Appunti di Fondamenti di Informatica

Giacomo Simonetto

Primo semetre 2023-24

Sommario

Appunti del corso di Fondamenti di Informatica della facoltà di Ingegneria Informatica dell'Università di Padova.

Indice

1 Storia dell'informatica

fine 1800	Si hanno i primi tentativi di **ricerca di un linguaggio formale**. La matematica è un sistema formale completo? Esiste un procedimento meccanico (passo-passo, finito) per dimostrare se una proposizione sia vera o falsa? Il primo tentavo di *"formalizzazione della matematica"* viene svolto da David Hilbert, con cui si scopre che la matematica possiede 23 problemi di formalizzazione chiamati *"23 problemi di Hilbert"*. La risposta alla prima domanda risale al 1931 quando Goedel, con il *"teorema di incompletezza"* conferma che la matematica non è un sistema formale.
1936	Church, Turing e Kleene elaborano dei formalismi meccanici tra cui la **Macchina di Turing** e la **Tesi di Church-Turing** che sostiene che tutto ciò che è computabile è computabile dalla macchina di Turing universale. La capacità computazionale tra una macchina di Turing e un computer odierno è la stessa (ecceto per il fatto che la macchina di Turing prevedeva uno spazio di archiviazione illimitato), cambia solo la velocità computazionale. Entrambe le macchine risolvono gli stessi problemi, ovvero tutti quelli che si possono risolvere con un algoritmo.
1943	Si arriva a costruire l'**ENIAC**, il primo computer (general purpose) della storia. Si programmava esclusivamente in binario, i circuiti si basavano sulle valvole termoioniche e occupava un palazzo di 5 piani.
1948	Walter Brattain, John Bardeen e William Shockley creano il primo **transistor** (MOSFET) in grado di sommare due bit. Grazie a ciò, durante gli anni '50 si riesce a ridurre le dimensioni dei computer a un piano.
1958	John Backus della IBM sviluppa il primo linguaggio di programmazione di alto livello **Fortran** per programmare uno dei computer sviluppati dall'IBM. Le novità erano quelle di poter programmare in un linguaggio simile all'inglese e l'introduzione delle selezioni e dei cicli.
1966	Viene formulato il **Teorema di Jacopini-Bohm**: qualsiasi algoritmo è implementabile utilizzando le strutture fondamentali di sequenza, selezione e ripetizione. In alrre parole ha senso investire nell'informatica come strumento per risolvere problemi di tipo algoritmico.
1969	Viene inventato l'**Internet** (a carattere).
1970-71	Niklaus Wirth inventa il **PASCAL**, il primo linguaggio strutturato in cui scompaiono il go-to, ma a differenza dei precedenti, non può essere usato per scrivere sistemi operativi.
1970-71	Federico Faggin sviluppa il **primo microprocessore**.
1973	Dennis Ritchie inventa il linguaggio **C**, simile al Pascal, ma con la possibilità di impiegarlo per sviluppare sistemi operativi.
1977	Steve Jobs e la Apple inventano il **primo personal computer**.
1979	Bjarne Stroustrup sviluppa il **C++**, ovvero il C con il paradigma a oggetti.
1979	Come risposta alla Apple, la IBM crea il suo primo PC. Non credendo nei PC, non volendo perdere tempo e non avendo un proprio sistema operativo, la IBM si rivolge alla Microsoft (nata nel 1974) chiedendole di sviluppare un sistema operativo per microprocessori. La Microsoft sviluppa **MS-DOS** (Microsoft Disk Operating System) chiedendo 50 euro per copia (praticamente nulla). Dopo 6 anni vengono vendute 300 milioni di copie e il ricavato viene investito per sviluppare Windows.
1991	Nasce **internet a interfaccia grafica** ed insieme ad esso c'è la necessità di avere programmi in grado di girare indipendentemente dal sistema operativo (Win, Mac OS, Unix). Si sviluppano le prime **Virtual Machine** in grado di e codici in grado di girare su qualsiasi macchina, grazie alle Virtual Machine.
1994	Linus Torvald pubblica la prima versione stabile del kernel **Linux** (creato nel 1991).
1995	James Gosling nella Sun Microsystems sviluppa il linguaggio **Java**, dotato della particolarità di generare un codice compilato in grado di essere eseguito su qualsiasi macchina grazie alla Java Virtual Machine.

2 Computer

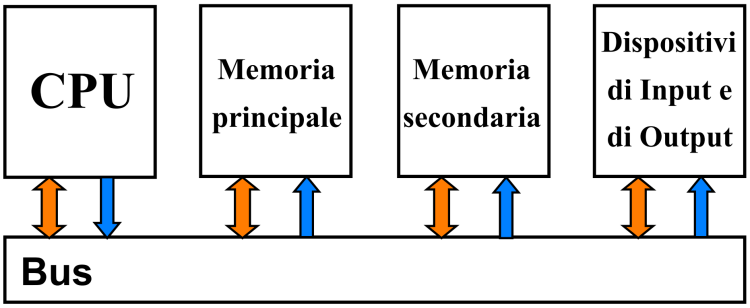
Per computer, o calcolatore, si intende un sistema di elaborazione e memorizzazione di informazioni che opera sotto il controllo di un programma. È composto da hardware (parte fisica) e software (programmi e dati). I dati possono essre di diverso tipo (immagini, testi, audio, video, ...) e sono rappresentati elettricamente in 0 e 1.

Esistono diversi tipi di computer (workstation, smartphone, ...) che possono svolgere diversi tipi di impieghi (elettrodomestici, giochi, fotografie, ...).

3 Modello di Von Neumann

Il modello di Von Neumann è una rappresentazione dell'archietttura di un elaboratore. Prevede la presenza di 4 blocchi: la CPU, la memoria primaria, la memoria secondaria e i dispositivi di I/O collegati insieme grazie al BUS.

Inoltre sono presenti due diversi flussi di informazioni: quello di dati è bidirezionale (nello schema è rappresetato dalle frecce arancioni), mentre quello degli indirizzi e dei segnali di controllo è unidirezionale con direzione CPU → altri blocchi (nello schema è rappresentato dalle frecce blu).



3.1 Central Processing Unit o CPU

La Central Processing Unit ha il compito di:

- individuare ed eseguire le istruzioni
- elaborare dati attraverso la ALU (Unità Logico Aritmetica)
- reperire dati di input e restituire dati di output

Componenti

È costituita da tre blocchi:

Control Unit	o *CU*, gestisce l'esecuzione dei programmi e i flussi di dati
ALU	o *Arithmetic Logical Unit*, elabora le espressioni logiche e algebriche
Registri	memorie temporanee per dati che devono essere subito elaborati: - l'**Accumulator**, o *ACC*, che memorizza i dati elaborati o che stanno per essere elaborati dalla *ALU* - il **Program Counter**, o *PC*, che memorizza l'indirizzo di memoria dell'istruzione successiva da eseguire - l'**Instruction Register**, o *IR*, che memorizza l'istruzione da decodificare - il **Memory Data Register**, o *MDR*, che memorizza i dati/le istruzioni lette o che stanno per essere scritte nella memoria primaria - il **Memory Adress Register**, o *MAR*, che memorizza l'indirizzo di memoria dell'istruzione da eseguire o del dato da utilizzare

Funzionamento

La CPU ha funzionamento ciclico che si divide in tre fasi. La velocità di una CPU, chiamata frequenza di clock è espressa in cicli al secondo (dell'ordine dei GHz) ed è scandita dal *Clock*. La velocità massima è dovuta ai limiti fisici della tecnologia disponibile.

1° fase	**fetch**	- viene letto l'indirizzo dell'istruzione da eseguire dal PC e viene salvato nel MAR - viene incrementato il PC in modo che punti all'istruzione successiva - viene letta e caricata l'istruzione prima nel MDR poi nell'IR
2° fase	**decode**	- la CU decodifica l'istruzione salvata nell'IR - se necessario viene caricato nel MAR l'indirizzo del dato da elaborare o della posizione in cui scrivere il dato elaborato
3° fase	**execute**	- viene eseguita l'istruzione: - se necessario viene caricato nel MDR il dato referenziato dal MAR - il dato può essere salvato nell'ACC o impiegato in un'operazione logico-algebrica eseguita dalla ALU - il risultato viene salvato nell'ACC - oppure il dato memorizzato nell'ACC viene scritto nell'indirizzo di memoria contenuto nel MAR

Limiti e parallelismo

I limiti della CPU sono principalmente due: la frequenza di clock e l'impossibilità di eseguire un'istruzione, finché non viene completata quella precedente. Per superare il secondo problema si sono cercate soluzioni come il parallelismo. Esistono due tipi di parallelismo:

- **parallelismo a livello di istruzioni**:
  detto anche pipeline o multiscalari, consiste nel suddividere il ciclo di un processore in 5 stadi (lettura, decodifica, recupero operandi, caricamento, esecuzione, invio risultati) e di eseguire contemporaneamente più istruzioni su stadi diversi. In questo modo non è necessario aspettare che l'esecuzione dell'istruzione precedente termini per iniziare quella della successiva, ma è sufficiente che sia completato il primo stadio.
- **parallelismo a livello di processori**:
  consiste nell'avere più processori che lavorano contemporaneamente in grado di eseguire più istruzioni nello stesso momento. In base all'archiettura di distinguono in multiprocessori (se sono presenti più processori che condividono la stessa memoria) o multicomputer (se sono più processori, ciascuno con la propria memoria dedicata, collegati tra loro).

3.2 Memoria primaria e secondaria

La memoria ha il compito di memorizzare dati e programmi, sia in maniera temporanea, che permanente.

Struttura

La memoria è composta da celle chiamate allocazioni di memoria. Ogni allocazione può contenere un preciso numero di bit. Un bit (abbreviazione di Binary Digit) è l'unità minima di dimensione della memoria e corrisponde allo spazio occupato da 0 o 1. Il bit è un sottomultiplo del byte, 1byte = 8bit. Il byte è l'unità minima di accesso singolo alla memoria ed è l'unità base per la misura della dimensione dello spazio di archiviazione.

Memoria primaria

La memoria primaria è la più veloce delle due, ma anche la più costosa. Ne esistono due tipi:

RAM	o *Random Access Memory*, memoria volatile, dotata della caratteristica di avere un tempo di accesso ad una cella indipendente dal luogo in cui essa si trova (tempo di accesso "casuale"). Viene impiegata per salvare dati temporanei derivati dall'esecuzione di programmi.
ROM	o *Read Only Memory*, memoria permanente di sola lettura in cui vengono salvati i programmi necessari all'avvio della macchina, es. BIOS (*Basic Input Output System*)
Cache	o *memoria di località*, memoria estremamente veloce che permette di memorizzare celle di memoria che potenzialmente potrebbero tornare utili nelle future elaborazioni. Esistono due tipi di località: - località temporale: accedere alla stessa cella in tempi vicini - località spaziale: accedere a celle limitrofe

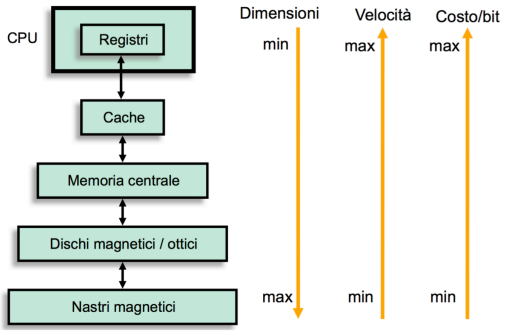
Memoria secondaria

Memoria non volatile, più lenta e molto meno costosa della memoria primaria. È riservata all'archiviazione di file, dati, programmi (tra cui anche il sistema operativo) che vengono trasferiti nella *RAM* al momento dell'esecuzione. Esistono diversi supporti di archiviazione di memoria secondaria:

- *HDD* o disco magnetico
- *SSD* o disco a stato solido (solid state drive)
- dischi ottici come *CD*, *DVD*, *Blue-Ray*
- chiavette USB
- nastri magnetici, impiegati per l'archiviazione di documenti, sono molto lenti, ma hanno costo molto basso ed elevata capacità di archiviazione

Gerarchie di memoria

Maggiore è la dimensione, minore è la velocità ed il costo.



