Programmazione ad oggetti

Programmazione ad oggetti

0 Introduzione agli appunti

Questi appunti non forniscono in alcun modo un metodo di studio.

Il codice scritto negli esempi non é stato mai compilato, quindi potrebbe contenere errori di sintassi.

Il prof Ferrara (che tiene il corso) **NON** é in nessun modo affiliato alla repo da cui viene questo file e di conseguenza **NON** é responsabile di eventuali errori presenti in esso
~~(molto probabilmente non é nemmeno a conoscenza di questi appunti)~~

0.1 Licenza (CC BY-SA 4.0)

0.2 Scelta di determinati termini inglesi

Gli appunti sono totalmente in italiano, tuttavia in diverse parti troverete parole inglesi che non sono state tradotte, questo perché:

1. Non é tanto la traduzione letterale ad assumere un senso quanto la spiegazione del concetto che c'é dietro
2. Alcune non hanno nemmeno una traduzione in italiano
3. Come ho giá detto in sostanza potresti non sapere nulla di inglese e consultare comunque questi appunti imparando il concetto che c'é dietro a certe sequenze di lettere a te sconosciute ma se non sai un minimo di inglese ti vedo male nel mondo della scienza.

Detto questo spero vi possa essere utile questo ripasso di OOP <3

1 Introduzione

Una classe é una fabbrica di oggetti
Un oggetto ha dei campi e dei metodi, i campi si possono toccare solo attraverso metodi definiti da chi ha programmato la classe

1.1 Perché Java?

- Orientato ad oggetti
- Community molto grande e tante librerie
- Portable (gira indipendentemente dall'OS)

file.java = javac (compiler) => Bytecode.jar/.class = VM => EXEC

JRE - (Java Runtime Environment) Serve per runnare
JDK - (Java Developement Kit) Serve per compilare

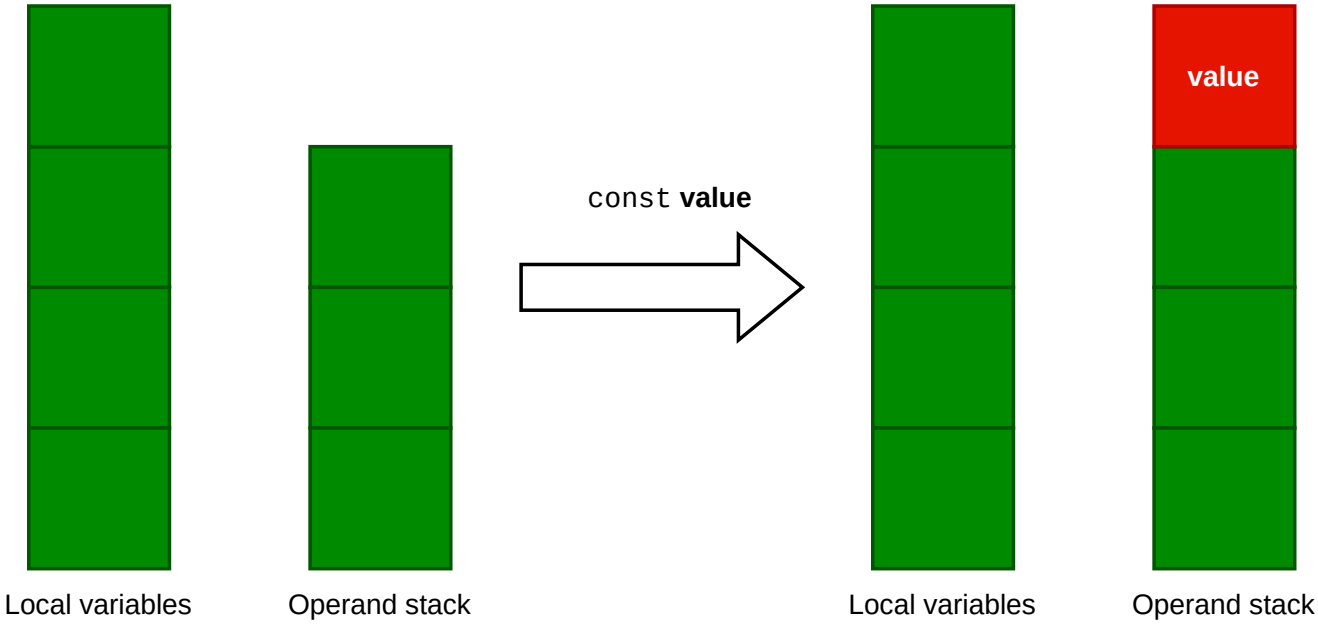
1.2 Java ByteCode

- é:
  - machine-independent low-level language
  - object-oriented
  - garbage-collection-based
- L'esecuzione é data da:
  - uno stack di vari frame (one per chiamata a metodo) contenenti:
    - una pool di variabili locali contententi ognuna i propri valori
    - uno stack degli operandi
  - una memoria contenente gli oggetti

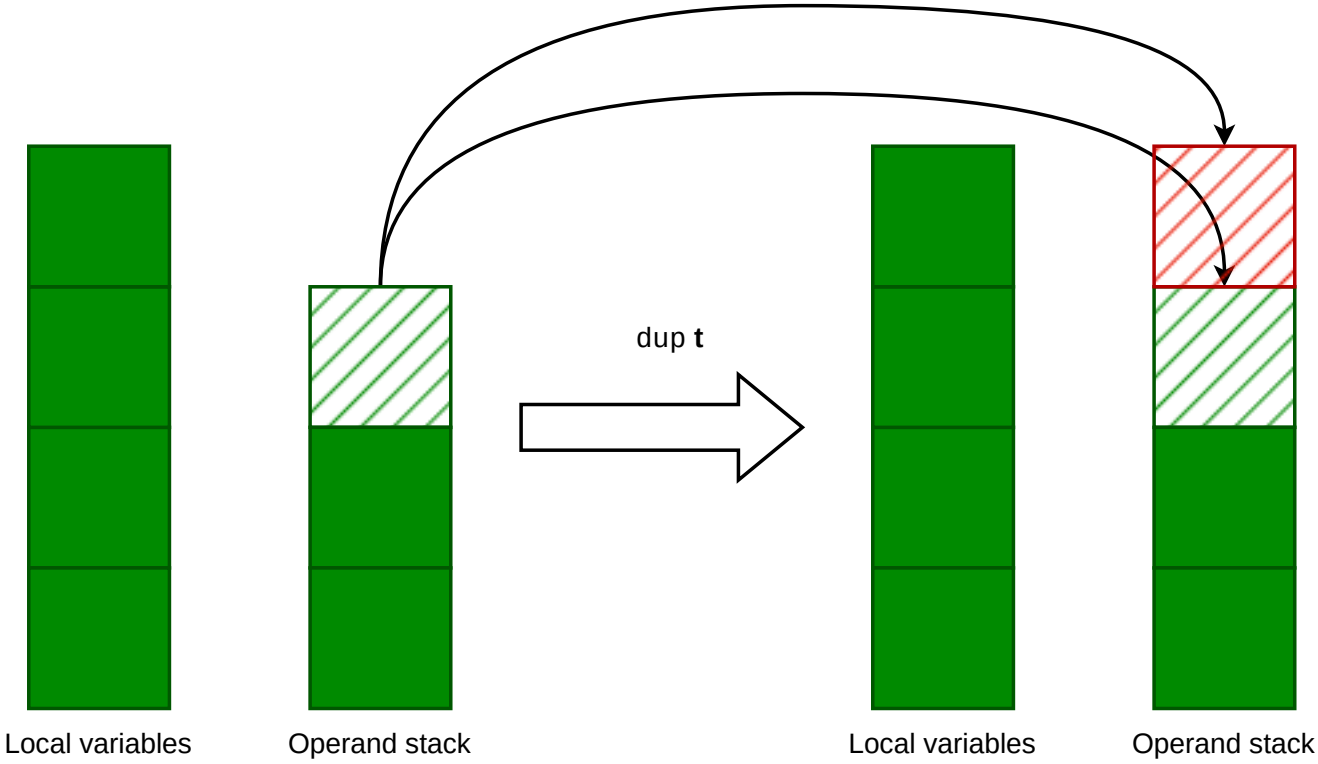
Il bytecode é un linguaggio molto stabile e divisibile in varie categorie di operazioni:

- salvare o caricare variabili locali
- scrivere o leggere zone di memoria heap (non lo stack)
- invocare metodi
- eseguire operazioni aritmentiche
- effettuare valutazioni condizionali

