:
Per convertire stringhe in numeri si può essere utilizzato il metodo Integer.valueOf() come nell'esempio seguente:

```
String str = "25";

try{
        Integer number = Integer.valueOf(str);
}
catch (NumberFormatException ex){
        ex.printStackTrace();
}
```

Nota: valueOf restituisce un oggetto!!
Se vogliamo che ad essere restituito sia il dato primitivo int possiamo utilizzare Integer.parseInt(), a cui dobbiamo passare una stringa contenente un numero intero valido altrimenti darà errore. Pertanto è necessario l'utilizzo di un blocco try-catch. Esempio:

```
String str = "25";
try{
        int number = Integer.parseInt(str);
}
catch (NumberFormatException ex){
        ex.printStackTrace();
}
```

CLASSI WRAPPER (SIMPLE DATA OBJECT)

In Java per ogni tipo primitivo (i cui nomi iniziano tutti rigorosamente con la prima lettera minuscola) esiste un corrispondente "Simple Data Object", anche detta classe wrapper (il cui nome inizia con la lettera maiuscola, tranne nel caso di Integer e Character che oltre alla prima lettera cambiano anche il nome):

byte —> Byte
short —> Short
int —> Integer
long —> Long
float —> Float
double —> Double
char —> Character
boolean —> Boolean

Fondamentalmente, una classe wrapper è come un involucro (wrap) che ha lo scopo di contenere un valore primitivo, trasformandolo in un oggetto (quindi utilizzabile con le Java Collection, in quanto una collection può contenere solo oggetti e non tipi primitivi) e dotandolo di metodi di utilità.

Esempio: Integer n = new Integer (2); oppure Integer n = 2;

EVOLVING INTERFACE

Si usano quando si modificano interfacce e servono a non modificare direttamente quest'ultime creando problemi a chi le stava utilizzando.

Esempio: public interface DoitPlus extends Doit{….} // Doit interfaccia da evolvere

DEFAULT METHODS (METODI DI DEFAULT)

Consentono di aggiungere nuove funzionalità alle interfacce delle librerie e di garantire la compatibilità con il codice scritto per le versioni precedenti di tali interfacce.

Un metodo di default non dipende direttamente dall'implementazione.

METODI E CLASSI FINAL

Si usa final in una dichiarazione di metodo per indicare che il metodo non può essere sovrascritto da sottoclassi.

Una classe final non può essere sottoclassata, un esempio ne è la classe String.

GENERICS

In qualsiasi progetto software non banale, i bug sono semplicemente un dato di fatto.

I Generics aggiungono stabilità al tuo codice rendendo rilevabili più bug in fase di compilazione, in quanto effettua controlli sui tipi più rigorosi in fase di compilazione ed elimina il cast tra tipi di dato; inoltre consente ai programmatori di implementare algoritmi generici che funzionano su raccolte di tipi diversi e possono essere personalizzati rendendoli anche più facili da leggere.

Il seguente snippet di codice senza generics richiede il casting

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Quando riscritto per utilizzare i generici, il codice non richiede il casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

DEFINIZIONE:

Una *generic class* è definita con il seguente formato:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

La sezione del tipo di parametro delimitata da parentesi angolari (<>) segue il nome della classe. Specifica i tipi dei parametri T1,T2,…,Tn.

GENERIC TYPES:
Un tipo generico è una classe generica o un'interfaccia parametrizzata sui tipi. La seguente classe Box verrà modificata per dimostrare il concetto.
Iniziamo esaminando una classe box non generica che opera su oggetti di qualsiasi tipo. Deve solo fornire due metodi: set che aggiunge un oggetto al box, e get che lo recupera:

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Poiché i suoi metodi accettano o restituiscono un Object, sei libero di passare quello che vuoi, a condizione che non sia uno dei tipi primitivi. Non c'è modo di verificare, in fase di compilazione, come viene utilizzata la classe. Una parte del codice potrebbe inserire un Integer nella casella e aspettarsi di estrarne Integer, mentre un'altra parte del codice potrebbe erroneamente passare un String portando ad un errore di runtime.

Per aggiornare la classe Box in modo che utilizzi i generics, crea una dichiarazione di tipo generico modificando il codice "public class Box" in "public class Box<T>". Questo introduce la variabile di tipo, T , che può essere utilizzata ovunque all'interno della classe.
Con questa modifica la classe Box diventa:

```
public class Box<T> {
    // T sta per "Tipo"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Come puoi vedere, tutte le occorrenze di Object vengono sostituite da T. Una variabile di tipo può essere qualsiasi tipo non primitivo specificato: qualsiasi tipo di classe, qualsiasi tipo di interfaccia, qualsiasi tipo di array o anche un'altra variabile di tipo.
Questa stessa tecnica può essere applicata per creare interfacce generiche.

CONVENZIONI DENOMINAZIONE PARAMETRI DI TIPO
Per convenzione, i nomi dei parametri di tipo sono singole lettere maiuscole.
I nomi dei parametri di tipo più comunemente utilizzati sono:

- E - Elemento (ampiamente utilizzato da Java Collections Framework)
- K - Chiave
- N - Numero
- T - Tipo
- V - Valore
- S,U,V ecc. - 2°, 3°, 4° tipo

INVOCAZIONE E ISTANZIAZIONE DI UN TIPO GENERICO
Per fare riferimento alla classe Box generica dal codice, è necessario eseguire *un'invocazione di tipo generico*, che sostituisce T con un valore concreto, come Integer:

```
Box<Integer> integerBox;
```

Puoi pensare a un'invocazione di tipo generico come simile a un'invocazione di metodo ordinaria, ma invece di passare un argomento a un metodo, stai passando un *argomento di tipo* (Integer in questo caso) alla classe Box stessa.

Per quanto riguarda la terminologia, parametro di tipo (Type Parameter) e argomento di tipo (Type Argument) non sono gli stessi. Durante la codifica, si forniscono argomenti di tipo per creare un tipo parametrizzato. Pertanto, la T in <T> è un parametro di tipo e la String in <String> è un argomento di tipo.

Tornando all'esempio, come in qualsiasi altra dichiarazione di variabile quel codice in realtà non crea un nuovo oggetto Box. Dichiara semplicemente che integerBox conterrà un riferimento a un " Box of Integer ", che è il modo in cui viene letto Box<Integer>.
Per istanziare questa classe, si usa la parola chiave new ,come al solito, ma inserendo <Integer> tra il nome della classe e la parentesi come segue:

```
Box<Integer> integerBox = new Box<Integer>();
```

IL DIAMANTE: è possibile sostituire gli argomenti di tipo richiesti per richiamare il costruttore di una classe generica con un set vuoto di argomenti di tipo (<>) purché il compilatore possa determinare o dedurre gli argomenti di tipo dal contesto. Questa coppia di parentesi angolari, <>, è chiamata informalmente *il diamante*.
Ad esempio, puoi creare un'istanza di Box<Integer> con la seguente istruzione:

```
Box<Integer> integerBox = new Box<>();
```

PARAMETRI DI TIPO MULTIPLI
Una generic class può avere più parametri di tipo, come nell'esempio seguente:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
```

PARAMETERIZED TYPES (TIPI PARAMETRIZZATI)
È inoltre possibile sostituire un parametro di tipo (ovvero K o V) con un tipo con parametri (ovvero List<String>). Ad esempio, utilizzando l'esempio OrderedPair<K, V>:
OrderedPair<String, **Box<Integer>**> p = new OrderedPair<>("primes", new Box<Integer>(...));

