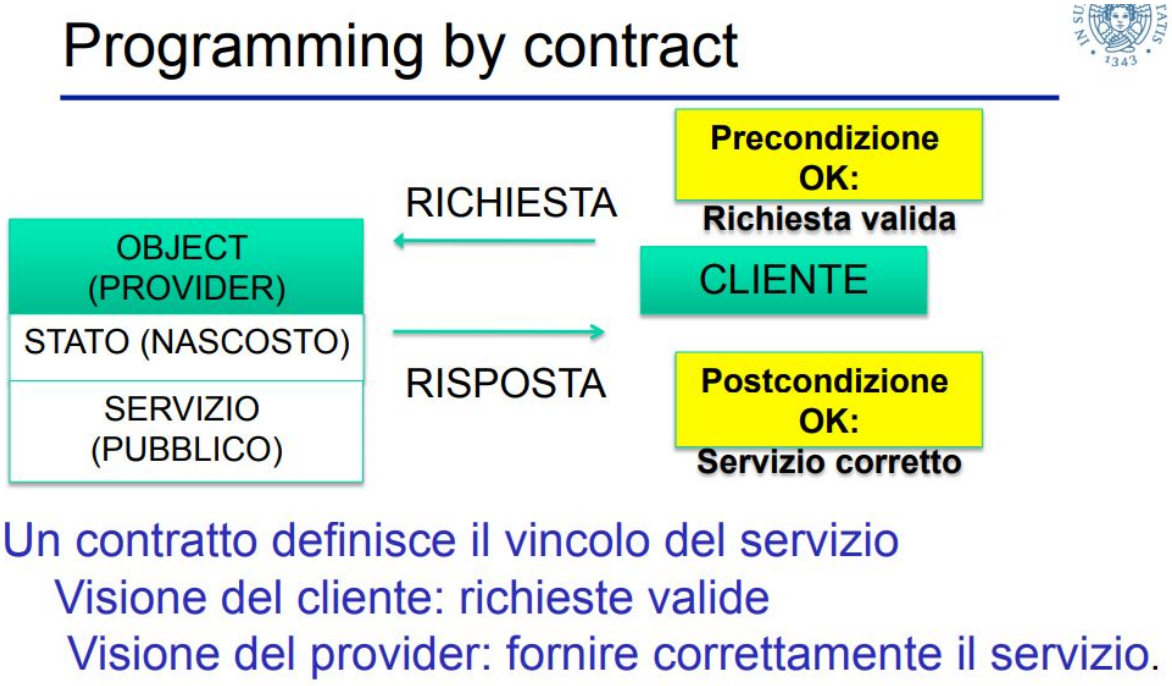
Program Development in Java (Addison Wesley 2000)
datato ma per aspetti concettuali
Data Structures and Abstraction (Pearson 2017)
Bruni, Corradini, Gervasi Programmazione in Java (Apogeo 2017)
Gabbrielli, Martini Linguaggi di Programmazione (McGraw-Hill 2006)

Paradigmi di programmazione: imperativo (C), funzionale (caml), a oggetti (java)

___

Information Hiding (*nascondere implementazione ma non interfaccia. Ciò permette di presentare all'utilizzatore come utilizzare per realizzare ciò che vuole senza che abbia bisogno di sapere come viene realizzato ciò. Cioè l'utilizzatore conosce parametri e valore di ritorno ma non il codice in sè. Più tecnicamente: nascondere gli aspetti dell'implementazione che sono più soggetti a variazione e presentare quelli che quasi sicuramente non cambieranno*)

Dichiarazione di <u>cosa</u> si fa e non <u>come</u> lo si fa
Protezione dei clienti dalle modifiche
Facilita manutenzione e modifica



Astrazione
Usare linguaggi ad alto livello è già un'astrazione di per sé, perché astraggono i linguaggi macchina/assembler/... rendendo più semplice la scrittura di programmi. Uso costrutto linguaggio alto livello invece di una lunga sequenza di istruzioni macchina equivalenti.

- **Astrazione Procedurale**: definizione e chiamata di procedure (funzione)
  Separare definizione e chiamata rende disponibile nel linguaggio i due meccanismi di astrazione fondamentali:
  - **Astrazione attraverso Parametrizzazione**
    Astrae dall'identità di alcuni dati "coprendoli" con i parametri (si generalizza un parametro

per poterlo usare in contesti diversi)
Cioè io definisco una procedura ma non specifico una particolare computazione (un particolare gruppo di variabili) ma variabili generiche quindi un potenziale insieme infinito di variabili

- Il programma seguente descrive una particolare computazione

  x * x + y * y

- La definizione

  fun(x, y: int) = (x * x + y * y)

  descrive tutte le computazioni che si possono ottenere chiamando la funzione, cioé applicando la funzione a una opportuna coppia di valori

- **Astrazione attraverso Specifica**
  Astrae dai dettagli implementativi della procedura per limitarsi a considerare il comportamento (ciò che fa, non come lo fa).
  Si presta ad astrazioni più potenti della parametrizzazione:
  - Possiamo astrarre dalla specifica implementazione e cioè associare ad ogni procedura la semantica più che la sintassi
  - e derivare la semantica della procedura dalla specifica piuttosto che dal corpo della procedura
  - non è di solito supportata dai linguaggi di programmazione, se non in parte tramite le specifiche di tipo (come nei linguaggi funzionali della famiglia ML)

  Es:

```
float sqrt (float coef) {
        //REQUIRES: coef > 0
        //EFFECTS: ritorna una approssimazione della radq di coef
        float ans = coef/2.0;
        for (int i = 1; i < 7; i++) {
                ans = ans-((ans*ans-coef)/(2.0*ans));
        }
        return ans;
}
```

