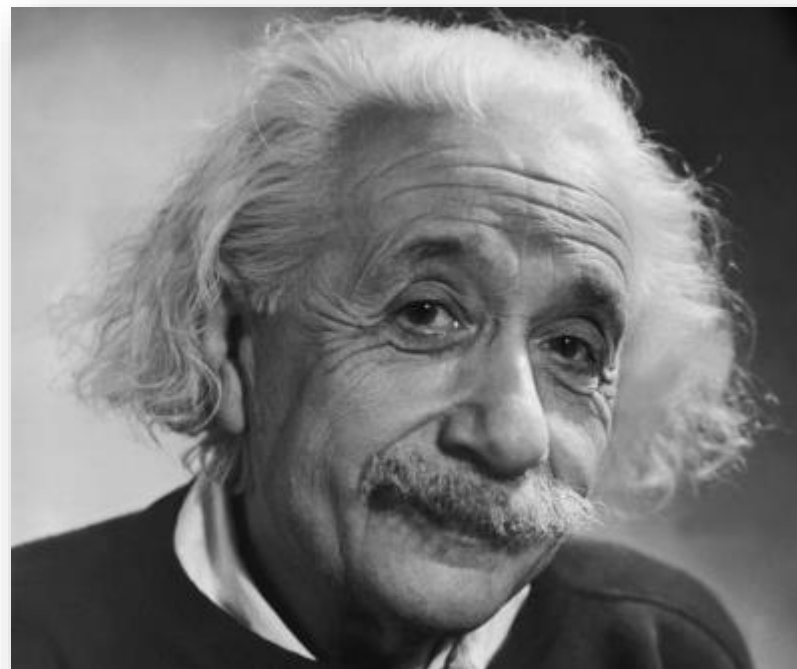


problemi - algoritmi
programmi - paradigmi

problema

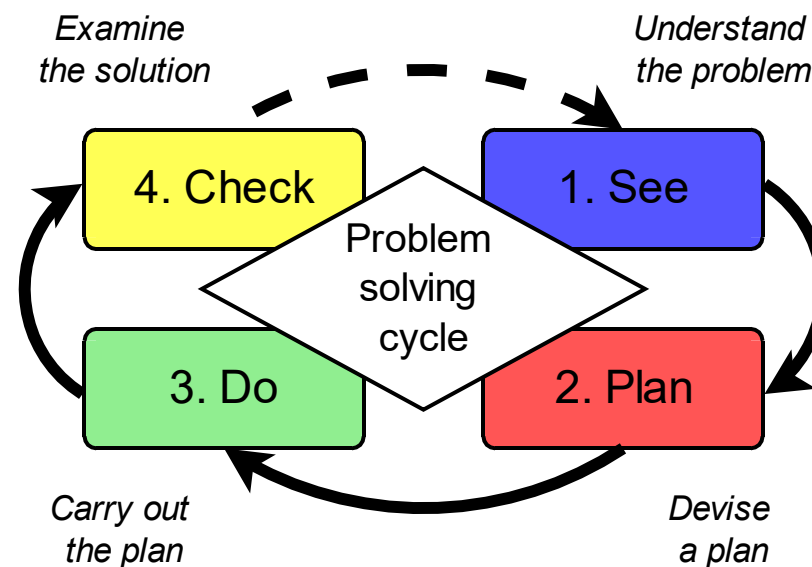
- risolvere un problema:
 - passare da uno stato iniziale
 - a uno stato finale
 - attraverso stati intermedi



problem solving - see

(1) see

- capire il problema
- quali sono i dati, quali le incognite?
- quali sono le condizioni? sono soddisfacibili, ridondanti, contraddittorie?



Make things as simple as possible, but not simpler. (A. Einstein)

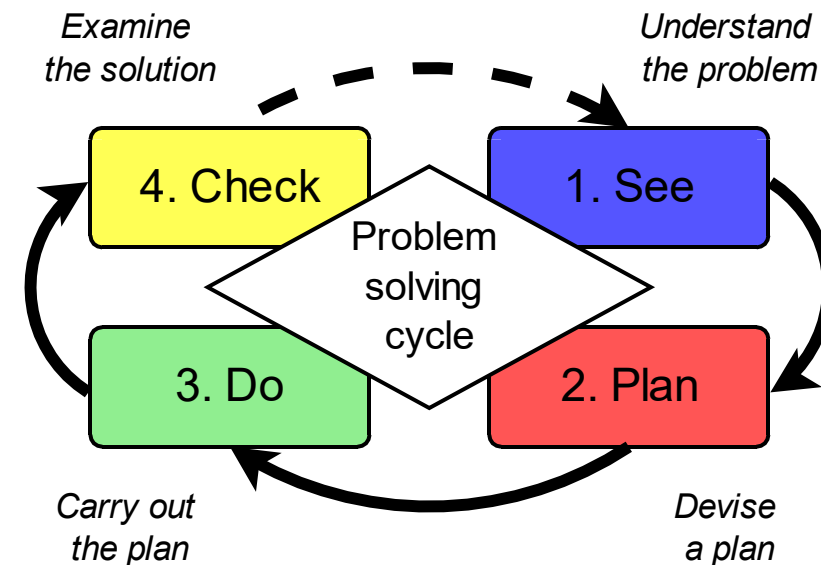
For every complex problem there is an answer that is clear, simple, and wrong.