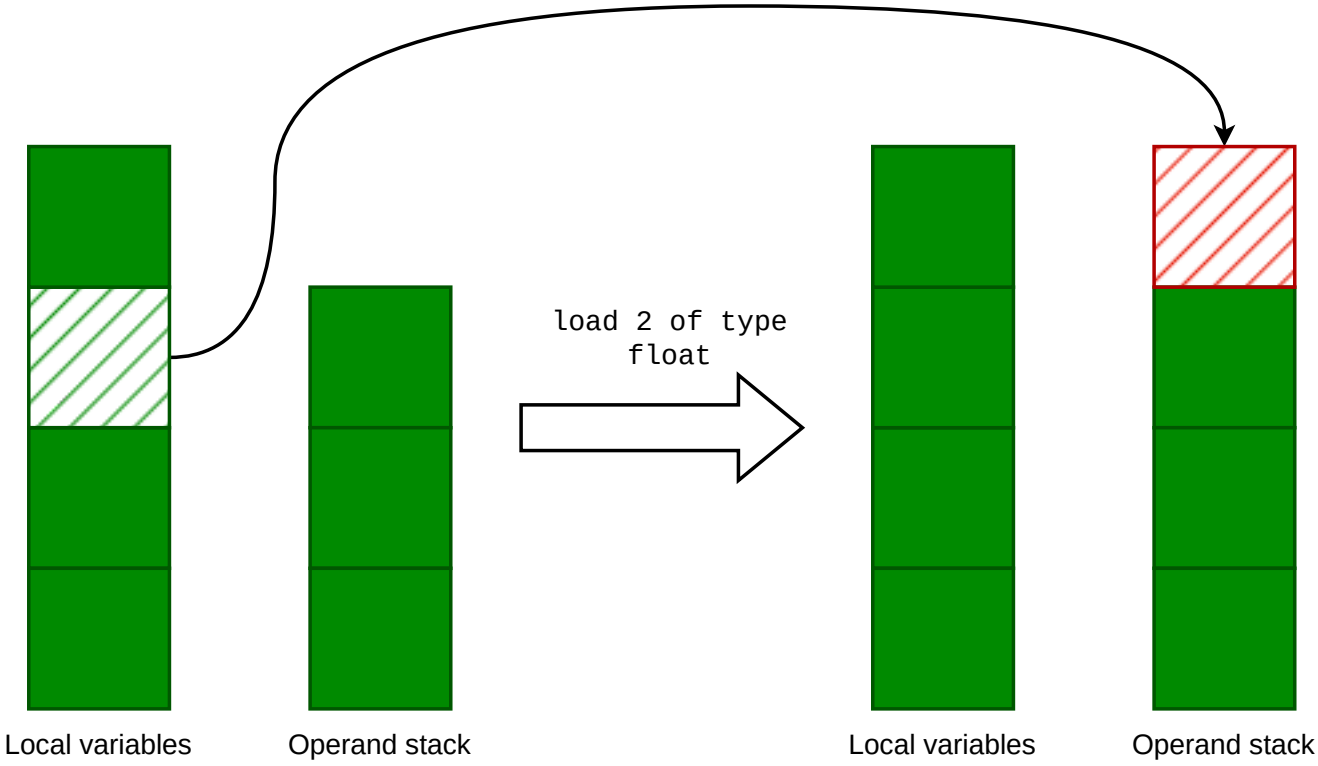
1.2.1 const



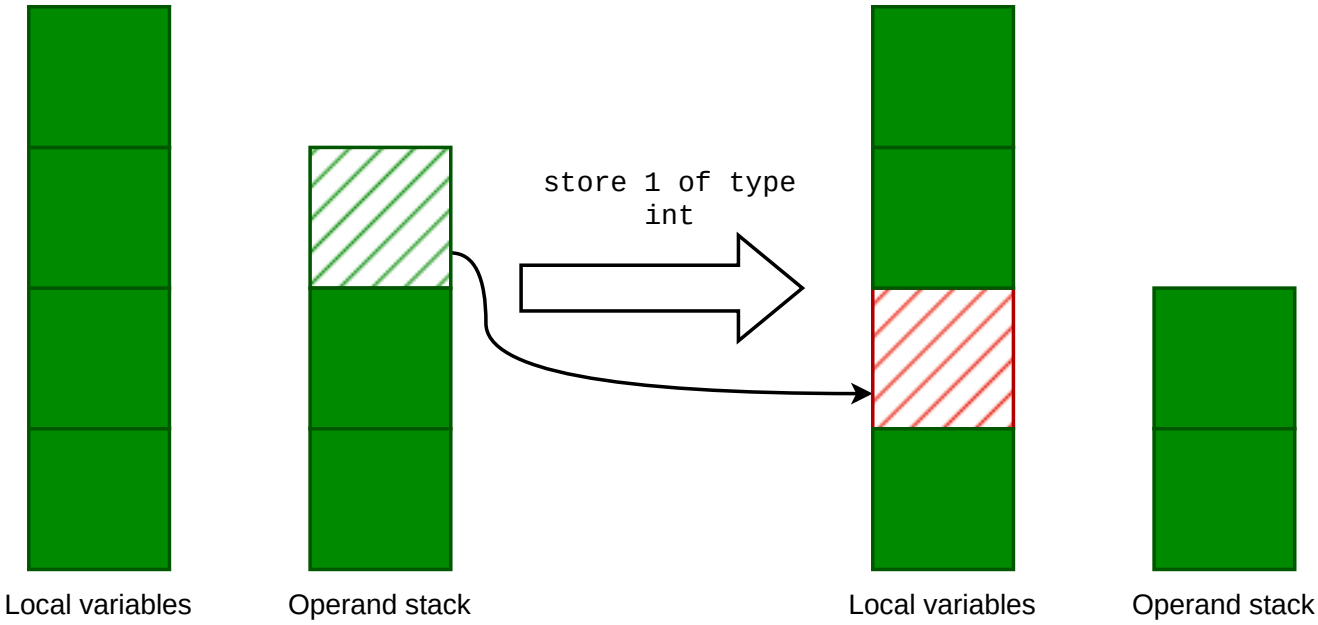
1.2.2 dup



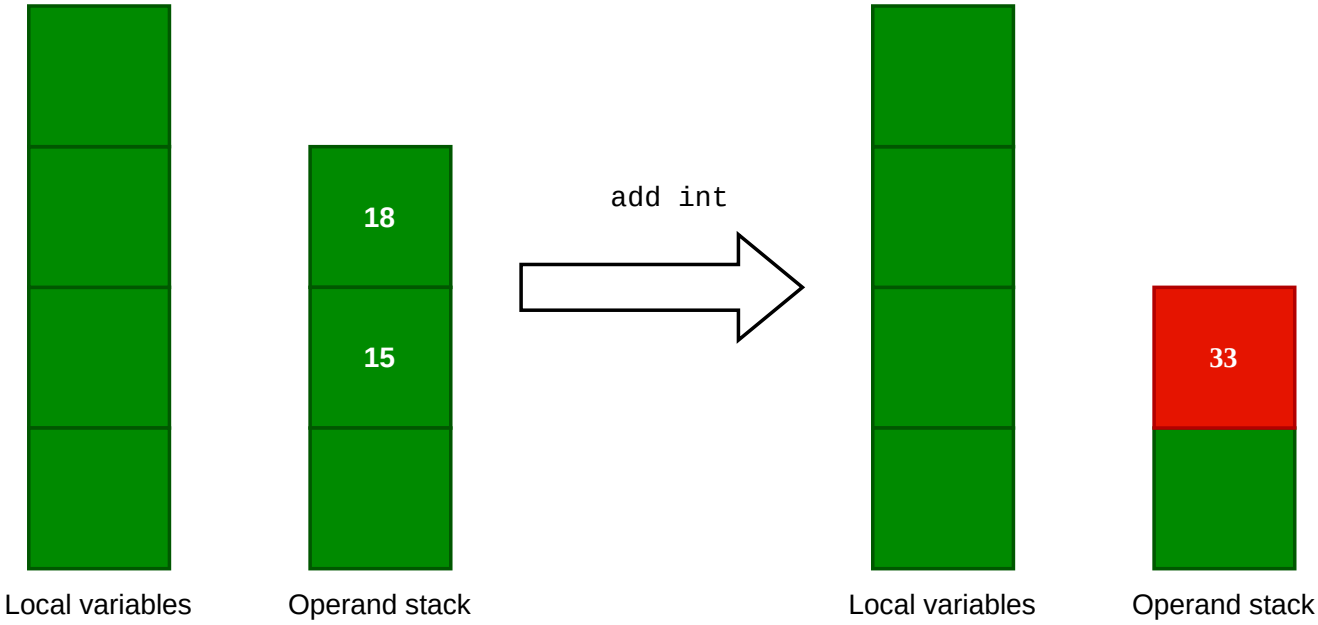
1.2.3 load



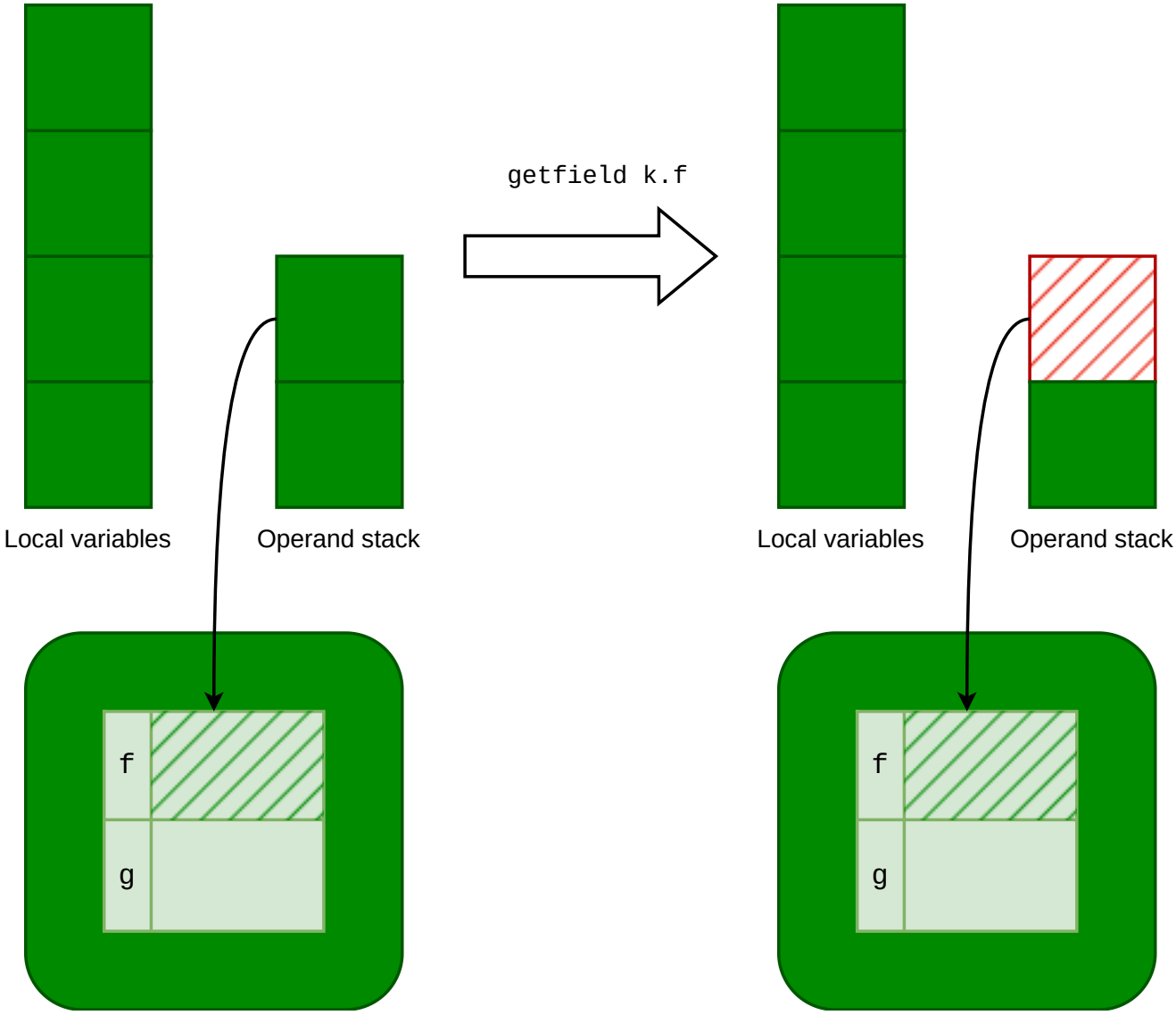
1.2.4 store



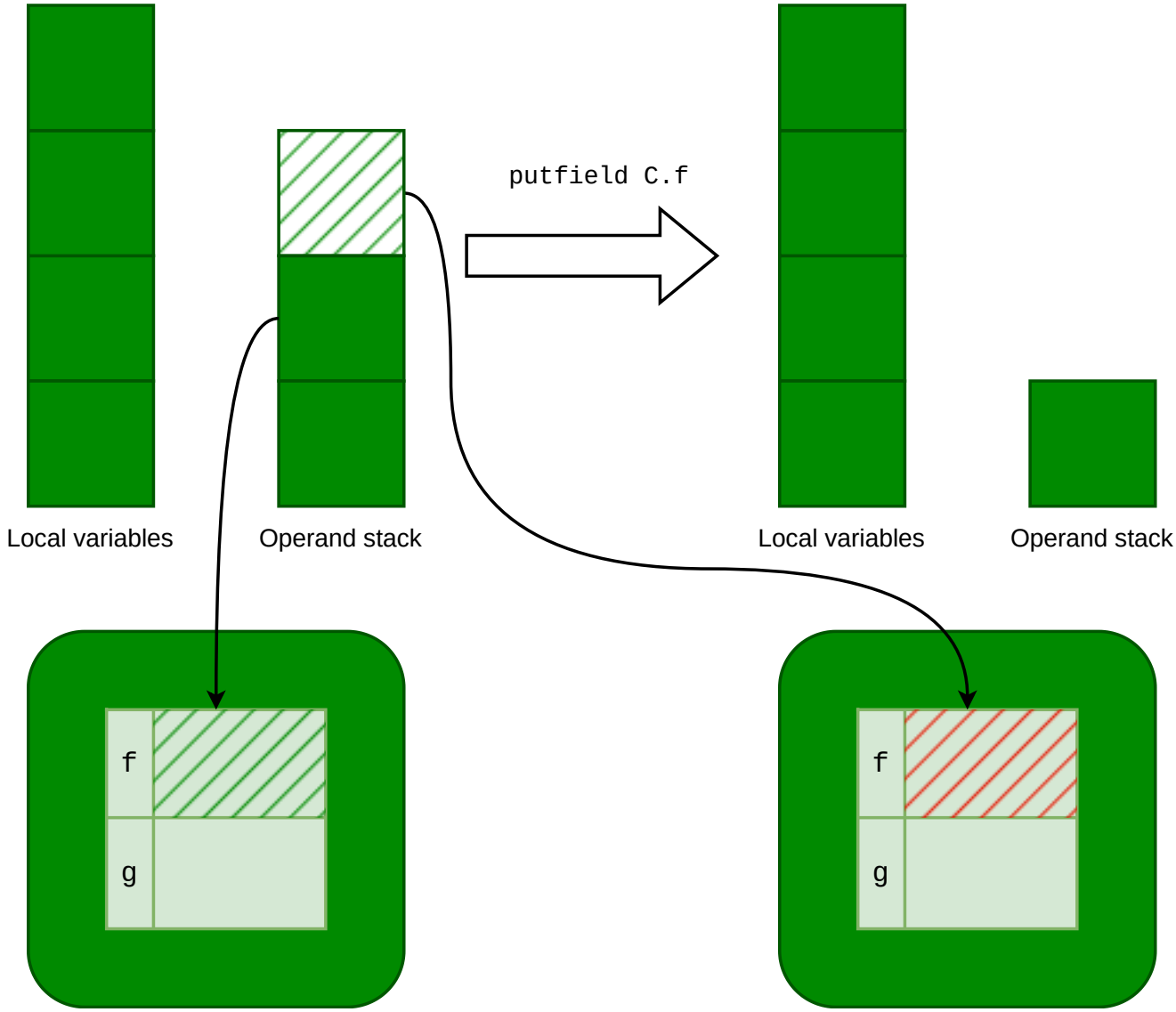
1.2.5 add



1.2.6 getfield



1.2.7 putfield



1.3 Java History

- Sviluppato da James Goslin alla Sun Corp.
- 2010: Oracle compra Sun
- da java 9 ogni 6 versioni una é LTS (long term support)
  - Java 9, 10, 12, 13, 14, 15 non sono piú supportate
  - Java 11, 17, 23, ... sono LTS
- Oracle é uno dei fornitori di JDK e JRE (quello ufficiale) ma esistono anche altri come OpenJDK

1.4 Tipi valore VS Tipi Reference

In java abbiamo due categorie di tipi:

- Valore (primitivi):
  - int
  - long
  - float
  - double

    - booleans
    - char
- Reference:
    - Oggetti (spoiler)
    - array
    - stringe
    - ...