  Precondizione (REQUIRES): deve essere verificata quando si chiama la procedura
  Postcondizione (EFFECTS): tutto ciò che possiamo assumere che valga quando la procedura termina se al momento della chiamata era verificata la precondizione

  L'utente vede solamente

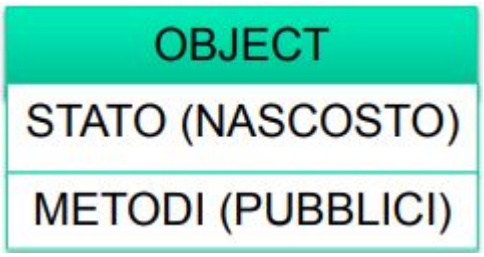
```
float sqrt (float coef) {
        //REQUIRES: coef > 0
        //EFFECTS: ritorna una approssimazione della radq di coef
        ...
```

}
Esso non deve preoccuparsi di capire come lo fa, astraendo dalle computazioni descritte nel corpo (che possono essere molto complesse)
Non può osservare il corpo e da questo dedurre proprietà diverse da quelle descritte nelle asserzioni

Parametrizzazione e specifica consentono di definire vari tipi di astrazione:

- **Astrazione procedurale**: si aggiungono nuove operazioni
  Viene fornita da tutti i linguaggi ad alto livello, aggiunge operazioni nuovo a quelle fornite come primitive
- **Astrazione di dati**: si aggiungono nuovi tipi di dato
  Fornita da tutti i linguaggi ad alto livello moderni, aggiunge nuovi tipi di dato e le relative operazioni:
  - L'utente non deve interessarsi all'implementazione ma solo fare riferimento alle proprietà presenti nella specifica
  - le operazioni sono astrazioni definite da asserzioni logiche
  - la specifica descrive relazioni tra operazioni
- **Iterazione astratta**: permette di iterare su elementi di un insieme, senza sapere come sono ottenuti (es: iteratore di un vettore in Java). Nasconde il flusso di controllo dei cicli
- **Gerarchie di tipo**: permette di astrarre da specifici tipi di dato a famiglie di tipi correlati
  Fornita da linguaggi d'alto livello moderni. I tipi di una famiglia condividono alcune operazioni (definite nel **supertype**, del quale tutti i tipi della famiglia sono **subtype** ed ereditano).
  Una famiglia di tipi astrae i dettagli che rendono diversi tra loro i vari tipi della famiglia.

Il tipo più importante è l'astrazione sui dati: gli iteratori astratti e le gerarchie di tipo sono basati sui tipi di dati astratti.
L'astrazione è il meccanismo fondamentale alla base della OOP.



Oggetto: insieme strutturato di **variabili d'istanza** (**stato**) e di **metodi** (**operazioni**)
Classe: **modello** (**template**) per la creazione di oggetti
Ogni oggetto è caratterizzato da:

- Stato
- Identità (nome)
- Ciclo di vita (Creati, riferiti, disattivati)
- Locazione di memoria
- Comportamenti

La definizione di una classe specifica:

- **Tipo** e **valore iniziale** dello stato locale degli oggetti (le variabili d'istanza)
- **Insieme delle operazioni** che possono essere eseguite (metodi)
- **Costruttori** (uno o più), cioè il codice da eseguire al momento della creazione di un oggetto

Ogni oggetto è istanza di una classe e può opzionalmente implementare un'**interfaccia**.
Ogni oggetto ha una **interfaccia** verso l'esterno ben definita: **stato e metodi** pubblici sono visibili, il resto no.
L'implementazione non è visibile all'esterno dell'oggetto (**information hiding**)

Java Bytecode
è il linguaggio della JVM: load & store, aritmetica, logica, stack, invocazione e ritorno metodi...
visualizzabile con javap

Uguaglianza in Java
Due operatori:

- o1 == o2 è **true** se e solo se le variabili o1 e o2 denotano lo stesso riferimento (**Pointer equality**)

- o1.equals(o2) è **true** se e solo se le variabili o1 e o2 denotano due oggetti identici (**Deep equality**)

Abstract Stack Machine
Modello computazionale per Java che permette di descrivere la nozione di stato modificabile.
Struttura della Java ASM:

- **Workspace** per la memorizzazione dei programmi in esecuzione
- **Stack** per la gestione dei binding
- **Heap** per la gestione della **memoria dinamica**

Oltre a questi componenti la ASM ha uno spazio di memoria dinamica in cui vengono memorizzate le tabelle dei metodi degli oggetti.
La ASM è una visione **semplificata** della macchina astratta per la realizzazione del linguaggio, permette di trattare in modo omogeneo la gestione degli oggetti e definisce chiaramente come sono gestite le strutture dati.

Nello stack viene memorizzato il nome della variabile e il riferimento alla cella di memoria dello heap.

Ereditarietà
L'ereditarietà è uno strumento tipico dell'OOP per **riusare codice e creare una gerarchia di astrazioni**.

- **Generalizzazione**: una **superclasse** generalizza una **sottoclasse** fornendo un comportamento che è condiviso da tutte le sottoclassi
- **Specializzazione**: una **sottoclasse** specializza (concretizza) il comportamento di una **superclasse**

L'ereditarietà è importante perché permette di specializzare il comportamento di una classe, prevedendo nuove funzionalità ma contemporaneamente mantenendo le vecchie, senza influenzare codice cliente già scritto.
Soprattutto **subtype polymorphism**: tramite l'ereditarietà una variabile può assumere tipi di classi differenti.
Una funzione con parametro formale T può operare con un parametro formale S a patto che S estenda T.
es.: public void ExMethod(<S extends T> var);

Subtyping
B è un sottotipo di A: ogni oggetto che soddisfa l'interfaccia B soddisfa anche l'interfaccia A
Obiettivo: il codice scritto guardando la specifica A opera correttamente anche se viene usata dalla specifica B.
B è un sottotipo di A: B può essere sostituito per A. Questo perché una istanza del sottotipo soddisfa le proprietà del supertipo e può avere maggiori vincoli.