3.3 Dispositivi di I/O

Permettono l'interazione dell'essere umano con la macchina. Comprendono mouse, tastiera, touchpad, schermo, stampante, ... Le operazioni relative ai dispositivi i I/O sono:

polling	o controllo da programma, consiste nel ripetuto e periodico controllo dello stato dei dispositivi
interrupt	richiama l'attenzione della CPU attraverso un'interruzione del flusso di esecuzione
DMA	o *Direct Memory Access*, dispositivo indipendente dalla CPU che gestisce il flusso di dati dei dispositivi di I/O ed alleggerisce il carico della CPU (la CPU indica solo indirizzi e dati da spostare)

4 Sistemi operativi

Cos'è e a cosa serve

Un sistema operativo, abbreviato *SO*, è un insieme di software che fornisce all'utente una serie di comandi e servizi per usufruire della potenza di calcolo di un elaboratore elettronico, inoltre garantisce l'operatività di base di un elaboratore, coordinando e gestendo le risorse hardware di elaborazione e memorizzazione, le periferiche, le risorse/attività software e facendo da interfaccia con l'utente, senza il quale quindi non sarebbe possibile l'utilizzo del computer stesso e dei programmi. Ogni sistema operativo è legato ad uno specifico hardware.

Bootstrap

Il bootstrap è la fase in cui viene avviato il sistema operativo, generalmente all'avvio del computer. La procedura di bootstrap è memorizzata nella *ROM*, all'avvio del computer:

- la CPU legge le istruzioni dalla ROM
- recupera il sistema operativo dal disco (memoria secondaria)
- carica il sistema operativo nella RAM
- avvia l'esecuzione dei programmi che ne permettono il funzionamento

Struttura a cipolla

Il sistema operativo è organizzato su più strati (come una cipolla), ciascuno con la caratteristica di poter interfacciarsi soltanto con quelli più interni. Questo garantisce modularità, flessibilità e più facile manutenzione. Gli strati, dal più interno sono:

nucleo o core	gestisce le risorse fisiche, comunica con l'hardware ed è scritto in linguaggio macchina
gestore I/O	gestisce i dispositivi di input e output e si occupa di trasferire i dati tra le diverse memorie del computer
gestore memoria	gestisce l'allocazione delle memorie durante l'esecuzione dei programmi
gestore archiviazione	anche chiamato filesystem, organizza la struttura si archiviazione dei file
interfaccia utente	permette all'utente di interagire con la macchina attraverso un'interfaccia grafica (GUI) o a linea di comando (CLI)

Comandi e linguaggi di controllo

Ogni sistema operativo possiede un linguaggio di controllo, ovvero un insieme di comandi che permette di interfacciarsi con il sistema operativo, eseguire operazioni o programmi, controllare le attività in corso e lo stato della macchina. I comandi sono impartiti dall'utente attraverso il terminale o attraverso l'interazione con l'interfaccia grafica. In Windows i comandi riprendono il vecchio sistema MS-DOS.

4.1 Unix - Linux

Introduzione

Linux è un sistema operativo sviluppato nel 1994 da Linus Torvald, basandosi su UNIX. Unix è un sistema operativo proprietaro, Linux è la corrispettiva versione di Unix, ma open source.

Utenti e permessi

Da sempre Linux e Unix sono sistemi multiutente, ovvero ciascun file ha un utente proprietario e ogni utente più accedere e modificare solo dove è permesso. L'utente che non ha limitazioni è chiamato `root`.

Filesystem

Il filesystem è organizzato con una struttura ad albero capovolto, in cui la cartella, o *directory*, principale, che contiente tutti i file e le directory del sistema, è chiamata `root`. Ogni elemento nel filesystem è raggiungibile attraverso un percoso chiamato *path*. Il percorso della cartella `root` è `\`.

I path si distinguono in:

- **percorso assoluto**:
  ovvero il percorso che separa la cartella `root` dal file in questione, inizia con `\`, ovvero il simbolo della cartella `root`, cioè con `/`, es. `/user/nomeutente/home/desktop/file.txt`
- **persorso relativo**:
  ovvero il percorso che separa una cartella diversa dalla `root` dal file in questione, inizia con il nome della cartella di partenza, es. `desktop/file.txt` rispetto alla `home`

Il percorso per accedere alla stessa cartella è `./`, quello per accedere alla cartella di livello superiore è `../`

Shell o CLI

La Shell è l'interfaccia utente a linea di comando. In Linux/Unix sono presenti diverse shell: *bash*, *csh*, *ksb*, *zsh*, in base alla distribuzione utilizzata (in Windows è quella di *MS-DOS*).

I comandi della CLI si dividono in *builtin*, che sono presenti di default nell'OS, ed *esterni* che possono essere installati in un secondo momento dall'utente.

In Linux/Unix sono presenti dei metacaratteri come il simbolo `*`, che rappresenta una sequenza di uno o più caratteri, e il simbolo `?`, che rappresenta un singolo carattere.

Alcuni comandi di Linux/Unix sono:

`cd`	change directory
`ls`	list files and subdirectory of the current directory
`cp`	copy file
`mv`	move or rename file
`rm`	remove file
`mkdir`	make new directory
`rmdir`	remove directory
`kill`	end process
`sudo`	per eseguire comandi dall'utente root
`man`	manuale
`appropos`	ricerca comandi
`whatis`	descrizione comando

5 Rappresentazione delle informazioni nei calcolatori

tipo di dato	**rappresentazione**
numeri naturali $\mathbb{N}$	rappresentazione secondo il sistema posizionale
numeri interi $\mathbb{Z}$	rappresentazione modulo - segno rappresentazione in complemento a due
numeri reali $\mathbb{R}$	rappresentazione in virgola fissa rappresentazione in virgola mobile (singola e doppia precisione)
caratteri	tabella ASCII o Unicode

Dato che un computer può elaborare soltanto un numero finito di informazioni, mentre i numeri sono infiniti, significa che ci sarà un valore massimo e un valore minimo rappresentabile. Inoltre viene introdotto un errore dato dal fatto che non tutti i nmeri hanno un numero di cifre limitate (es. π o $\sqrt{2}$).

5.1 Base binaria, decimale, ottale ed esadecimale

La rappresentazione decimale utilizza le cifre da 0 a 9 (10 simboli), quella binaria solo 0 e 1 (2 simboli). Le altre rappresentazioni si comportano allo stesso modo. Nella base ottale vengono impiegate le cifre da 0 a 7 (8 simboli), in quella esadecimale da 0 a F (15 simboli). Verranno approfonditi soltanto i diversi sistemi di rappresentazione binaria e le conversioni con quella decimale.

5.2 ASCII e UNICODE

Il codice ASCII e quello UNICODE sono sistemi che associano ad ogni simbolo (carattere) un numero senza segno. Nello standard ASCII vengono riservati 7 bit per ogni carattere per un massimo di 128 simboli (poi estesi a 256 con l'extended ASCII), mentre per l'UNICODE si impiegano 2 byte per un massimo di 65536 simboli rappresentabili. Il sistema UNICODE, ad oggi il più usato, comprende il vecchio codice ASCII.

5.3 Rappresentazione in sistema posizionale

Si sfrutta il principio che ogni cifra possiede un peso dato dalla posizione relativa nel numero. Nel sistema decimale ogni cifra ha, come peso, una potenza di 10, in quello binario si usano le potenze di 2.

Rappresentazione

DEC	234_{10}	$= 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$
BIN	11101010_2	$= 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$

Valori rappresentabili

Il range di valori rappresentabili con n cifre è $[0, 2^n - 1]$.

Conversione decimale - binario

Per eseguire la conversione dal sistema decimale a quello binario, bisogna usare l'algoritmo di conversione:

```
while (numero != 0)
    resto_nesimo = numero % base
    numero = numero / base
```

Il numero converito si ottiene giustapponendo i resti ottenuti al contrario, in modo che l'ultimo resto diventi la cifra più significativa e il primo resto diventi quella meno significativa.

Conversione binario - decimale

Per convertire un numero dal sistema binario a quello decimale, basta associare ciascuna cifra al suo peso in potenza di 2 e sommare i valori ottenuti.

$$\begin{aligned} 11101010_2 &= 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 1 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 \\ &= 234_{10} \end{aligned}$$

5.4 Rappresentazione in modulo - segno

Con il sistema posizionale non è possibile rappresentare valori negativi, per cui viene introdotta la rappresentazione modulo-segno. Tale sistema prevede di riservare il primo bit al segno del numero ed i restanti per il modulo rappresentato con il sistema decimale.

Rappresentazione

numeri ≥ 0	$+12_{10} = +1 \cdot 1100_2 = 0 + 1100_2 = 01100_{2MS}$
numeri ≤ 0	$-12_{10} = -1 \cdot 1100_2 = 1 + 1100_2 = 11100_{2MS}$

Valori rappresentabili

Il range di valori rappresentabili con n cifre è $\left[-\left(2^{n-1} - 1\right); +\left(2^{n-1} - 1\right)\right]$.
Si osserva che lo 0_{10} possiede due rappresentazioni: 10000_{2MS} e 00000_{2MS} per $n = 4$.

Conversioni

Per le conversioni da sistema decimale a binario e viceversa si ricorre al procedimento illustrato nel sistema posizionale, con l'unica particolarità di far corrispondere uno 0 davanti al numero binario se il numero decimale è positivo, oppure un 1 se il numero decimale è negativo.

Criticità

Questo sistema di rappresentazione non viene utilizzato in quanto l'algoritmo per eseguire somme (e sottrazioni) è poco efficiente e complesso.

5.5 Rappresentazione in complemento a 2

Con la rappresentazione in complemento a 2 è possibile rappresentare numeri interi positivi e negativi, eliminando la doppia rappresentazione dello zero e semplificando l'algoritmo di somma (e differenza), avendo sempre i positivi e lo 0 che inziano per 0 e i negativi che iniziano per 1.

Rappresentazione

numeri ≥ 0	$+12_{10} = 0 + 1100_2 = 01100_{C2}$
numeri < 0	$-12_{10} = 10100_{C2} \quad \rightarrow \quad -12 + 32 = 20_{10} = 10100_2$

Valori rappresentabili

Il range di valori rappresentabili con n cifre è $\left[-2^{n-1}; 2^{n-1} - 1\right]$.