1.5 Classi

Sono blueprint, modellano quindi come deve essere fatta una determinata cosa, questa cosa é detta: **Oggetto**

Oggetto: Un istanza della classe
Una classe puó essere istanziata varie volte, ogni istanza ha uno stato diverso.

Le variabili che contengono questi oggetti sono delle reference alla memoria heap (che contiene il vero oggetto)

1.5.1 Definire una classe

```
class <class_name> {
    <field1_type> field1_name [ = <initial_value>];
    <field2_type> field2_name [ = <initial_value>];

    <method1_returntype> <method1_name>(<method1_pars>){
        <method1_body>
    }
    <method2_returntype> <method2_name>(<method2_pars>)
    {
        <method2_body>
    }

}
```

1.5.2 Instanziare una classe

```
ClassName ObjName = new ClassName() // costruttore (spoiler)
```

1.5.2.1 Costruttore

il costruttore é il metodo usato per costruire una classe, é definito all'interno della stessa e ce ne puó essere piú di uno in Java con diversi parametri

Esempio:

```
class Bottle{
    String content;
    double amount;
    double capacity;

    Bottle(String content,double amount,double capacity){
        this.content = content;
        this.amount = amount;
        if(capacity >= amount){
            this.capacity = capacity;
        }else{
            this.capacity = -1
            /*
              non é il modo corretto
              per gestire queste cose,
              ma vedremo dopo come fare
            */
        }

}
```

1.5.2.2 This

`This` é una keyword di java

- punta all'oggetto corrente
- passa una reference dello stesso ad altri metodi
- puó essere usata per accedere ai campi e i metodi dell'oggetto corrente (come abbiamo fatto sopra)
- permette di chiamare un altro costruttore con `this(params)` da un altro costruttore a patto che sia la prima riga del costruttore che chiama

1.5.2.3 new

`new` é una keyword di java usata per istanziare oggetti ed esegue varie operazioni:

1. Alloca la memoria nella heap
2. Inizializza i campi dell'oggetto da istanziare
3. invoca il costruttore specificato
4. ritorna il puntatore all'oggetto

(siccome abbiamo il garbage collector anche se viene allocata memoria se la vede java per le free)

1.5.3 Overview su variabili locali, parametri e campi

```
class Bottle{
    //questi sono campi
    String content;
    double amount;
    double capacity;

    Bottle(
        //questi sono parametri
        String content,double amount,double capacity
    ){
        this.content = content;
        this.amount = amount;
        if(capacity >= amount){
            this.capacity = capacity;
            //questa é una variabile locale
            double fullPercentage = capacity * amount / 100;
        }else{
            this.capacity = -1
        }
    }
}
```

1.6 Package

Le classi possono essere raggruppate in `package`

- Ogni package é una unitá software che puó essere distribuita da sola o combinata con altri package o integrata in interi programmi.
- hanno la keyword `package` all'inizio.
- vanno importati prima di qualsiasi dichiarazione di classe (supportano il wildchar)
- hanno come naming convention sottodomini url al contrario:
    - (Esempio: it.wapeety.po1)
      Ovviamente i file delle classi dovranno stare in it/wapeety/po1

La struttura ad albero delle directory riflette la gerarchia e la dipendenza fra pacchetti, tutto ció che serve ad un pacchetto deve stare sotto esso

2 Incapsulation e information hiding

2.1 Abstraction e interfacce

Una classe definisce un "contratto di utilizzo" specificando ció che ogni suo oggetto puó fare, attraverso due sistemi:

- Firma dei metodi (parametri, funzioni e ritorni)

    - il costruttore (cosa vuole) e le varie funzioni come ad esempio getter e setter ci permettono di capire cosa dobbiamo passare e cosa riceveremo in ritorno.
- semantica dei metodi (documentazione esterna)
    - descrive semanticamente cosa fa ogni funzione e quindi il significato dei parametri e dei ritorni.

Tutto ció allo stesso tempo ci permette di **NASCONDERE COMPLETAMENTE** come lavora la nostra classe e quindi l'utilizzatore dovrá solo preoccuparsi di seguire le specifiche date.

2.2 Encapsulation

L'encapsulation é un concetto fondamentale della programazione ad oggetti.

Il termine Encapsulation esprime vari concetti:

- i dati su cui vengono fatte le operazioni sono legati internamente con i metodi che li usano.
- informazioni non necessarie esternamente alla classe sono nascoste dalla visualizzazione e dall'editing esterno.
  Questo permette di rendere l'interfaccia esterna di cui parlavamo prima piú chiara e piú facilmente riutilizzabile

se la classe fa tutto internamente trasportarla da un'altra parte non é un problema al fine dei suoi calcoli. Paradossalmente potrei usare la stessa classe macchina in un videogioco di corse o in un programma di una concessionaria virtuale, l'importante é che la mia classe sappia sempre "cosa é" e "come costruire/usare" una macchina (oggetto di classe car)

L'encapsulation:

- permette di garantire la consistenza dei dati e delle operazioni della nostra classe
- non é limitata alle classi (potremmo avere un **[spoiler]** `Package` composto da varie classi)

2.2.1 Modifiers

- Access modifiers:
    - sono:
        - `public`
        - `<none> (default)`
        - `protected`
        - `private`
    - si applica a campi e metodi
    - solo `public` é usato per le classi
- Concurrecy modifiers (non coperti da questo corso)
    - si applica a campi e metodi

- Static
  - si applica a campi e metodi
- Final:
  - si applica a campi, metodi, classi
- Abstract:
  -si applica a metodi, classi

Lista riassuntiva degli accessi possibili con i modifiers

	Stessa classe	Stesso Package	Sottoclassi	Dappertutto
`public`	SI	SI	SI	SI
`protected`	SI	SI	SI	NO
`<default>`	SI	SI	NO	NO
`private`	SI	NO	NO	NO

Differenti punti di vista

- Gli access modifiers definiscono diversi punti di vista della stessa classe
  - `public` tutto ció che rappresenta l'interfaccia esterna della classe
  - `protected/default` ció che deve "collaborare" nella stessa porzione logica di codice
  - `private` dati e funzioni di supporto per calcoli che riguardano solo l'implementazione interna che in un futuro potrá essere cambiata senza per questo costringere chi usa la classe/funzione a modificare qualcosa

Esempio:
Costruire una macchina

Per fare una macchina, abbiamo bisogno di definire un serbatoio di carburante e allo stesso tempo per fare un serbatoio di carburante abbiamo bisogno di definire il carburante.

Possiamo quindi dividere il tutto in due `package`, uno che riguarda il carburante e uno che riguarda la macchina:

- Package Fuel:
  - ```
    //File: it/unive/dais/po1/fuel/FuelType.java

    package it.unive.dais.po1.fuel;

    class FuelType {
       private String name;
       private double costPerLiter;
    ```
    ```

```
        private double fuelConsumption;

        public FuelType(...) {...}
    }
```

- ```
    //File: it/unive/dais/po1/fuel/FuelTank.java

    package it.unive.dais.po1.fuel;
    class FuelTank {
    private FuelType type;
    private double amount;

    FuelTank(...) {...}
    }
  ```

- Package car:

  ```
    //File: it/unive/dais/po1/car/Car.java
    package it.unive.dais.po1.car;

    /*
      importiamo dall'esterno
      tutto ció che riguarda il carburante
    */
    import it.unive.dais.po1.fuel.*;

    class Car {
       double speed;

       Car(...){...}
       void accelerate(double a) {...}
       void fullBreak() {...}
    }
  ```

Parlando di modifiers dobbiamo sottolineare infine che NON possono esistere classi `protected` o `private` , NON possono esistere classi `static` e non possono esistere campi `abstract` (non avrebbero semplicemente senso)

In oltre, non possono co-esistere i modificatori `abstract` e `static` oppure `abstract` e final (maledetti `static`!)

2.2.2 Getters

Un accesso read-only ad un valore, di solito é un semplice: `return this.field` .
  ```

- Ha il vantaggio di poter essere controllato e da la possibilitá di assicurare l'integritá del valore stesso.
- potenzialmente potrebbe essere anche un valore che non esiste realmente nella nostra implementazione ma viene calcolato runtime.
- Svantaggio: é computazionalmente meno efficiente.

2.2.3 Setters

Un accesso write-only ad un valore, di solito é un semplice:
`this.field = val` dove val é l'unico parametro della funzione.

- Permette di controllare eventuali restrizioni prima di mutare l'oggetto e potenzialmente rompere il suo stato funzionale inserendo valori non corretti.
- Svantaggio: é computazionalmente meno efficiente.

3 Documentazione e JavaDoc

Grazie a questa modularitá del codice possiamo distribuire i nostri package con facilitá sotto forma di librerie JAR

3.1 Jar

Gli archivi Jar oltre a poter essere inclusi in altri progetti possono anche essere veri e propri eseguibili

contengono `META-INF/MANIFEST.MF`, un file che contiene vari valori utili al corretto funzionamento e alla corretta implementazione dell'archivio.

```
Manifest-Version: 1.0
Specification-Title: Java Platform API Specification
Specification-Version: 11
Specification-Vendor: Oracle Corporation
Implementation-Title: Java Runtime Environment
Implementation-Version: 11.0.6
Implementation-Vendor: Oracle Corporation
Created-By: 10 (Oracle Corporation)
```

3.2 Commenti

Quella cosa che non fai mai perché
si tanto sono giusto 3 righe per fare una cosa al volo e poi mi ricordo il codice che scrivo, no?

oppure col codice aziendale

ma si, é semplice, anche i miei futuri colleghi capiranno...
e poi ora ho la release fra poco e non ho tempo di commentare,
poi facciamo refactor la prossima release e io staró sempre qua in azienda a spiegare al junior

developer di turno che cosa ho implementato e come, tanto mi ricordo la codebase aziendale di ogni lavoro, no?

e poi bestemmi tanto, tanto, tanto in entrambi i casi.

Scherzi a parte...

I commenti si dividono in due tipi:

- Commenti per il codice sorgente
  - a singola linea `// commento a singola riga`
  - multilinea `/* iniziano cosí e finiscono con */`
- Documentazione (JavaDoc) per spiegare le API di una libreria
  - `/** come i commenti multiriga ma con due asterischi all'inizio */`

i commenti JavaDoc possono essere usati per generare pagine di documentazione in HTML e vari IDE li usano come suggerimenti

3.2.1 JavaDoc

JavaDoc é uno standard nella documentazione di librerie java e produce pagine HTML consultabili e navigabili.
Ogni sezione documenta un elemento con l'ordine mostrato di seguito:

- Classi
- Campi
- Metodi

3.2.1.1 JavaDoc elements

I commenti JavaDoc possono contenere tag HTML e in piú usano:

- `@literal <text>` stamperá il testo nell'esatta forma in cui é scritto
- `@code <code>`come `@literal` ma aggiunge la formattazione del codice
- `@link <elem>` aggiunge un collegamento all'elemento (classe, campo, metodo)
- `@see <elem>` aggiunge un collegamento all'elemento nella sezione *vedi anche...*

3.2.1.2 Documentare una classe

La JavaDoc di una classe contiene:

- `@author` l'autore della classe
- `@version` la versione della classe
- `@since` la versione di java in cui la classe é stata aggiunta la prima volta

3.2.1.3 Documentare un metodo

L'interfaccia di un metodo consiste in:

- valore di ritorno `@return`
- parametri `@param <name>`
- eccezioni `@throws`
- se un elemento é stato deprecato `@deprecated`

3.2.1.4 Documentare un campo

La descrizione della variabile, non ci sono tag speciali

3.2.1.5 Cosa documentare con javadoc

Metodi/campi/classi pubblici sicuramente e a seconda della situazioni anche cose protected se non é un package totalmente nostro.