Sottotipo è un nozione semantica
B è un sottotipo di A se e solo se un oggetto B si può mascherare come uno di A in tutti i possibili contesti.

Ereditarietà è una nozione di implementazione
Creare una nuova classe evidenziando (codice nuovo) solo le differenze)
class B extends A {...}
In Java l'ereditarietà è semplice: una classe può implementare più interfacce ma estendere una sola superclasse. Ad esempio in C++ questo non vale.

Aspetto critico: **il costruttore non viene ereditato**. Tipicamente il costruttore della sottoclasse deve accedere anche alle variabili d'istanza della superclasse: per fare ciò Java fornisce il meccanismo **super**. Es:

```
class A {
        private int n;

        public A(int n) {
                this.n = n;
        }
```

```
}

class B extends A {
        private float f;

        public B(int nn, float f){
                super(nn);
                this.f = f;
        }
}
```
//qua B.n esiste perché B è sottoclasse di A, ed ha valore nn perché inizializzato col costruttore di A chiamato tramite super(nn);

Mentre all'interno di un metodo o di un costruttore la parola chiave **this** permette di riferire l'oggetto corrente, cioè in cui il metodo o costruttore viene chiamato.

Upcasting & Downcasting
Supponendo T extends S

- **Upcasting**: un oggetto di tipo T può essere legato ad una variabile di tipo S.
  Implicito.
- **Downcasting**: un oggetto di tipo S può essere legato ad una variabile di tipo T
  Deve essere esplicito, non posso fare operazioni di casting fuori dalla struttura descritta dalla gerarchia

Tipi

- **Tipo statico di una variabile**: è il tipo della classe (o interfaccia) che definisce quali oggetti possono essere legati a quella variabile. Es:
  ```
  public class C {...}
  C c = new C(...);
  ```
  C è il tipo statico della variabile c
- **Tipo statico di una espressione**: è il tipo che descrive il valore calcolato dall'espressione, solamente in base alla struttura testuale (senza valutarla)
- **Tipo dinamico di un oggetto**: è il tipo della classe di cui l'oggetto è istanza.
- **Tipo dinamico di una variabile o espressione:** è **sempre** un sottotipo del tipo statico. (chiaro dal concetto di ereditarietà)

Dynamic Dispatch
La dichiarazione di una variabile non determina in maniera univoca il tipo di oggetto al quale si riferisce. In esempio:

```
Class B extends A {...} //L'estensione riscrive il metodo m()
A a = new A()
```

Faccio diverse operazioni anche con oggetti di tipo B. Invocando m(), qual è il metodo effettivamente invocato? Quello di A perché A è il tipo dinamico di a. Viene quindi cercato il metodo lungo la gerarchia **a partire dal tipo dinamico dell'oggetto**
A tempo di esecuzione viene utilizzato il tipo dinamico dell'oggetto per determinare nella gerarchia di classi **qual è il metodo più specifico da invocare**.

Per comprendere meglio si estende la ASM di Java con la **tabella dei metodi**. Essa contiene il codice dei metodi e tutte le componenti statiche definite nella classe, oltre ad un puntatore alla classe padre.
Quindi nell'esempio di prima, la tabella di B include un puntatore ad A che include un puntatore a Object.

Le tabelle sono allocate sullo heap (memoria dinamica). Il costruttore alloca sullo heap la tabella dei metodi della classe dell'oggetto creato (se non è già presente). Ogni oggetto sullo heap contiene un puntatore alla tabella dei metodi del suo tipo **dinamico**.
Quindi l'invocazione di un metodo (esempio m() di prima) usa questa tabella per effettuare l'**operazione di dispatch**:

- ricerca sulla gerarchia a partire dalla tabella associata al tipo dinamico dell'oggetto invocato
- si nota l'utilizzo di **this** per indicare l'oggetto che invoca il metodo

Alla slide 23 di http://pages.di.unipi.it/levi/PR2-18-005.pdf c'è una schematizzazione del concetto

Eccezioni

Un metodo può richiedere argomenti **soddisfacenti determinate precondizioni** per procedere con l'esecuzione. Es: m(List L) con L != null
Componenti esterni potrebbero fallire. Es: File non esistente
Implementazioni parziali.
Queste situazioni dove si verificano errori si gestiscono con diverse tecniche:

- parser per gli errori sintattici (ci pensa il compilatore)
- analisi statica per gli errori semantici (type checking, compilatore)
- test covering & best practice
- *ignorare gli errori*

Le **eccezioni sono oggetti**, meccanismi del linguaggio che permettono, dove viene rilevato l'errore, di trasferire il controllo del programma al codice per permette di gestirlo.