Conversione decimale - binario

Per i numeri positivi, compreso lo 0, si impiega il classico sistema posizionale (aggiungendo uno 0 davanti al numero per il segno), mentre per i numeri negativi è necessario:

1. sommare 2^n con n numero di cifre in binario, in modo da rendere il numero positivo
2. convertire il risultato secondo il sistema posizionale (se i conti sono giusti il numero inizierà per 1)

$$
\begin{aligned}
&+12_{10} = 0 + 1100_2 = 01100_{C2} \\
&-12_{10} \rightarrow -12 + 2^5 = 20_{10} = 10100_2 \\
&-12_{10} = 10100_{C2}
\end{aligned}
$$

In alternativa se il numero è negativo

1. convertire il modulo secondo il sistema posizionale
2. aggiungerci uno 0 davanti
3. invertire le cifre (gli 0 diventano 1 e gli 1 diventano 0)
4. sommarci 1 (se i conti sono giusti il numero inizierà per 1)

$$
\begin{aligned}
&-12_{10} \rightarrow 1100_2 \rightarrow 00011 \rightarrow 10011 \rightarrow 10011 + 1 \rightarrow 10100 \\
&-12_{10} = 10100_{C2}
\end{aligned}
$$

Conversione binario - decimale

Per i numeri che iniziano per 0 basta eseguire la conversione per sistema posizionale, per quelli che inizano con 1 bisogna converire il numero secondo il sistema posizionale e toglierci 2^n con $n =$ cifre del numero.

$$
\begin{aligned}
01100_{C2} &= 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^3 + 0 \cdot 2^1 + 0 \cdot 2^0 \\
&= 8 + 4 \\
&= 12_{10}
\end{aligned}
$$

$$
\begin{aligned}
10100_{C2} &= 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^3 + 0 \cdot 2^1 + 0 \cdot 2^0 \quad - 2^5 \\
&= 16 + 4 \quad - 32 \\
&= 28 - 32 \\
&= -12_{10}
\end{aligned}
$$

5.6 Rappresentazione in virgola fissa

La rappresentazione in virgola fissa riprende il principio del sistema posizionale, usando potenze con esponenti negativi per le cifre dopo la virgola.

Rappresentazione

DEC	234.56_{10}	$= 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 + 5 \cdot 10^-1 + 6 \cdot 10^-2$
BIN	11101010.1001_2	$= 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 +$ $+1 \cdot 2^-1 + 0 \cdot 2^-2 + 0 \cdot 2^-3 + 1 \cdot 2^-3$

Valori rappresentabili

Il range di valori rappresentabili è $\{0\} \cup [2^{-m}, 2^n - 1]$, con n cifre intere, m cifre decimali.

Conversione decimale - binario

Per eseguire la conversione dal sistema decimale a quello binario, è necessario dividere la parte intera da quella dopo la virgola. Per la prima basta convertirla con l'algoritmo visto per il sistema posizionale, mentre per la parte decimale è necessario applicare il seguente "algoritmo":

```
while (numero != 0)
    parteIntera_n-esima = parteIntera(numero)
    numero = parteDecimale(numero) * base
```

Per ottenere il numero convertito è necessario prendere le parti intere in ordine (senza invertirle). Il risultato potrebbe essere un numero illimitato.

Conversione binario - decimale

Per convertire un numero dal sistema binario a quello decimale, basta associare ciascuna cifra al suo peso in potenza di 2 e sommare i valori ottenuti.

$$
\begin{aligned}
11101010.1001_2 &= 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + \\
&\quad + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} \\
&= 1 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 + \\
&\quad + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 1 \cdot \frac{1}{16} \\
&= 128 + 64 + 32 + 8 + 2 + 0.5 + 0.0625 \\
&= 234.5625_{10} \\
&\approx 234.56_{10}
\end{aligned}
$$

Criticità

Non tutti i numeri sono rappresentabili in un numero finito di cifre. Alcuni numeri che nel sistema decimale hanno un numero finito di cifre, nel sistema binario potrebbero essere illimitati, per cui la loro rappresentazione potrebbe essere un'approssimazione, ad esempio $4.35_{10} = 100.010\overline{1100}_2$.
Inoltre è poco efficiente in quando per rappresentare numeri molto grandi, la parte decimale sarebbe poco significativa e i bit riservati a tale parte si potrebbero usare per la parte intera. Viceversa per numeri prossimi allo 0.

5.7 Rappresentazione in virgola mobile

Utilizza la notazione esponenziale con mantissa ed esponente che permette maggiore flessibilità per numeri molto grandi e numeri prossimi allo 0. Lo standard *IEEE 754* prevede due rappresentazioni: a singola e a doppia precisione.

Rappresentazione

DEC	$234,56_{10}$	$= 0,23456 \cdot 10^4$
BIN	$11101010,1001_2$	$= 0,111010101001 \cdot 2^8$

In Java un numero di virgola mobile è rappresentato nella forma $s \cdot m \cdot 2^{(e-N+1)}$ con:

- s segno, può assumere soltanto i valori $-1, +1$
- m mantissa nell'intervallo $\left[0, 2^N - 1\right]$
- e esponente nell'intervallo $\left[-(2^{K-1} - 2), 2^{K-1} - 1\right]$
- N numero di bit riservati alla base
- K numero di bit riservati all'esponente

Suddivisione in bit

precisione	segno	mantissa	esponente	complessivo
singola	"1 bit"	24 bit	8 bit	32 bit
doppia	"1 bit"	53 bit	11 bit	32 bit

Densità

La densità nel sistema in virgola mobile è la distanza tra due numeri adiacenti in formato binario.

Per la singola precisione, avere 23 bit di mantissa, significa che posso suddividere l'intervallo da $0.000\ldots1_2 \cdot 2^n, 0.111\ldots1_2 \cdot 2^n$ in 2^{23} parti. Queste parti saranno più fitte per esponenti bassi e meno fitte per esponenti alti. Per cui la densità è data dalla formula $\delta = 2^{-23} \cdot 2^E$, dove E è l'esponente binario dei numeri da rappresentare.

Valori rappresentabili

precisione	mantissa	esponente	minimo	massimo	densità
singola	$\left[0, 2^{24} - 1\right]$	$[-126, 127]$	$1.4 \cdot 10^{-45}$	$3.4 \cdot 10^{38}$	$\delta = 2^{-23} \cdot 2^E \approx 1.2 \cdot 10^{-7} \cdot 2^E$
doppia	$\left[0, 2^{53} - 1\right]$	$[-1022, 1023]$	$4.9 \cdot 10^{-324}$	$1.8 \cdot 10^{308}$	$\delta = 2^{-52} \cdot 2^E \approx 2.2 \cdot 10^{-16} \cdot 2^E$

Valori limite

Alcune combinazioni di mantissa ed esponente sono catalogate per valori particolari come:

0: esponente minimo, mantissa 0

∞: esponente massimo, mantissa 0

NaN: esponente massimo, mantissa 1 (*Not A Number*)

Criticità

Non tutti i numeri sono rappresentabili con un numero finito di cifre, ma vengono introdotte approssimazioni, come nel caso di 4.35 (come visto precedentemente). Per questo motivo, quando si fanno confronti tra valori in virgola mobile è necessario introdurre un errore entro cui due numeri sono uguali.

Inoltre nel caso in cui si lavora con numeri molto grandi a cui vengono sommati numeri molto piccoli, si rischia di non avere abbastanza precisione per eseguire correttamente la somma, rischiando di perdere il valore dell'addendo minore. Questo avviene soprattutto quando l'addendo più piccolo è minore della precisione del numero più grande.

6 Introduzione alla programmazione

6.1 Algoritmo

Un algoritmo è un metodo di risoluzione di un problema che:

- deve essere eseguibile
- non deve essere ambiguo
- deve concludersi in un numero finito di passi

6.2 Computational Thinking

Per computational thinking, o pensiero computazionale, si intende l'insieme delle abilità che permettono di astrarre il problema e tradurlo in algoritmo. Comprende le tecniche di astrazione/risoluzione di problemi algoritmici tra cui la decomposizione di problemi complessi e la modularità.

6.3 Programmazione

Per programmazione si intende il processo di progettazione e codifica di programmi per calcolatori.
Per programma si intende l'implementazione di algoritmi in un qualche linguaggio per calcolatore.
Un programma è corretto se possiede correttezza semantica, correttezza sintattica e correttezza logica.
Il teorema di Jacopini-Bohm sostiene che un programma/algoritmo è composto da istruzioni imperative, istruzioni condizionali, o decisioni, e iterazioni, o cicli.

6.4 Linguaggi di programmazione

Per linguaggio di programmazione si intende un sistema per la scrittura di programmi/algortimi eseguibili da un calcolatore. Deve essere rigoroso e non avere ambiguità.

Linguaggio macchina

Il linguaggio macchina è il primo esempio di linguaggio di programmazione, è diverso per ogni tipo di processore e le istruzioni sono composte da sequenze di bit. Le istruzioni si dividono in tre categorie:

- trasferimento dati (es. `LOAD` per registri, `STORE` per memorie)
- operazioni aritmetiche e logiche (es. `ADD`, `SUB`, `MUL`, `DIV`, `AND`, `OR`, `NOT`)
- salti (es. `JUMP`, `JZ` se `== 0`, `JGZ` se `> 0`)

Linguaggi assembly e di basso livello

Permettono di scrivrere codice macchina attraverso codici mnemonici più simili al linguaggio umano, per cui più semplice da utilizzare. C'è una corrispondenza biunivoca tra l'instruction set del processore e il linguaggio assembly (o basso livello), per cui è diverso per ogni processore. Le istruzioni sono convertite in linguaggio macchina da un assembler. Rimane sempre il problema della portabilità.

Linguaggi di alto livello

Sono linguaggi formali (artificiali), espressivi, non ambigui, efficienti e molto più leggibili e versatili dei precedenti in quanto simili al linguaggio parlato. Posseggono precise regole grammaticali (lessicali, sintattiche e semantiche) definite dalla notazione EBNF (Extended Backus-Naur form). Sono convertiti in linguaggio macchina da un compilatore. Si dividono in:

- **linguaggi compilati**:
  il programma (indipendente dalla CPU) è convertito in codice eseguibile (dipendente dalla CPU) da un compilatore, garantisce alta velocità, ma bassa portabilità
- **linguaggi interpretati**:
  le istruzioni tradotte in linguaggio macchina da un interprete durante l'esecuzione, garantisce alta portabilità, ma lenta velocità di esecuzione

Breve storia sui linguaggi di programmazione

anni 50	primi linguaggi di programmazione ad alto livello (FORTRAN, BASIC, COBOL), viene coniato il termine *spaghetti code* per l'inappropriato uso delle istruzioni di salto (`GOTO`)
anni 60-70	programmazione strutturata con istruzioni imperative, selezioni e iterazioni, come previsto dal teorema di Jacopini-Bohm (1966), compaiono nuovi linguaggi strutturati PASCAL (1968) e C (1970-75)
anni 80-90	si sviluppa il paradigma di programmazione ad oggetti con nuovi linguaggi come C++ (1979) e Java (1991) in grado di supportare la creazione di classi e oggetti

7 Java

7.1 Introduzione

`Java` è un linguaggio di programmazione ad oggetti, sviluppato da James Gosling nel 1995. La sua particolarità è di essere un linguaggio fortemente tipizzato come il C o C++, ma più semplice e flessibile in quanto la gestione della memoria non deve essere fatta dal programmatore, ma dal Garbage Collector e presenta librerie standard molto ricche.