3.2.2 JML Java Modelling Language

Se consideriamo un contratto come una promessa di eseguire un qualcosa a patto di certe condizioni nella maggior parte dei casi possiamo giá assumere dai tipi di ritorno, dai parametri e dal contesto tuttavia non vi é nulla di formale in tutto ció.

In sostanza JML permette di stipulare formalmente un contratto fra l'utilizzatore di una libreria e il programmatore della stessa.

i suoi costrutti principali sono:

- `requires` specifici constraint
- `ensures` cosa ritorna
- `loop invariants` per verificare il programma

3.2.4 Doxygen

Siccome nessuno usa JML ci atteniamo allo standard Doxygen, supporta molti piú linguaggi ed é meno stretto in quanto specifiche, anche l'input e l'output possono essere in vari formati (pdf, latex, html, ...)

3.3 Ereditarietá (Inheritance)

L'ereditarietá in programmazione é il processo di derivazione e "trasmissione" di elementi, proprietá e/o funzioni da classi padri a classi figlie

3.3.1 Aggregazione

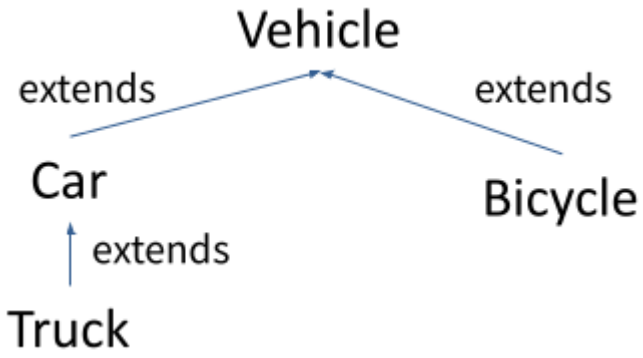
Se un oggetto si riferisce ad un altro ha con quest'ultimo una relazione quindi, due oggetti distinti, rappresentanti due concetti diversi si relazionano fra loro

3.3.2 Inheritance

Un oggetto puó estendere le funzionalitá di un altro, cioé senza riscrivere il codice di una classe una sua "figlia" puó creare oggetti che fanno ció che é descritto nella classe madre **ma** anche altro.

Ogni classe puó estendere solo una classe e questo puó essere fatto piú volte

quindi potremo avere un qualcosa del tipo:



la gerarchia é quindi rappresentabile come un albero.

In java la keyword per estendere una classe é `extends`

esempio:

```
class A{
    ...
}

class B extends A{
    ...
}
```

3.3.2.1 La keyword `super`

Una sottoclasse puó chiamare il costruttore della superclasse usando `super` con le parentesi tonde per i parametri.

`super` é un'altra keyword riservata di java molto utile nel campo dell'ereditarietá.

esempio:

```
public class Coke{
    /*sono protected
      cosí sono accessibili alle
      sottoclassi ma non fuori,
      se non te lo ricordavi torna dietro
      alla sezione 2.2.1 (modifiers) :P
    */
```

```
    protected double amount;
    protected sugar = true;

    Coke(double amount){
        this.amount = amount;
    }
    ...
}

public class DietCoke extends Coke{

    DietCoke(double amount){
        //il supercostruttore DEVE essere la prima cosa!
        super(amount);
        //qui vediamo un secondo utilizzo di super!
        super.sugar = false;
        /*l'esempio é stupido e non realistico
          ma non mi sto laureando in chimica e non
          lavoro nel mondo della cucina
        */
    }
    ...
}
```

in sostanza possiamo vedere `super` come un `this` che peró punta alla superclasse e ai suoi elementi.
OCCHIO SEMPRE ALLA VISIBILITÁ CHE QUESTI HANNO

per rispettare i canoni della buona programmazione non tutto deve essere accessibile all'esterno **ma ci sono cose che devono essere nascoste anche alle sottoclassi**.
Se serve modificare un valore si puó sempre fare una funzione `protected` (non accessibile fuori dal package) nella superclasse che con i giusti controlli modifica quel valore e chiamarla con `super.funz()` dalla classe figlia.

3.3.2.2 La keyword `abstract`

In java é possibile creare classi che NON potranno mai essere istanziate tramite la keyword `abstract`.

Il senso?

Se possiamo vedere le classi come dei blueprint che ci permettono di fare un oggetto le classi astratte sono dei "modelli preconfezionati" su come fare questi blueprint.

Le classi astratte possono implementare metodi o sempre tramite la keyword `abstract` lasciare che sia la classe figlia a farlo, ció che é importante peró é che per estendere una classe astratta **TUTTI** i metodi abstract devono essere implementati o, in alternativa, la classe figlia deve essere a sua volta astratta.

Note:
La firma di un metodo é composta da questo schema:

```
<tipo_di_ritorno> <nome_funzione>(<parametri>)
```

che quindi **NON** include il corpo dello stesso, firma e corpo sono la definizione del metodo.

esempio:

```
/*in questo caso noi non vogliamo che una
    bevanda anonima possa essere stanziata
    come oggetto, quindi la mettiamo abstract

*/
abstract public class Beverage{
    protected double amount;

    Beverage(double amount){
        this.amount = amount;
    }

    abstract public void drink(double drinkedAmount){
        //funzione abstract non implementata
    }
}

public class Coke extends Beverage{
    protected sugar = true;

    Coke(double amount){
        super(amount);
    }

    public void drink(double drinkedAmount){
        if(drinkedAmount > amount){
            this.amount = 0;
        }
        else{
            this.amount = this.amount - drinkedAmount;
```

```
        }
    }
    ...
}
```

3.3.2.3 Overriding

Nulla vieta ad una classe figlia di riscrivere un comportamento della classe madre, questa procedura é chiamata overriding.

esempio:

```
public class Coke{
    protected double amount;
    protected sugar = true;

    Coke(double amount){
        this.amount = amount;
    }

    public void drink(double drinkedAmount){
        if(drinkedAmount > amount){
            this.amount = 0;
        }
        else{
            this.amount = this.amount - drinkedAmount;
        }
    }
}

public class DietCoke extends Coke{

    DietCoke(double amount){
        super(amount);
        super.sugar = false;
    }

    public void drink(double drinkedAmount){
        /*quando bevo diet coke
          preferisco berla in due sorsi
        */
        super.drink(drinkedAmount/2);
        super.drink(drinkedAmount/2);
    }
```

```
    //non mi picchiate se gli esempi fanno schifo
}
```

3.3.2.4 Overloading

Importante da non mischiare con l'overriding é l'overloading

piú metodi con lo stesso nome possono avere piú implementazioni, questo puó essere fatto nella stessa classe cosí come in una classe figlia

esempio:

```
public class Coke{
    protected double amount;
    protected sugar = true;

    Coke(double amount){
        this.amount = amount;
    }

    public void drink(double drinkedAmount){
        if(drinkedAmount > amount){
            this.amount = 0;
        }
        else{
            this.amount = this.amount - drinkedAmount;
        }
    }

    //metodo in overload
    public void drink(double drinkedAmount, int sips){
        for(int i=0; i < sips, i++)|{
                if(drinkedAmount/sips > amount){
                    this.amount = 0;
                }
                else{
                    this.amount = this.amount - (drinkedAmount/sips);
                }
        }
    }
}
```

3.3.2.5 Accessibilitá dei metodi overridden

Come giá implicitamente detto prima i modificatori non sono parte della firma del metodo, tuttavia esistono delle regole da rispettare nell'overriding.

I modificatori possono essere "aperti" maggiormente, quindi un metodo `protected` puó diventare `public` ed estendere la propria visibilitá

ci sono delle eccezioni:

- i metodi final non possono subire override
- i metodi statici non possono subire override
  (entrambe cose di cui non abbiamo parlato)

3.3.3 Sostituire una classe con una sua figlia

Se abbiamo come parametro di una funzione una classe madre possiamo passare anche una classe figlia proprio per la proprietá per cui una classe che ne estende un'altra puó solo allargare l'interfaccia della madre (di questo ne parleremo meglio nel capitolo 6 - Subtyping)

4 Final

`final` é una keyword in java fin troppo usata e il significato puó cambiare molto (come vedremo successivamente) a seconda del contesto in cui viene utilizzata.

4.1 Final applicato ai metodi

Per evitare che un metodo possa subire l'override in una classe figlia si usa `final` prima della sua dichiarazione

Metodi astratti e costruttori **non** possono essere final

esempio:

```
final public void doSomethingImportant(){
    ...
}
```

4.2 Final applicato ai campi

Un campo con anteposto il modificatore `final` non potrá modificare il suo valore che dovrá obbligatoriamente essere assegnato o nella dichiarazione o nel costruttore.

esempio:

```
public class FooClass{
    final public boolean prod = true;
    final private int grade;

    FooClass(int grade){
        this.grade = grade;
```

```
    }
}
```

4.3 Final applicato alle classi

La keyword `final` applicata alle classi invece permette di renderla non estendibile, quindi saremo sicuri che non esisterá una classe "figlia" che prende le proprietá dalla nostra classe.

Per via di questo comportamento alcune regole di visibilitá risultano inutili o molto "strette", infatti, impostare una classe a final é di persé molto limitante e i campi `protected` risulteranno uguali a quelli con il modifier di default.

Un approccio migliore sarebbe impostare (come visto prima) tutti i metodi che vogliamo a final ma lasciare aperta la possibilitá di estensione.

5 Static

`static`, a differenza di `final` é una keyword che non cambia molto il suo significato in base all'applicazione, puó essere applicato solo a metodi e campi.

5.1 Static applicato ai metodi

metodi statici appartengono alla classe stessa e dovrebbero quindi essere chiamati con `NomeClasse.nomeMetodoStatico()` e non con `ObjDiNomeClasse.nomeMetodoStatico()` poiché agiscono **su tutta la classe** e non su una sua specifica istanza.

possono accedere sono a campi statici (ne parleremo fra poco)

5.2 Static applicato ai campi

I campi `static` sono condivisi (sia logicamente che a livello di memoria) fra tutte le istanze di una classe, anche loro dovrebbero essere chiamati con `NomeClasse.nomeCampoStatico` e non con `ObjDiNomeClasse.nomeCampoStatico` poiché agiscono **su tutta la classe** e non su una sua specifica istanza.

Nulla vi vieta tecnicamente di usare la seconda notazione ma se vi sgamo vi vengo a prendere a sotto casa <3.

I campi `static` devono essere inizializzati tramite lo `static` constructor che ha questa forma:

```
static{
    //qua inizializzo tutto
    ...
}
```

6 Subtyping (questo non ha traduzione)

Java, come giá detto, ha dei tipi predefiniti questi sono conosciuti a tempo di compilazione e quindi in caso di casting impliciti illegali o assegnazioni di dati a tipi non compatibili il programma non compila.

Le operazioni devono quindi essere compatibili col tipo.
Tutto é tipato, non sono variabili e campi

6.1 Sostituire una classe con una sua figlia

Secondo il principio per cui una sottoclasse puó solo estendere o al massimo modificare l'implementazione dell'interfaccia un istanza di una superclasse puó essere sostituita da una sottoclasse

esempio:

```
abstract public class Item{

    public double price;

    Item(double price){
        this.price = price;
    }
}

public class Book extends Item{

    protected String name;
    protected String author;
    protected String ISBN;

    Book(double price, String name, String author, String ISBN){
        super(price);
        this.name = name;
        this.author = author;
        this.ISBN = ISBN;
    }
}

public class Tshirt extends Item{

    protected char size;
    protected String imgPath;

    Tshirt(double price, char size, String imgPath){
        super(price);
```

```
        this.size = size;
        this.imgPath = imgPath;
    }
}
/*
    a questa funzione getPrice posso passare
    sia un Book che una Tshirt siccome so che
    entrambi per ereditarietá avranno il campo
    price
*/
double getPrice(Item item){
    return item.price;
}
```

6.2 Polimorfismo

Si parla di polimorfismo (in questo caso per inclusione) quando descritto un dato di tipo A per ci ritroviamo ad assegnargli un valore di tipo B sottoclasse di A.