RAW TYPES (TIPO NON ELABORATO)
Un raw type è il nome di una classe o interfaccia generica senza argomenti di tipo. Ad esempio, data la classe generica Box:

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}

Box<Intero> intBox = new Box<>();       //tipo parametrizzato Box<T>

Box rawBox = new Box();                 //argomento del tipo omesso,
                                        creazione di un raw type di Box<T>
```

L'utilizzo dei raw types permette di riutilizzare codice precedente (tipi grezzi vengono visualizzati nel codice perché molte classi API come le classi Collections non erano generiche prima di JDK 5.0).

GENERIC METHODS (METODI GENERICI)
I metodi generici sono metodi che introducono i propri parametri di tipo. È simile alla dichiarazione di un tipo generico, ma l'ambito del parametro type è limitato al metodo in cui viene dichiarato. Sono consentiti metodi generici statici e non statici, nonché costruttori di classi generiche.

La sintassi per un metodo generico include un elenco di parametri di tipo, racchiusi tra parentesi angolari, che viene visualizzato prima del tipo restituito dal metodo. Per i metodi generici statici, la sezione del parametro di tipo deve essere visualizzata prima del tipo restituito del metodo.
La classe Util include un metodo generico, compare, che compara due oggetti Pair:

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
      return p1.getKey().equals(p2.getKey()) &&
           p1.getValue ().equals(p2.getValue());
   }
}

public class Pair<K, V> {
   private K key;
   private V value;

   public Pair(K key, V value) {
      this.key = key;
      this.value = value;
   }

   public void setKey(K key) { this.key = key; }
   public void setValue(V value) { this.value = value; }
   public K getKey()   { return key; }
   public V getValue() { return value; }
}
```

La sintassi completa per invocare questo metodo sarebbe:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

Il tipo è stato fornito esplicitamente, come mostrato in grassetto. Generalmente, questo può essere tralasciato e il compilatore dedurrà il tipo necessario:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

Questa funzionalità, nota come *inferenza del tipo*, consente di richiamare un metodo generico come metodo ordinario, senza specificare un tipo tra parentesi angolari.

LIMITAZIONI SU PARAMETRI DI TIPO (Bounded Type Parameters)
In alcuni casi potrebbe essere necessario limitare i tipi che possono essere utilizzati come argomenti di tipo in un tipo con parametri. Ad esempio, un metodo che opera sui numeri potrebbe voler accettare solo istanze di Number e delle sue sottoclassi. Questo è lo scopo dei Bounded Type Parameters (parametri di tipo limitato).

Per dichiarare un parametro di tipo limitato elenca il nome del parametro di tipo, seguito dalla parola chiave extends, seguita dal suo *limite superiore* , che in questo esempio è Number. Si noti che, in questo contesto, extends è usato in senso generale per significare "estende" (come nelle classi) o "implementa" (come nelle interfacce). Esempio:

```
public <U extends Number> void inspect(U u){
      System.out.println("T: " + t.getClass().getName());
      System.out.println("U: " + u.getClass().getName());
   }

integerBox.inspect("some text"); // error: this is still String!
```

Possono essere introdotte dal programmatore anche limitazioni multiple tramite extends di più elementi: <T extends B1 & B2 & B3>

NOTA: Box<Integer> non è un sottotipo di Box<Number> anche se Integer è un sottotipo di Number. Vediamo:

```
Box<Number> box = new Box<Number>();
```

Che tipo di argomento accetta? Accetta un singolo argomento il cui tipo è Box<Number>. Si possono passare Box<Integer> o Box<Double>? La risposta è no perché Box<Integer> e Box<Double> non sono sottotipi di Box<Number>.

GENERIC CLASS E SUBTYPING(SOTTOTIPIZZAZIONE)

È possibile creare un sottotipo di una classe o interfaccia generica estendendola tramite parola chiave extends o implementandola tramite parola chiave implements.

Esempio: Immaginiamo ora di voler definire la nostra interfaccia di lista chiamata PayloadList , che associa un valore opzionale di tipo generico P a ciascun elemento. La sua dichiarazione potrebbe assomigliare a

```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
  ...
}
```

Allora le seguenti parametrizzazioni di PayloadList sono sottotipi di List<String>:

```
PayloadList<String,String>
PayloadList<String,Integer>
PayloadList<String,Exception>
```

TYPE INFERENCE (INFERENZA DI TIPO)

La type inference '(inferenza del tipo) è la capacità di un compilatore Java di esaminare ogni invocazione del metodo e la dichiarazione corrispondente per determinare l'argomento (o gli argomenti) del tipo che rendono applicabile l'invocazione. L'algoritmo di inferenza determina i tipi degli argomenti e, se disponibile, il tipo a cui verrà assegnato o restituito il risultato. Infine, l'algoritmo di inferenza tenta di trovare il tipo più specifico che funzioni con tutti gli argomenti. Per illustrare quest'ultimo punto, nell'esempio seguente, l'inferenza determina che il secondo argomento passato al metodo pick è di tipo Serializable:

```
static <T> T pick(T a1, T a2) { return a2; }
Serializable s = pick("d", new ArrayList<String>());
```

WILDCARDS

Nel codice generico, il punto interrogativo (?), chiamato *wildcard*, rappresenta un tipo sconosciuto. Il carattere jolly può essere utilizzato in diverse situazioni: come tipo di parametro, campo o variabile locale; a volte come tipo restituito (sebbene sia meglio essere più specifici come pratica di programmazione). Il carattere jolly non viene mai utilizzato come argomento di tipo per l'invocazione di un metodo generico, la creazione di un'istanza di classe generica o un supertipo.

UPPER BUONDED WILDCARDS: Per dichiarare una wildcard con limite superiore, utilizzare il carattere jolly (' ? '), seguito dalla parola chiave extends , seguita dal suo *limite superiore*:

```
public static void process(List<? extends Foo> list) { /* ... */ }
```

UNBOUNDED WILDCARD: Il tipo unbounded wildcard viene specificato utilizzando il carattere jolly (?), ad esempio List<?>. Questo è chiamato lista di tipi sconosciuti.

LOWE BOUNDED WILDCARDS: In modo simile, una wildcard con limite inferiore limita il tipo sconosciuto a un tipo specifico o a un supertipo di quel tipo. Un carattere jolly con limite inferiore viene espresso utilizzando il carattere jolly (' ? '), seguito dalla parola chiave super, seguita dal suo *limite inferiore*: <? super A>.

WILDCARD CAPTURE: In alcuni casi, il compilatore deduce il tipo di un carattere jolly. Ad esempio, una lista può essere definita come List<?> ma, quando valuta un'espressione, il compilatore deduce un tipo particolare dal codice. Questo scenario è conosciuto come wildcard capture.

USO DELLE WILDCARD: Ecco alcune linee guida per capire quale tipologia di wildcard utilizzare, sia "in" una variabile fornita dal codice come ad esempio in un metodo copia l'elemento da copiare, e sia "out" una variabile contenente dati da utilizzare altrove, nell'esempio di un metodo copia corrisponde al valore copiato in uscita. Allora abbiamo che:

- Una variabile "in" viene definita con upper bounded wildcard, utilizzando la parola chiave extends .
- Una variabile "out" viene definita con un lower bounded wildcard, utilizzando la parola chiave super .
- Nel caso in cui sia possibile accedere alla variabile "in" utilizzando i metodi definiti nella classe Object , utilizzare un unbounded wildcard.
- Nel caso in cui il codice debba accedere alla variabile sia come variabile "in" che come variabile "out", non utilizzare wildcard.