Un'eccezione è un particolare tipo di oggetto usato per rappresentare e catturare le condizioni anomale:

- **Throwing** un'eccezione significa programmare una sorta di uscita di emergenza nell'esecuzione di un programma
- **Catching** un'eccezione significa programmare le azioni da eseguire per gestire l'anomalia

Formato dei messaggi d'errore:
[exception class]: [additional description of exception] at [class].[method]([file]: [line number])

Nel caso di una situazione anomala che **provoca la terminazione del programma in esecuzione**, le eccezioni (di run-time in questo caso) sono dette **unchecked exception**.
Per gestire queste eccezioni come un normale problema di programmazione esistono meccanismi come throw + try-catch in Java o failwith in OCaml, che consentono di codificare una situazione di errore e contenerla, gestirla.
In Java, un'eccezione si solleva tramite una primitiva specifica, **throw**. Essa richiede come argomento un sottotipo qualunque di **throwable**, classe che contiene tutti i tipi di errore ed eccezione.
http://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html
throw new NullPointerException()

Se un metodo contiene codice che può generare un'eccezione allora si deve esplicitare nella firma del metodo:

- public void myMethod(...) throws Exception {...}

L'eccezione diventa un componente della firma del metodo: questo tipo di eccezioni sono chiamate **checked exceptions** e "rappresentano eccezioni frequentemente considerate non fatali per l'esecuzione di un programma". Esse devono essere gestite, il compilatore verifica che siano sollevate (throw) e gestite (catch).

In Java esiste anche il try-catch per gestire l'eccezione ed indicare codice da eseguire in caso si verifichi:

```
try {
        <codice che può lanciare eccezioni>
} catch (<T extends Throwable> e) {
```

```
        <codice per gestire l'eccezione di tipo T>
} catch (<U extends Throwable> e) {
        <codice per gestire l'eccezione di tipo U>
} ...
} finally {
        <codice di clean-up che viene sempre eseguito, qualsiasi sia l'eccezione lanciata e anche nel caso non siano avvenute eccezioni>
}
```

Se un nuovo tipo di eccezione estende la classe Exception è Checked:

- VA gestita con catch
- VA specificata nella firma del metodo
- (altrimenti non compila)
- se un chiamante riceve come ritorno un'eccezione sollevata, deve gestirla. Il chiamante può propagarla (throw) solo se è elencata tra le sollevabili.

Se estende RuntimeException è Unchecked:

- può non essere specificata nella firma del metodo
- se un chiamante riceve come ritorno un'eccezione sollevata, può essere gestita o propagata

Con un **throw** il metodo termina sollevando l'eccezione indicata. Il controllo non riprende dal codice che segue la chiamata (come in un return normale) ma viene trasferito ad un pezzo di codice che gestisce l'eccezione (con try-catch oppure propagandolo al codice chiamante)

Quando l'eccezione si solleva all'interno di un try, il controllo passa alla prima clausola catch che identifica l'eccezione o un suo supertipo. Il try-catch è utile anche per catturare eccezioni unchecked.

Se obj chiama un metodo che ritorna un'eccezione anche obj torna un'eccezione se:

- viene propagata automaticamente (stessa eccezione)
- viene catturata e ne viene sollevata un'altra (possibilmente diversa)

Le eccezioni non sono necessariamente errori ma metodi per sollevare l'attenzione in situazioni particolari (classificate come eccezioni). Errori ad un certo livello possono non esserlo a livelli d'astrazione superiori. Il compito primario è ridurre al minimo i vincoli nella strutturazione del programma in modo da gestire e codificare informazione su terminazioni particolari.

Le istruzioni checked offrono maggior protezione degli errori (più facili da catturare, compilatore controlla che vengano gestite)
Le istruzioni checked sono più pesanti da gestire quando siamo abbastanza sicuri che non vengano sollevate (questo perché esistono modi efficienti e convenienti per evitarle o per i contesti d'uso limitati, solo in questi casi si opta per un'istruzione unchecked)

Nella **defensive programming** si incoraggia il programmatore a verificare l'assenza di errore ogni volta che è possibile.

Checked vs. unchecked

- Pro Checked Exceptions
  - Compiler enforced catching or propagation of checked exceptions make it harder to forget handling that exception
- Pro Checked Exceptions
  - Unchecked exceptions makes it easier to forget handling errors since the compiler doesn't force the developer to catch or propagate exceptions (reverse of 1)
- Pro Unchecked Exceptions
  - Checked exceptions that are propagated up the call stack clutter the top level methods, because these methods need to declare throwing all exceptions thrown from methods they call
- Pro Checked Exceptions
  - When methods do not declare what unchecked exceptions they may throw it becomes more difficult to handle them
- Pro Unchecked Exceptions
  - Checked exceptions thrown become part of a methods interface and makes it harder to add or remove exceptions from the method in later versions of the class or interface

http://pages.di.unipi.it/levi/PR2-18-006b.pdf es. operazionale di eccezione

Astrazione sui dati in Java: specifica di tipi di dato astratti
Data Abstraction: separazione delle proprietà logiche dei dati dai dettagli della loro rappresentazione
Quindi un utente vede solo **cosa** fa il programma, il programmatore vede anche **come** lo fa.

Contratti d'uso
Un contratto è l'insieme dei vincoli d'uso che vengono concordati tra l'utente di un software e chi lo implementa.

- Separation of concern: i dettagli implementativi sono mascherati all'utente che vede solo la funzionalità offerta (e si suppone richiesta)
- Facilitano manutenzione e ri-uso del software

In **Java** le **interfacce** permettono di definire esplicitamente il confine tra cliente e implementatore. Le interfacce definiscono sintassi e tipo dei metodi ma non definiscono comportamento e effetto atteso dell'esecuzione (quindi sintassi e tipi forniscono un'informazione limitata ai clienti)
Estendere le interfacce con informazioni sul codice effettivo, ma il codice:

- è complesso per il cliente, che non deve sapere nel dettaglio come è implementato un metodo ma avere solo un'**astrazione del comportamento**
- può cambiare nel tempo e il cliente non è interessato alle modifiche puntuali

Abstract Data Type (ADT)
Un ADT è una collezione di elementi il cui comportamento logico è definito da un dominio di valori e da un insieme di operazioni su quel dominio.