Non è né un linguaggio interpretato, né un linguaggio compilato. Un file con un programma scritto in java si presenta con l'estensione `.java`, viene pseudocompilato da un compilatore java e convertito in un file con estensione `.bytecode`. Il file bytecode può essere eseguito da un computer con installata la Java Virtual Machine, o JVM, ovvero l'ambiente di esecuzione che funge da interprete del file bytecode.

Per questo motivo è più veloce di un linguaggio interpretato, ma meno veloce di uno compilato e meno portabile di uno interpretato, ma più portabile di uno compilato.

Per compilare un file java si utilizza il comando `javac NomeFile.java`, per eseguire un file bytecode si usa `java NomeFile`

7.2 Struttura di un programma

Un programma in java è composto da diversi elementi (introduzione):

classi	contenitori di metodi e variabili, possono essere eseguibili se contengono un metodo `main`, come fabbriche di oggetti o entrambe (anche se poco raccomandabile); le classi sono organizzabili in pacchetti e sono importabili in file diversi da quello in cui sono implementate attraverso la parola chiave `import`
metodi	sequenze di istruzioni, chiamati anche funzioni o procedure in altri linguaggi; devono necessariamente essere contenuti in una classe, il metodo principale `main` viene eseguito all'avvio del programma
variabili	spazi di memoria riservati per memorizzare dati, ogni variabile può contenere solo un particolare tipo di dato es. `int`, `double`, `char`, ... il valore di una variabile può essere modificato nel corso dell'esecuzione del programma
costanti	spazi di memoria riservati per memorizzare valori che rimangono costanti per tutta la durata del programma, come le variabili, anche le costanti sono tipizzate
literals	insieme di valori numerici e stringhe di testo che sono contenuti nel programma
oggetti	istanze di una classe, sono variabili particolari in grado di avere dei metodi in grado di compiere operazioni su di esse, es. stringhe, array, strutture dati
commenti	parti di testo che vengono ignorate dal compilatore al momento della compilazione, si utilizzano per descrivere la funzione di una determinata parte di codice
strutture logiche	insieme di costrutti che permettono di compiere selezioni e iterazioni, es. `if`, `else`, `while`, `for`, ...
parole chiave	insieme di parole riservate con una funzione logica ben precisa, es. le strutture logiche, tipi delle variabili, `import`, `final`, `private`, `public`, `new` ...

7.3 Variabili

Una variabile è uno spazio di memoria riservato per memorizzare dati (numerici, logici o caratteri) che possono variare durante l'esecuzione del programma. Una variabile può contenere solo dati di specifico tipo indicato nella dichiarazione.

Dichiarazione, inizializzazione e assegnazione

- **dichiarazione**: `int a;`
  viene riservato uno spazio in memoria in grado di contenere valori del tipo indicato al posto di `tipo`
- **inizializzazione**: `int a = 0;`
  viene dichiara una variabile e le viene assegnato un valore inziale
- **assegnazione**: `a = 345;`
  viene salvato un valore in una variabile

Visibilità

Le variabili vengono create al momento della loro dichiarazione e vengono distrutte alla chiusura del blocco in cui sono state create. Inoltre sono accessibili solo all'intendo del blocco (e sottoblocchi) in cui sono state dichiarate.

Tipi di dato

tipo	min	max	mem	descrizione
byte	-128	127	8 bit	numeri interi in complemento a due
short	-32 768	32 767	16 bit	numeri interi in complemento a due
int	$-2.15 \cdot 10^9$	$2.15 \cdot 10^9$	32 bit	numeri interi in complemento a due
long	$-9.22 \cdot 10^{18}$	$9.22 \cdot 10^{18}$	64 bit	numeri interi in complemento a due
float	$1.40 \cdot 10^{-45}$	$3.40 \cdot 10^{38}$	32 bit	numeri decimali in singola precisione
double	$4.94 \cdot 10^{-324}$	$1.80 \cdot 10^{308}$	64 bit	numeri decimali in doppia precisione
char	0	65535	16 bit	numeri interi ≥ 0 per standard UNICODE
boolean	false	true	1 bit	valori booleani true, false

Promozione e casting

Per convertire un valore di un determinato tipo di dato in un altro, esistono due operazioni:

- **promozione**: `double a = 123`
  sempre concesso e non genera errori, avviene quando si converte un numero con minore precisione in un formato con maggiore precisione
- **casting**: `int a = (int)123,45`
  se non correttamente gestito può generare errori di perdita di precisione o overflow, avviene quando si converte un numero con maggiore precisione in un formato con minore precisione, per eseguire il casting è necessario indicare tra parentesi il nuovo tipo di dato, davanti al valore da convertire

Oveflow

L'overflow avviene quando si supera il valore massimo (o minimo) consentito da un determinato tipo di dato. Nei numeri interi, i valori hanno comportamento "ciclico" quando si supera il valore superiore, si ricomincia dal valore inferiore e viceversa.

Operazioni

somma	`a + b` `a++`	viene eseguita la somma (possibile overflow) al variabile viene incrementata di uno
differenza	`a - b` `a--`	viene eseguita la differenza la variabile viene decrementata di uno
prodotto	`a * b`	viene eseguito il prodotto
divisione	`a / b`	viene eseguita la divisione, se entrambe le variabili sono di tipo int viene eseguita la divisione intera (tralasciando la parte decimale), se almeno una delle due è double, allora viene eseguita la divisione reale, se b è 0 e le variabili sono intere, viene lanciata una `ArithmeticException`
modulo	`a % b`	calcola il resto della divisione tra a e b, se b è 0 e le variabili sono intere, viene lanciata una `ArithmeticException`

confronti	`a == b`
	`a < b` `a <= b`
	`a > b` `a >= b`
		confronta il valore contenuto nelle due variabili

and	`a && b`	restituisce `true` se e solo se entrambi i membri sono `true`
or	`a || b`	restituisce `true` se almeno uno dei due membri è `true`
not	`!a`	restituisce `true` se `a` è `false` e viceversa

Precedenze

In ordine di precedenza:

1. operatori unari: di incremento e decremento, not logico, byteshift, ...
2. prodotto, divisione e modulo
3. somma e sottrazione
4. confronti con uguaglianza
5. operatore logico and
6. operatore logico or
7. assegnazioni

Costanti

Le costanti sono un tipo particlare di variabili che contengono un valore fisso per tutta la durata dell'esecuzione. il valore è assegnato al momento della dichiarazione. In genere le costanti si indicano con lettere maiuscole. `final double PI = 3,14...`

Literals

I Literals sono l'insieme di numeri e stringhe che vengono utilizzate nel programma e che non sono contenuti in una variabile. Ad esempio i valori di inizializzazione delle variabili o i messaggi di testo da mandare in output.

I numeri interi vengono interpretati di tipo `int`, quelli decimali come `double`, i caratteri racchiusi da singoli apici es. `'c'` come `char` e le sequenze di caratteri racchiuse tra doppi apici `"text"` come oggetti della classe String.

7.4 Metodi

I metodi sono blocchi di codice racchiusi da parentesi graffe che contengono istruzioni che risolvono un determinato problema. Un programma eseguibile deve necessariamente avere un metodo main definito come `public static void main(String[] args) { }`.

Struttura

```
tipoRestituito nomeMetodo (parametriFormali) {
    istruzioni;
    return valoreRestituito;
}
```

La prima riga del metodo viene chiamata firma.

Un metodo può ricevere in input dei parametri espliciti indicati tra le parentesi tonde e restituire in output un valore di ritorno il cui tipo è specicificato prima del nome del metodo e il valore viene indicato alla fine del metodo dopo la parola return. Se un metodo non restituisce nessun valore si utilizza `void` al posto del tipoRestituito e il return viene omesso o indicato senza specificare il valoreRestituito.

Overloading

Due metodi possono avere lo stesso nome, ma avere parametri differenti (per tipo e per numero). Quando vengono richiamati, il compilatore riconosce il metodo corretto da utilizzare in base alla corrispondenza del tipo e del numero di parametri tra la chiamata e la firma del metodo.

7.5 Classi e oggetti

Una classe è un contenitore di metodi e variabili. Sono alla base del paradigma di programmazione ad oggetti (OOP) e permettono un maggiore livello di astrazione per risolvere problemi complessi più facilmente.

Ogni file Java deve contenere una classe pubblica con lo stesso nome del file. Questa classe può essere eseguibile se ha un metodo main, altrimenti viene detta non eseguibile.

Una classe può essere istanziabile, ovvero è possibile crearne delle copie (più o meno indipendenti) che prendono il nome di oggetti o istanze della classe.

Struttura

Una classe è composta da:

variabili d'istanza	memorizzano le informazioni e lo stato di un oggetto
metodi costruttori	contengono le istruzioni da eseguire alla creazione dell'oggetto, es. inizializzazione delle variabili
metodi di accesso	servono per accedere alle variabili d'istanza, senza poterle modificare
metodi modificatori	servono per modificare le variabili d'istanza secondo dei precisi criteri

Incapsulamento

Per evitare che l'oggetto venga messo in uno stato non valido (es. lati di una figura negativi), si sceglie di nascondere le variabili d'istanza, riducendone l'accessibilità, e creando dei metodi di accesso e di modifica. Questo principio di "nascondere" le variabili è chiamato incapsulamento e prevede i seguenti vantaggi:

- impedire che l'oggetto venga messo in uno stato inconsistente
- l'utilizzatore non deve preoccuparsi dei dettagli implementativi
- se si modifica l'implementazione di un metodo, non serve modificarne i programmi in cui è impiegato
- se c'è un errore nelle variabili d'istanza, va cercato nei metodi della classse o nelle loro invocazioni

Accessibilità

public	accessibile da qualsiasi classe
package	(default) accessibile da qualsiasi classe contenuta nello stesso pacchetto
protected	accessibile da qualsiasi sottoclasse della classe in cui è creato
private	accessibile solo dalla classe in cui è stato creato

Variabili statiche e non statiche

static	variabile unica comune a tutte le istanze di una classe, se viene modificata da un'istanza, si modifica per tutte le istanze
non-static	variabile propria di una specifica istanza, indipendente dalla corrispettiva in istanze diverse

Metodi statici e non statici

static	non può accedere alle variabili d'istanza e non ha parametri impliciti, si invoca indicando `NomeClasse.nomeMetodo(parametri espliciti)`
non-static	può accedere alle variabili d'istanza e ha come parametro implicito l'oggetto invocante si invoca indicando `nomeOggetto.nomeMetodo(parametri espliciti)`

Struttura base di una classe generatrice di oggetti

```
public class NomeClasse {
    // variabili d'istanza
    private int variabileIstanza1;
    ...

    // costruttore
    public NomeClasse() {
        variabileIstanza1 = 0;
        ...
    }

    // metodi di accesso
    public int getVariabileIstanza1(){
        return variabileIstanza1;
    }
    ...

    // metodi di modifica
    public void setVariabileIstanza1(int v1) {
        variabileIstanza1 = v1;
    }
    ...

    // metodi statici
    public static int metodoStatico(int a) {
        return a;
    }
    ...
}
```

Instanziamento di un oggetto e richiamo metodi

```
// istanziamento di un oggetto della classe NomeClasse
NomeClasse nomeOggetto = new NomeClasse(parametri);

// invocazione del metodo metodo1 dall'oggetto nomeOggetto
nomeOggetto.metodo1(parametri);

// invocazione del metodo metodoStatico della classe NomeClasse
NomeClasse.metodoStatico(parametri);
```

Riferimenti ad oggetti

Quando si dichiara un nuovo oggetto, si crea uno spazio in memoria in cui vengono salvati i dati dell'oggetto (es. variabili d'istanza e metodi) e viene creata una variabile riferimento che punta alla posizione in memoria dell'oggetto.