Queste linee guida non si applicano al tipo restituito di un metodo.

TYPE ERASURE (CANCELLAZIONE DEL TIPO)

I Generics sono stati introdotti nel linguaggio Java per fornire controlli di tipo più rigorosi in fase di compilazione e per supportare la programmazione generica. Per implementare i generics, il compilatore Java applica il type erasure a tutti i tipi generics che incontra cancellando i tipi dei loro parametri e sostituendoli con il tipo associato ad ognuno se il tipo del parametro è unbounded(illimitato). Il bytecode prodotto, quindi, contiene solo classi, interfacce e metodi ordinari. Se necessario inserisce il cast dei tipi (type casts) per preservare il tipo stesso. Inoltre genera metodi bridge utili a preservare il polimorfismo nei tipi generics estesi.
Nota: tramite il type erasure le annotazioni esplicite vengono rimosse dal programma prima che venga compilato ed eseguito.

BASIC I/O

I programmi utilizzano byte streams per input e output di byte a 8 bit. Tutte le classi di byte stream discendono da InputStream e OutputStream.
La piattaforma Java memorizza i valori dei caratteri utilizzando le convenzioni Unicode. Il flusso di caratteri I/O traduce automaticamente questo formato interno da e verso il set di caratteri locale. Nelle impostazioni locali occidentali, il set di caratteri locale è solitamente un superset ASCII a 8 bit. Per la maggior parte delle applicazioni, l'I/O con flussi di caratteri non è più complicato dell'I/O con flussi di byte. L'input e l'output eseguiti con le stream class si traducono automaticamente da e verso il set di caratteri locale. Un programma che utilizza flussi di caratteri al posto di flussi di byte si adatta automaticamente al set di caratteri locale ed è pronto per l'internazionalizzazione.
Per non essere troppo sovraccaricata la piattaforma Java implementa flussi I/O bufferizzati.
Esempio di utilizzo dei buffer:

```
inputStream = new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"))
```

Ci sono quattro buffered stream classes usate per "imballare" uno stream non bufferizzato: BufferedInputStream e BufferedOutputStream creano byte streams bufferizzati, mentre BufferedReader e BufferedWriter creano character streams bufferizzati.

SCANNING

Oggetti di tipo Scanner sono utili per scomporre l'input in token e tradurre ogni token nel rispettivo tipo di dato. Per utilizzare oggetti di tipo Scanner bisogna importare la java.util.Scanner.

Esempio utilizzo Scanner:

```
Scanner scanner = new Scanner(System.in);
System.out.println("Come ti chiami?");
String nome = scanner.nextLine(); //utilizzo di nextLine() perché è
                                  di tipo String il contenuto di
                                  scanner
System.out.println("Quanti anni hai?");
int eta = scanner.nextInt();      //utilizzo di nextInt() perché è di tipo
                                  String il contenuto di scanner
scanner.nextLine();               //senza questo "si blocca"
System.out.println("Ciao "+nome+" hai "+eta+" anni");
```

FORMATTAZIONE:

Gli Stream Object che implementano la formattazione sono istanze di PrintWriter (character stream class) o PrintStream (byte stream class). Entrambi implementano lo stesso insieme di metodi per convertire i dati interni in output formattato. Sono forniti due livelli di formattazione:

1) Print e println per formattare i singoli valori in modo standard.
   Richiamare print o println per restituire un singolo valore dopo aver convertito il valore utilizzando il metodo toString appropriato.
2) format per formattare un numero qualsiasi di valori in base a una stringa di formato con varie opzioni di formattazione(quali %d per interi, %f per valori float, %n simbolo terminatore ecc..)

NOTA: Gli unici oggetti PrintStream di cui probabilmente avrai bisogno sono System.out e System.err.
NOTA: Quando è necessario creare un flusso di output formattato istanziare PrintWriter e non PrintStream.

I/O DA LINEA DI COMANDO

Input e output da linea di comando possono essere gestiti tramite Standard Streams o tramite Console.

- Gli Standard Streams sono una funzionalità di molti sistemi operativi. Per impostazione predefinita, leggono l'input dalla tastiera e scrivono l'output sul display.
  La piattaforma Java supporta tre flussi standard: Standard Input, a cui si accede tramite System.in; Standard Output, accessibile tramite System.out; e Standard Error, a cui si accede tramite System.err. Questi oggetti vengono definiti automaticamente e non è necessario aprirli. System.out e System.err sono definiti come oggetti PrintStream(stream di byte e non di caratteri), ma anche se tecnicamente è un flusso di byte PrintStream utilizza un oggetto flusso di caratteri interno per emulare molte delle funzionalità dei flussi di caratteri.
  System.in invece è un flusso di byte senza funzionalità di flusso di caratteri. Per utilizzare l'input standard come flusso di caratteri, System.in racchiudere InputStreamReader come da esempio:

  ```
  InputStreamReader cin = new InputStreamReader(System.in);
  ```

- Un'alternativa più avanzata agli Standard Streams è la Console. Si tratta di un singolo oggetto di tipo predefinito Console che presenta la maggior parte delle funzionalità fornite dagli Standard Stream e altre ancora. La Console è particolarmente utile per l'immissione sicura della password(grazie al suo metodo readPassword). L'oggetto Console fornisce inoltre flussi di input e output che sono veri e propri flussi di caratteri, tramite i suoi metodi reader e writer. Prima che un programma possa utilizzare la Console, deve tentare di recuperare l'oggetto Console invocando System.console(). Se l'oggetto Console è disponibile, questo metodo lo restituisce.

DATA STREAMS (FLUSSI DI DATI)
I data streams supportano I/O binario di valori di tipi di dato primitivi (boolean, char, byte, short, int, long, float, double) e anche valori String.
Tutti i data stream implementano o l'interfaccia DataInput o l'interfaccia DataOutput. Le implementazioni più utilizzate sono DataInputStream e DataOutputStream.
Esempio:

Tipo di dati	Metodo di output	Metodo di input
double	DataOutputStream.writeDouble	DataInputStream.readDouble
int	DataOutputStream.writeInt	DataInputStream.readInt
String	DataOutputStream.writeUTF	DataInputStream.readUTF

OBJECT STREAMS (FLUSSI DI OGGETTI)
Proprio come i data streams supportano l'I/O di tipi di dati primitivi, gli object streams supportano l'I/O degli oggetti.
Le classi di object stream sono ObjectInputStream e ObjectOutputStream, queste classi implementano ObjectInput e ObjectOutput che sono sottointerfacce di DataInput e DataOutput. Ciò significa che tutti i metodi primitivi di I/O dei dati trattati in Data Streams sono implementati anche nei flussi di oggetti.
Esempio di utilizzo semplice:

```
Object ob = new Object();
out.writeObject(ob);
Object ob1 = in.readObject();
```

RIEPILOGO(I/O)
Il package java.io contiene molte classi che i tuoi programmi possono utilizzare per leggere e scrivere dati. La maggior parte delle classi implementa flussi di accesso sequenziale(sequential access streams). I flussi di accesso sequenziale possono essere divisi in due gruppi: quelli che leggono e scrivono byte e quelli che leggono e scrivono caratteri Unicode. Ogni flusso di accesso sequenziale ha una specialità, come leggere o scrivere su un file, filtrare i dati mentre vengono letti o scritti o serializzare un oggetto.