(H.L. Mencken)

*problem solving - **plan***

(2) **plan**

- elaborare un piano
- mettere in relazione dati e incognite
- metodologie: divide et impera, composizione, astrazione...
- computational thinking
- cominciare a risolvere un problema più semplice
 - (vincoli rilassati)



*If you can't solve a problem...
then there is an easier problem you can solve: find it. (G. Polya)*

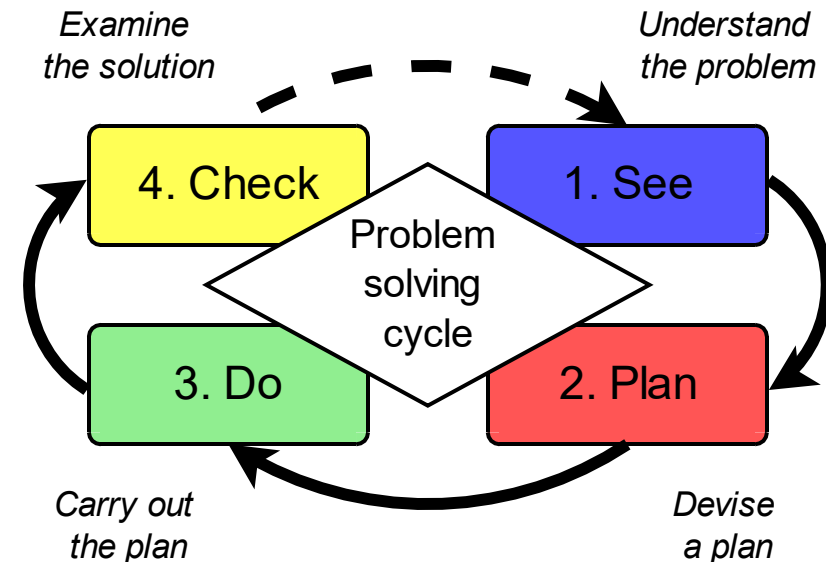
problem solving - do & check

(3) **do**

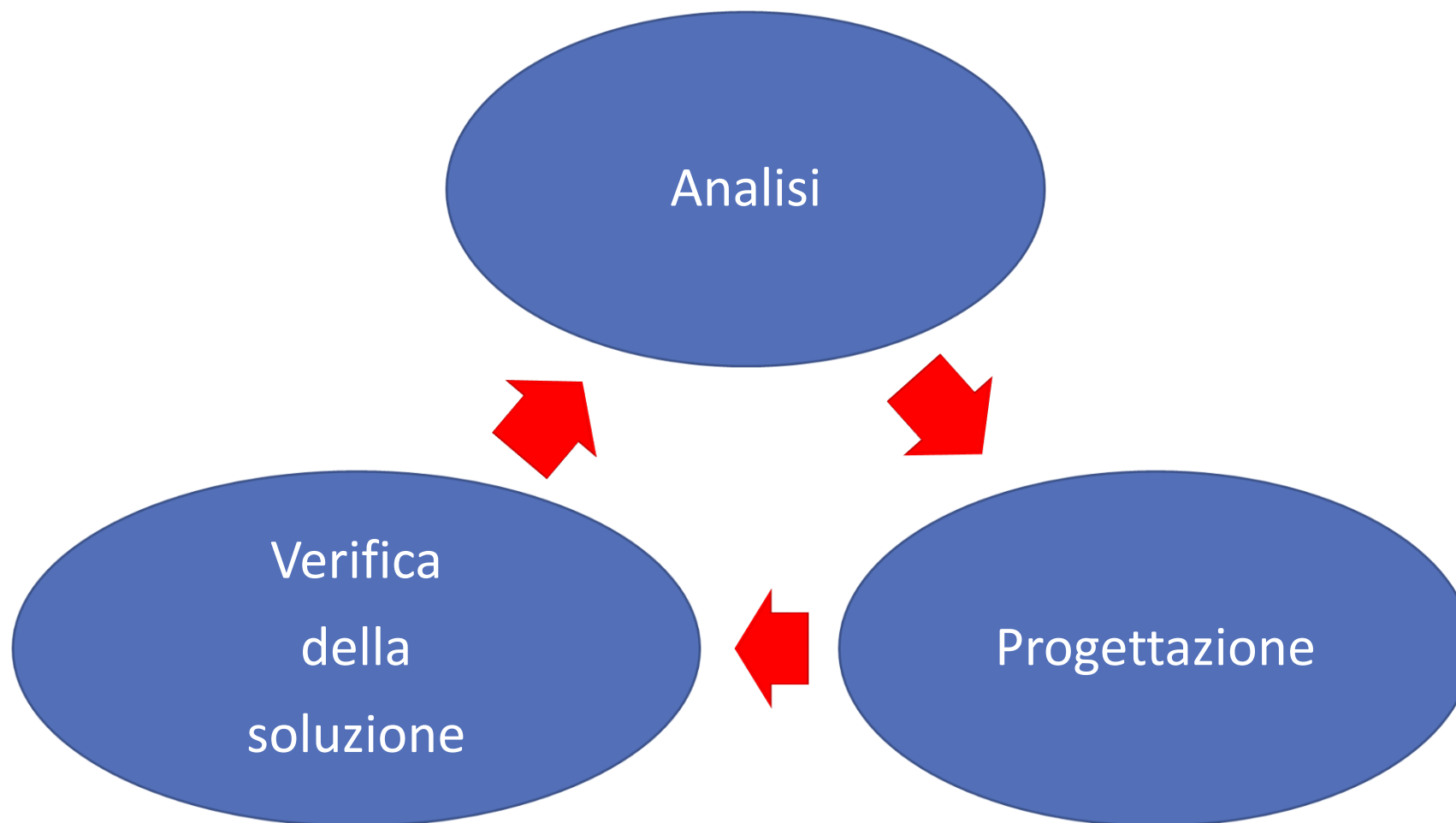
- eseguire il piano
- controllare ogni passo
 - è corretto?