Per cui se creo un oggetto `A a = A();` e uno `A b = a;`, le modifiche che faccio ad `a` si riflettono anche su `b` in quanto entrambi `a` e `b` puntano alla stessa posizione in memoria, per cui allo stesso oggetto.

Quando viene passato un oggetto ad un metodo come parametro esplicito, le modifiche che vengono fatte all'oggetto all'interno del metodo, si riscontrano anche sull'oggetto nel metodo chiamante.

Riferimento this

In una classe, il riferimento all'istanza della classe è contenuto nel parametro `this`. Quando viene invocato un metodo non statico su un oggetto `nomeOggetto.nomeMetodo(...)`, il riferimento dell'oggetto invocante, chiamato parametro implicito, è contenuto nella parola chiave `this`.

Quando si deve chiamare un metodo costruttore da un altro costruttore (dello stesso oggetto), si utilizza `this(...)`, in cui nelle tonde sono specificati i parametri richiesti dal costruttore.

Quando avviene un "overloading" di variabili e una variabile locale del metodo nasconde quella della classe (perché hanno lo stesso nome), per utilizzare quella della classe, bisogna fare `this.nomeVariabile`.

Ereditarietà

Tra diverse classi ci può essere un rapporto di madre-figlia (superclasse-sottoclasse), quando una classe (la sottoclasse) estende un'altra classe (la superclasse) ereditandone tutti i metodi e tutte le variabili d'istanza. Per realizzare una sottoclasse, è necessario aggiungere `extends SuperClasse` nella firma della sottoclasse, dopo il nome della sottoclasse. Una sottoclasse ha un'unica superclasse, non è possibile estendere due classi contemporaneamente.
All'interno dell'implementazione dei metodi della sottoclasse si può:

- chiamare il costruttore della superclasse, tramite il riferimento `super`: `super(parametri);`
- accedere ai metodi non private della superclasse (se ci sono overloading di metodi è necessario utilizzare il riferimento `super` per chiamare quelli della superclasse)
- accedere alle variabili non private della superclasse (se ci sono overloading di variabili è necessario utilizzare il riferimento `super` per indicare quelle della superclasse)
- aggiungere nuovi metodi statici e non ed in caso eseguire l'overloading di metodi della superclasse
- aggiungere nuove variabili, in caso con lo stesso nome di quelle della superclasse

Attraverso l'istanza della sottoclasse (oggetto della sottoclasse) è possibile chiamare i metodi della superclasse solo se non è stato fatto l'overloading. La superclasse non può accedere alle variabili e ai metodi della sottoclasse.

Tutte le classi, in Java, sono sottoclassi della classe `Object`, da cui ereditano i seguenti metodi (riportati solo quelli ritenuti più importanti):

- `protected Object clone()` crea e restituisce una copia dell'oggetto chiamante
- `boolean equals(Object obj)` confronta i riferimenti dell'oggetto chiamante e di quello passato come parametro
- `int hashCode()` restituisce il codice hash dell'oggetto chiamante
- `String toString()` restituisce la rappresentazione in stringa dell'oggetto chiamante

È buona pratica eseguire l'overloading dei metodi ereditati dalla classe Object prima di utilizzarli, in modo da renderli più adatti all'oggetto chiamante (es. modificare toString in modo che non restituisca classe e indirizzo di memoria o modificare equals in modo che non confronti gli indirizzi, ma le variabili d'istanza).

Una classe non estendibile ha attributo `final` prima del nome della classe.

Per sapre se una classe è un'istanza di una specifica classe o di una sua sottoclasse si utilizza la parola chiave `instanceof` (usata nel seguente modo `Classe1 instanceof Classe2`). L'espressione restituisce true se Classe1 è una classe dello stesso tipo di Classe2, o una sua sottoclasse, altrimenti restituisce false.

Casting tra oggetti

È sempre possibile eseguire il casting da una sottoclasse ad una sua superclasse, senza generare errori in compilazione o in esecuzione. Al contrario per eseguire il casting da una superclasse ad una sua sottoclasse è necessario esplicitare il tipo della sottoclasse tra parentesi tonde e, se non è possibile eseguire la conversione, viene lanciata `ClassCastException` durante l'esecuzione. Prima di eseguire la conversione esplicita è raccomandato verficare che sia possibile utilzzando `instanceof`.

Polimorfismo

Quando viene eseguito il casting da una sottoclasse ad una superclasse, il "nuovo" riferimento permette di utilizzare solo i metodi definiti per la superclasse, pena un errore in compilazione.
Se non è stato fatto l'overloading del metodo della superclasse, allora in fase di eseguzione verrà chiamato il metodo della superclasse. Se, invece, è stato eseguito l'overloading del metodo in qualche sottoclasse, in fase di esecuzione verrà chiamato il metodo oveloaded della sottoclasse più specifica. Es. se è stato fatto l'overloading di `toString` e viene chiamato da un oggetto non Object dopo il cast ad Object, verrà chiamato l'overloading del `toString` della classe originaria.

Questo fenomeno è chiamato polimorfismo e deriva dal fatto che quando viene eseguito il casting da sottoclasse a superclasse, il nuovo riferimento del tipo superclasse punta sempre allo stesso oggetto della sottoclasse e, quando viene invocato un metodo della superclasse di cui è stato eseguito l'overloading, l'interprete (in fase di esecuzione) riconosce il tipo della sottoclasse ed invoca il metodo della sottoclasse.

Questo è utile per creare dei metodi che eseguano operazioni generiche su diversi tipi di oggetti. Ad esempio il metodo `println` riceve come parametro un oggetto (es. un String) a cui viene fatto un cast implicito a Object e invoca il metodo `toString`. Se non è stato eseguito l'overloading del metodo, viene chiamato il `toString` definito in Object (non è questo il caso), altrimenti viene chiamato l'overloading della sottoclasse (in questo caso definito in String).

Classi interne

È possibile definire ed implementare una classe (classe interna) all'interno di un'altra classe (classe esterna), per cui la classe interna e quella esterna condividono un rapporto di "fiducia" in quanto possono reciprocamente accedere a tutti i metodi e tutte le variabili, anche se sono stati definiti private.

Non è mai possibile creare nuove istanze della classe interna, se non nella classe esterna o nelle classi che la estendono, invece è sempre concesso restituire riferimenti di istanze di classe interna, indipendentemente se la classe interna sia public o private.

In un metodo al di fuori della classe esterna (es. nel main), se la classe interna è public:

- è possibile chiamare i metodi public della classe interna su riferimenti di classe interna
- è possibile creare riferimenti di classe interna con `ClasseEsterna.ClasseInterna nomeOggetto;`
- i nuovi riferimenti del tipo classe interna possono puntare a null o ad istanze della classe interna create nei metodi della classe esterna o sue sottoclassi

Viceversa se la classe interna è private non è possibile fare nulla (non si possono creare riferimenti e non è possibile chiamare i metodi, anche se questi sono definiti public).

Pacchetti e organizzazione

Più classi che svolgono compiti simili vengono raggiuppatte in pacchetti. Ad esempio il pacchetto `java.lang` racchiude la classe `System`, la classe `String` e la classe `Math`.

Per utilizzare delle classi che si trovano in file diversi in cartelle diverse, bisogna inserire all'inizio del programma (fuori dalle classi) la parola chiave `import` seguita dalla classe che si vuole importare, precisandone il relativo pacchetto. Il pacchetto `java.lang.*` è importato di default su ogni progetto.

Classe java.lang.String

Classe che memorizza una stringa di testo e contiene i metodi per manipolarla. Ogni stringa di testo è un'istanza di `String` e ogni istanza di `String` è immutabile, ovvero non è possibile modificare la stringa memorizzata se non creando una nuova istanza.

Alcuni metodi forniti dalla classe:

`String()`	costruttore vuoto
`char charAt(int)`	restituisce il carattere all'indice passato come parametro
`int compareTo(String)`	restituisce 1 se il parametro esplicito precede quello esplicito, -1 se avviene l'oppposto e 0 se sono uguali
`boolean equals(Object)`	restituisce true se le due stringhe sono uguali, false altrimenti
`boolean equalsIgnoreCase(String)`	analogo al metodo prima, ingnorando maiuscole e minuscole
`int indexOf(char)`	restiuisce l'indice di un dato carattere/stringa nell'intervallo di indici specificato (se specificato)
`int length()`	restituisce la lunghezza della stringa
`String substring(int)`	restituisce la sottostringa dal primo indice (incluso) alla fine
`String substring(int, int)`	restituisce la sottostringa dal primo indice (incluso) al secondo indice (escluso)
`String toLowerCase()`	converte tutte le maiuscole in minuscole
`String toUpperCase()`	converte tutte le minuscole in maiuscole

Classe java.util.Scanner

Classe che permette di facilitare le operazioni di input, output, lettura da file e suddivisione di stringhe in token.

Alcuni metodi forniti dalla classe:

`Scanner(InputStream)`	scanner per leggere da standard input: `new Scanner(System.in)`
`Scanner(String)`	scanner per leggere da una stringa
`Scanner(File)`	scanner per leggere da file
`boolean hasNext...()`	true se ci sono ancora dati da leggere, false altrimenti
`String next...()`	estrae e restituisce un dato dal buffer
`void close()`	metodo per chiudere lo scanner, dopo aver finito di utilizzarlo

Classe java.io.FileReader

Classe per leggere da file, utilizzabile come oggetto della classe `File` per utilizzare lo scanner.

Alcuni metodi forniti dalla classe:

`FileReader(File)`	costruttore che riceve un oggetto file da cui leggere
`FileReader(String)`	costruttore che riceve il nome del file da cui leggere
`void close()`	metodo per chiudere il file di input, dopo aver finito di utilizzarlo

Siccome può lanciare `IOException`, è necessario inserirlo in un try-with-resources (così viene chiuso in automatico). Questa struttura verrà approfondita più avanti.

Classe java.io.PrintWriter

Classe per scrivre su file utilizzando i metodi normalmente impiegati per lo standard output.
Alcuni metodi forniti dalla classe:

`PrintWriter(File)`	costruttore che riceve un oggetto file su cui scrivere
`PrintWriter(String)`	costruttore che riceve il nome del file su cui scrivere
`void print(...)`	metodo che scrive quanto passato come parametro nel file
`void println(...)`	come il metodo precedente, alla fine va a capo
`void close()`	metodo per chiudere il file di input, dopo aver finito di utilizzarlo

Classe java.util.Random

Classe utilizzata per generare numeri pseudocasuali
Alcuni metodi forniti dalla classe:

`Random()`	istanzia un generatore di numeri casuali, il seme è preso dall'orologio del computer
`Random(long)`	istanzia un generatore di numeri casuali, il seme è passato come parametro
`int nextInt()`	restituisce un numero casuale intero nell'intervallo $\left[0, 2^{32}\right)$
`int nextDouble()`	restituisce un numero casuale decimale nell'intervallo $[0, 1)$
`int nextInt(int)`	restituisce un numero casuale nell'intervallo $[0, x)$, con x passato come parametro

Paradigma di programmazione ad oggetti - riassunto

Obietti del OOP

- robustezza: gestire situazioni inattese
- adattabilità: possibilità di evolvere ed implementare nuove funzionalità e di avere portabilità
- riusabilità: creazione di componenti riusabili in altri programmi o applicazioni

Principi del OOP

- astrazione: distillare concetti di un sistema o un oggetto
- information hiding: o incapsulamento, nascondere informazioni critiche per evitare di avere oggetti inconsistenti
- modularità: organizzazione del sistema in componenti funzionali separati

Mezzi per l'implementazione

- classi, oggetti e incapsulamento: per information hiding
- ereditarietà: estensione e specializzazione di una classe
- polimorfismo: l tipo di una variabile non determina completamente il tipo dell'oggetto a cui essa si riferisce

7.6 Interfacce

Le interfacce sono classi astratte non istanziabili e non implementate, che contengono soltanto metodi pubblici (non serve specificare `public`, dato che è implicito). Servono per indicare i metodi che devono avere determinate classi che implementano l'interfaccia.