I/O & FILE
Il package java.nio.file fornisce un supporto completo per l'I/O di file e file system. Si tratta di un'API molto completa con i seguenti punti chiave:
- La classe Path dispone di metodi per manipolare un percorso.
- La classe File dispone di metodi per operazioni sui file, come spostamento, copia, eliminazione e anche metodi per recuperare e impostare attributi di file.
- La classe FileSystem dispone di una varietà di metodi per ottenere informazioni sul file system.

MAVEN

Apache Maven è uno strumento di automazione build usato principalmente per i progetti Java, sviluppato e gestito dalla Apache Software Foundation. Rende più facile gestire e mantenere grandi progetti fornendo una struttura coerente e una serie di convenzioni su come organizzare il progetto, aiutando gli sviluppatori ad automatizzare il processo di build, test e distribuzione del software.

Una delle caratteristiche principali di Maven è la sua capacità di gestire le dipendenze. Maven tiene traccia di tutte le librerie e di altre dipendenze di cui un progetto ha bisogno e le scarica automaticamente quando sono necessari. Ciò rende facile per gli sviluppatori utilizzare librerie esterne nei loro progetti senza doverle scaricare e gestirle manualmente.

Maven usa un approccio dichiarativo per specificare la build e le dipendenze del progetto. Maven utilizza un file XML chiamato pom.xml (project object model) per gestire le dipendenze del progetto, i plugin e la configurazione di build. Fornisce inoltre una serie di plug-in integrati per attività comuni e può essere esteso con plugin personalizzati.

ECLIPSE

Ctrl + Shift + L	lista di shortcuts utili
Ctrl + Shift + P	per vedere dove inizia e finisce un blocco
Ctrl + Space	[syso->System.out.println();]

Per generate automaticamente costruttore, getters e setters da una riga all'interno della classe premere tastoDestro>Source>QuelloCheVuoiInserire

MODULO2 LMP

CONCETTI BASE PROLOG:

Il Prolog è un linguaggio di programmazione che adotta il paradigma di programmazione logica, è quindi un linguaggio dichiarativo e non funzionale(python è funzionale).

Le variabili in prolog sono variabili matematiche e non spazi di memoria.
Per notazione le variabili in prolog si indicano con stringhe che iniziano per lettere maiuscole o per underscore.
Le variabili si usano in fatti e regole; fatti e regole insieme generano dei predicati.

Al concetto di assegnazione viene sostituito il concetto di unificazione.

Le costanti iniziano con una lettera minuscola.

Importante: l'algoritmo in prolog legge le espressioni dall'alto verso il basso e da sinistra verso destra.

Il prolog mediante il backtracking esplora le possibili strade alternative finchè non ne trova una che porta alla destinazione finale. (Il backtracking è una tecnica per trovare soluzioni a problemi in cui devono essere soddisfatti dei vincoli. Questa tecnica enumera tutte le possibili soluzioni e scarta quelle che non soddisfano i vincoli).

In prolog il simbolo "=" è simbolo di unificazione ed è diverso dall'uguale in matematica, pertanto 4+3=7 in prolog restituisce false perché viene visto come +(4,3) = +(X,Y).

LINGUAGGIO DICHIARATIVO

Indica "cosa" serve per arrivare alla soluzione desiderata, ma non il "come", cioè l'implementazione utilizzata.
Si contrappone ai linguaggi imperativi e procedurali (Java, C, C++, ecc)

TERMINI

Atomi: nomi che iniziano con lettera minuscola, sequenze di caratteri tra ', numeri preceduti da caratteri. Esempi: andrea , 'Corso di Prolog' , c1p8.

Numeri: 12345

Variabili: nomi che iniziano con lettera MAIUSCOLA o con _
Esempi: Tizio , _andrea , _

Termini composti: somma(1,2,X)

PREDICATI

Espressi tramite la notazione f(t1,…,tn) , f è un atomo che prende il nome di funtore mentre t1,…,tn sono gli argomenti e sono dei termini (predicato f con n argomenti ha arità n).

CLAUSOLE

Le clausole: fatti e regole. I fatti sono regole senza corpo.

```
FATTI =     parent(ben,jim).

            friend(luke,daisy).

REGOLE =    grandparent(X,Y):-
                parent(X,Z),
                parent(Z,Y).
```

REGOLE

Head :- Body. significa che affinchè la Head sia vera deve essere vero il Body (e quindi i predicati che la compongono).

Nel body ci sono 1 o più predicati separati da , (and) o da ; (or)

Ogni regola termina con .

FATTI

Un fatto è un predicato seguito da .

Un fatto può essere composto da più termini: amico(fratello(alice,X),bob).

PROGRAMMA LOGICO

Insieme di regole/fatti. Risponde alle query con tue o false e assegna dei valori alle variabili.

ESECUZIONE DI UN PROGRAMMA

Prolog cerca nel proprio database di regole e fatti, quelli che soddisfano la nostra query, istanziando le variabili.Ogni variabile, una volta istanziata (unificata), non può assumere un secondo valore.

COMANDI UTILI A SWI-PROLOG

- edit.
  apre l'editor per modificare/aggiungere fatti e regole al file in esame
- consult('nome_file').
  carica un file con i suoi dati
- reconsult(‘nome_file’) .
  ricarica il file con i suoi dati
- trace / notrace .
  abilita / disabilita la stampa di tutti i passaggi intermedi (molto utile per seguire lo svolgersi del programma)

DA DOCUMENTAZIONE

member(?Elem, ?List)

True if Elem is a member of List. The SWI-Prolog definition differs from the classical one. Our definition avoids unpacking each list element twice and provides determinism on the last element

get(+Stream, -Char)

Read the next non-blank character from Stream.

Simboli usati:

+ termine che deve essere già istanziato
- termine che viene istanziato dalla regola
? termine che può o meno essere già istanziato

ESECUZIONE DI UN PROGRAMMA

- Analizza i fatti/regole dall'alto verso il basso (quindi è importante l'ordine con cui vengono scritti)
- Utilizzo del BACKTRACKING per tornare indietro a prima che una variabile fosse unificata o che una certa regola fosse esplorata
- Utilizzo della ricorsione per chiamare le altre regole
- Per avere altre risposte, e quindi forzare il backtracking anche se il programma ne ha già trovata una che funziona, basta premere **;**

OSSERVAZIONE

Ha molta importanza l'ordine delle clausole e nelle clausole:

```
path(X,Y):- path(X,Z),path(Z,Y).
path(X,Y):- edge(X,Y).
```

Genera un loop infinito!!!

LISTE IN PROLOG

Esempio introduttivo: [a,b,c,d]=[H|T] dove H=a , T=[b,c,d]
[a]=[H|T] dove H=a , T=[]
[]=[H|T] NON UNIFICABILE!

Una lista che normalmente sarebbe rappresentata come segue:

[aldo , punto(1,2) , gatto , radice_di(25)]

in progol viene rappresentata così:

.(aldo , .(punto(1,2) , .(gatto , .(radice_di(25), nil))))

Dove la lista vuota è [] e in generale le liste hanno struttura [Head|Tail] (in questo caso Testa=aldo e Coda contiene il resto della lista).

Nota: si possono estrarre più elementi contemporaneamente, infatti in [H1,H2|T] H1 e H2 sono il primo e il secondo elemento, mentre T è la lista rimanente(lista iniziale senza i primi 2 elementi).