(4) **check**

- controllare la soluzione
- è corretta?
- è ottenibile in altro modo?
- il risultato è utilizzabile per altri problemi?

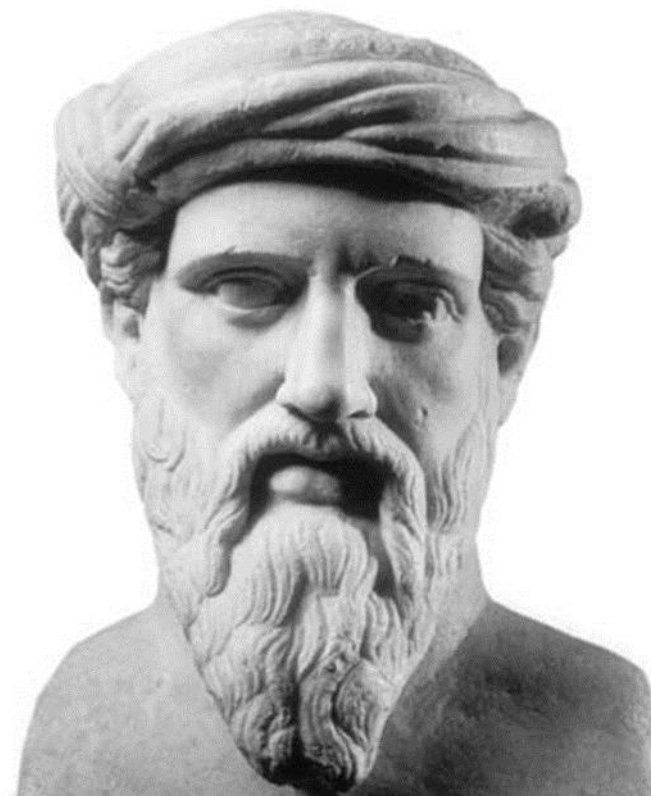


sintesi



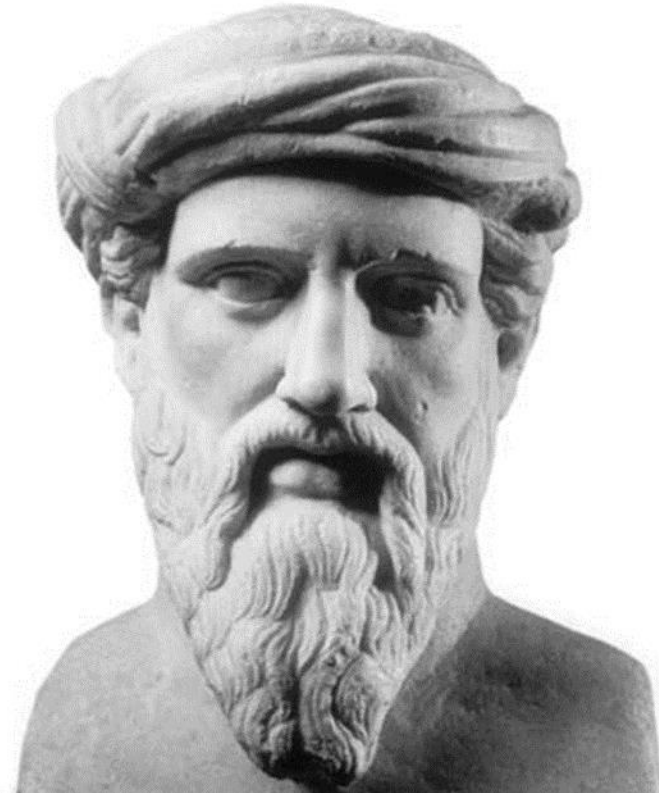
esempio: analisi

- l'***analista*** deve raccogliere le informazioni necessarie per definire il problema
- individua le ***informazioni iniziali*** significative
- individuare le ***informazioni finali*** (risultato)
- esempio: Pitagora identifica come ***obiettivo*** la ricerca del valore dell'ipotenusa di un triangolo rettangolo e come ***dati iniziali*** significativi i valori dei due cateti



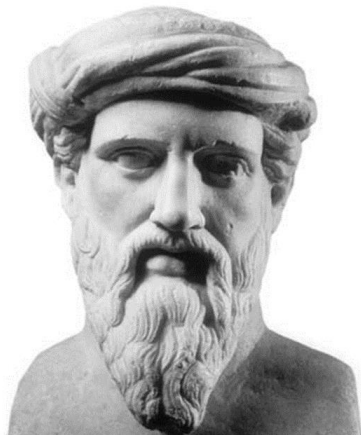
esempio: progettazione

- il **progettista** fornisce una descrizione del procedimento che porta alla soluzione del problema (*algoritmo*)
- specifica le azioni da eseguire per passare dai dati iniziali ai dati intermedi ai risultati finali
- esempio:
 - calcola il quadrato del primo cateto
 - calcola il quadrato del secondo cateto
 - somma i due valori ottenuti
 - calcola la radice quadrata del valore ottenuto