- Nome
- Valori
- Operazioni
- Semantica delle operazioni

Le operazioni sono caratterizzate da:

- **Precondition**: formula logica che caratterizza le proprietà è il valore degli **argomenti**
- **Postcondition**: formula logica che caratterizza il **risultato rispetto al valore degli argomenti**

s.push(t) //porta ad uno stack s1
/** PRECOND: s è un'istanza valida di Stack && s non è full
* POSTCOND: s1 è un'istanza valida di Stack && s1 == s esteso con t
* come elemento top
**/

Data abstraction via specifica

Con la specifica astraiamo dall'implementazione del tipo di dato. Avendo gli oggetti insieme alle operazioni l'astrazione diventa possibile:

- la rappresentazione è nascosta all'utente ma visibile all'implementazione delle operazioni
- se una rappresentazione viene modificata, devono essere modificate le implementazioni delle operazioni ma non le astrazioni che la utilizzano.

Realizzazione della specifica:

- Java (parte sintattica) con:
  - classe o interfaccia
  - nome della classe
  - metodi (inclusi construttori)
- specifica del tipo
  - clausola Overview che descrive i valori astratti degli oggetti e alcune proprietà
- specifica di metodi con REQUIRES (precond) e EFFECTS (postcond)

```
public class NuovoTipo {
        //OVERVIEW:     Gli oggetti di nuovo tipo sono... con proprietà...

        //Costruttore
        public NuovoTipo();
        //REQUIRES:                 ...
        //EFFECTS:                  ...
        ...
}
```

```
public class IntSet {
  // OVERVIEW: un IntSet è un insieme modificabile
  // di interi di dimensione qualunque
  //costruttore
    public IntSet( )
     // EFFECTS: inizializza this all'insieme vuoto
  // metodi
    public void insert(int x)
      // EFFECTS: aggiunge x a this
    public void remove (int x)
      // EFFECTSS: toglie x da this
    public boolean isIn(int x)
      // EFFECTS: se x appartiene a this ritorna true,
      // false altrimenti
...
}
```

http://pages.di.unipi.it/levi/PR2-18-007.pdf 19+ esempi

Quindi:

- definire specifica
    - scheletro con header, overview, precond e postcond per tutti i metodi

Abstract Data Type

- Insieme di **valori**
- Insieme di **operazioni** che possono essere applicate in modo uniforme ai valori
- **NON** è caratterizzato dalla **rappresentazione** dei dati:
    - La rappresentazione è privata, senza effetto sul codice che utilizza quel tipo di dato
    - se l'ADT è **mutabile** allora la rappresentazione è **modificabile**

La specifica di un ADT è quindi un **contratto** che definisce **valori**, **operazioni**, parametri tipo, effetti ossservabili.

Separation of concerns:

- Progettazione e realizzazione ADT
- Progettazione applicazione che usa ADT

Specifiche e implementazioni

Se IMPL è una possibile implementazione della specifca S, allora

- **Impl soddisfa S** se
    - Ogni comportamento di Impl è un comportamento premesso dalla specifica S
    - cioè **i comportamenti di impl sono un sottoinsieme dei comportamenti specificati da S**
- **Impl non soddisfa S** allora Impl o S non sono "corretti"
  Preferibile cambiare l'implementazione rispetto alla specifica

Possono naturalmente venir fuori due specifiche diverse della stessa implementazione. Possibile il confronto.

Una specifica **forte** è difficile da soddisfare (più vincoli sull'implementazione) ma è facile da usare (si possono fare più assunzioni sul comportamento)
Una specifica **debole** invece è facile da verificare (molte implementazioni la verificano, pochi vincoli) ma difficile da usare per il minor numero di assunzioni che si possono fare
Formalmente: S1 è **più forte** di S2 se S1 ⇒ S2

Nell'implementazione delle astrazioni sui dati, la scelta fondamentale è **come** i valori del tipo astratto sono implementati in termini di altri valori:

- Tipi primitivi o già implementati
- Nuovi tipi astratti che facilitano l'implementazione
    - Questi tipi vengono specificati

La scelta deve tenere conto di come implementare in modo efficiente costruttori e metodi
L'implementazione del metodo e dei costruttori **viene dopo** questa scelta.

Per verificare e validare un'implementazione:

- **funzione di astrazione**
- **invariante di rappresentazione**
- **dimostrazione tramite induzione su dati**

Proprietà dell'astrazione

- modificabilità
    - i tipi non modificabili sono più sicuri ma anche più inefficienti, perché richiedono di copiarli spesso e complica la vita al garbage collector
    - la scelta deve tenere conto delle caratteristiche dei concetti matematici o degli oggetti modellati dal mondo reale
    - Un tipo non modificabile **può essere implementato utilizzando strutture modificabili**, facendo attenzione agli effetti collaterali (come restituire la rappresentazione modificabile cioè **esporre la rappresentazione**)
- categorie di operazioni
    - **creatori** che creano oggetti del loro tipo dal nulla (costruttori)
    - **produttori** che prendono come parametri oggetti del loro tipo e ne costruiscono altri (costruttori o metodi, ad esempio initialize o cose così)
    - **modificatori** modificano oggetti del loro tipo (add, remove…)
    - **osservatori** che prendono oggetti del loro tipo e restituiscono altri valori per ottenere info (length, isIn, charAt)
    - **Sempre almeno un creatore**, qualche **produttore** se tipo non modificabile, qualche **modificatore** se tipo modificabile, qualche **osservatore**
- dimostrare proprietà
    - aspetto significativo dello sviluppo sw safe: **garantire le proprietà delle astrazioni**
    - per dimostrare proprietà si usano specifiche tramite **induzione strutturale**
        - Si dimostra proprietà valida sui valori costruiti dai creatori

- si dimostra che se vale prima vale anche dopo ogni applicazione di modificatore o produttore

Dimostrare la correttezza dell'implementazione significa dimostrare che la rappresentazione rappresenta in modo corretto i valori del tipo di dato astratto (definito in OVERVIEW) e che i metodi soddisfano le rispettive specifiche (definite nelle pre-post condizioni)

Funzione di astrazione

Per ogni valore concreto restituisce il valore astratto

Cattura l'intenzione del progettista nello scegliere una particolare rappresentazione.