Una classe, per implementare l'interfaccia, deve contenere `implements NomeInterfaccia` che segue il nome della classe nell'implementazione della classe stessa. Una classe può implementare diverse interfacce.

Struttura base di un'interfaccia

```
public interface NomeInterfaccia {
    // metodi pubblici
    void metodo1(parametri){
        ...
    }

    int metodo2(parametri){
        ...
    }
    ...
}
```

Interfaccia Comparable

L'interfaccia `Comparable`, contenuta in `java.lang`, definisce il metodo `boolean compareTo(Object)` utilizzato per l'ordinamento di oggetti. Gli oggetti di una classe che implementa tale interfaccia sono ordinabili. Alcune classi che implementano `Comparable` sono `String`, `Integer`, `Double`.

7.7 Commenti e JavaDoc

I commenti sono parti di testo che il compilatore ignora. Si usano per descrivere quali operazioni vengono effettuate in una determinata parte del codice. Possono essere su una singola riga `\\ commento` o su più righe `/* commento */`.

Esiste uno standard di commentazione che permette di creare una documentazione nello stesso stile di quella di java, con il comando `javadoc NomeFile.java`.Si prevede un commento su più righe posto prima di un metodo o una classe e in tale commento vengono descritti la funzione della classe, i parametri (preceduti da `@param`), il valore di ritorno (preceduto da `@return`) ed eventuali eccezioni (precedute da `@throws`)) che possono essere lanciate.

Per generare la documentazione basta eseguire il comando `javadoc NomeFile.java` e verranno creati i file necessari per visualizzare la documentazione nello stesso stile di quella di Java.

```
/**
 * Descrizione del metodo
 *
 * @param p1 parametro 1
 * @param p2 parametro 2
 * @return valore che viene restituito
 * @throws Exception eccezione lanciata
 */
public int metodo(int p1, double p2) {
    ...
}
```

7.8 Operazioni logiche

Operazioni logiche

Le operazioni logiche sono particolari operazioni vengono eseguite tra tipi di dato boolean e danno risultato true o false. Le operazioni che restituiscono un boolean sono, ad esempio, le operazioni di confronto o di uguaglianza ($>, \geq, <, \leq, =$) e le operazioni tra dati booleani sono l'and, l'or e il not.

In Java le operazioni si traducono nel seguente modo:

$<$	`<`	$>$	`>`	$=$	`==`	and	`&&`
$\leq$	`<=`	$\geq$	`>=`	$\neq$	`!=`	or	`||`
						not	`!`

Per calcolare i risultati delle operazioni si utilizzano le tabelle di verità, come le seguenti:

a	b	a and b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a or b
true	true	true
true	false	true
false	true	true
false	false	false

a	not a
true	false
false	true

Confronti tra double / float

Dato che nelle rappresentazioni a virgola mobile si introduce un certo errore dato dalla precisione e dalla limitatezza del sistema di rappresentazione, eseguire il confronto risulta essere particolarmente delicato. In Java $2 \neq \sqrt{2}^2$ in quanto $\sqrt{2}^2 = 2.0000000000000004 \neq 2$ e $\sqrt{2}^2 - 2 = 4.440892098500626 \cdot 10^{-16} \neq 0$

Per questo motivo la verifica dell'uguaglianza tra due double deve prevedere un certo errore:

```
a == b   ⇔   Math.abs(a - b) <= Math.max(Math.abs(a), Math.abs(b)) * 1E-14
```

Il valore 10^{-14} deriva dal calcolo della "densità" della rappresentazione double: $2^{-52} \approx 2.2 \cdot 10^{-16} \approx 10^{-14}$

Confronti tra oggetti

Il confronto tra due oggetti usando il simbolo `==`, confronta gli indirizzi, per cui per eseguire un corretto confronto tra oggetti, si utilizza il metodo `boolean equals(Object obj)` (ereditato dalla classe Object) di cui spesso si fa l'overloading per adattarlo alla classe in cui viene utilizzato.

Per ordinare due oggetti si utlizza il metodo `int compareTo(Comparable obj)`, che ogni classe che implementa l'interfaccia `Comparable` deve avere.

De Morgan e doppia negazione

Le leggi di De Morgan e della doppia negazione servono per eseguire semplificazioni in operazioni logiche:

```
!(a and b) = !a or !b     !(a or b) = !a and !b     !(!a) = a
```

Cortocircuito logico

Dato che l'operazione and è vera se e solo se entrambe le condizioni/variabili sono vere, se la prima è falsa, il risultato sarà sicuramente falso indipendentemente dal valore della seconda. Analogamente l'operazione or è vera se e solo se almeno una delle due condiozioni/variabili è vera, per cui se la prima è vera, il risultato sarà vero.

L'elaboratore termina la valutazione dell'espressione logica non appena si ottiene il risultato, per cui nelle situazioni sopra, la seconda condizione/variabile non viene nemmeno controllata

Questo permette di evitare il lancio di `NullPointerException` o `ArithmeticException`, ad esempio `s != null && s.length() < 3` se `s` punta a null, non è necessario eseguire la seconda condizione, che lancerebbe appunto l'eccezione sopra, caso analogo per `n != 0 && 1/n == 0`.

7.9 Selezioni

Le selezioni sono delle strutture logiche composte da una condizione e una serie di istruzioni che vengono eseguite solo se la la condizione è verificata (`if`). Opzionalmente è possibile aggiungere un secondo blocco di istruzioni da eseguire solo se la condizione non è verificata (`else`).

If - else

Struttura logica di selezione semplice

```
if (condizione) {
    // istruzioni eseguite se la condizione e' true
    ...
}
else {
    // istruzioni eseguite se la condizione e' false
    ...
}
```

È possibile annidare più if-else, uno dentro l'altro, creando selezioni annidate. È possibile omettere le parentesi graffe se il blocco contiene una singola istruzione.

L'else fa riferimento sempre all'if immediatamente prima, per cui nel seguente caso, l'istruzione contenuta nell'else è eseguita se a è true e se b è false (è legata al secondo if). Questa situazione è chiamata problema dell'else sospeso.

```
if (a)
    if (b)
        ...
else
    ...
```

If - else if - else

Struttura logica per creare condizioni mutualmente esclusivamente

```
if (condizione) {
    // istruzioni eseguite se la condizione1 e' true
    ...
}
else if( condizione2 ) {
    // istruzioni eseguite se la condizione1 e' false e condizione2 e' true
    ...
}
else {
    // istruzioni eseguite se entrambe le condizioni sono false
    ...
}
```

Switch - case

Struttura logica che permette di eseguire blocchi di istruzioni in base al valore assunto da una variabile.

```
switch (variabile) {
    case a:
        // istruzioni eseguite se variabile == a
        ...
        break;
    case b:
        // istruzioni eseguite se variabile == b
        ...
        break;
    defalut:
        // istruzioni eseguite in tutti gli altri casi
        ...
        break;
}
```

Se viene omesso il `break;` viene eseguito anche il blocco di istruzioni successivo.

7.10 Iterazioni

Le iterazioni sono strutture logiche composte da una condizione (il cui esito varia nel corso dell'esecuzione) e un blocco di istruzioni che vengono eseguite ripetutamente, finché la condizione risulta vera. Se la condizione rimane true, si parla di iterazione infinita ed è fonte di un possibile errore concettuale.

While

Struttura di iterazione semplice, non a conteggio.

```
while (condizione) {
    // istruzioni eseguite finche' condizione e' true
    ...
}
```

Do - while

Struttura di iterazione semplice, non a conteggio, con la particolarità che la prima volta il blocco di istruzioni viene eseguito almeno una volta a prescindere dal risultato della condizione.

```
do {
    // istruzioni eseguite almeno una volta e finche' condizione e' true
    ...
} while (condizione);
```

For

Struttura di iterazione a conteggio che prevede l'inizializzazione (e l'eventuale dichiarazione) di una variabile contatore che viene incrementata di un certo valore (generamente 1). In questo modo si tiene traccia di quante iterazioni sono state svolte.

```
for (dichiarazione e inizializzazione; condizione; incremento) {
    // istruzioni eseguite finche' condizione e' true
    ...
}
```

Break e continue

La parola chiave `break` serve per interrompere un ciclo prima della valutazione della condizione. Se sono presenti due cicli annidati e break è presente sul ciclo più interno, viene interrotto solo quello più interno.

La parola chiave `continue` serve per interrompere l'esecuzione dell'iterazione e ritornare alla valutazione della condizione.

Dato che le istruzioni `break` e `continue` interrompono il normale andamento dell'iterazione, è consigliato un uno moderato per evitare di incorrere in programmi in stile "spaghetti-code".

7.11 Note sulla memoria in Java

Stack e Heap

La memoria di esecuzione, in Java, si divide in due parti, lo stack e l'heap. Entrambi si trovano finisicamente nella RAM, lo stack si espande dalla posizione iniziale dello spazio riservato al programma verso la memoria alta con indirizzi crescenti, mentre l'heap si espande dalla posizione finale dello spazio riservato al programma verso la memoria bassa con indirizzi decrescenti.

Quando lo stack e l'heap si incontrano, la memoria a disposizione del programma finisce e il programma termina con uno `StackOverflowError`.

Nello stack (o record di attivazione) vengono memorizzate le relative chiamate ali metodi, i riferimenti ai valori di ritorno dei metodi ed eventuali riferimenti ai parametri dei metodi. Nell'heap vengono memorizzati fisicamente i valori delle variabili e le istanze delle classi (gli oggetti), con le relative variabili.

Quando una variabile o un'istanza di un oggetto non possiede più il riferimento nello stack, la relativa posizione occupata nell'heap viene liberata (marcata come disponibile) dal Garbage Collector.

Record di attivazione

Il record di attivazione è lo stack (pila, verrà approfondita nelle strutture dati) delle chiamate ai metodi. Alla base della pila si trova il record del metodo main, ad ogni chiamata di un nuovo metodo (durante l'esecuzione del main), si sospende il record relativo al main e se ne apre uno nuovo per il metodo chiamato. Questa azione si ripete ad ogni chiamata di un metodo. Quando un metodo termina, relativo record nello stack viene chiuso ed eliminato dalla pila e si riapre il record sottostante del metodo chiamante. In questo modo è possibile tenere traccia di tutte le chiamate dei vari metodi.