esempio: programmazione

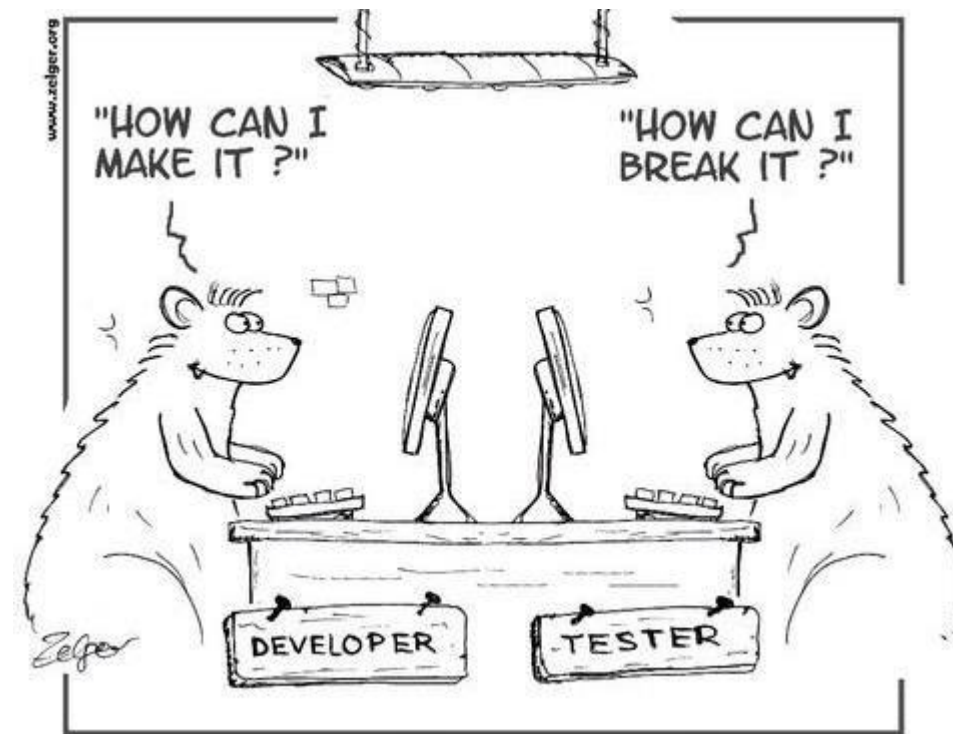
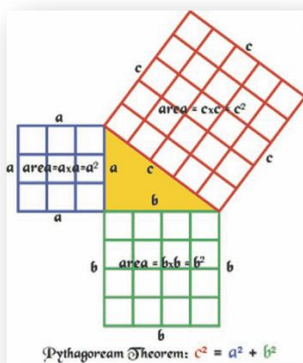
- se il *risolutore* è un computer
 l'algoritmo deve essere tradotto
 in un *linguaggio di
 programmazione*



```
""" pitagora """
import math
# dati di input
c1 = float(input("primo cateto: "))
c2 = float(input("secondo cateto: "))
# calcola il quadrato del primo cateto
q1 = math.pow(c1,2)
# calcola il quadrato del secondo cateto
q2 = math.pow(c2,2)
# somma i due valori ottenuti
s = q1 + q2
# calcola la radice quadrata del valore ottenuto
ip = math.sqrt(s)
# dati di output
print("ipotenusa",ip)
```

*esempio: **verifica** della soluzione*

- il **tester** verifica che i risultati ottenuti non generino alcuna contraddizione con i dati iniziali
- in caso contrario si deve ripartire dall'analisi per poi passare di nuovo alla progettazione finché la verifica della soluzione non ha dato esito positivo



definizioni

- ***algoritmo***
 - procedimento che risolve un determinato problema attraverso un numero *finito* di passi *elementari* (al-Khwarizmi, جعفر محمد خوارزمی ~800)
- ***dati***
 - *iniziali* (istanza problema), *intermedi*, *finali* (soluzione)
- ***passi elementari***
 - azioni *atomiche* non scomponibili in azioni più semplici
- ***processo*** (esecuzione)
 - sequenza ordinata di passi