Operazione delete per eliminare un elemento dalla lista:

```
#del(X,L,NewL)  X elemento da cancellare, NewL è L senza X
del(X,[X|Tail],Tail).
del(X,[Y|Tail],[Y|Tail1]):-
    del(X,Tail,Tail1).
```

Operazione insert per inserire non deterministicamente l'elemento X nella lista:

```
insert(X,L,[X|L]).
insert(X,[Y,L],[Y,NewL]):-
    insert(X,L,NewL).
```

Operazione sublist per vedere se la lista S è sottolista di L:

```
sublist(S,L):-
    append(_,L2,L),
    append(S,_,L2).
```

Operazione di permutazione di una lista:

```
permutazione([],[]).
permutazione([Testa|Coda],ListaPermutata):-
    permutazione(Coda,CodaPermutata),
    insert(Testa,CodaPermutata,ListaPermutata).
```

(Vedi Esercizio 29 per permutazione Zanzotto fatta a lezione.)

Ordinamento:

```
ordinata(L,LO):-
    permutazione(L,LO),
    /*vera se L è una permutazione degli elementi di LO*/
    ordinata(LO).
    /*vera se L è una lista ordinata*/

ordinata([]).
ordinata([H1]).
ordinata([H1,H2|R]):- H1 >= H2 ,!, ordinata([H2|R]).
```

OPERATORI IN PROLOG (e precedenza)

La sintassi del prolog prevede tre tipi di operatori: infissi, prefissi e postfissi.
Un operatore infisso può essere utilizzato per strutture binarie e compare sempre tra i suoi due argomenti.
Gli operatori prefissi e postfissi invece possono essere usati solamente per strutture unarie e figurano sempre prima o dopo il loro argomento:

Prefisso (unario) fx fy
Infisso (binario) xfx xfy yfx
Postfisso (unario) xf yf

Precedenza degli operatori:
x → la sua priorità deve essere minore di quella dell'operatore
y → la sua priorità deve essere minore o uguale a quella dell'operatore

xfx significa che l'operatore deve avere sia a dx che a sx delle espressioni il cui funtore principale ha una precedenza strettamente minore dell'operatore che stiamo definendo e quindi stiamo dicendo che l'operatore in questione non è associativo.

Esempio1 per capire come funziona la precedenza:
con + e – di tipo yfx se abbiamo a-b+c
allora la precedenza sarà (a-b)+c , cioè +(-(a,b),c)

Esempio2:
>> tipo yfk
<< tipo xfy
1 >> 2 >> 3 viene considerato come (1 >> 2) >> 3
1 << 2 << 3 viene considerato come 1 << (2 << 3)

Esempio3:
con not di tipo fy possiamo avere not(not G) perchè l'espressione a destra ha stessa priorità del primo not e questo è ammesso dalla specifica fy.
Se not fosse stato di tipo fx allora prolog non avrebbe ammesso la scrittura not not G per via della specifica fx.

COME DEFINIRE UN OPERATORE IN PROLOG

Puoi utilizzare ?-help(op)
per capire come definire un operatore op(+Precedence, +Type, :Name).

Per definire un operatore si usa all'inizio del programma la direttiva seguente

:- op (P , T , N)

Dove: P indica la precedenza (valore che va da 1 a 1200),
T indica l'associatività (ad esempio xfy),
N indica il nome dell'operatore che stiamo definendo.

Esempi:
:- op(1050, xfy, ->).
:- op(300, yfx, [-,%,&]).

IS - OPERAZIONI ARITMETICHE – OPERATORI BUILT-IN

Con ?- X=1+2 prolog fa il matching(simile a unificazione) tra le due sottopressioni che compaiono a sinistra e a destra dell'operatore uguale, quindi restituisce la risposta calcolata che ad X associa l'espressione 1+2 , quindi restituisce in questo caso X=1+2.
Per risolvere questo possiamo utilizzare l'operatore "is" al posto dell'uguale.

L'OPERATORE "is" FORZA LA PARTE ALLA SUA DESTRA AD ESSERE COMPUTATA E CONSENTE L'UNIFICAZIONE DELLA PARTE DESTRA COMPUTATA CON LA PARTE ALLA SUA SINISTRA
NOTA: alla sua destra in cui deve esserci una espressione calcolabile
NOTA: essendoci bisogno di "tempo" la somma è fatta in maniera non dichiarativa.

?-X is 1+2 restituirà X=3.
?-X is 1+Y resituirà un errore se Y non istanziato.

Alcuni operatori built-in in prolog sono:

[Aritmetici]

+ , - , * , / , **	addizione,sottrazione,…
// , mod	operazioni su interi
sin , cos , log	funzioni standard

[Comparazione]

X>Y
X<Y
X >= Y
X <= Y
X=:=Y (X e Y sono numericamente uguali)
X =\= Y (X e Y non sono numericamente uguali)

[Confronto]

=
is
=:= (confronto espressioni aritmetiche uguali)
=\= (confronto espressioni aritmetiche diverse)
== (uguaglianza, ma non assegnazione)
\== (disuguaglianza)
@< (ordinamento lessicografico)

PROGRAMMI VISTI A LEZIONE:

1)

```
genitore(mario,giovanni).
genitore(mario,luca).
fratello(X,Y):-genitore(Z,Y),genitore(Z,X).
```

2)

```
word(h,o,s,e,s).
word(s,n,a,i,l).
word(l,a,s,e,r).
word(s,t,e,e,r).
word(e,a,r,n).
word(s,a,m,e).
word(r,u,n).
word(y,e,s).
word(n,o).
word(b,e).
word(u,s).
word(i,t).
word(h,i,k,e).
word(e,a,t).
word(s,u,n).
word(s,h,e,e,t).
word(a,l,s,o).
word(i,r,o,n).
word(l,e,t).
word(t,e,n).

solution( X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12) :-
        word( X1, X2, X3, X4, X5),
        word( X3, X6, X9, X12),
        word( X5, X7, X11),
        word( X8, X9, X10, X11).
```

```
% A crossword puzzle
%
%       X1  X2  X3  X4  X5
%               X6      X7
%           X8  X9 X10 X11
%               X12
```

3)

```
edge(a,b).
edge(b,c).
edge(c,d).
edge(a,k).
edge(k,e).
edge(k,c).
path(X,Y):-edge(X,Y).
path(X,Y):- path(X,Z),edge(Z,Y).
```

4) (member, da sapere a memoria)

```
appartiene(X,[X|_]).
appartiene(X,[_|T]):-appartiene(X,T).
```

```
?-appartiene(x,L)
?-appartiene(x,[a,b,c,d])
```

```
member(X,L): X is member of L

member(X,[X|_]).
member(X,[_|L]):-
        member(X,L).
```

5)

```
append([],L,L).
append([Hx|Tx],L,[Hx|Tz]):-
        append(Tx,L,Tz).

/*
?- append([1,2,3],[4,5,6],[1,2,3,4,5,6]) deve restituire true
?- append([1,3],[4,5],[1,2,3,4,5,6]) deve restituire false

append([],L,L). è il caso base
append([Hx|Tx],L,[Hx|Tz]):-append(Tx,L,Tz). con il caso base si parla di induzione
strutturale
*/
/*
Si può anche scrivere come concatenazione
conc([],L,L).
conc([X|L1],L2,[X|L3]):-
        conc(L1,L2,L3).
*/
```

6)

```
rivoltata([],[]).
rivoltata([HX|TX],R):-
        append(TR,[HX], R),
        rivoltata(TX,TR).

?-rivoltata([1,2],[2,1]) true

/*
predicato rivoltata vero se L è la lista rivoltata di RL
rivoltata(L,RL)
rivoltata è una proprietà
rivoltata è vera se il primo elemento di L e l'ultimo elemento di RL coincidono e se RLR
corrisponde con L
*/
```

