TITOLO DEL DOCUMENTO

November 29, 2025

INDICE

1 Concetti di Base della Programmazione

1.1 Variabili e Tipi di Dato

Le variabili rappresentano porzioni di memoria che contengono valori. Ogni variabile ha un tipo che stabilisce:

- la dimensione in memoria,
- il formato del dato,
- le operazioni consentite.

Tipi primitivi:

- *int*: numeri interi a 32 bit.
- *double*: numeri con la virgola a 64 bit.
- *boolean*: valori logici `true`/`false`.
- *char*: carattere Unicode a 16 bit.

Tipi reference:

- Oggetti,
- Array,
- String (che pur essendo trattata come primitivo è un oggetto).

```
int x = 10;
double price = 19.99;
boolean isReady = true;
String name = "Java";
char letter = 'A';
```

1.2 Strutture di Controllo

Le strutture di controllo determinano il flusso dell'esecuzione.

```
if (x > 10) {
    // blocco eseguito se la condizione è vera
} else {
    // alternativa
}

for (int i = 0; i < 5; i++) {
    System.out.println(i);
}

while (condition) {
    // ripete finché la condizione è vera
}

do {
    // eseguito almeno una volta
} while(condition);
```

1.3 Funzioni e Metodi

I metodi rappresentano blocchi riutilizzabili di codice.

- *Firma del metodo*: tipo di ritorno + nome + parametri.
- *Return*: restituisce un valore al chiamante.
- *Overloading*: più metodi con lo stesso nome ma parametri diversi.

```
int sum(int a, int b) {
    return a + b;
}

int sum(int a, int b, int c) {
    return a + b + c;  // esempio di overloading
}
```

2 Fondamenti della Programmazione a Oggetti

2.1 Classi e Oggetti

Una classe è un modello per creare oggetti. Gli oggetti sono istanze che contengono:

- attributi (stato),
- metodi (comportamento).

```
class Car {
    String model;
    int year;

    void info() {
        System.out.println(model + " - " + year);
    }
}

Car c = new Car();
c.model = "Fiat";
c.year = 2020;
c.info();
```

2.2 Incapsulamento

Permette di nascondere i dettagli interni e proteggere lo stato dell'oggetto.

- *private*: accesso solo dalla classe.
- *public*: accesso libero.
- *protected*: accesso da sottoclassi.

```
class User {
    private String name;
    private int age;

    public String getName() { return name; }
    public void setName(String n) { name = n; }
```

```
    public int getAge() { return age; }
    public void setAge(int a) {
        if (a > 0) age = a;
    }
}
```

2.3 Ereditarietà

Consente di riusare e specializzare codice.

- una classe può estendere una sola classe (ereditarietà singola),
- supporta overriding dei metodi,
- permette polimorfismo.

```
class Vehicle {
    void move() { System.out.println("Moving..."); }
}

class Bike extends Vehicle {
    @Override
    void move() { System.out.println("Bike moving"); }
}
```

2.4 Polimorfismo

Il polimorfismo consente di utilizzare oggetti di classi diverse tramite un'interfaccia comune, permettendo al codice di essere più flessibile e riutilizzabile. In pratica, lo stesso codice può chiamare metodi su oggetti diversi e ottenere comportamenti differenti, senza sapere esattamente di quale classe specifica si tratta.

- *Polimorfismo di sottotipo (runtime)*: un oggetto di una sottoclasse può essere trattato come oggetto della superclasse. Il metodo corretto viene scelto dinamicamente.
- *Polimorfismo di compile-time (overloading)*: stesso nome del metodo con parametri diversi. La scelta avviene a tempo di compilazione.

Esempio runtime:

```
class Animal {
    void speak() { System.out.println("Some sound"); }
}

class Dog extends Animal {
    @Override
    void speak() { System.out.println("Bark"); }
}

class Cat extends Animal {
    @Override
    void speak() { System.out.println("Meow"); }
}

Animal a1 = new Dog();
Animal a2 = new Cat();

a1.speak(); // Output: Bark
a2.speak(); // Output: Meow
```

Spiegazione: anche se ‘a1‘ e ‘a2‘ sono dichiarati come ‘Animal‘, il metodo corretto di ciascuna sottoclasse viene eseguito, grazie al polimorfismo runtime.