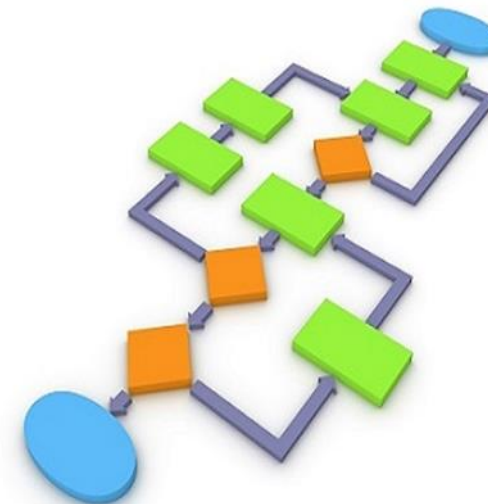


linguaggi algoritmici

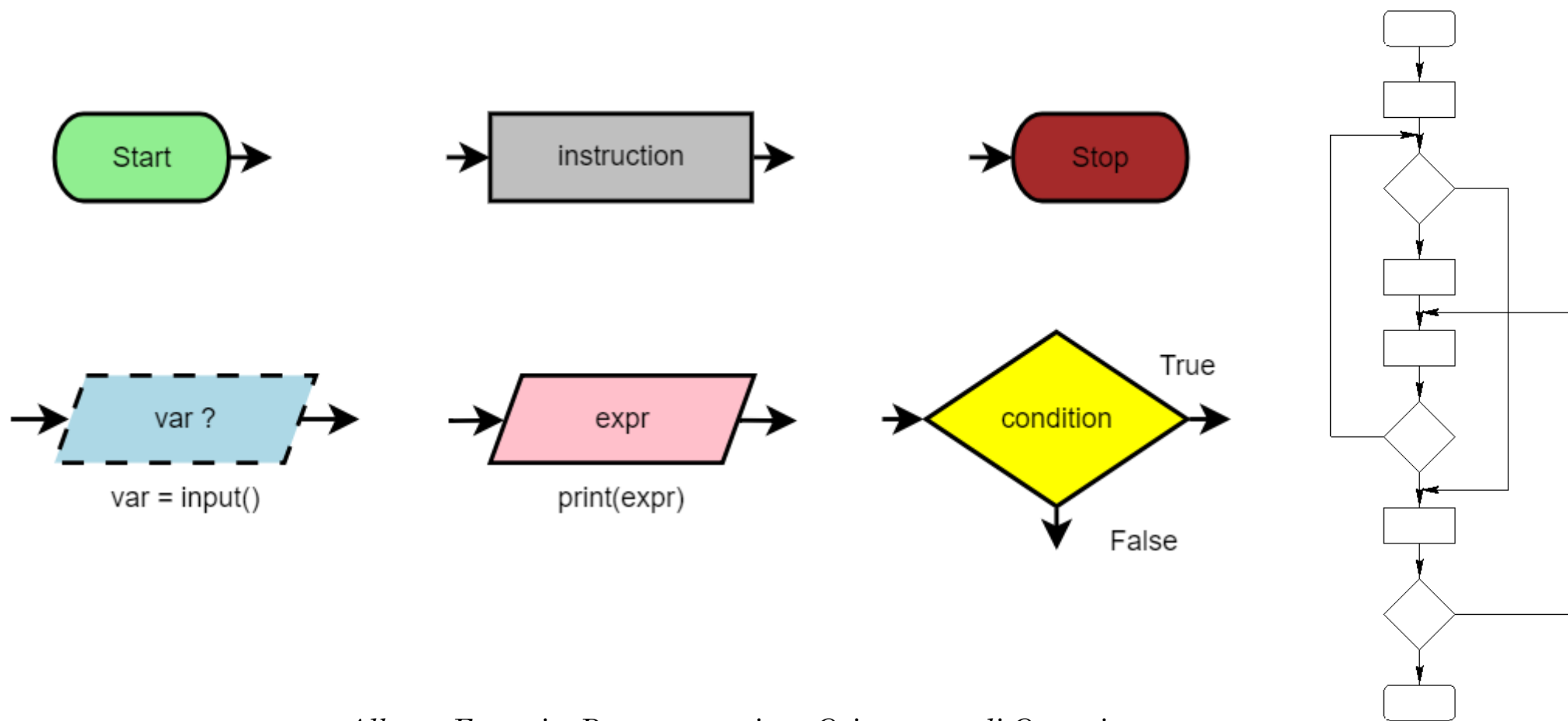
- caratteristiche di un linguaggio algoritmico
 - ***non ambiguità***
 - capacità di esplicitare il ***flusso*** di esecuzione delle istruzioni
- deve contenere istruzioni di tipo:
 - ***operativo*** (fare qualcosa)
 - ***input/output*** (comunicare con il mondo esterno)
 - ***decisionale*** (variare il flusso di esecuzione)

diagrammi di flusso

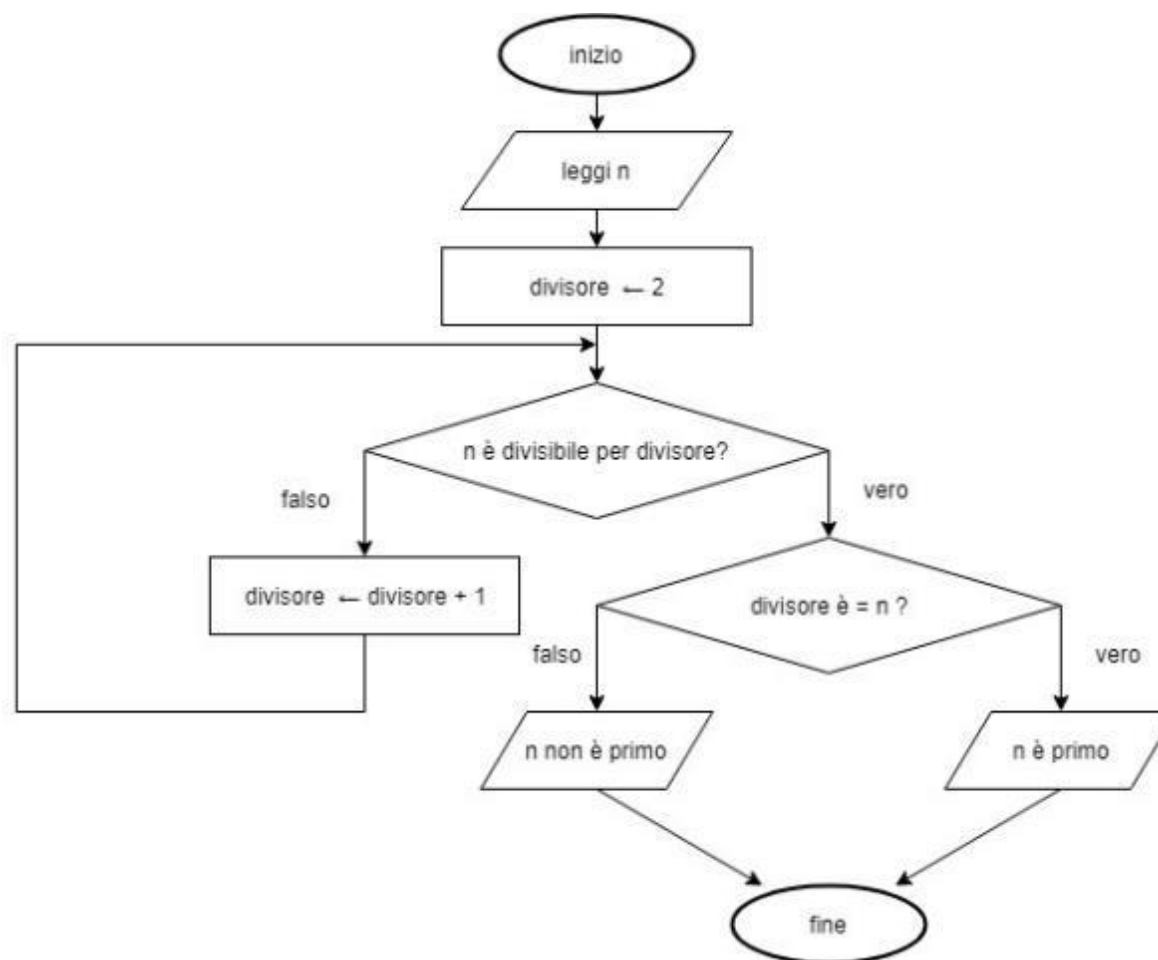
- diagramma di flusso
(*flow-chart*):
 - rappresentazione grafica di algoritmi
 - più *efficace* e *meno ambigua* di una descrizione a parole
- è un grafo orientato
- due tipi di entità:
 - *nodi*
 - *archi*