La **funzione di astrazione** f : C -> A porta da uno stato concreto (stato di un oggetto dell'implementazione definito in termini della sua rappresentazione) ad uno stato astratto (ai valori dell'oggetto astratto definito della OVERVIEW). Può essere molti-a-uno e **deve essere sempre definita**:

- perché è parte importante delle decisioni relative all'implementazione
- sintatticamente è inserita come commento all'implementazione, dopo le dichiarazioni delle variabili d'istanza che definiscono la rappresentazione
- senza di essa non si può dimostrare la correttezza della rappresentazione

Per definirla formalmente dobbiamo avere una notazione per i valori astratti: quando è necessario la si indica nella OVERVIEW "*//Typical element*".

Quindi:

- stabilisce come interpretare la struttura dati concreta dell'implementazione
- definita solamente sui valori che rappresentano l'IR
- Guida per chi implementa/modifica/verifica: ogni operazione deve preservare la funzione di astrazione

Invariante di rappresentazione

Quali valori concreti che rappresentano valori astratti

l'IR è un predicato I : C -> bool che è verificato **solo per gli stati concreti che sono rappresentazioni legittime di uno stato astratto**.

Insieme alla funzione di astrazione riflette le scelte relative alla rappresentazione ed è quindi inserita nella documentazione come commento insieme alla funzione di astrazione.

Viene accompagnata da una dimostrazione formale che garantisce che tutti i metodi preservano l'invariante.

Può anche essere verificata dinamicamente con un metodo speciale chiamato **`repOk`** che diventa inutile in presenza della dimostrazione formale.

Verifica:

- **base** dimostriamo che invariante vale per gli oggetti dei costruttori
- **induttivo** dimostriamo che vale per tutti i metodi (produttori e modificatori)
    - assumendo che valga per this e per tutti gli argomenti del tipo
    - dimostriamo che quando il metodo ritorna vale
        - per this
        - per gli argomenti del tipo
        - per gli oggetti del tipo ritornati
- induzione sul numero di invocazioni dei metodi usati per produrre il valore corrente dell'oggetto. La base è fornita dai costruttori.

Quindi:

- Stabilisce se un'istanza è **ben formata**
- stabilisce l'insieme concreto dei valori dell'astrazione (ovvero l'insieme dei valori che sono implementazione dei valori astratti)
- Guida per chi implementa/modifica/verifica l'implementazione delle astrazioni: nessun oggetto deve violare l'IR

Gerarchie di Tipi

B è **sottotipo** di A = ogni oggetto che soddisfa l'interfaccia B soddisfa anche l'interfaccia A.
Metodologicamente, il codice scritto per la specifica di A opera correttamente anche sotto la specifica B.
-> B può essere sostituito per A
Quindi **B è sottotipo di A se e solo se un oggetto di B si può mascherare come oggetto di A in tutti i possibili contesti**.
Sottotipo è semantica, l'**ereditarietà** è implementazione (creare una nuova classe evidenziando solo il codice nuovo)

Specifica del supertipo

- Come specifica classi già vista
- unica differenza è che può essere parziale, ad es. possono mancare i costruttori

Specifica di un sottotipo

- data relativamente alla specifica dei suoi supertipi
- non vengono ridefinite le parti che non cambiano dalla specifica del supertipo
- vanno specificati solo
    - costruttori del sottotipo
    - metodi nuovi forniti dal sottotipo
    - metodi del supertipo che il sottotipo ridefinisce

Implementazione del supertipo

- Può non essere implementato affatto
- Può avere implementazione parziale
- Può fornire informazioni a potenziali sottotipo dando accesso a variabili o metodi d'istanza

Implementazione del sottotipo

- Implementati come estensioni dell'implementazione del supertipo
- la rep degli oggetti del sottotipo contiene anche le variabili d'instanza definite nell'implementazione del supertipo
- alcuni metodi possono essere ereditati
- altri metodi possono essere ridefiniti

Il supertipo fornisce in ogni caso la specifica del tipo: nel caso dell'interfaccia fornisce solo la specifica, nel caso della classe può fornire parte dell'implementazione.
Nel caso delle classi, esse possono essere:

- astratte (forniscono implementazione parziale)
    - Non hanno oggetti
    - codice esterno non può richiamare i loro costruttori
    - metodi astratti la cui implementazione è lasciata a qualche sottoclasse
- concrete (forniscono implementazione completa)

Entrambe possono contenere **metodi finali**, cioè metodi che i sottotipi non possono reimplementare
Le interfacce invece definiscono solo la specifica e non implementano nulla:

- contengono solo specifica dei metodi (pubblici, non statici, astratti)

Una sottoclasse **dichiara** la superclasse che estende (e/o le interfacce che implementa)

- ha tutti metodi superclasse con stessi nomi e segnature
- può implementare i metodi astratti e reimplementare quelli non final
- qualunque metodo sovrascritto deve avere segnatura identica a quella della superclasse (ma possono sollevare meno eccezioni)

Un oggetto della sottoclasse è rappresentato da variabili d'istanza proprie e da quelle della superclasse (che non possono essere accedute direttamente se sono dichiarate private)