Un record di un metodo è composto da indice dell'istruzione chiamante (punto di ritorno una volta che il metodo è terminato), i parametri espliciti ed impliciti del metodo, il valore di ritorno ed eventuali riferimenti a variabili e oggetti locali.

Questo spiega il concetto di visibilità di variabili: una variabile è accessibile solo se il relativo riferimento è contenuto nel record di attivazione che al momento è "aperto".

7.12 Reindirizzamento dei flussi i/o

Reindirizzamento del flusso di input

È possibile reindirizzare il flusso di standar input in modo che i dati di ingresso non vengano forniti dall'utente, ma provengano da un file. Questo permette un risparmio di tempo in fase di debug nell'evitare di dover reinserire molte volte gli stessi dati. Per fare ciò è necessario eseguire il programma specificando il file da cui prendere i dati nel seguente modo:

```
java NomeFile.java < FileDiInput.txt
```

Reindirizzamento del flusso di ouput

Analogo discorso per il flusso di standard output, è possibile salvare i messaggi di output e i risultati di elaborazione in un file specificando il file in cui scrivere al momento dell'esecuzione:

```
java NomeFile.java > FileDiOutput.txt
```

È possibile reindirizzare il flusso di input e quello di output contemporaneamente nel seguente modo:

```
java NomeFile.java < FileDiInput.txt > FileDiOutput.txt
```

Canalizzazioni o pipes

Le pipeline e le concatenazioni sono funzioni che possiedono i terminali dei sistemi operativi (NB su Windows PowerShell hanno un funzionamento differente e non è possibile reindirizzare il flusso di i/o come illustrato sopra). Servono per fare in modo che l'output di un programma sia l'input di un altro programma:

```
java Programma1.java < FileDiInput.txt | java Programma2.java > FileDiOutput.txt
```

che equivale a:

```
java Programma1.java < FileDiInput.txt > temp.txt
java Programma2.java < temp.txt > FileDiOutput.txt
```

7.13 Eccezioni in Java

Le eccezioni sono uno strumento del linguaggio Java per gestire errori che possono incorrere durante l'esecuzione del programma. Vengono lanciate da un metodo di fronte un errore durante l'esecuzione di tale metodo. L'eccezione si propaga nello stack dei record di attivazione facendo terminare anticipatamente ogni metodo finché non viene catturata e gestita opportumanete. Se non viene catturata il programma termina l'esecuzione riportando l'eccezione lanciata.

Throw

Per lanciare un'eccezione, si utilizza la parola chiave `throw` seguita dal costruttore dell'eccezione che si vuole lanciare ad esempio `throw new IllegalArgumentException()`.

Try - catch

Per catturare un'eccezione è necessario inserire i metodi che possono lanciarla all'interno del blocco `try-catch` definito come segue:

```
try {
    // metodi che possono lanciare un'eccezione
    ...
} catch (NomeEccezioneDaCatturare e) {
    // istruzioni da eseguire se viene lanciata l'eccezione indicata tra parentesi
    ...
}
```

Try with resources

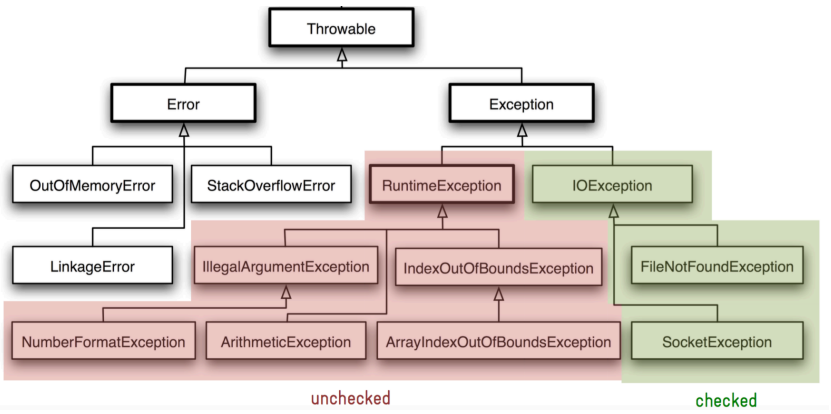
Nel caso in cui vengono utilizzate delle "risorse" ovvero classi che devono essere chiuse (Scanner, FileReader, PrintWriter), è possibile utilizzare la struttura try-with-resources specificando le risorse tra le parentesi del try. Le risorse vengono chiuse in automatico al lancio di un'eccezione o al termine del try.

```
try (/* dichiarazione delle risorse */ ... ) {
    // metodi che possono lanciare un'eccezione e utilizzo delle risorse
    ...
} catch (NomeEccezioneDaCatturare e) {
    // istruzioni da eseguire se viene lanciata l'eccezione indicata tra parentesi
    ...
}
```

Eccezioni checked e unchecked

Le eccezioni sono classi che estendo la classe `Exception` ed hanno una specifica struttra gerarchica. È possibile creare una eccezione personalizzata estendo `RuntimeException`. Si dividono in tre categorie:

1. **Errors**: lanciate in caso di problemi della JVM
2. **RuntimeException**: unchecked - lanciate per un errore nel codice dovuto al programmatore
3. **IOException**: checked - lanciate per un problema di risorse esterne al programma



8 Ricorsione

La ricorsione è una tecnica di programmazione che prevede la chiamata di un metodo durante l'esecuzione del metodo stesso, in modo da creare istanze multiple del metodo in questione, che prende il nome di metodo ricorsivo.

Per risolvere un problema ricorsivamente è necessario individuare:

- **caso base**
  situazione in cui si sa già la soluzione al problema in quanto è immediata, è il momento in cui termina la ricorsione
- **passo ricorsivo**
  situazione in cui non è possibile ottenere la soluzione in maniera immediata, ma ci si riconduce ad un problema più semplice e si richiama il metodo ricorsivamente per risolvere il caso più semplice; è importante che il caso più semplice converga, in un numero finito di iterazioni, al caso base, altrimenti si genera una ricorsione infinita che termina con `StackOverflowError`

Ci sono diverse tecniche di ricorsione:

- **ricorsione in coda**
  quando la chiamata ricorsiva avviene come ultima operazione del metodo (es. nel `return`)
- **ricorsione multipla**
  quando sono presenti più chiamate ricorsive dello stesso metodo che avvengono contemporaneamente (es. Fibonacci), questo tipo di ricorsione è estremamente lento
- **ricorsione strutturale**
  quando la ricorsione è applicata ad una struttura dati (es. stringa o array)
- **ricorsione mutua**
  quando ci sono due metodi ricorsivi che si chiamano a vicenda alternandosi (raramente usata)

Esempio di un metodo ricorsivo (Fibonacci):

```
public int fibonacci(int n) {
    if (n == 0) {
        // istruzioni da eseguire nel caso base
        return 1;
    } else {
        // istruzioni da eseguire nel passo ricorsivo
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

9 Complessità computazionale

La complessità computazionale è uno strumento di calcolo che permette di confrontare le prestazioni (asintotiche) di due programmi in modo da individuare quello più efficiente. Si divide in complessità temporale (relativa al tempo di esecuzione) e complessità spaziale (relativa alla memoria occupata). Si esprime come ordine di grandezza in funzione della dimensione dei dati di input.

Analisi sperimentale e complessità asintotica

Per misurare la complessità di un programma si può procedere con un'analisi sperimentale, ovvero misurando il tempo di esecuzione medio e la memoria occupata dal programma (o di un metodo o parte di codice) in funzione del numero di dati in input, oppure con un'analisi teorica sul comportamento asintotico della complessità.

L'analisi sperimentale è limitante in quanto bisogna già aver implementato l'algoritmo, dipende dal linguaggio di programmazione utilizzato, dalle condizioni di memoria e CPU dell'elaboratore e dal contenuto dell'input. Per questo motivo, si adotta la complessità asintotica che permette di calcolarla teoricamente (non è necessario aver implementato il programma, se non in pseudocodice) e il risultato (approssimato) è indipendente dal linguaggio, dalle condizioni della macchina e dal contenuto dell'input del programma. Inoltre è di più facile calcolo.

La complessità asintotica (per $n \to \infty$) si indica con $O(f(n))$, dove O è l'ordine di grandezza della funzione e $f(n)$ è la funzione che indica la complessità teorica in funzione di n dimensione dell'input.
Per definizione matematica:

- $f(n) \in O(g(n))$ se $\exists c > 0, N > 0$ t.c. $f(n) \leq c \cdot g(n)\ \forall n \geq N$
- $f(n) \in \Omega(g(n))$ se $\exists c > 0, N > 0$ t.c. $f(n) \geq c \cdot g(n)\ \forall n \geq N$
- $f(n) \in \Theta(g(n))$ se $f(n) \in O(g(n))$ e $f(n) \in \Omega(g(n))$

Nei polinomi: $O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n!) \subset O(n^n)$

Dato che si cerca la complessità asintotica per $n \to \infty$, è possibile che per dimensione di input piccola, un algoritmo con complessità asintotica maggiore sia più rapido di uno con complessità asintotica minore.

Classi di complessità

In base alla complessità asintotica di algoritmi, è possibile catalogarli in due classi:

- classe P: algoritmi di cui si conosce una soluzione polinomiale
- classe NP: algoritmi di cui non si conosce una soluzione polinomiale

In generale un buon algoritmo ha complessità inferiore a $O(n^2)$.

Esempi di algoritmi e relativa complessità (temporale) asintotica

- $\log n$: ricerca binaria
- n: scansione lineare di un Array
- $n \log n$: mergeSort
- n^2: selectionSort
- 2^n: torri di Hanoi (ricorsivo)
- $n!$: anagrammi

9.1 Complessità temporale

La complessità temporale è la misura del tempo di esecuzione teorico di un programma. Per calcolarla è necessario definire delle istruzioni a tempo costante e calcolare il numero di tali istruzioni in base alla dimensione dell'input. Le istruzioni a tempo costante sono:

- dichiarazioni
- inizializzazioni
- operazioni algebriche e logiche
- confronti e selezioni
- accesso a variabili o celle dell'array

9.2 Complessità spaziale

La complessità spaziale è la misura della dimensione di memoria occupata da un programma. Per calcolarla è necessario definire delle istruzioni che occupano memoria in tempo costante e calcolare il numero di tali istruzioni in base alla dimensione dell'input. Le istruzioni a "memoria costante" sono le inizializzazioni di una variabile, un array di n elementi è considerato come inizalizzazione di n variabili.

10 Strutture dati

Le strutture dati sono concetti di programmazione che permettono di memorizzare e organizzare oggetti in maniera efficiente, permettendo di risparmiare sulla memoria occupata, sul tempo di implementazione e sul tempo di accesso. Dato che queste proprietà sono inversamente proporzionali (tanto più è efficiente una, tanto meno lo è l'altra), si avranno diverse strutture dati tra cui scegliere in base alla situazione.
Le **strutture dati** si dividono in:

- strutture dati o (DT): strumenti di programmazione per la memorizzazione di dati
- strutture dati astratte (ADT): strutture dati con determinate proprietà e compromessi implementate con le DT

Le **strutture dati (DT)** si dividono in:

- lineari: ogni elemento può avere al massimo un elemento che lo precedere e un elemento che lo segue, ad esempio array e liste concatenate
- non lineari: non esiste un vincolo nel numero di precedenti o successivi, ad esempio alberi e grafi

Le **strutture dati astratte (ADT)** si dividono in:

- pila
- coda
- mappa
- multimappa o dizionario
- tabelle e hashTable
- insiemi

10.1 Array

Un array è una sequenza di celle di memoria di numero costante, ordinata in quanto si accede in base ad un indice e omogenea dato che i valori contenuti devono essere tutti dello stesso tipo. Per dichiarare un array è necessario specificare il tipo di dato da memorizzare e il numero di celle. Le celle sono numerate da 0 a $n-1$ con n dimensione dell'array e per accedere ad una specifica cella è necessario indicarne l'indice racchiuso tra le parentesi quadre. Il tempo di accesso ad una cella di un array è di tempo costante (non dipende dalla posizione relativa della cella nll'array). La dimensione dell'array è contenuta nella variabile d'istanza (public final) `length` ed essendo costante non può essere modificata.