tipi di nodi



«determinare se un numero è primo» - flow chart



programmazione

- l'esecutore è un computer
 - pregi
 - velocità
 - correttezza
 - ...
 - difetti
 - «rigidità»
 - comprensione del linguaggio



linguaggi di programmazione

- linguaggi ad **alto livello**
 - più simili al linguaggio umano
 - gestiscono automaticamente alcuni aspetti del computer
 - «indipendenti dal tipo di hardware»
- linguaggi a **basso livello**
 - offrono ai programmatori un maggiore controllo sul computer
 - richiedono conoscenze minuziose del suo funzionamento
 - possono non essere compatibili con hardware diversi
- linguaggio **macchina**
 - istruzioni elementari scritte in binario

linguaggi ad alto livello

- relativa rapidità di scrittura
- relativa facilità di comprensione
- spesso la velocità di esecuzione è abbastanza alta
- possono essere usati su hardware diversi
- non richiedono la conoscenza dell'hardware

```
n=20
numeri=[]

for i in range(n):
    numero=int(input('Inserisci un numero: '))
    numeri.append(numero)
```

linguaggi a basso livello

- controllo diretto delle funzioni hardware
- possono aumentare la velocità di codici che necessitano del massimo della potenza di calcolo
- richiedono una comprensione dell'hardware
- i programmi girano solo su processori simili

```
MOV DS AX  
INT 22H  
ADD DS AX
```

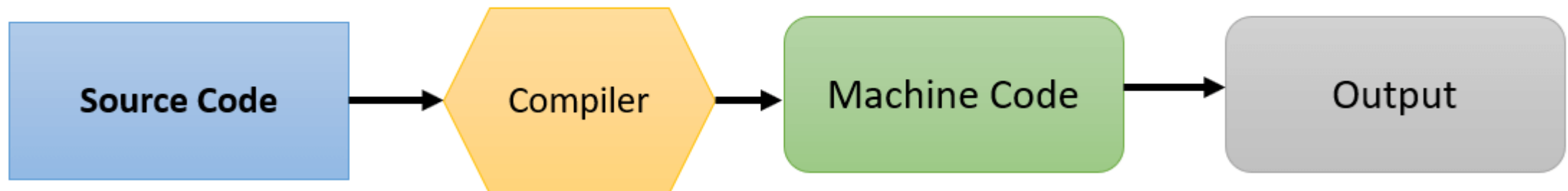
linguaggi compilati e interpretati

- nei programmi scritti con linguaggi di programmazione ad alto livello ogni istruzione si traduce in un insieme spesso corposo di istruzioni a livello macchina
- i linguaggi di programmazione si dividono in due tipologie:
 - linguaggi *compilati*
 - linguaggi *interpretati*
- la differenza è il modo in cui il linguaggio ad alto livello viene tradotto in istruzioni in codice macchina

linguaggi compilati

- il **compilatore** è un software che traduce il codice sorgente in codice macchina e lo memorizza in un file (**codice oggetto**)
- un programma eseguibile contiene istruzioni in codice macchina **specifiche** di un processore
- esempi di linguaggi **compilati** sono **C** e **C++**