Per fare questo in java necessitiamo di:

- Ereditarietá
- Subtyping

7 Tipi statici e dinamici

Ogni espressione in java ha in realtá due tipi:

- tipo statico
- tipo dinamico

Questo comportamento é fondamentale al fine di far funzionare il polimorfismo e quindi a cascata possiamo dire che permette subtyping ed ereditarietá.

Il tipo statico é definito a tempo di compilazione mentre il tipo dinamico viene conosciuto a tempo di esecuzione e per la stessa espressione possono essere differenti.

Ma come!? Java non era fortemente tipato? perché adesso si comporta diversamente?

In realtá il fatto che questi possono essere differenti ha una condizione... il tipo dinamico, se diverso, **deve** essere un sottotipo di quello statico e le operazioni che verranno fatte saranno tutte giá disponibili nell'interfaccia di tipo statico.

mi spiego peggio...

immaginiamo di avere una classe `Person` definita come segue:

```
public class Person{
    Person(...){
        ...
    }
    public void walk(int meters){
        ...
    }
}
```

e di avere due sottoclassi `Dancer` e `Singer` che estendono `Person` così:

```
public class Dancer extends Person{
    Dancer(...){
        ...
    }
    public void dance(...){
        ...
    }
}
```

```
public class Singer extends Person{
    Singer(...){
        ...
    }
    public void sing(...){
        ...
    }
}
```

creiamo anche una funzione `doSomething(Person p)` definita così:

```
void doSomething(Person p){
    p.walk(100);
}
```

Se nel main chiamando questa funzione potremo passare:

```
//qui il tipo statico == tipo dinamico
Person p = new Person(...);

doSomething(p);
```

ma anche:

```
//qui il tipo statico != tipo dinamico
Person d = new Dancer(...);
```

```
doSomething(d);
```

ma anche:

```
//qui il tipo statico != tipo dinamico
Person s = new Singer(...);

doSomething(s);
```

Questo perché per le regole dell'ereditarietá noi sappiamo che l'interfaccia potrá solo essere espansa dal tipo dinamico ma di fatti giá a compile time noi sappiamo che il tipo statico sará in grado di effettuare tutte le operazioni contenute in `doSomething()`.

Ovviamente se in `doSomething(Person p)` ci fosse un qualcosa che `Person` non sa fare o se dovessimo provare a passare come parametro qualcosa che non é un oggetto di tipo `Person` o un sottotipo dello stesso il programma non compilerebbe proprio.

7.1 Casting ad un sottotipo del tipo statico

Possiamo effettuare il casting di un'espressione ad un sottotipo del suo tipo statico usando il classico:

`(<type>) <expression>`

é possibile anche fare casting ad un supertipo ma ció é tecnicamente inutile e anzi ci da un Warning!

NON possiamo castare ad un tipo che non é un sottotipo

Durante l'esecuzione, se il tipo dinamico non dovesse essere compatibile viene lanciata un'eccezione (spoiler, ne parliamo al capitolo XXXX)

esempio:

```
Person p = new Singer(...);

p.walk(400);

//casto una persona al suo sottotipo
Singer s = (Singer) p;

s.sing(...);

//ILLEGALE!!!, Dancer non é un sottotipo di Singer!
Dancer d = (Dancer) s;

/* tecnicamente é legale e compila ma...
```

```
// CRASHA runtime, un Singer **non** puó
// diventare magicamente un Dancer
*/
Dancer d = (Dancer) p;
```

7.2 Instanceof

La keyword `instanceof` permette di verificare il tipo dinamico di un espressione, ha una risposta booleana e ritorna true solo se il tipo dinamico dell'espressione data é un tipo o un sottotipo del tipo dato come confronto.

é inutile controllare su un supertipo.

é vietato controllare su un tipo che non é un sottotipo.

esempio:

```
Person p = new Singer(...);

//true
if(p instanceof Singer){ ... }

Dancer d = new Dancer(...);

//inutile
if(d instanceof Person){ ... }

//vietato
if(d instanceof Singer){ ... }
```

8 Interfaces

Ogni classe in java puó estenderne solo un'altra quindi non c'é modo di implementare una stessa funzionalitá in due sottoclassi di natura diversa (se non riscrivendo codice)

esempio:

```
public class Tire{
    Tire(...){
        ...
    }

    public void inflate(double prAmount){
        //do stuff that adds pressure
    }
```

```
    public void deflate(double prAmount){
        //do stuff that removes pressure
    }
}

public class SafetyVest{

    SafetyVest(...){
        ...
    }

    public void inflate(double prAmount){
        //do stuff that adds pressure
    }

    public void deflate(double prAmount){
        //do stuff that removes pressure
    }
}
```

come vediamo, sia una ruota che un giubbotto di salvataggio implementano un sistema per gonfiarsi e sgonfiarsi, tuttavia da un lato non hanno abbastanza elementi in comune da poter definire una superclasse ma dall'altro abbiamo una ripetizione di codice che in un contesto reale potrebbe rivelarsi molto fastidiosa.

per questo possiamo usare un'interfaccia

esempio:

```
interface Inflatable{
    public void inflate(double prAmount);

    public void deflate(double prAmount);
}

public class Tire implements Inflatable{
    Tire(...){
        ...
    }

    public void inflate(double prAmount){
        //do stuff that adds pressure
    }
```

```
    public void deflate(double prAmount){
        //do stuff that removes pressure
    }
}

public class SafetyVest implements Inflatable{

    SafetyVest(...){
        ...
    }

    public void inflate(double prAmount){
        //do stuff that adds pressure
    }

    public void deflate(double prAmount){
        //do stuff that removes pressure
    }
}
```

nelle interfacce originariamente era possibile definire solo la firma delle funzioni e non tutta l'implementazione, cosí come non era possibile definire campi (come vediamo nell'esempio sopra).

8.1 implementazione delle `Interfaces`

Da java 8 questo limite é stato parzialmente superato tramite le implementazioni `default`, sicome peró una classe puó implementare diverse interfacce ci sono delle **limitazioni**.

anteponendo ad un metodo dell'interfaccia la keyword `default` potremo fornire una prima implementazione.

anche i campi possono essere definiti ma solo se sono:

- public
- static
- final

esempio:

```
interface Inflatable{
    default void inflate(double prAmount){
        //default implementation
    }

    default void deflate(double prAmount){
```

```
        //default implementation
    }
}
```

Come abbiamo detto (forse) si possono implementare anche piú interfacce in una sola classe usando una virgola fra queste dopo `implements`, cosí:

```
public class Example implements Interface1, Interface2{
    ...
}
```

8.1.1 default implementation multipla

Se abbiamo due interfacce con la stessa firma e una default implementation non potremo implementarle nella stessa classe, il compilatore java vieta questo comportamento.

8.2 Estendere le interfaccce

Il sistema funzione piú o meno come le classi ma abbiamo una grande differenza, **é possibile estendere piú interfacce in un unica sottointerfaccia**

8.3 Classi astratte o interfacce?

In breve:

- Pro delle classi astratte:
  - possono avere uno stato
  - possiamo implementare realmente i metodi
- Contro delle classi astratte:
  - estensione singola

- Pro delle interfacce:
  - estensione multipla
- Contro delle interfacce:
  - non possono avere uno stato
  - non possiamo implementare metodi (parzialmente risolto da `default`)

9 Dispatching di Java

A differenza di altri argomenti trattati fino ad ora, in questo caso abbiamo bisogno di introdurre un significato al titolo di questo capitolo. Innanzitutto, **cosa é il Dispaching**?

In informatica, il Dispatching é definito come il modo in cui il linguaggio collega le chiamate a funzione alle giuste definizioni delle stesse.

In sostanza quando noi chiamiamo una funzione `foo()` il dispatching permette al programma di entrare nella stessa ed eseguirne le istruzioni.

In Java, data la possibilitá di fare overload, override, etc... questo non risulta sempre semplice e deve seguire regole ben precise per evitare confusione, facciamo un esempio:

Una classe potrebbe avere piú metodi con lo stesso nome ma con parametri differenti, Il dispatcher di java andrá a chiamare il metodo con il corretto numero di parametri e con i tipi piú "compatibili" fra quelli scritti.

9.1 Invocazione di metodi e accesso ai campi

Il pattern che usa java per accedere a metodi e campi di una classe o un'istanza della stessa (un oggetto) é la *dot notation* ovvero:

`<class/obj>.<method/field>`

potremmo anche parlare di:

`<reciever>.<member signature>`

La firma di un metodo (come giá detto al capitolo 2 sez 1 (Abstraction e Interfacce)) é composta da:

- nome del metodo
- lista dei parametri con il loro tipo

mentre per i campi abbiamo solo il nome del campo stesso.

Per quanto riguarda il `<reciever>` é bene sottolineare due cose:

- Il dispatcher vede il tipo dinamico quindi verrá chiamata la versione del metodo piú "vicina" alla (eventuale) sottoclasse su cui stiamo lavorando.
- puó essere a volte implicito (ad esempio il this omesso se siamo nella classe stessa)

9.2 "algoritmo" di risoluzione dei metodi

1. Estrai il tipo dinamico del `<reciever>`
2. Cerca un metodo nella classe che abbia una firma compatibile con quella data
3. se lo trovi ritornalo
4. altrimenti ripeti questo passaggio in tutte le eventuali superclassi della classe corrente.

(il tutto é semplificato)

9.3 Problemi con l'overloading

Le firme dei metodi possono sovrapporsi e la situazione puó diventare difficile in situazioni in cui esistono due metodi con:

- stesso nome
- stesso numero di argomenti
- tipi di argomenti diversi ma uno il sottotipo dell'altro

esempio:

```
public class Foo{
    Foo{
        ...
    }

    public bar(Foo obj){
        ...
    }
}

public class FooExt extends Foo{
    FooExt{
        ...
    }

    public bar(FooExt obj){
        ...
    }
}
```

Come fa il dispatcher a scegliere quale eseguire?

...

9.4 Dispatching statico

- Il dispatcher entra in azione a tempo di compilazione quindi, solo i tipi statici sono conosciuti
- non permette polimorfismo (vedi cap 6 sez 2)
- Linguaggi ad oggetti si rifanno invece al dispatching dinamico

9.5 Dispatching dinamico

- Il dispatcher entra in funzione a runtime quindi anche i tipi dinamici sono conosciuti
- Reindirizza automaticamente al metodo con l'implementazione piú "specifica" che trova
- permette il polimorfismo
- non c'é bisogno di conoscere a priori la sottoclasse (ammesso che esista)
- crea un overhead durante l'esecuzione

9.6 Associazione dei parametri

(titolo originale *Matching method arguments*, questa é la migliore traduzione a cui sono arrivato)

Nella stessa classe potremmo avere due metodi con:

- stesso nome
- parametri diversi

Come scegliere quello da invocare?
Come giá detto, in primis conta il numero di parametri e poi il tipo.

In java i parametri sono analizzati ed associati staticamente, questa procedura é chiamata "single dispatch".

9.7 Chiamate a metodi statici

Se le chiamate sono fatte accedendo ai metodi tramite la classe (`Foo.staticMethod()`) allora non ci sono problemi, il dispatcher avrá meno lavoro, altrimenti andrá applicato lo static dispatching (a tempo di compilazione quindi).
Come abbiamo giá detto peró non é una buona pratica accedere a elementi statici tramite un'istanza di una classe quindi **non fatelo!!**

10 Tipi generici

Molte volte capita di dover implementare strutture dati complesse che peró debbano poter accogliere al loro interno (in diverse istanze) diversi tipi di dato.
In C++ la questione viene affrontata tramite i `template` e in java abbiamo qualcosa di simile dal punto di vista concettuale ma di radicalmente diverso nella sua fattezza pratica ovvero i `generic types`

java a differenza di C++ non crea a tempo di compilazione una nuova classe per ogni istanza di template usata ma a runtime fa i dovuti controlli e cast per far funzionare il tutto, tuttavia una classe generica esiste e viene creata e questo lo possiamo evincere anche dalla definizione ufficiale di `generic` in java:

A generic type is a generic class or interface that is parameterized over types.

10.1 Tipi generici come parametri

Un `generic` puó essere visto come un altro parametro passato al costruttore di una classe quando questa viene istanziata.
L'istanza della classe lo implementerá int tutte le firme dei metodi e dei campi.
Possiamo passare piú di un `generic` ad una classe ed é possibile passarli anche a superclassi.