```
// dichiarazione di un array di interi di dimensione 10
int[] vettore = new int[10];

// assegnazione del valore 7 alla cella di indice 4
vettore[4] = 7;

// accesso e stampa del valore della cella di indice 4
System.out.println(vettore[4]);

// stampa del contenuto dell'array
for (int i = 0; i < vettore.length; i++) {
    System.out.println(vettore[i]);
}
```

In Java un array è un oggetto, per cui quando viene passaggio come parametro, viene passato come riferimento, inoltre se ho due riferimenti allo stesso oggetto (anche con due nomi diversi), se lo modifico attraverso uno dei due, si modifica anche per l'altro.

Dato che è un oggetto, come tutti gli oggetti, alla sua creazione i valori delle celle sono inizializzati a 0 se il dato memorizzato è un tipo numerico (char, int, double, ...), altrimenti `null` se sono memorizzati riferimenti ad oggetti.

Se l'indice per accedere ad un elemento è negativo o maggiore-uguale alla dimensione, viene lanciata l'eccezione `ArrayIndexOutOfBoundException`. Se l'array viene solo dichiarato, ma non inizializzato (niente `new ...`), viene lanciata l'eccezione `NullPointerException`.

Algoritmo di copia

Dato che è un oggetto, non è sufficiente creare un nuovo riferimento e assegnargli il valore del riferimento vecchio, è necessario copiare i valori singolarmente. Si suppone che gli array abbiano la stessa dimensione. In alternativa si può utilizzare il metodo `arraycopy` nella classe `System`.

```
// copia del contenuto dell'array v1 nell'array v2
for (int i = 0; i < v1.length; i++) {
    v2[i] = v1[i];
}

// metodo public static void arraycopy(Object src, int srcPos, Object dest, int destPos,
    int length)
System.arraycopy(v1, 0, v2, 0, v1.length);
```

Algoritmo di ridimensionamento

Dato che gli array sono a dimensione costante, per ridimensionare un array è necessario crearne uno nuovo di dimensione maggiore, copiarne i valori e assegnare il riferimento di quello ridimensionato al vecchio array. Per una migliore efficienza computazionale la nuova dimensione deve essere un multiplo di quella vecchia (ad esempio il doppio).

```
// ridimensionamento dell'array di interi v1
int[] v2 = new int[v1.length * 2];
System.arraycopy(v1, 0, v2, 0, v1.length);
v1 = v2;
```

Array riempiti a metà

In alcuni casi, per evitare continui ridimensionamenti e non avere celle vuote, viene creato un array con un numero sufficientemente grande di celle e si tiene traccia del numero di celle occupate (dimensione logica). Ogni volta che viene aggiunto un elemento all'array, viene incrementata la dimensione logica. Nel caso in cui è necessario passarlo come parametro, serve passare anche la dimensione logica.

Questa tecnica è utile qunado è necessario memorizzare un numero indefinito di dati (es. input di numeri da parte dell'utente) di cui non si conosce a prescindere la quantità.

Input da riga di comando

È possibile eeseguire l'input di dati direttamente da riga di comando inserendoli, separati da uno spazio, immediatamente dopo il nome della classe eseguibile: `java NomeClasse param1 param2 param3 ...`

I parametri vengono memorizzati nell'array di stringhe `args` presente nella firma del main.

10.2 Matrici

Le matrici sono array bidimensionali, rappresentabili con una scacchiera. Per accedere ad una cella è necessario utilizzare due indici distinti, uno per le righe, l'altro per le colonne.

```
// dichiarazione di una matrice di interi di dimensione 5x6
int[][] matrice = new int[5][6];

// assegnazione del valore 7 alla cella in posizione (2,3)
matrice[2][3] = 7;

// accesso e stampa del valore della cella in posizione (2,3)
System.out.println(matrice[2][3]);

// stampa del contenuto della matrice
for (int i = 0; i < matrice.length; i++) {
    for (int j = 0; j < matrice[i].length; j++) {
        System.out.print(matrice[i][j] + " ");
    }
    System.out.println();
}
```

10.3 Linked List o lista concatenata

Struttura dati che implementa il concetto di sequenza. Elimina il vincolo della dimensione fissa, ma rimane ordinata e omogenea. Il tempo di accesso non è costante, ma dipende dalla posizione relativa della cella nella lista. È composta da nodi che contengono:

- il riferimento al dato associato
- il riferimento al nodo successivo

Il nodo di testa è chiamato header, punta al nodo primo nodo della lista e ha `null` come dato associato. Il nodo di coda è chiamato tail e ha `null` come riferimento al nodo successivo e al dato associato. Quando nodo di testa e nodo di coda coincidono la lista è vuota.

Implementazione nodo

```
public class ListNode {
    private Object element;
    private ListNode next;

    public ListNode(Object elm, ListNode next);
    public ListNode();

    public Object getElement();
    public ListNode getNext();

    public void setElement(Object elm);
    public void setNext(ListNode next);
}
```

Implementazione lista

```
public class LinkedList {
    private ListNode header;
    private ListNode tail;

    public LinkedList();

    public boolean isEmpty();
    public void makeEmpty();

    public Object getFirst();
    public Object getLast();

    public void addFirst(Object elm);
    public Object removeFirst();

    public void addLast(Object elm);
    public Object removeLast();

    public ListIterator getIterator();
}
```

Implementazione iteratore lista

```
private class LinkedListIterator {
    private ListNode current;
    private ListNode previous;

    public Object next();
    public boolean hasNext();
    public void add(Object obj);
    public void remove();
}
```

La classe ListIterator è interna alla classe LinkedList ed è privata.
Si immagini l'iteratore come il cursore in un editor di testo.

Complessità

- getFirst O(1)
- getLast O(1)
- addFirst O(1)
- removeFirst O(1)
- addLast O(1)
- removeLast O(n)

Lista vs Array

...

10.4 ArrayList (no esame)

Classe presente nel pacchetto `java.util`, implementato con array riempito in parte e ridimensionabile.

Inizializzazione

```
ArrayList<OggettiDaSalvare> nomeArray = new ArrayList<OggettiDaSalvare>();
```

Metodi

Alcuni dei metodi principali (`T` è il tipo di oggetti da salvare specificato nella dichiatazione):

`void add(T obj)`	aggiunta alla fine, tempo costante secondo l'analisi ammortizzata
`T remove(int index)`	rimozione elemento ad un dato indice
`T get(int index)`	accesso ad un elemento dato l'indice
`T set(int index, T obj)`	modifica di un elemento già presente
`int size()`	dimensione logica dell'array

11 Algoritmi di ordinamento

...

11.1 SelectionSort

...

11.2 MergeSort

...

11.3 InsertionSort

...

11.4 Confronto complessità

...

12 Algoritmi di ricerca

...

12.1 Ricerca lineare

...

12.2 Ricerca binaria

...

13 Strutture dati astratte - ADT

Tutte le strutture dati astratte sono definite attraverso delle interfacce, che poi verranno implementate sfruttando array (riempiti in parte, a dimensione fissa, ridimensionabili, ...) e linked list.

13.1 Container

Struttura dati generica.

```
/**
 * Interfaccia che definsice i metodi di una struttura dati generica
 */
public interface Container {
    /**
     * Metodo che verfica se la struttura dati e' vuota o no
     * @return true se la struttura dati e' vuota, false altrimenti
     */
    public boolean isEmpty();

    /**
     * Metodo che svuota la struttura dati
     */
    public void makeEmpty();
}
```

13.2 Stack o pila

Struttura dati Last-In-First-Out (LIFO).

```
/**
 * Interfaccia che definisce i metodi di una pila
 */
public interface Stack extends Container {
    /**
     * Metodo che inserisce un nuovo elemento nella pila
     * @param obj oggetto da inserire alla pila
     * @throws FullStackException se la pila e' piena
     */
    public void push(Object obj);

    /**
     * Metodo che restituisce il prossimo elemento da estrarre
     * @return prossimo elemento da estrarre
     * @throws EmptyStackException se la pila e' vuota
     */
    public Object top();

    /**
     * Metodo che rimuove un elemento dalla testa della pila
     * @return elemento rimosso
     * @throws EmptyStackException se la pila e' vuota
     */
    public Object pop();
}
```

Complessità computazionale:

- inserimento O(1)
- accesso O(1)
- rimozione O(1)

13.3 Queue o coda

Struttura dati First-In-First-Out (FIFO).

```java
/**
 * Interfaccia che definisce i metodi di una coda
 */
public interface Queue extends Container {
    /**
     * Metodo che inserisce un nuovo elemento nella coda
     * la condizione in cui la coda e' vuota e' front == back, siccome quella in
     * cui la coda e' piena sarebbe la stessa, per evitare problemi si sceglie
     * di scprecare un elemento e considereare la coda piena quando si ha un
     * solo elemento libero, ovvero se back+1 = front
     * @param obj oggetto da inserire alla coda
     * @throws FullQueueException se la coda e' piena
     */
    public void enqueue(Object obj);

    /**
     * Metodo che restituisce il prossimo elemento da estrarre
     * @return prossimo elemento da estrarre
     * @throws EmptyQueueException se la coda e' vuota
     */
    public Object getFront();

    /**
     * Metodo che rimuove un elemento dalla testa della coda
     * @return elemento rimosso
     * @throws EmptyQueueException se la coda e' vuota
     */
    public Object dequeue();
}
```

Complessità computazionale:

- inserimento O(1)
- accesso O(1)
- rimozione O(1)

13.4 Map

Definizione

Struttura dati associativa che memorizza coppie di dati composte da chiave e valore. L'associazione chiave-valore è biunivoca (1 chiave - 1 valore)

Metodi - interfaccia

Complessità computazionale

13.5 Multimap - Dictionary

Definizione

Struttura dati associativa che memorizza coppie di dati composte da chiave e valore. Possono essere presenti più valori associati alla stessa chiave.

Metodi - interfaccia

Complessità computazionale

13.6 Set

Struttura

Definizione

Metodi - interfaccia

Complessità computazionale

13.7 Table

Definizione

Metodi - interfaccia

Complessità computazionale

13.8 Hash Table

Definizione

Metodi - interfaccia

Complessità computazionale

13.9 Albero - no esame

Struttura dati in cui un elemento può avere più successori, se ogni elemento ha due successori, si dice albero binario.

13.10 Grafi - no esame

Struttura dati in cui un elmeento può avere più successori e più predecessori, creando delle reti di nodi. Si puù parlare di grafi diretti/indiretti, semplici/multigrafi, connessi/non connessi, pesati/non pesati, ...