8)

```
num_elem([],0).
num_elem([H|T],N):-num_elem(T,N1) , N is N1+1.

?-num_elem([],0)         -----> TRUE
?-num_elem([a,b,c],3)    -----> TRUE
?-num_elem([a,b,c],4)    -----> FALSE

/* num_elem(L,N) vero se L è una LISTA e N rappresenta il NUMERO di elementi che ci
sono nella lista */
```

9)

```
sommatoria([],0).
sommatoria([H|T],N):-sommatoria(T,N1), N is H+N1.

?-sommatoria([2,2,1],5) ----> TRUE
?-sommatoria([2,2,1],1) ----> FALSE
?-sommatoria([],0)      ----> TRUE

/* sommatoria(L,N) vero se L è una lista di numeri e N è la somma di tutti
i numeri presenti nella lista */
```

10)

```
media([],0).
media(L,N):-sommatoria(L,M), num_elem(L,K), N is M/K.

?-media([2,6],4)  ---> TRUE

/* media(L,N) è vero se L è una lista di numeri e N è la media dei numeri nella lista */
```

15)

```
valida(L):-ordinata(L).

ordinata([]).
ordinata([_]).
ordinata([X1,X2|L]):- X1<X2, ordinata([X2|L]).

edge([[H1|T1], L2, L3],
    [T1, [H1|L2], L3 ]):-
   valida([H1|L2]).
edge([L1, [H2|T2], L3],
    [[H2|L1], T2, L3 ]):-
   valida([H2|L1]).
edge([[H1|T1], L2, L3],
    [T1, L2, [H1|L3] ]):-                                    (TORRE DI HANOI)
   valida([H1|L3]).
edge([L1, [H2|T2], L3],
    [L1, T2, [H2|L3] ]):-
   valida([H2|L3]).
edge([L1, L2, [H3|T3]],
    [L1, [H3|L2], T3 ]):-
   valida([H3|L2]).
edge([L1, L2, [H3|T3]],
    [[H3|L1], L2, T3 ]):-
   valida([H3|L1]).

path(A,B,[A,B]):-
   edge(A,B).
path(A,B,[A|P1]):-
   path(X,B,P1),
   edge(A,X).

/*     ?- path([[],[],[1,2,3]],[[1,2,3],[],[]],X).      */
```

CONCETTI DI CUT, FAIL e NOT

Gli operatori CUT, FAIL e NOT sono predicati esterni alla logica del primo ordine e pertanto chiamati operatori extralogici. CUT, FAIL e NOT appartengono alla logica del secondo ordine, in quanto sono predicati il cui argomento è ancora un predicato.

1) **CUT** (taglio)
[Informalmente: i cut possono essere inseriti ovunque all'interno di una clausola per evitare il backtracking ai sotto-obiettivi precedenti.]

L'operatore cut è sintatticamente rappresentato dal simbolo "!". Lo scopo del cut è quello di tagliare una parte dell'albero di ricerca. Tagliando una parte dell'albero di ricerca è possibile che si alteri la completezza del sistema, in quanto se sono possibili unificazioni nella parte di albero di ricerca tagliato tramite il cut, tali unificazioni non vengono prese in considerazione. Il cut serve per introdurre una forma di controllo sul meccanismo di backtracking automatico del motore inferenziale del Prolog.

Si consideri il predicato $A:- B_1,B_2,\ldots,B_{i-1}, B_i,\ldots,B_n$. Tale predicato è vero se i letterati B_1,B_2, B_3 fino a B_n hanno unificato. Se l'unificazione con B_i non riesce, viene applicato il fail su B_i ed il redo su B_{i-1} per tentare l'applicazione di un'altra regola presente nella KB (backtracking). Volendo inserire il cut dopo la i-sima clausola, si ha: $A:- B_1,B_2,\ldots,B_{i-1}, ! ,B_i,\ldots,B_n$. Il ruolo del cut è impedire il backtracking su B_i. Una volta applicato il fail su B_i, non può venire applicato il redo.

La prima volta che il CUT viene incontrato (cioè scendendo lungo l'albero di ricerca) viene unificato. Se è stato incontrato il CUT la prima volta, vuol dire tutte le variabili presenti nelle clausole da B_1 a B_{i-1} sono state unificate. Se per unificare una qualsiasi delle variabili presenti nelle clausole da B_i a B_n bisogna liberare una delle variabili delle clausole da B_1 a B_{i-1} tutto il predicato fallisce, in quanto viene impedito il backtracking su quelle variabili. Il predicato A risulterà quindi vero per una qualsiasi unificazione delle clausole da B_1 a B_{i-1} ma risulterà falso al primo tentativo di unificazione non riuscito su una delle variabili $B_i,\ldots,B_n$. Vengono quindi potenzialmente perse delle soluzioni, rendendo il sistema incompleto.

Il cut può essere usato essenzialmente in due modi diversi:

i. CUT VERDE
   Se è nota la posizione dell'unificazione nell'albero di ricerca, si può usare il cut per evitare esplorazioni successive inutili.

ii. CUT ROSSO
   Un altro uso del cut è quello di tagliare una parte dell'albero di ricerca in cui sono presenti delle soluzioni, in quanto si è interessati soltanto ad alcune di esse. In questo caso, il cut viene deliberatamente utilizzato per alterare la completezza del sistema, e si parlerà di cut rosso. Il cut rosso può essere facilmente fonte di effetti collaterali difficili da gestire.

   (PER DIFFERENZA TRA CUT VERDE E CUT ROSSO:
   Se dopo aver utilizzato il cut scambiamo l'ordine dei predicati e il significato dichiarativo non cambia allora si parla di cut verde, altrimenti si parla di cut rosso)

Esempio di CUT ROSSO:

```
a(1).
a(2).

b(1).
b(2).

c(1).
c(2).
c(3).

d(1).
d(2).
d(3).

e(8).

p(X,Y) :- a(X),b(X),c(Y),d(Y).
p(X,Y) :- e(X),e(Y).
```

La query p(X,Y) fornisce 7 unificazioni.

?- p(X,Y).

X = 1 Y = 1 ;
X = 2 Y = 2 ;
X = 1 Y = 2 ;
X = 2 Y = 3 ;
X = 1 Y = 3 ;
X = 8 Y = 8 ;
X = 2 Y = 1 ;

```
a(1).
a(2).

b(1).
b(2).

c(1).
c(2).
c(3).

d(1).
d(2).
d(3).

e(8).

p(X,Y) :- a(X),b(X),!,c(Y),d(Y).
p(X,Y) :- e(X),e(Y).
```

La query p(X,Y) fornisce 3 unificazioni.

?- p(X,Y).

X = 1 Y = 1 ;
X = 1 Y = 2 ;
X = 1 Y = 3 ;

Nel programma logico in cui non è presente il cut, il motore inferenziale ha trovato sette unificazioni. L'inserimento del cut ha alterato il numero di soluzioni trovate.
Trovate le prime tre unificazioni va effettuato un passo di backtracking sulla clausola b(X). Tuttavia il CUT fa fallire il redo sulla clausola b(X), cioè impedisce che la variabile X, attualmente unificata, venga liberata in favore di una successiva unificazione, rendendo impossibili tutte le unificazioni che si sarebbero potute successivamente verificare.
E' da notare che durante la prima unificazione (call) il CUT viene ignorato.