10.2 Subtyping con i tipi generici

in java non é possibile assegnare un espressione con un tipo generico ad una variabile con un altro tipo generico.

10.3 Tipi generici per i metodi

Indipendentemente dalla classe é possibile implementare un tipo generico in un metodo della stessa anteponendolo al tipo di ritorno nella definizione del metodo stesso fra parentesi angolari (`<T>`) e si puó usare per:

- tipo di parametro
- tipo di ritorno
- tipo di variabile locale

esempio:

```
public class List<V> {
    public void add(V el){
        ...
    }

    public boolean contains(V el) {
        ...
    }
    public V get(int i) {
        ...
    }


    public static <T> List<T> toList(T value) {
        List<T> result = new List<T>();
        result.add(value);
        return result;
    }

    public static <T> T getFirst(List<T> list) {
        return list.get(0);
    }
}
```

10.4 Deduzione di tipo sui generics

Non é necessario in molti casi passare il tipo generico quando istanziamo una classe o chiamamo metodi, java ha un metodo di deduzione dei tipi

esempio:

```
/*
immaginiamo di avere una classe Beverage che viene estesa da Coke e Pepsi.
```

```
*/

/*qui non specifico nella parte destra dell'assegnamento di cosa deve essere
la nuova lista che sto istanziando ma siccome andrá in una lista di Beverage
la deduzione di tipo saprá cosa mettere
*/
List<Beverage> FridgeCokes = new List<>();

/*come vediamo dopo anche se siamo in una lista di Beverage posso mettere
una Coke siccome é una sottoclasse.
*/
FridgeCokes.add(new Coke(...));

/*Anche in questo caso, prendo una Coke e la metto in una variabile di tipo
Beverage (se hai dubbi vedi Capitolo 6 - Subtyping)
*/
Beverage tastyBeverage = List.getFirst(FridgeCokes);

List<Beverage> FridgePepsi = List.toList(new Pepsi(...));
```

Ovviamente come per i casting impliciti é meglio evitare (anche per sicurezza) certe operazioni.

10.5 Restringere i possibili generics

é possibile restringere il campo di possibili tipi in ingresso tramite una clausola implementata con la keywork `extends` vicino la dichiarazione del tipo generico

esempio:

```
<T extends Beverage> void drink(T tastyBeverage){
    ...
}
```

Solo i sottotipi di Beverage adesso saranno ammessi.

10.6 Wildcards

É possibile usare anche le wildcards ovvero senza definire un tipo generico ma accettare quello ricevuto.

Anche le wildcards possono essere ristrette (vedi qua sopra 10.5)

e le relazioni fra sottotipi rimangono, quindi:

`List<? extends Coke>` é un sottotipo di `List<? extends Beverage>`

11 Gerarchia dei tipi, Oggetti, String, tipi nativi.

`Object` é la superclasse di riferimento "universale", se una classe non estende esplicitamente un'altra allora implicitamente sta estendendo `Object`, questo permette di fornire operatori universali comuni a tutte le classi.

11.1 `equals`

é un metodo fornito da `Object` (su cui possiamo fare l'override).
Indica se un oggetto é uguale a quello su cui chiamo il metodo

`Obj1.equals(Obj2) => Obj1 == Obj2`

Proprietá comuni dell'uguaglianza:

- Riflessiva: `x.equals(x)`
- Simmetrica: `x.equals(y) <==> y.equals(x)`
- Transitiva: `x.equals(y) && y.equals(z) => x.equals(z)`

Di seguito un esempio di implementazione di equals:

esempio:

```
public class Gassosa{
    private double fullPercentage;

    Gassosa(double fullPercentage){
        this.fullPercentage = fullPercentage;
    }

    public boolean equals(Object o){
        if(o != null && o instanceof Gassosa) {
            Gassosa g = (Gassosa) o;
            return g.fullPercentage == this.fullPercentage;
        }
        return false;
    }
}
```

IMPORTANTE:

```
Gassosa g1 = new Gassosa(100);
Gassosa g2 = new Gassosa(100);
g1.equals(g2) //true
g1 == g2 //false
```

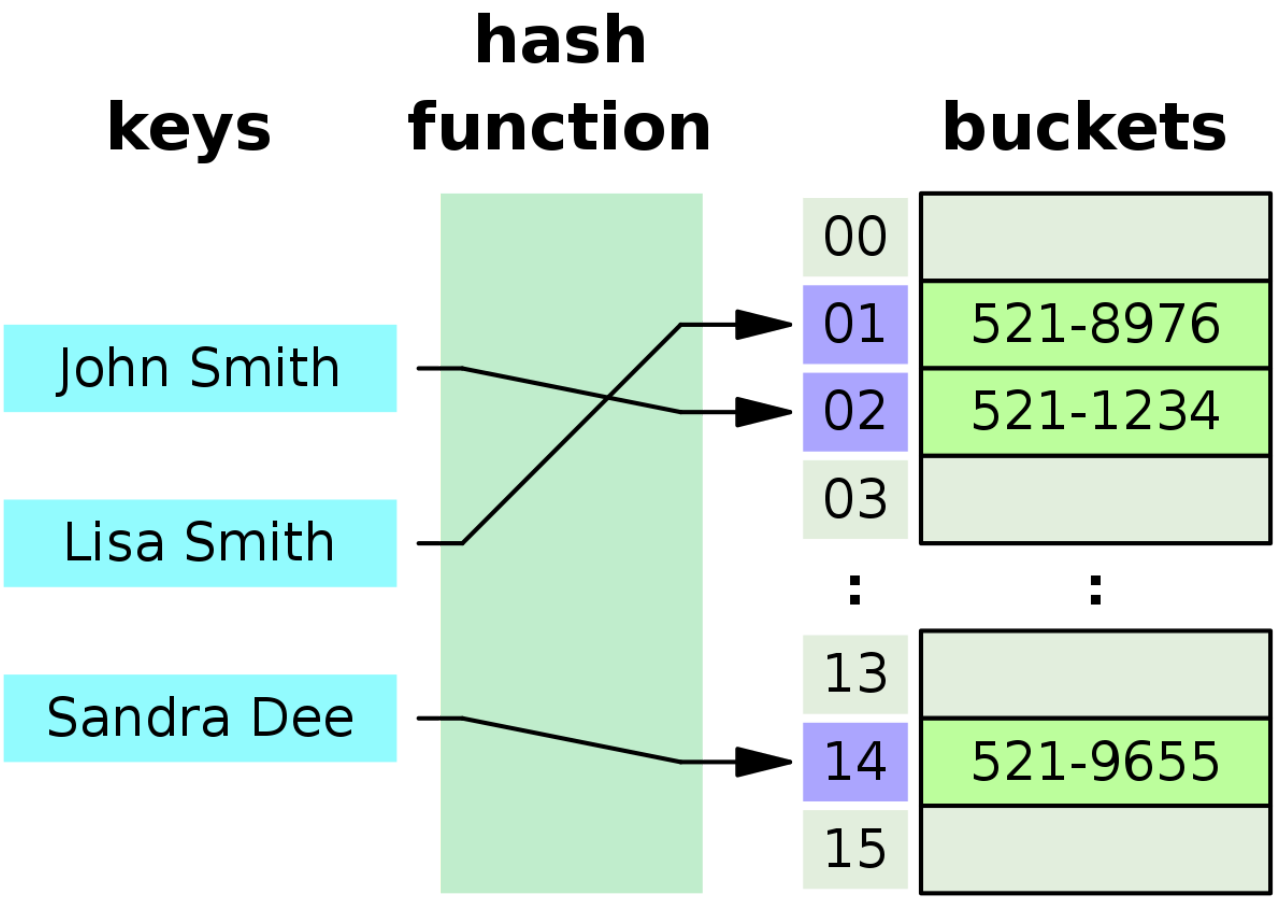
Anche se sono due oggetti uguali non sono lo stesso oggetto!!

11.2 HashCode

11.2.1 Hash table

Una hash table utilizza una funzione di hashing per generare un indice (detto hash code) riferito ad un array di buckets contenenti valori, il principio di funzionamento é il seguente:

preso un valore se ne calcolo l'hash sapró la sua posizione nell'array e quindi dove questo é salvato



Jorge Stolfi - CC BY-SA 3.0 from Wikipedia

Ricerca se Obj é in tabella:

1. Genero l'hash code (c) di obj
2. Prendo gli elementi all'indirizzo c
3. Vedo se obj é lí

11.2.2 hashCode in java

Il metodo hashcode ritorna un valore da usare come "marcatore".
La base di funzionamento non é avere un id univoco ma accomunare oggetti che potrebbero avere abbastanza caratteristiche in comune da essere considerati uguali.

esempio

```
public Class Bottle{
    private double liters;

    Bottle(double liters){
        this.liters = liters;
    }

    protected double getLiters(){
        return this.liters;
    }

}

public class Gassosa{
    private double fullPercentage;
    private Bottle bottle;

    Gassosa(double fullPercentage,bottleLiters){
        this.fullPercentage = fullPercentage;
        this.bottle = new Bottle(bottleLiters);
    }

    public int hashCode(){
        return bottle.getLiters();
    }
}
```

in questo caso, controllando il risultato di `hashCode()` avró tutte le gassose contenute in bottiglie dalla stessa grandezza.
Questa peró é una faccia della medaglia, infatti, ogni volta che si effettua l'override di `hashCode()` é necessario farlo anche per `equals()` (e viceversa).

11.3 Clonare oggetti

La copia di un oggetto (e quindi di un area di memoria) in qualsiasi linguaggio é sempre stato un argomento da toccare con le pinze e in java non abbiamo differenze.

Ritorna un oggetto differente anche se uguale nel contenuto (eseguendo `equals` dopo dovrebbe ritornare true)

se ci sono oggetti fra i campi si divide in shallow e deep copy:

- Shallow copy: I due oggetti condivideranno l'oggetto contenuto nel campo

- Deep copy: Verrá fatta una copia anche dell'oggetto contenuto nel campo

il metodo `clone()` di default ha visibilitá `protected`

esempio:

```
public Class Bottle{
    private double liters;

    Bottle(double liters){
        this.liters = liters;
    }

    protected double getLiters(){
        return this.liters;
    }

}

public class Gassosa{
    private double fullPercentage;
    private Bottle bottle;

    Gassosa(double fullPercentage,bottleLiters){
        this.fullPercentage = fullPercentage;
        this.bottle = new Bottle(bottleLiters);
    }

    protected Object clone(){
        return new Gassosa(this.fullPercentage,bottle.getLiters());
    }
}
```

In questo caso anche la bottiglia viene "clonata", questo é un caso di deep copy.

11.4 Set

I set in java sono raccolte che non contengono duplicati.
La base di comparazione é la funzione `equals()`.

11.4.1 HashSet

Esistono vari tipi di Set fra cui HashSet che é l'implementazione dell'interfaccia Set implementata tramite una hash table.
Non garantisce l'ordine degli elementi (come farebbe un array).

esempio:

```
public class BeverageCatalog{
    Set<T extends Beverage> catalog;
    BeverageCatalog(){
        this.catalog = new HashSet<T extends Beverage>();
    }

    BeverageCatalog(Set<T extends Beverage> items){
        catalog = items;
    }

    boolean addToCatalog(Beverage item){
        //il metodo add di HashSet ritorna true se l'elemento viene aggiunto
        return catalog.add();
    }

}
```

11.4.2 TreeSet e comparatori

Il TreeSet segue un ordine definito da un ordinamento "naturale" o da un Comparator passato tramite costruttore quando viene creato il set.

La classe da inserire nel set deve implementare l'interfaccia `Comparable` che richiede un metodo `compareTo`

Ha un ordinamento totale
x.compareTo(y) == 0 <==> x.equals(y)

Dato l'ordinamento anche l'iterazione é ordinata

esempio:

```
public class BeverageCatalog{
    Set<T extends Beverage> catalog;
    BeverageCatalog(){
        this.catalog = new TreeSet<T extends Beverage>();
    }

    BeverageCatalog(Set<T extends Beverage> items){
        catalog = items;
    }

    boolean addToCatalog(Beverage item){
```

```
        //il metodo add di HashSet ritorna true se l'elemento viene aggiunto
        return catalog.add();
    }

}

public class Beverage implements Comparable<Beverage>{
    ...
    public int compareTo(Beverage b) {
        if(amount==b.amount)
            return 0;
        else return amount-o.amount;
    }
}
```

11.4.3 Problemi con i set

Abbiamo detto che non si possono aggiungere due oggetti identici ad un set... MA:
Se aggiungiamo due oggetti diversi e poi modifichiamo uno dei due potremmo trovarci in una situazione in cui il set perde l'univocitá alla sua base.