La superclasse può lasciare parti della sua implementazione accessibili alle sottoclassi (dichiarando metodi/variabili come protected)

- implementazioni delle sottoclassi più efficienti
- si perde astrazione completa, che dovrebbe consentire di reimplementare superclasse senza influenzare sottoclassi
- entità protected visibili anche all'interno dell'eventuale package
- meglio interagire con le superclassi attraverso le interfacce pubbliche

Classi astratte come supertipi

- implementazione parziale del tipo
    - la parte generica dell'implementazione è fornita dalla superclasse
    - le sottoclassi forniscono i dettagli
- può avere variabili d'istanza e uno o più costruttori
- **non** ha oggetti
- costruttori chiamati solo dalle sottoclassi per inizializzare la parte di rappresentazione della superclasse
- può contenere metodi astratti
- può contenere metodi regolari usando anche i metodi astratti

Principio di Sostituzione
Un oggetto del sottotipo **può essere sostituito da un oggetto del supertipo** senza influire sul comportamento dei programmi che utilizzano il tipo.

Regola della segnatura: gli oggetti del sottotipo devono avere tutti i metodi del supertipo, le segnature dei metodi del sottotipo devono essere compatibili con le segnature dei corrispondenti metodi del supertipo
Regola dei metodi: le chiamate dei metodi del sottotipo devono comportarsi come le chiamate dei corrispondenti metodi del supertipo

- in generale un sottotipo può indebolire le precond e rafforzare le postcond. Per mantenere la compatibilità tra specifiche del super e del sottotipo:
    - PREsuper => PREsub
      Codice scritto per usare il supertipo, verifica presuper e di conseguenza presub
    - PREsuper && POSTsub => POSTsuper
      Codice scritto per usare il supertipo, assume effetti specificati nel postsuper e se chiamata soddisfa precond forte (presub) allora gli effetti includono quelli del super

Regola delle proprietà: il sottotipo deve preservare tutte le proprietà che possono essere provate sugli oggetti del supertipo

- Per le proprietà invarianti dobbiamo dimostrare che tutti i metodi nuovi costruttori inclusi preservano invariante e provare che creatori e produttori stabiliscano invariante (induzione sul tipo)
- Per le proprietà di evoluzione provare che ogni metodo del sottotipo le preserva

Le regole **riguardano la semantica**.

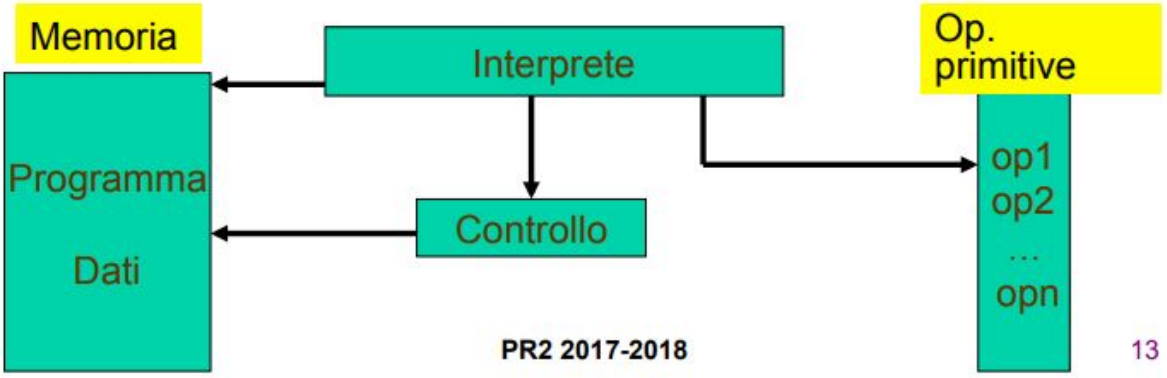
Per la regola della segnatura, garantita dal compilatore Java, se una chiamata è typecorrect per il supertipo lo è anche per il sottotipo. Può sollevare meno eccezioni del supertipo o ritornare un tipo più specifico.
Le altre due regole non vengono garantite dal compilatore poiché hanno a che fare con la semantica.

Codice Sorgente (programmatore) -> **Compilatore** -> **Codice macchina** -> **Interprete** -> **Esecuzione**