Vantaggi:

- Codice più modulare e riutilizzabile.
- Riduce if/else per gestire comportamenti diversi.
- Facilita l’estensione del software senza modificare codice esistente.

```
Vehicle v = new Bike();
v.move();    // dinamicamente: stampa "Bike moving"
```

2.5 Astrazione

Consiste nel definire concetti generici tramite classi astratte o interfacce.

```
abstract class Shape {
    abstract double area();
}

class Square extends Shape {
    double side;
    Square(double side) { this.side = side; }
    double area() { return side * side; }
}
```

3 Concetti Avanzati OOP

3.1 Interfacce

Un’interfaccia definisce un contratto senza implementazione.

```
interface Drawable {
    void draw();
}

class Circle implements Drawable {
    public void draw() { System.out.println("Draw Circle"); }
}
```

3.2 Overloading e Overriding

- *Overloading*: tempo di compilazione, stessa classe.
- *Overriding*: tempo di esecuzione, ereditarietà.
- Per fare override occorre l’annotazione @Override.

3.3 Generics

Permettono di creare classi e metodi tipizzati.

```
class Box<T> {
    private T value;
    void set(T v) { value = v; }
    T get() { return value; }
}

Box<Integer> b = new Box<>();
b.set(5);
```

4 Design Patterns Fondamentali

4.1 Singleton

Scopo: garantire che una classe abbia una sola istanza e fornire un punto di accesso globale.
Concetti chiave:

- Istanza privata e statica.
- Costruttore privato per evitare creazioni esterne.
- Metodo pubblico statico per ottenere l'istanza.

```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) instance = new Singleton();
        return instance;
    }
}

// Utilizzo
Singleton s1 = Singleton.getInstance();
Singleton s2 = Singleton.getInstance();
// s1 e s2 puntano alla stessa istanza
```

Vantaggi:

- Controllo centralizzato.
- Evita duplicazioni di risorse.

—

4.2 Factory Method

Scopo: creare oggetti senza esporre la logica di costruzione, favorendo estensibilità.
Concetti chiave:

- Creator astratto con metodo di creazione astratto.
- ConcreteCreator implementa la creazione specifica.
- Il client usa solo il creator, senza conoscere il prodotto concreto.

```
abstract class Product {}
class ConcreteProduct extends Product {}

abstract class Creator {
```

```
    abstract Product create();
}

class ConcreteCreator extends Creator {
    Product create() { return new ConcreteProduct(); }
}

// Utilizzo
Creator creator = new ConcreteCreator();
Product p = creator.create();
```

Vantaggi:

- Decoupling tra creazione e utilizzo.
- Facile aggiungere nuovi prodotti senza cambiare il client.

4.3 Strategy

Scopo: permettere di cambiare il comportamento di un oggetto a runtime senza modificare la classe.
Concetti chiave:

- Interfaccia comune per tutti i comportamenti.
- Implementazioni concrete del comportamento.
- Classe Context usa una strategia e può cambiarla dinamicamente.

```
interface Payment { void pay(); }

class Paypal implements Payment {
    public void pay() { System.out.println("PayPal"); }
}

class Card implements Payment {
    public void pay() { System.out.println("Card payment"); }
}

class Context {
    private Payment method;
    void setStrategy(Payment m) { method = m; }
    void execute() { method.pay(); }
}

// Utilizzo
Context ctx = new Context();
ctx.setStrategy(new Paypal());
ctx.execute(); // PayPal
ctx.setStrategy(new Card());
ctx.execute(); // Card payment
```

Vantaggi:

- Comportamenti intercambiabili senza cambiare Context.
- Facilita test e manutenzione.

—

4.4 Observer

Scopo: permettere a un oggetto (Subject) di notificare automaticamente più oggetti interessati (Observer) quando cambia il suo stato.

Concetti chiave:

- *Subject*: mantiene lo stato e la lista di Observer.
- *Observer*: implementa il metodo `update()`.
- *Loose coupling*: Subject non conosce i dettagli degli Observer.

```
interface Observer { void update(String message); }

interface Subject {
    void addObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers();
}

class NewsAgency implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String news;

    public void addObserver(Observer o) { observers.add(o); }
    public void removeObserver(Observer o) { observers.remove(o); }

    public void setNews(String news) {
        this.news = news;
        notifyObservers();
    }

    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(news);
        }
    }
}

class NewsChannel implements Observer {
    private String news;
    public void update(String news) {
        this.news = news;
        System.out.println("Received: " + news);
    }
}

// Utilizzo
NewsAgency agency = new NewsAgency();
NewsChannel channel1 = new NewsChannel();
NewsChannel channel2 = new NewsChannel();

agency.addObserver(channel1);
agency.addObserver(channel2);

agency.setNews("Breaking news!");
```

Vantaggi:

- Aggiornamento automatico di più oggetti.
- Riduce dipendenze tra Subject e Observer.
- Facilita estensione e manutenzione del codice.