Esempi di CUT VERDE:

a)

```
leggival(N):-
        N is 1,!,write('hai inserito 1.').
leggival(N):-
        N is 2,!,write('hai inserito 2.').
leggival(N):-
        N is 3,!,write('hai inserito 3.').

leggival(N):- write('Hai inserito un valore non valido.').
```

b)

Il programma logico proposto per l'eliminazione da una lista ha una prima unificazione che è quella che ci si aspetta. Tuttavia, a causa del fatto che le clausole sono poste in OR non esclusivo tra loro, vengono trovate altre unificazioni che non corrispondono ad eliminazioni.

Ciò che si vuole fare è quindi eliminare la possibilità di unificazioni successive alla prima. Per fare questo è possibile modificare il programma logico inserendo un CUT VERDE.

```
cancella(El,[],[]).
cancella(El,[El|C],C1) :- !,cancella(El,C,C1).
cancella(El,[T|C],[T|C1]) :- cancella(El,C,C1).
```

Ricapitolando quindi il cut ha 2 effetti:

1. non puoi tornare indietro
2. la regola chiamata non può essere richiamata una seconda volta

Nota: il cut è sempre vero.

2) **FAIL**

Il FAIL è un predicato che si usa per inserire un controllo sul backtracking. Lo scopo del FAIL è complementare a quello del cut. Serve infatti a forzare un fallimento sull'unificazione in modo da consentire al backtracking di seguire un nuovo ramo.

3) **NOT**

```
not(P) :- P, !, fail.
not(P).
```

Se il predicato P è vero, la prima clausola risulta falsa a causa del fail, ed il cut fa fallire tutto il predicato. Viceversa, se il predicato P è falso, la prima clausola è falsa, ma la seconda è vera (clausola di catch-all). È da notare che il predicato NOT funziona soltanto se l'ordine dei predicati in or è quello indicato e se l'ordine delle clausole all'interno del primo predicato è quello indicato. L'ordine dei predicati e delle clausole nei predicati diventa importante quando le clausole contengono operatori che non sono del primo ordine.

Il significato del NOT è diverso dal not booleano:

a) Il not booleano ha successo sse il suo argomento è falso
b) Il NOT extralogico ha successo sse non è possibile alcuna unificazione delle variabili del predicato.

Esempio:

not($A(x_1, x_2, x_3, \ldots, x_n)$)
è vero se non esiste alcuna unificazione contemporanea di tutte le varibili $x_1, x_2, x_3, \ldots, x_n$

NEGATION AS FAILURE

Come possiamo rappresentare in Prolog la frase "Mary ama tutti gli animali, ma non i serpenti"?

La parte complicata è il "ma non i serpenti"

L'idea è formularla nel seguente modo: se X è un serpente, allora a Mary non piace, altrimenti se è un animale si:

```
likes(mary, X):-
        snake(X),
        !,
        fail.

likes(mary, X):-
        animal(X).
```

fail, usato insieme al !, permette di esprimere il concetto di non riuscita di una regola, impedendo il backtracking e quindi l'utilizzo della regola successiva

Ragionamento analogo per la regola different che verifica se due termini sono differenti o meno:

```
different(X,X):-
        !,
        fail,
different(_,_).
```

Sfruttando la regola per il not, è possibile riscrivere il different in modo più semplice:

```
different2(X,Y):-
not(X=Y).
```

CONTINUO PROGRAMMI:

16)

16.1)

```
12  prova(a):-riprova(X),write(X),nl,fail.
13
14  riprova(c).
15  riprova(d).
16  riprova(e).
17  riprova(f).
18  riprova(g).
```

/*FAIL INTRODUZIONE FATTA A LEZIONE

CON IL FAIL ESPLORA TUTTO, SE TOGLIAMO A QUESTO CODICE IL FAIL DICE SOLO IL PRIMO RISULTATO
CON FAIL OBBLIGHIAMO A ESPLORARE TUTTO LO SPAZIO
*/

16.2)

/*PRIMO APPROCCIO PER VEDERE "COME SI FA LA NEGAZIONE?" */

```
27  prova(X):-riprova(X),!,write(X),nl,fail.
28  prova(_).
29
30  riprova(c).
31  riprova(d).
32  riprova(e).
33  riprova(f).
34  riprova(g).
```

?-prova(c). FALSE
?-prova(k). TRUE

16.3)

```
46  not_member(X,[X|_]):-!,fail.
47  not_member(X,[_|M]):- !,not_member(X,M).
48  not_member(_,_).
49
50  non_vero(P):-P,!,fail.
51  non_vero(_).
```

/*LA NEGAZIONE IN GENERALE SI FA COSI
?-non_vero(member(b,[a,b,c]))
*/

16.4)

/* USANDO IL CUT CHE SI INDICA CON ! (CUT SEMPRE VERO) */

```
62  prova(X):-riprova(X),write(X),nl,fail.
63
64  prova1(X):- riprova(X),write(X),nl,false.
65
66  prova2(X,B,C):-riprova(X,B),!,riprova2(X,C).
67
68  riprova(d,1).
69  riprova(d,2).
70  riprova(e,3).
71  riprova(c,4).
72  riprova(f,5).
73  riprova(g,6).
74  riprova2(d,2).
75  riprova2(e,2).
```

PREDICATI ASSER – RETRACT – LISTING

In prolog posso modificare il programma durante l'esecuzione modificando sia fatti che regole (quindi non solo la parte dati ma anche la parte programma), quindi in un certo senso prolog può evolvere se stesso. Nell'esecuzione del programma nella parte in cui evolve nel tempo è possibile modificare, togliere e mettere datti e regole (programmare il programma col programma stesso).
2 predicati di base sono assert e retract che non agiscono solo su fatti, ma consentono di asserire e retrattare regole (sono utilizzati anche nella programmazione dinamica).
Nota: non si può asserire un fatto negato.
Assert e retract possono essere utilizzati per trovare tutte le soluzioni di un predicato (con o senza ripetizioni).

- **ASSERT**

  assert(X) aggiunge un nuovo fatto o clausola al database. Il termine viene affermato come l'ultimo fatto o clausola con lo stesso predicato chiave.

  Esempio:

```
:- dynamic good/2.
:- dynamic bad/2.
assert(good(skywalker, luke)).
assert(good(solo, han)).
assert(bad(vader, darth)).

?- listing(good).
```

  Altre forme di assert sono asserta(clausola) e assertz(clausola):
  assertz aggiunge il fatto alla fine del database
  asserta aggiunge il fatto all'inizio del database

- **RETRACT e RETRACTALL**

  retract(X) rimuove il fatto o la clausola X dal database
  retractall(X) rimuove dal database tutti i fatti o le clausole per i quali la testa si unifica con X

  Esempio:

```
retract(bad(vader, darth)).
retractall(good(_, _)).
?- good(X, Y).
No
```

- **LISTING**

  Il meta-predicato listing può essere usato per verificare cosa c'è effettivamente nel database Prolog durante una sessione. Può essere usato quindi per vedere cosa si è asserito.

CONTINUO PROGRAMMI:

17)

```
 9   p(a,b)=L.
10   L=[p,a,b].
```

/*
PROLOG MODIFICARE PROGRAMMA DURANTE L'ESECUZIONE
-PREDICATI ASSERT E RETRACT
OPERATORE UNIV
?-assert(p(a,b)). TRUE
?-listing(p).
?-retract(p(a,b)).
?- listing(p).
*/

18)

```
4   p(a,b)=L.
5   L=[p,a,b].
6
7   ritrattaTutto(P):-retract(P),fail.
8   ritrattaTutto(_).
```

/*
?-listing(ritrattaTutto(p(A,B))).
?-listing(p).
*/

PREDICATI DINAMICI

I predicati il cui significato può essere modificato a runtime si dicono dinamici.
Alcuni interpreti Prolog (tra cui SWI) richiedono una dichiarazione dei predicati dinamici:

:- dynamic fatto/arità

dynamic è una direttiva che può essere inserita nel programma (non come query).