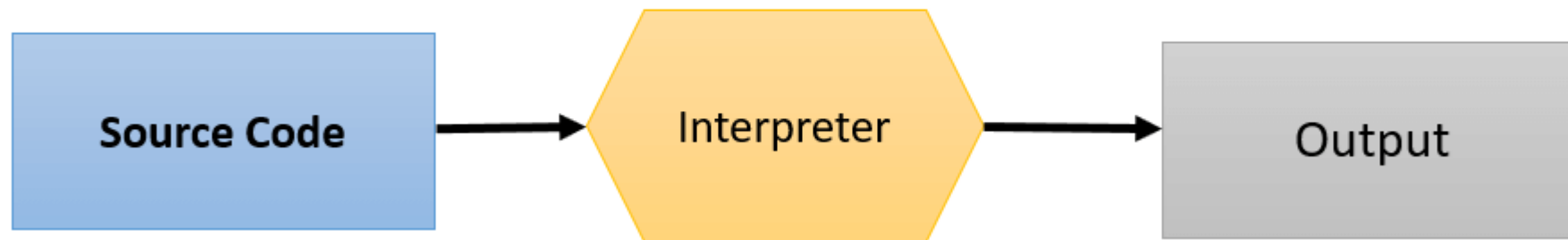
How Compiler Works



linguaggi interpretati

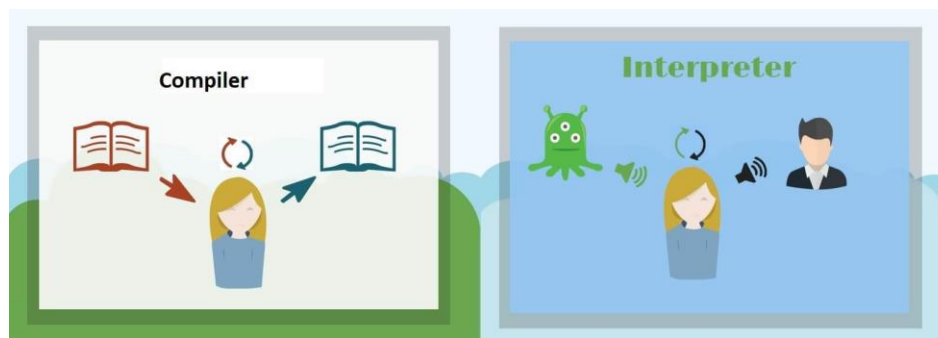
- il programma viene ***eseguito*** direttamente da un software (***interprete***) che esegue le istruzioni in codice macchina necessarie per le funzionalità richieste
- l'interprete simula il funzionamento di un processore
- esempi di linguaggi interpretati: ***Visual Basic, JavaScript***

How Interpreter Works



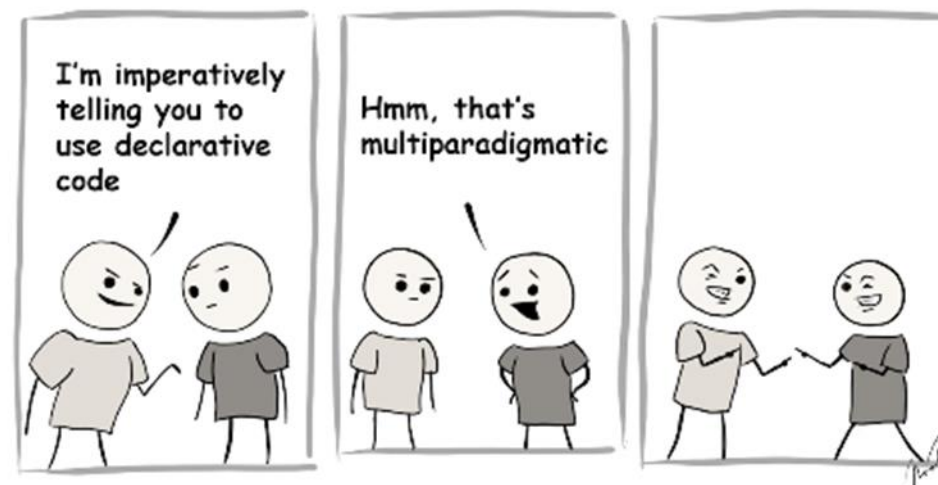
confronto

- i linguaggi interpretati sono generalmente ***più lenti*** dei linguaggi compilati
- i linguaggi interpretati offrono come vantaggio la ***rapidità di sviluppo***
- un programma interpretato è eseguibile immediatamente per essere provato dal programmatore



paradigma di programmazione

- il paradigma di programmazione definisce il modo in cui il programmatore concepisce il programma
- i vari paradigmi si differenziano
 - per le **astrazioni** usate per rappresentare gli elementi di un programma (funzioni, oggetti, variabili ...)
 - per i **procedimenti** usati per l'elaborazione dei dati (assegnamento, iterazione, gestione del flusso dei dati ...)



paradigmi di programmazione

- paradigma ***imperativo***
 - programmazione procedurale ('60)
 - programmazione strutturata ('60-'70)
- programmazione orientata agli ***eventi***
 - *interfacce grafiche*
- programmazione ***logica***
 - *intelligenza artificiale*
- programmazione ***funzionale***
 - *applicazioni matematiche e scientifiche*
- programmazione ***orientata agli oggetti***

paradigma imperativo

- il programma viene inteso come un insieme di istruzioni (direttive o **comandi**) ogni istruzione è un "**ordine**" che viene impartito al computer
- linguaggi imperativi:
 - *Cobol, C, Basic, Pascal, Fortran...*

```
utente = input("Come ti chiami?")  
if utente == "Alberto"  
    print("buongiorno prof")  
else  
    print("ciao", utente)
```

programmazione procedurale

- paradigma basato su **funzioni** (*procedure*) che contengono blocchi di programma riutilizzabili
- le funzioni possono avviare altre funzioni o riavviare se stesse
- la scomposizione in funzioni agevola lo sviluppo, il collaudo e la gestione dei programmi
- ogni funzione risolve un problema specifico ogni volta che si presenta nel programma
 - può essere scritta una volta sola e riutilizzata più volte
- la maggior parte dei linguaggi più diffusi consentono la programmazione procedurale

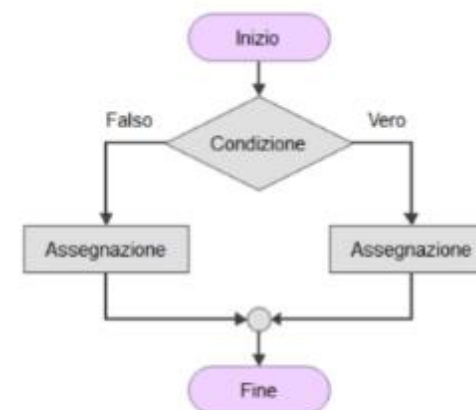
programmazione strutturata

- strutture di controllo:

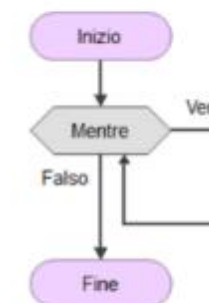
- *sequenza*
- *selezione*
- *iterazione*