5 Algoritmi Fondamentali

5.1 Bubble Sort

```
void bubbleSort(int[] arr) {
    boolean swapped;
    do {
        swapped = false;
        for (int i = 0; i < arr.length - 1; i++) {
            if (arr[i] > arr[i+1]) {
                int t = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = t;
                swapped = true;
            }
        }
    } while (swapped);
}
```

Complexity:

- Worst-case: $O(n^2)$
- Best-case (array già ordinato): $O(n)$
- Space: $O(1)$

Come funziona: Bubble Sort confronta coppie di elementi vicini e li scambia se sono nell'ordine sbagliato. Ripete tutto finché non ci sono più swap.
Costo: Molto alto per array grandi perché ogni elemento può essere confrontato con tutti gli altri.
Quando usarlo: Solo per array piccoli o quasi ordinati, oppure come esercizio didattico.

5.2 Selection Sort

```
void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        int min = i;
        for (int j = i+1; j < arr.length; j++) {
            if (arr[j] < arr[min]) min = j;
        }
        int t = arr[i];
        arr[i] = arr[min];
        arr[min] = t;
    }
}
```

Complexity:

- Worst-case: $O(n^2)$
- Best-case: $O(n^2)$ (non migliora se l'array è già ordinato)
- Space: $O(1)$

Come funziona: Trova il minimo nella parte non ordinata e lo sposta all'inizio, ripetendo fino a ordinare tutto.
Costo: Alto, ma meno swap rispetto a Bubble Sort.
Quando usarlo: Quando vuoi ridurre il numero di swap rispetto a Bubble Sort, ma comunque non per array grandi.

5.3 Insertion Sort

```
void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }
}
```

Complexity:

- Worst-case: $O(n^2)$
- Best-case (quasi ordinato): O(n)
- Space: O(1)

Come funziona: Inserisce ogni elemento nella posizione corretta tra quelli già ordinati, come se stessimo ordinando le carte in mano.

Costo: Molto basso per array piccoli o quasi ordinati.

Quando usarlo: Perfetto per array quasi ordinati o piccoli set di dati; spesso usato come subroutine in algoritmi più complessi come Merge Sort.

5.4 Binary Search

```
int binarySearch(int[] arr, int target) {
    int low = 0, high = arr.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

Complexity:

- Worst-case: O(log n)
- Best-case: O(1)
- Space: O(1)

Come funziona: Divide a metà l'array ordinato e decide quale metà continuare a cercare. Ripete finché trova l'elemento o finisce l'array.

Costo: Super efficiente per array grandi, ma funziona solo su array già ordinati.

Quando usarlo: Sempre su array ordinati; ottimo per lookup rapidi in database, liste ordinate, o qualsiasi struttura in cui cerchi spesso valori.

5.5 Merge Sort