Se andiamo ad inserire una clausola di programma che non esisteva nel programma originario questa viene considerata dinamica da prolog.
Affinchè un predicato già consultato nel programma prolog sia manipolato durante il runtime, deve essere dichiarato come dinamico. Lo facciamo aggiungendo la direttiva dynamic all'inizio del codice.
Se il predicato è "nuovo" allora è automaticamente dinamico e non deve essere dichiarato.

(Nota: La programmazione dinamica è utilissima quando ci sono dei predicati ricorsivi che dentro di se usano risultati precedentemente computati)

SETOF() BAGOF() E FINDALL()

var¬
setof(schema,goal,listaRisultati)
bagof(// , // , //)

Esempio: setof(k(A),p(A),L).

Più dettagliatamente:

bagof(Term,Goal,L) : produce una lista L di tutte le istanze di Term che si ottengono soddisfando Goal in tutti i modi possibili. Se Goal ha altre variabili oltre a quelle che compaiono anche in Term, bagof fornirà diverse soluzioni. Fallisce se Goal fallisce.

setof(Term,Goal,L) : si comporta come bagof, ma la lista L è ordinata e senza duplicati.

Esempio:

```
figlio(carlo,mario).
figlio(carlo,maria).
figlio(mario,alberto).
figlio(maria,alberto).

discendente(X,Y) :- figlio(X,Y).
discendente(X,Y) :- figlio(X,Z), discendente(Z,Y).

?- bagof(X,discendente(carlo,X),Antenati).
Antenati = [mario, maria, alberto, alberto].

?- setof(X,discendente(carlo,X),Antenati).
Antenati = [alberto, maria, mario].
```

findall(Term,Goal,L): produce una lista L di tutti gli oggetti di Term che si ottengono soddisfando Goal.

Esempio:

```
 ?- findall(X, member(X, [1,2,3,4]), Results).
Results = [1,2,3,4]
yes
```

Ricapitolando:

- **setof/3** works very much like `findall/3`, except that:
  - It produces the ***set*** of all results, **with any duplicates removed**, and the results ***sorted***.
  - If any variables are used in the goal, which do not appear in the first argument, `setof/3` will return a separate result for each possible instantiation of that variable:

```
| ?-setof(Child, age(Child,Age),Results).
    Age = 5,
    Results = [ann,tom] ? ;
    Age = 7,
    Results = [peter] ? ;
    Age = 8,
    Results = [pat] ? ;
    no
```

```
age(peter, 7).
age(ann, 5).
age(pat, 8).
age(tom, 5).
age(ann, 5).
```

Knowledge base

bagof/3 is very much like `setof/3` except:

- that the list of results might contain duplicates,
- and isn't sorted.

```
| ?- bagof(Child, age(Child,Age),Results).
     Age = 5, Results = [tom,ann,ann] ? ;
     Age = 7, Results = [peter] ? ;
     Age = 8, Results = [pat] ? ;
     no
```

bagof/3 is different to `findall/3` as it will generate separate results for all the variables in the goal that do not appear in the first argument.

```
| ?- findall(Child, age(Child,Age),Results).
     Results = [peter,pat,tom,ann,ann] ? ;
     no
```

CONTINUO PROGRAMMI:

19)

```
:-dynamic appoggio/1.
appoggio([]).
tuttelesoluzioni(_):-
                        p(A),
                        appoggio(L),
                        assert(appoggio([A|L])),
                        retract(appoggio(L)),
                        fail.
tuttelesoluzioni(L):-   appoggio(L1),
                        reverse(L1,L),
                        retract(appoggio(L1)),
                        assert(appoggio([])).
p(a).
p(b).
p(c).
p(c).
p(d).
```

```
/*
Con   p(a)
      p(b)
      p(c)
      p(c)
      p(d)
tuttelesoluzioni() è TRUE se L=[a,b,c,c,d]

?-tuttelesoluzioni(X).
prova ?-setof(k(A),p(A),L). per vedere funzionamento di setof
prova ?-bagof(k(A),p(A),L). per vedere funzionamento di bagof
*/
```

20)

```
p(a,1).
p(b,2).
p(c,3).
p(c,4).
p(d,5).

p(X):-p(_,X).
```

```
/*
CON setof() VOGLIAMO AVERE LA LISTA [1,2,3,4,5] con
p(a,1)ecc...

?- setof(k(B),A^p(A,B),L).
*/
```

21)

```
?- 3+4 = +(A,B).
A = 3,
B = 4.

?- 3 + 4 + 5 = +(A,B).
A = 3+4,
B = 5.

?- 3 + (4 + 5) = +(A,B).
A = 3,
B = 4+5.

?- 3+4*5 = +(A,*(B,C)).
A = 3,
B = 4,
C = 5.

?- 3+4*5 = +(*(B,C),A).
false.

C is +(4,5).
C = 9.
```

```
MODIFICA OPERATORE:
:-op(500,xfy,+).
scambiato yfx con xfy,
quindi se prima avevamo A=3+4,B=5
ora abbiamo A = 3, B = 4+5


CREAZIONE OPERATORE
:-op(200,xfx,[loves,hates]).
mario loves maria.
mario hates what.

?- mario hates what.
true.

?- mario hates maria.
false.

?- mario loves  maria.
true.
```

22)

```
ESERCIZIO:DEFINIRE INSIEME OPERATORI CHE CI PERMETTONO
          DI INTERPRETARE LE FRASI.


:-op(200,xfx,[mangia,porta]).
:-op(100,xfx,con).
:-op(150,xfx,di).

mario mangia mela di gianni con coltello.
mario mangia pera di michele.
mario mangia zucchina.
mario mangia panino con mortadella.

mario porta borsa di pelle di leopardo.


ESEMPIO QUERY

?- mario mangia mela di Chi con Cosa.
Chi = gianni,
Cosa = coltello.

?- mario mangia mela di gianni con coltello.
true.
```

29)

PERMUTAZIONE:

```
perm([],[]).
perm(L,[X|LPX]):-
    appartiene(X,L,LX),
    perm(LX,LPX).

appartiene(X,[X|L],L).
appartiene(X,[Y|L],[Y|L1]):-
    appartiene(X,L,L1).
```

OPERATORE UNIV

Operatore infisso in che si scrive con =.. nella seguente maniera:

Term = .. L

Univ è vero se L è una lista che contiene il funtore principale di Term seguito dai suoi argomenti.

Il suo scopo è quello di dato un termine scomporlo in una lista in cui il primo elemento è il funtore del termine e gli altri elementi sono i suoi argomenti.

Esempio:	?-f(a,b)=..L	?-T=.. [is_blue,sam,today]
	L=[f,a,b] ?	T=is_blue(sam,today) ?
	yes	yes

VERSIONE QUASI COMPLETA MANCA SOLO:

Vedi esercizi da 24 a 26
+
inserire concetto di albero in prolog con anche visite dell’albero utilizzando video prof su yt!!!! Esercizio d’esame!

/*Domande salva esame:
* faccia una reverse
* faccia una permutazione
* domande del genere
* faccia una member */

/*Esercizi d'esame:
* Problema con grafi
* Unificazione
* Grammatiche*/
/*

UTILE:https://www.youtube.com/watch?v=mw5pzuM-0d0&list=PL_JeImB8DWObszSniVu51k6ZK3S4brFG8&index=10