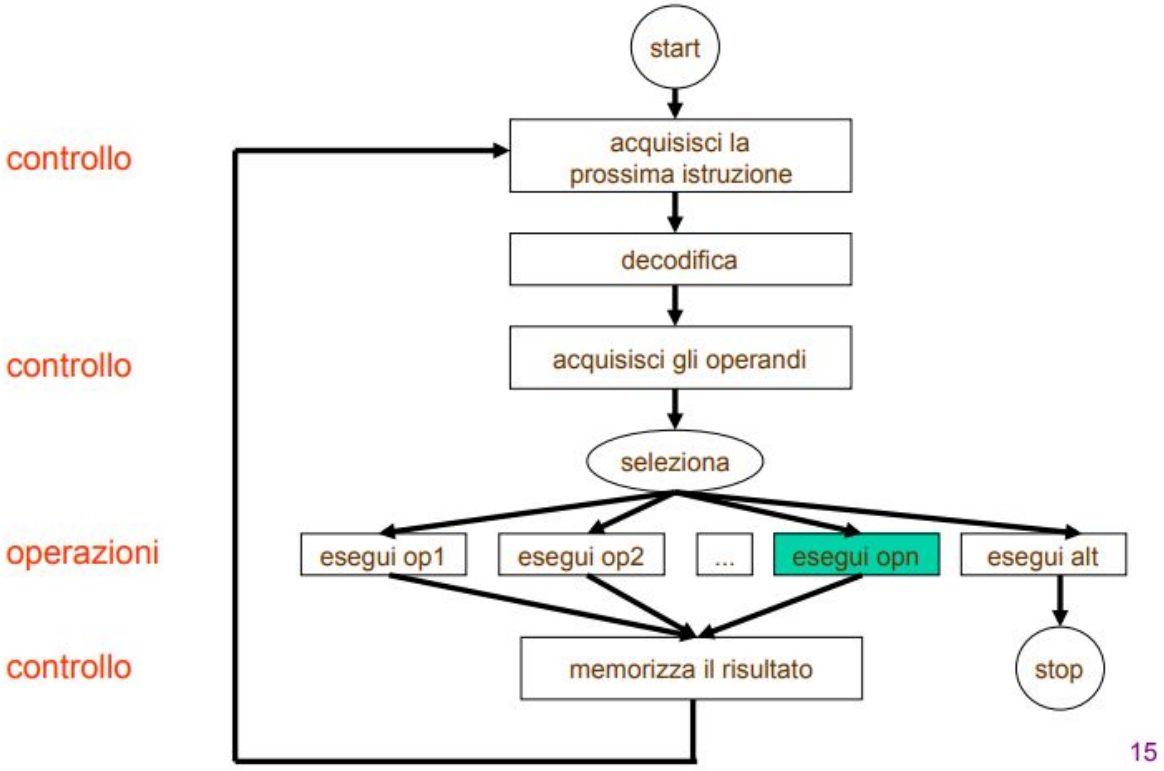
Macchina Astratta

Un sistema virtuale che rappresenta il comportamento di una macchina fisica individuando precisamente l'insieme delle risorse necessarie per l'esecuzione di programmi

- collezione di strutture dati e algoritmi in grado di **memorizzare** ed **eseguire programmi**
- componenti principali:
    - Interprete
    - Memoria (dati e programmi)
    - Controllo
        - collezione di strutture dati e algoritmi per:
            - Acquisire prossima istruzione (fetch)
            - Acquisire operandi e memorizzare risultati operazioni
            - Gestire chiamate e ritorni dai sottoprogrammi
            - Mantenere le associazioni fra nomi e valori denotati
            - Gestire dinamicamente la memoria
            - ...
    - Operazioni primitive



Interprete



Linguaggio Macchina

- **M** macchina astratta
- **LM** Linguaggio macchina di M
    - è il linguaggio che ha come stringhe legali tutti i programmi interpretabili dall'interprete di M
- I programmi sono particolari dati su cui opera l'interprete
- Alle componenti di M corrispondono componenti di LM
    - tipi di dato primitivi
    - costruttori di controllo
        - per controllare ordine esecuzione
        - per controllare acquisizione dati

I **componenti di M** sono **realizzati mediante strutture dati e algoritmi implementati nel linguaggio macchina di una macchina ospite** Mo già esistente ed implementata.
La realizzazione dell'interprete di M può coincidere con l'interprete di Mo (M è realizzata come estensione di Mo, gli altri componenti possono differire) o può essere diverso (M è realizzata su Mo in modo interpretativo, altre componenti possono essere uguali).

Da Linguaggio a Macchina Astratta

L linguaggio -> ML macchina astratta di L
Implementazione di L = **realizzare ML su una macchina ospite Mo**
Se L è di alto livello e Mo una macchina fisica: l'interprete di ML è necessariamente diverso dall'interprete di Mo:

- ML realizzata su Mo in modo interpretativo
- L'implementazione di L si chiama interprete
- Esiste una soluzione alternativa basata su tecniche di traduzione (compilatore?)

Implementare un linguaggio

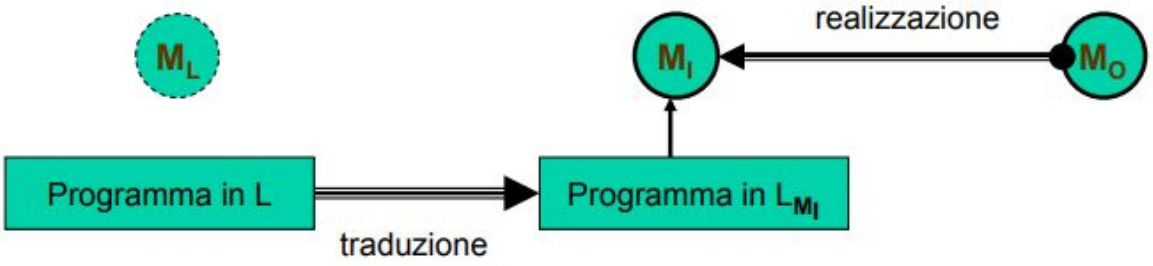
L linguaggio ad alto livello

ML macchina astratta di L

Mo macchina ospite

- **Interprete puro**
  - ML realizzata su Mo in modo interpretativo
  - Scarsa efficienza soprattutto per colpa dell'interprete (ciclo di decodifica)
- **Compilatore puro**
  - Programmi di L tradotti in programmi funzionalmente equivalenti nel linguaggio macchina di Mo (**ML non viene realizzata**)
  - Il problema è la dimensione del codice prodotto

Entrambi casi limite che nella realtà non esistono quasi mai



MI Macchina intermedia

Traduzione dei programmi da L al linguaggio intermedio LMI, realizzazione della macchina intermedia MI su Mo

ML = MI Interprete (puro)

Mo = MI compilatore (puro)

- Possibile solo se la differenza fra Mo e ML è molto limitata (L Linguaggio Assembler di Mo)
- In tutti gli altri casi c'è sempre una macchina intermedia che estende eventualmente la macchina ospite in alcuni componenti