```
void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
```

```
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int[] L = new int[n1];
    int[] R = new int[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
```

Complexity:

- Worst-case: O(n log n)
- Best-case: O(n log n)
- Space: O(n) (richiede memoria aggiuntiva per array temporanei)

Come funziona: Merge Sort divide ricorsivamente l'array a metà fino a singoli elementi, poi li ricombina ordinandoli. È un algoritmo **divide and conquer**.

Costo: Più alto in termini di memoria rispetto a Insertion o Selection Sort, ma molto efficiente per array grandi.

Quando usarlo: Ideale per array grandi o strutture dati che non entrano in memoria contigua. Perfetto se serve stabilità nell'ordinamento (non cambia l'ordine relativo di elementi uguali).

Esempio pratico: Ordinare una lista di 100.000 numeri. Bubble/Selection/Insertion sarebbero troppo lenti ($O(n^2)$), mentre Merge Sort rimane O(n log n), quindi molto più veloce.

6 Comprendere i Costi Computazionali (Big O) in Termini Pratici

Quando parliamo di algoritmi, spesso si usa la **notazione Big O** per indicare quanto "costa" un algoritmo in termini di tempo o spazio, cioè quanto cresce il lavoro da fare quando aumenta la dimensione del problema (n). Ecco una spiegazione chiara con esempi concreti:

6.1 O(1) - Tempo Costante

- **Cosa significa:** L'algoritmo impiega sempre lo stesso tempo, indipendentemente dalla dimensione dell'input.
- **Esempio:** Leggere il primo elemento di un array.
- **Analogia:** Cercare un libro su uno scaffale sapendo già esattamente dove si trova.

6.2 O(n) - Tempo Lineare

- **Cosa significa:** Il tempo cresce proporzionalmente al numero di elementi. Se hai 10 elementi ci mette X tempo, se ne hai 100 ci mette circa 10 volte X.
- **Esempio:** Cercare un numero in un array non ordinato (linear search).
- **Analogia:** Controllare ogni libro su uno scaffale uno per uno finché trovi quello giusto.

6.3 O(n²) - Tempo Quadratico

- **Cosa significa:** Il tempo cresce come il quadrato del numero di elementi. Raddoppiare n fa quadruplicare il tempo.
- **Esempio:** Bubble Sort o Selection Sort su un array di n elementi.
- **Analogia:** Se devi controllare ogni coppia di libri per vedere quale è più grande, per 10 libri fai 45 confronti, per 100 libri fai 4950 confronti.

6.4 O(log n) - Tempo Logaritmico

- **Cosa significa:** L'algoritmo riduce l'input a metà ad ogni passo. Anche aumentando molto n, il tempo cresce lentamente.
- **Esempio:** Binary Search in un array ordinato.
- **Analogia:** Cercare un libro in una biblioteca seguendo l'alfabeto: apri a metà della sezione, poi metà della metà, ecc. Anche se la biblioteca è enorme, bastano pochi passaggi.

6.5 O(n log n) - Tempo Lineare-Logaritmico

- **Cosa significa:** L'algoritmo fa un lavoro proporzionale a n volte log n. Cresce più veloce di lineare ma molto più lento di quadratico.
- **Esempio:** Merge Sort, Quick Sort (medio caso).
- **Analogia:** Dividere una lista di libri in metà, ordinarle e poi unirle ordinatamente. Ogni livello di divisione richiede un passaggio lineare per ricombinare.

6.6 Perché è importante

Sapere queste complessità aiuta a:

- Prevedere se un algoritmo è adatto a grandi dataset.
- Capire quali algoritmi saranno lenti o veloci in pratica.
- Scegliere il giusto compromesso tra tempo e spazio.

7 Low-Level Concepts

7.1 Stack and Heap

- **Stack:** Memoria veloce per funzioni e variabili locali. Viene liberata automaticamente quando la funzione finisce.
- **Heap:** Memoria per oggetti creati dinamicamente. Gestita dal garbage collector (Java) o manualmente (C/C++).

7.2 Pass-by-Value

In Java, tutto è passato **by value**:

- Per tipi primitivi: la funzione riceve una copia del valore.
- Per oggetti: la funzione riceve una copia della reference. Modificare l'oggetto dentro la funzione cambia l'oggetto originale, ma assegnare la reference a un nuovo oggetto non lo fa.

```
void change(int x) { x = 10; }
```

Costo: O(1) per la copia del valore/reference.
Quando usarlo: Sempre in Java; è il comportamento standard per chiamate di funzione.

7.3 Memoria e Garbage Collector

Il GC libera memoria eliminando oggetti non più referenziati.

7.4 Eccezioni

- Checked: devono essere gestite.
- Unchecked: errori di runtime.