Soluzioni:

- oggetti immutabili
- adottare altre librerie
  - Apache commons collection
  - TransformedSet
  - CloneTransformer

11.5 Tipi primitivi

I tipi primitivi non hanno come superclasse la classe `Object`.

Possiamo fare le classiche operazioni su di loro:

- + - * / % su numeri interi e numeri in virgola mobile
- && || ! su valori booleani

Tipo	Bit	Floating point	Valori assumibili
Boolean	1	No	true/false
byte	8	No	
char	16	No	‘a’, ‘b’, ‘\n’, ‘\t’
short	16	No	

Tipo	Bit	Floating point	Valori assumibili
int	32	No	12, 564, -436
long	64	No	12L, 564L, -436L
float	32	Si	1.23F, -54.3F, 1F
double	64	Si	1.23D, -54.3D, 1D

11.5.1 Conversioni di tipo implicite

Ogni valore numerico puó essere assegnato a qualsiasi variabile numerica che supporta un range piú largo (ha piú bit per rappresentare quell'informazione)

char puó sostituire un int (16bit vs 8bit)
float puó diventare un double (32bit vs 64bit)
valori interi (di qualsiasi tipo) possono diventare floating point.

MA NON VICEVERSA

infatti tentando l'operazione inversa di queste tre descritte sopra si rischia una perdita di informazioni, quindi java implicitamente non permette queste conversioni.

11.6 Tipo Stringa

java fornisce una sua classe stringa, non abbiamo puntatori a char.
Le stringhe sono formate da caratteri e ogni stringa da noi dichiarata é implementata come un'istanza della classe che contiene un array di bytes. (come un array di caratteri)

11.6.1 Metodi sulla classe string

Come abbiamo giá detto ogni istanza della classe String rappresenta una stringa.

definiamo `String s1 = "str"` e `String s2 = "ing"`

Su ogni stringa é possibile performare varie operazioni fra cui:

- `s3=s1.concat(s2)` == `s1 + s2` => `string` (ritorna string)
- `s1.replace('s','z')` => `ztr` (ritorna string)
- `s3.substring(1,3)` => `tri` (ritorna string) (parto da 1 e avanzo per 3 caratteri)
- `s3.split("r")` => ["st","ing"] (ritorna array di string)

11.6.2 stringhe sono costanti

Come abbiamo visto tutte le operazioni su oggetti della classe String ritornano qualcosa ma non modificano mai la stringa su cui sono applicati, questo perché **non possono farlo**.

In java per evitare problemi le stringhe sono oggetti immutabili, una volta create non possono essere modificate.

Anche facendo `s1=s1.concat(s2)` quello che fará java all'atto pratico sará creare una nuova stringa `s1+s2` e cambiare successivamente la reference a cui punta `s1` quindi di fatto cambiando oggetto non modificherá mai praticamente `s1` originale.

11.7 Boxing e Unboxing

Per ogni tipo primitivo esiste una sua versione "oggetto".
ovviamente é meno efficiente ma piú strutturata e quindi con vari metodi utili come `min()` e `max()` o altri per facilitarne la conversione da un tipo ad un altro.

La conversione implicita fra un tipo primitivo (unboxed) ed un tipo wrapped (boxed) non esiste (un tipo int puó diventare solo un tipo Integer)... quindi `Double val = 5` non é ammesso.

12 Iterare sui set

Si usa un for speciale (molto simile ad un foreach) con questa forma:
`for(<tipo> <v1>: <v2>)`

Dichiariamo una variabile di `<tipo>` tipo chiamata `v1` (che possa contenere ovviamente elementi di `<v2>`) e `<v2>` che deve essere ovviamente un tipo iterabile.

esempio

```
public class BeverageCatalog{
    Set<T extends Beverage> catalog;
    BeverageCatalog(){
        this.catalog = new HashSet<T extends Beverage>();
    }

    BeverageCatalog(Set<T extends Beverage> items){
        catalog = items;
    }

    public boolean addToCatalog(Beverage item){
        //il metodo add di HashSet ritorna true se l'elemento viene aggiunto
        return catalog.add();
    }

    public void printSet(){
        for(Beverage el: this.catalog){
            System.out.println(el);
        }
    }
}
```

13 Stampare oggetti

Prima abbiamo usato `printSet(){...}` per stampare oggetti e stranamente nonostante abbiamo passato un oggetto a `System.out.println()` vi posso assicurare che funziona.

Java fornisce di suo un metodo per stampare oggetti (ovviamente modificabile tramite l'Override) che restituisce una stringa e viene chiamato automaticamente da `System.out.println()`, questo si chiama `toString()`.

Di default (quindi anche nel nostro esempio precedente) restituisce:

- La classe a cui appartiene l'oggetto (tramite la keyword `istanceof`)
- L'hashCode dell'oggetto

14 Eccezioni

Durante l'esecuzione ci possono essere diversi errori
Per esempio:

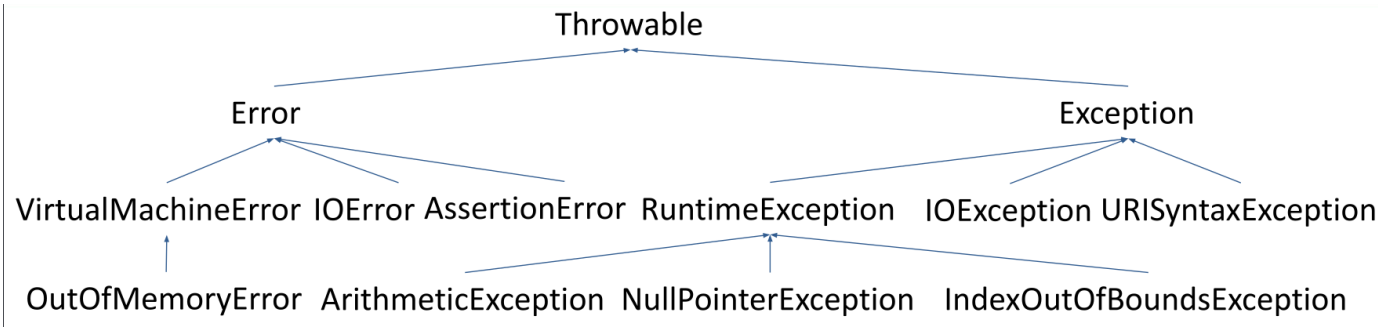
- Tentare operazioni su reference invalide o `null`
- Dividere un numero per zero
- Provare ad accedere ad un campo di un array fuori dimensioni massime
- Allocare memoria quando non disponibile
- etc ...

Ovviamente diversi errori significa anche un diverso livello di "gravitá" della situazione e non é detto che il programma debba crashare.
java quando incontra un errore lancia una `exception` che puó essere intercettata e trattata in vari modi.

14.1 Focus: quando l'esecuzione incontra un errore

- Un'eccezione viene lanciata
- L'esecuzione normale del programma é sospesa
- L'eccezione ripercorre e logga tutto lo stack delle chiamate
- Un messaggio di errore viene fornito con tutto il log appena creato

Questo aiuta in fase di debug.



14.2 Throwable

tramite la keyword `throw` é possibile creare un oggetto di tipo `Throwable` che tramite un gerarchia ben definita di oggetti puó lanciare diversi tipi di errore e gestire anche un'eventuale fase di stampa del problema.

14.3 Definire e lanciare le proprie eccezioni

é possibile creare le proprie eccezioni estendendo `Throwable` (o una sua sottoclasse)

Le nostre eccezioni possono avere una loro "interfaccia esterna" ma una good practice é quella di stampare sempre un messaggio di errore.

Perché creare le proprie exception?
Perché ogni eccezione rappresenta un errore specifico e java di suo ha solo eccezioni ad errori comuni.

É possibile esplicitamente lanciare eccezioni tramite `throw <expr>`, ovviamente `<expr>` dovrá essere un istanza di `Throwable` o di una sua sottoclasse.

14.3.1 Dichiarare eccezioni lanciabili da un metodo

Ogni metodo deve dichiarare le eccezioni che potrebbe lanciare tramite la keyword `throws` fra la fine della firma dello stesso e l'inizio dell'implementazione, quindi:

`<nomemetodo> throws <ecc1>,<ecc2>,...,<eccN> { ... }`

e tutti i chiamanti del metodo dovrebbero a loro volta farlo (per best practice) e per lo stesso motivo andrebbe inserito anche nella documentazione e nei commenti.

esempio:

```
public class LiquidOverflowException extends Exception {
    public LiquidOverflowException(double amount,double capacity) {
        super(“Too much liquid in the bottle! \n amount: “+ amount + "\n
capacity: " + capacity);
    }
}
```

```
class Bottle{
    String content;
    double amount;
    double capacity;

    Bottle(String content,double amount,double capacity) throws
LiquidOverflowException{
            this.content = content;
            this.amount = amount;
      if(capacity < amount){
                throw new LiquidOverflowException(amount,capacity)
      }
            this.capacity = capacity;
    }
}
```

14.4 Commentare le eccezioni

Commentando un metodo con un eccezione si usa `@throws <exception>` (é un tag ufficialmente supportato da javaDoc).
Esiste anche `@exception` che é perfettamente uguale.

14.5 Estendere metodi che dichiarano eccezioni

Quando una classe con un metodo che dichiara eccezioni viene estesa se quel metodo verrá reimplementato (override) allora per il principio che l'interfaccia debba rimanere uguale o piú ampia anche l'eccezione dovrá essere scritta altrimenti la sottoclasse non compilerá.

```
class SodaBottle extends Bottle{
    Bottle(double amount,double capacity) throws LiquidOverflowException{
            super("Soda",amount,capacity);
    }
}
```

14.6 Eccezioni checked and unchecked

Non tutte le eccezioni possibilmente lanciate devono essere dichiarate!

Altrimenti per esempio ogni costruttore dovrebbe dichiarare una possibile `OutOfMemoryException`, no?

Di fatti in java le eccezioni sono divise in due categorie principali:

- Checked: vanno dichiarate, estendono `Exception` ma no `RuntimeException`

- Unchecked:
  - non dichiarate, estendono `Error` o `RuntimeException`.

La differenza sta nel fatto che eccezioni unchecked vengono sollevate quando c'é una falla logica nel programma che in nessun modo puó essere recuperata.

14.6.1 Focus: Gli errori

Come abbiamo detto c'é differenza fra errori ed eccezioni.
Gli errori rappresentano situazioni in cui l'ambiente su cui gira il programma potrebbe essere compromesso e quindi l'esecuzione **non** puó essere ripresa.

Se vengono intercettati per good practice andrebbero rilanciati e non andrebbero mai lanciati dal programma ma solo dalla JVM e dalle librerie standard.

14.7 Intercettare le eccezioni

Tutti i tipi di eccezioni possono essere intecettati tramite i blocchi `try{...}` `catch(<exc-type> e) {}` .

Il funzionamento si basa su un approccio condizionale, ovvero, prova ad eseguire il codice fra le graffe di `try` e se viene lanciata un'eccezione di tipo `<exc-type>` intercettala e fai quello che c'é nelle graffe di `catch`

Alcune Good practice:

- Non provare ad intercettare eccezioni generiche, non sapresti comunque come gestirle dopo.
- I blocchi `Catch` non dovrebbero essere vuoti.

esempio:

```
createRandomSodaBottle(){
    try{
        //questo potrebbe triggerare un eccezione
        return new SodaBottle(Math.random()*10.0,Math.random()*10.0);
    }
    catch(LiquidOverflowException e){
        throw new IllegalArgumentException(“Provato a mettere troppo
liquido“, e);
    }
}
```

14.8 finally

La keyword `finally{}` rappresenta un blocco di codice (opzionale) eseguito indipendentemente se l'eccezione viene lanciata o no.
Puó essere usato per "resettare" lo stato del programma ad una condizione tale da poter essere riutilizzato dopo un'esecuzione (che sia andata bene o male).