SEQUENZA



SELEZIONE



ITERAZIONE

*Qualunque algoritmo può essere implementato utilizzando queste tre sole strutture
 (Teorema di Böhm-Jacopini, 1966)*

programmazione orientata agli eventi

- in un programma **tradizionale** l'esecuzione delle istruzioni segue **percorsi fissi**, che si ramificano soltanto in punti predefiniti dal programmatore
- nella **programmazione orientata agli eventi** il flusso del programma è determinato dal verificarsi di **eventi esterni**
 - il programma attende che accadano determinati eventi e, quando ciò avviene, avvia la sequenza specificata
- gli eventi possono essere azioni dell'utente, input proveniente da un sensore, messaggi di altri sistemi informatici ...
- linguaggi orientati agli eventi:
 - JavaScript, Scratch ...

```
<input type="button" value="Clicca qui!"  
onClick="showMessage();">
```

programmazione logica

- la ***programmazione logica*** adotta la logica del primo ordine
 - per rappresentare
 - per elaborare l'informazione
- richiede al programmatore di ***descrivere*** la struttura logica del problema piuttosto che il modo di ***risolverlo***
- linguaggi: ProLog ...

esempio prolog

```
padre(pippo,gino) .
padre(luca,pippo) .
padre(francesco,luca) .
padre(pippo,manuela) .
padre(luca,genoveffa) .
padre(luca,giuseppina) .
```

```
madre(giulia,gino) .
madre(lina,pippo) .
madre(gelda,luca) .
madre(giulia,manuela) .
madre(lina,genoveffa) .
madre(franca,giuseppina) .
```

```
genitore(X,Y) :- padre(X,Y) .
genitore(X,Y) :- madre(X,Y) .
```

```
antenato(X,Y) :- genitore(X,Y) .
antenato(X,Y) :- genitore(X,Z) , antenato(Z,Y) .
```

```
fratello_sorella(X,Y) :-
padre(Z,X) , padre(Z,Y) , madre(W,X) , madre(W,Y) , X \= Y.
fratellastro_sorellastra(X,Y) :-
genitore(W,X) , genitore(W,Y) , X \= Y.
zio_zia(X,Y) :- genitore(Z,Y) , fratello_sorella(X,Z) .
```

```
?- madre(X,gino) .
X = giulia ;
```

```
?- fratello_sorella(X,Y) .
```

```
X = gino
Y = manuela ;
```

```
X = pippo
Y = genoveffa ;
```

```
X = manuela
Y = gino ;
```

```
X = genoveffa
Y = pippo ;
```

programmazione funzionale

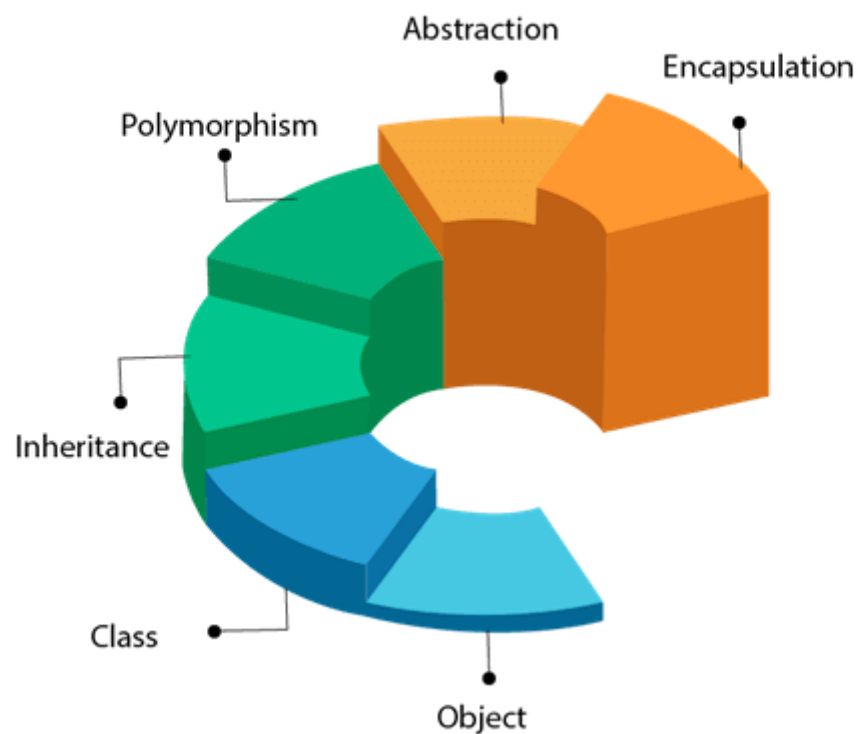
- nella ***programmazione funzionale*** il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche
- principale vantaggio:
 - mancanza di effetti collaterali (side-effect)
 - più facile verifica della correttezza e eliminazione di bug del programma
 - facilita la programmazione parallela
- linguaggi: LISP, Logo, Haskell ...

esempio Haskell

```
def create_pow(exponent: float):  
    """  
    Create and return a function, which calculates a power.  
    """  
    def result(base: float):  
        return base ** exponent  
    return result  
  
root = create_pow(0.5)  
cube = create_pow(3)  
  
print(root(3))  
print(root(4))  
  
print(cube(3))  
print(cube(4))
```

programmazione orientata agli oggetti

OOPs (Object-Oriented Programming System)



programmazione orientata agli oggetti

- la *programmazione orientata agli oggetti* (**OOP**, **O**bject **O**riented **P**rogramming) permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi



influenze fra linguaggi

- <https://exploring-data.com/vis/programming-languages-influence-network/>