Questo blocco viene eseguito anche se i blocchi `try`/`catch` presentano dei `return`.
Se `finally` presenta un `return` o lancia un'eccezione gli altri valori ritornati/le altre eccezioni vengono entrambi ignorati.

esempio:

```
createRandomSodaBottle(){
    try{
        //questo potrebbe triggerare un eccezione
        return new SodaBottle(Math.random()*10.0,Math.random()*10.0);
    }
    catch(LiquidOverflowException e){
        throw new IllegalArgumentException(“Provato a mettere troppo liquido“, e);
    }
    finally{
        //bello ma inutile, ritornerá sempre una bottiglia da 10.0 come capacity e amount.
        return new SodaBottle(10.0,10.0);
    }
}
```

14.9 "Algoritmo" di funzionamento di Try Catch e Finally

1. Esegui il codice di try
2. Se lancia un'eccezione di tipo o sottotipo intercettato:
    1. Esegui catch
3. Esegui il finally in ogni caso
4. Continua l'esecuzione normale del programma o rilancia l'eccezione basandoti su ció che é successo nel catch
5. Se lancia un'eccezione non di tipo o sottotipo intercettato
    1. Esegui il finally
    2. rilancia l'eccezione
6. Se non lancia eccezioni esegui il finally e continua

reference:

```
try {
<body>
}
catch(<exc-type> e) {
<catch-body>
}
finally {
<finally-body>
}
```

14.10 Catene di eccezioni

Un eccezione puó essere causata da un altra eccezione (come nell'esempio precedente)

In questo casso si parla di catena di eccezioni passando ad ognuna l'eccezione che l'ha causata tramite costruttore o chiamando il metodo `initCause()` che ovviamente potrá essere settato solo una volta.

esempio:

```
catch(LiquidOverflowException e){
    throw new IllegalArgumentException(“Provato a mettere troppo liquido“, e); //come secondo parametro abbiamo e ovvero l'eccezione precedente
    }
```

14.11 Assertions (Aspettative ??)

non so l'italiano :(

Le `assertions` sono speciali condizioni di controllo, possiamo tradurle come delle aspettative che dovrebbero essere sempre soddisfatte.
Sono usate per testare il codice, infatti anche se vengono "deluse" il codice continua ad andare e di default non sono nemmeno eseguite.

Per abilitarle si usa `java -ea` e possono mandare messaggi con:
`assert <condizione> [: <messaggio>];`

```
class Bottle{
    String content;
    double amount;
    double capacity;

    Bottle(String content,double amount,double capacity) throws LiquidOverflowException{
```

```
            this.content = content;
            this.amount = amount;
            assert capacity >= amount [:"piú liquido di quanto puó contenere
la bottiglia"]
      if(capacity < amount){
                throw new LiquidOverflowException(amount,capacity)
      }
            this.capacity = capacity;
    }
}
```

15 Annotazioni

Da Java 6+ esistono le annotazioni, sono "commenti" speciali che esprimono informazioni sintattiche sul programma e possono essere tenute (a differenza dei commenti normali) anche nel programma compilato.

Le annotazioni possono essere aggiunte a:

- classi
- campi
- metodi (e costruttori)
- parametri e variabili locali

Il package `java.lang.annotation` contiene tutto il necessario:

- `ElementType`: (quello di cui abbiamo parlato sopra)
- `RetentionPolicy`: se tenere l'annotazione nel compilato o no

Le annotazioni possono avere parametri.

15.1 Definire un tipo di annotazione "custom"

Di seguito il codice per definire un tipo di annotazione "custom":

esempio:

```
@interface Liquid{
    String type() default “Liters“;
    boolean sparkling();
}
```

come vediamo sopra: definiscono attributi (come se fossero metodi con un return) e possono avere un valore di default (che altrimenti va impostato ad ogni utilizzo)

Anche loro come le classi e le interfacce devono stare ognuna in un file .java separato

15.2 Usare le annotazioni

Per usare le annotazioni basta inserirle prima della dichiarazione di:

- classi
- campi
- firme dei metodi
- parametri o variabili

si usa per richiamarle:
`@<nome-annotation>(<nome-parametro>=<valore>,...)`
i parametri sono quelli definiti dall'annotation.

15.3 Restringere l'applicabilitá di un'annotazione

É possibile rendere le annotazioni applicabili solo a componenti specifici, per farlo si puó annotare l'annotazione (non sto scherzando) si usa per questo `java.lang.Target` che é un Array di `ElementType` e sará il compilatore a controllare queste restrizioni.

esempio:

```
@Target({
ElementType.FIELD,
ElementType.METHOD,
ElementType.PARAMETER
})
@interface Liquid{
    String type() default “Liters“;
    boolean sparkling();
}
```

15.4 Retention

La "retention" é una proprietá delle annotation che definisce fin quando queste vanno "incluse" nel codice, infatti abbiamo tre possibili livelli:

- SOURCE
    - Solo nel codice sorgente per documentazione
- CLASS
    - tenute nel file .class ma non visibili a runtime
- RUNTIME

- visibili a runtime, potrebbe modificare quindi anche il comportamento del programma.

esempio:

```
@Target({
ElementType.FIELD,
ElementType.METHOD,
ElementType.PARAMETER
})
@Retention(
    RetentionPolicy.SOURCE
)
@interface Liquid{
    String type() default “Liters“;
    boolean sparkling();
}
```

15.5 Annotazioni comuni

- `@Override`
  - override di un metodo dichiarato in una superclasse
  - aiuta l'editor a capire se l'override é fatto bene
- `@SuppressWarnings`
  - permette di non mostrare warnings del compilatore
  - prende come attributo un set di stringhe che definiscono i tipi di warning da nascondere
  - `@Deprecated`
    - permette di far capire che un metodo non andrebbe piú usato
    - prende come attributi `forRemoval` (in che versione verrá totalmente rimosso) e `since` (da che versione é considerato deprecato)

esempio:

```
class Bottle{
    protected String content;
    private double amount;
    private double capacity;
        private bool isEmpty;

    Bottle(String content,double amount,double capacity) throws
LiquidOverflowException{
            this.content = content;
            this.amount = amount;
```

```
            isEmpty = (amount == 0);

            assert capacity >= amount [:"piú liquido di quanto puó contenere
la bottiglia"]
      if(capacity < amount){
                throw new LiquidOverflowException(amount,capacity)
      }
            this.capacity = capacity;
    }

        @Deprecated(since = “2.0“)
        public void empty(){
            amount = 0;
            isEmpty = true;
        }

        @SuppressWarnings("unused")
        public void fill(){
            isEmpty = false;
            amount = capacity;
        }

        @Override
        public String toString(){
            return "Message in a bottle (the police)";
        }
}
```

16 Junit test

Junit é un framework per unit testing su programmi java

Funziona tramite varie annotazioni:

- `@Test`
  - specifica che la seguente funzione é un test
- `@BeforeEach` e `@AfterEach`
  - specifica cosa eseguire prima e dopo ogni test
- `@BeforeAll` e `@AfterAll`
  - specifica cosa eseguire prima e dopo tutti i test

Junit inoltre fornisce diverse classi fra cui `Assert` che reimplementa gli `assert` che abbiamo giá visto rendendoli piú specifici per i test.

17 JAXB

Java Architecture for XML Binding

JAXB fino a java 8 faceva parte del core adesso é una libreria esterna.
Permette di creare e leggere classi come file XML usando le annotazioni.

esempio:

```
@XmlRootElement
@XmlType
public class Person{

    @XmlElement
    private final String name;

    @XmlAttribute
    private double weight;
}
```

- `@XmlType` Definisce un tipo (di una classe) nello schema XML
- `@XmlRootElement` La classe che é root del documento XML
- `@XmlAttribute` Associa una propietá/campo ad un attributo XML
- `@XmlElement` Associa una propietá/campo ad un elemento XML

18 Reflection

La reflection permette di avere informazioni sul programma dal programma stesso!
In questo modo é possibile adattarsi a ció che é disponibile
Il package principale é `java.lang.reflect`

18.1 La classe Class

Accessibile tramite `NomeClass.class` espone tramite dei getters vari componenti, come:

- `getFields`
- `getMethods`
- `getConstructor`

che ritornano tutti array di `components` oppure tornano nomi/firme di metodi dichiarati o ereditati.
Altri getters ritornano la struttura dell'oggetto, come:

- `isInterface`
- `isArray`

- `isAnnotation`
- `getSuperclass`
- `getPackage`

esempio:

```
//questo é un dump della classe SodaBottle
Class<Vehicle> c = SodaBottle.class;

for(Constructor t : c.getDeclaredConstructors())
System.out.println(t);

for(Method m : c.getDeclaredMethods())
System.out.println(m);

for(Field f : c.getDeclaredFields())
System.out.println(f);

System.out.println(c.getSuperclass());

System.out.println(c.getPackage());
```

18.2 La classe Field

Ttramite la classe `Field` possiamo vedere tutti i campi siano essi private, public, protected o default e non solo...

Possiamo leggerne il valore con:

- `get`, `getDouble`/`Int`/`...`

Possiamo scriverne il valore con:

- `set`, `setDouble`/`Int`/`...`

Ottenere e modificare varie informazioni tramite:

- `getModifiers`
- `getType`
- (questo non ha senso ma permette di cambiare l'accessibilitá): `setAccessible`

```
class Bottle{
    protected String content;
    private double amount;
```

```
    private double capacity;
        private bool isEmpty;

    Bottle(String content,double amount,double capacity) throws
LiquidOverflowException{
            this.content = content;
            this.amount = amount;
            isEmpty = (amount == 0);

            assert capacity >= amount [:"piú liquido di quanto puó contenere
la bottiglia"]
      if(capacity < amount){
                throw new LiquidOverflowException(amount,capacity)
      }
            this.capacity = capacity;
    }

        public void empty(){
            amount = 0;
            isEmpty = true;
        }

        public void fill(){
            isEmpty = false;
            amount = capacity;
        }
}

SodaBottle s = new SodaBottle(0,10);
Class classSodaBottle = s.getClass();
Class classBottle = classSodaBottle.getSuperclass();

for(Field f : classSodaBottle.getDeclaredFields())
System.out.println(f);

for(Field f : classBottle.getDeclaredFields())
System.out.println(f);

//con queste due istruzioni posso rompere l'istanza della classe
Field amount = classBottle.getDeclaredField( "amount");
amount.setDouble(s, 0.0);
/*
infatti se prima l'unico modo per modificare l'amount era tramite metodi che
```

```
in caso di svuotamento completo avrebbero posto il valore booleano isEmpty a
true adesso potremmo avere isEmpty a false ma amount = 0.0!
*/
```

18.3 La classe Method

Ttramite la classe `Method` possiamo vedere tutti i campi siano essi private, public, protected o default e non solo...

Ci sono metodi che permettono di leggere la dichiarazione:

- `getReturnType`
- `getTypeParameters`
- `getGenericExceptionTypes`

é possibile anche invocarlo tramite `invoke`

Altre informazioni sono ottenibili tramite:

- `getModifiers`
- `isDefault`
- `setAccessible`

esempio:

```
SodaBottle s = new SodaBottle(0,10);
Class classSodaBottle = s.getClass();
Class classBottle = classSodaBottle.getSuperClass();

Method fill =
classBottle.getDeclaredMethod("fill");

Object res = fill.invoke(s);
```

18.4 La classe Constuctor

I costruttori sono metodi speciali, non ritornano un valore, non hanno un nome.
La classe `Constructor` funziona come la classe `Method` il getter peró non riceve il nome (non esiste) e non é possibile invocare il costruttore.

18.5 Ispezionare le annotazioni

Interface AnnotatedElement permette di ottenere praticamente tutto con:

- `getAnnotations`
- `getDeclaredAnnotations`

- `isAnnotationPresent`

Ovviamente é possibile ottenere le annotazioni applicate a qualsiasi cosa, dai campi ai package.

É possibile anche ottenere i valori delle annotations e dei parametri

18.6 Pro e contro della reflection

Pro:

- Possiamo accedere a componenti che non conosciamo a tempo di compilazione
- Molto utile per framework e testing (e.g., JUnit)
- Possiamo accedere a componenti non accessibili anche se contro i principi base della programmazione ad oggetti!!!

Contro:

- Molto verboso (alla fine stiamo reversando il programma)
- Nessuna garanzia di integritá in qualche modo otteniamo tutto (molto pericoloso)