```
try {
    risky();
} catch (Exception e) {
    e.printStackTrace();
}
```

8 HTTP e HTTPS

8.1 HTTP (HyperText Transfer Protocol)

Definizione: HTTP è un protocollo di comunicazione usato per trasferire dati tra client (browser) e server sul web.
Caratteristiche principali:

- **Stateless:** ogni richiesta è indipendente; il server non ricorda le richieste precedenti.
- **Testuale:** i dati sono inviati in chiaro, leggibili da chi intercetta la connessione.
- **Usato per:** navigazione web, invio/ricezione di dati tramite form, API non crittografate.

Funzionamento semplificato:

1. Il client invia una richiesta HTTP al server (GET, POST, ecc.).
2. Il server elabora la richiesta e invia una risposta.
3. I dati viaggiano in chiaro sulla rete.

Problemi principali:

- I dati possono essere intercettati facilmente (es. password, informazioni sensibili).
- Vulnerabile a attacchi come man-in-the-middle.

8.2 HTTPS (HTTP Secure)

Definizione: HTTPS è la versione sicura di HTTP. Aggiunge un livello di crittografia usando TLS/SSL per proteggere i dati in transito.

Caratteristiche principali:

- **Crittografia:** i dati sono cifrati, quindi non leggibili da chi intercetta.
- **Autenticità:** il certificato TLS/SSL garantisce che il server sia quello giusto.
- **Integrità:** i dati non possono essere modificati senza essere rilevati.

Come funziona (semplificato):

1. Il client si collega al server e richiede una connessione sicura.
2. Il server invia il certificato SSL/TLS.
3. Il client verifica il certificato.
4. Si stabilisce una chiave simmetrica tramite crittografia asimmetrica.
5. Tutti i dati successivi vengono cifrati con la chiave simmetrica (più veloce) per la durata della sessione.

Vantaggi di HTTPS:

- Protezione dei dati sensibili (password, numeri di carta, ecc.).
- Protezione contro attacchi di tipo man-in-the-middle.
- Migliore fiducia per gli utenti e SEO migliore sui motori di ricerca.

Esempio pratico: Quando inserisci la tua password su un sito bancario:

- Con HTTP: la password viaggia in chiaro → rischio intercettazione.
- Con HTTPS: la password è cifrata → impossibile leggerla senza la chiave privata del server.

Nota: HTTPS è ormai lo standard per qualsiasi sito moderno, anche per contenuti pubblici, perché protegge la privacy degli utenti e impedisce modifiche ai dati trasmessi.

9 Pila TCP/IP e Modello OSI

9.1 Modello OSI (Open Systems Interconnection)

Il modello OSI è un framework concettuale che divide la comunicazione di rete in **7 livelli**, ciascuno con responsabilità specifiche:

1. **Layer 7 – Application:** fornisce servizi di rete alle applicazioni (es. browser, email).
2. **Layer 6 – Presentation:** traduce i dati in formati comprensibili, cifratura, compressione.
3. **Layer 5 – Session:** gestisce le sessioni di comunicazione tra applicazioni.
4. **Layer 4 – Transport:** garantisce consegna affidabile dei dati (TCP) o veloce ma senza garanzia (UDP).
5. **Layer 3 – Network:** instrada i pacchetti tra reti diverse (IP).
6. **Layer 2 – Data Link:** gestione dei frame su reti locali, indirizzamento MAC, controllo errori.
7. **Layer 1 – Physical:** trasmissione dei bit su cavi, fibra, onde radio, ecc.

9.2 Pila TCP/IP

La pila TCP/IP è più pratica e usata nel mondo reale. Ha **4 livelli principali**:

1. **Application Layer:** HTTP, HTTPS, FTP, SMTP.
2. **Transport Layer:** TCP (affidabile), UDP (veloce).
3. **Internet Layer:** IP, ICMP.
4. **Network Access / Link Layer:** Ethernet, Wi-Fi, ARP.

Differenze principali tra OSI e TCP/IP:

- OSI ha 7 livelli teorici, TCP/IP 4 pratici.
- TCP/IP combina Session, Presentation e Application in un solo livello Application.
- TCP/IP è quello effettivamente implementato su Internet.

9.3 Grafico ASCII - Modello OSI

```
+--------------------+
|  Application (7)   |
+--------------------+
|  Presentation (6)  |
+--------------------+
|  Session (5)       |
+--------------------+
|  Transport (4)     |
+--------------------+
|  Network (3)       |
+--------------------+
|  Data Link (2)     |
+--------------------+
|  Physical (1)      |
+--------------------+
```

9.4 Grafico ASCII - Pila TCP/IP

```
+-----------------------------+
|  Application Layer          |
|  (HTTP, HTTPS, FTP, SMTP)   |
+-----------------------------+
|  Transport Layer            |
|  (TCP, UDP)                 |
+-----------------------------+
|  Internet Layer             |
|  (IP, ICMP)                 |
+-----------------------------+
|  Network Access / Link Layer|
|  (Ethernet, Wi-Fi, ARP)     |
+-----------------------------+
```

Spiegazione pratica: Quando mandi un pacchetto:

- Parte dall'Application Layer (es. browser vuole una pagina).
- Passa al Transport Layer (TCP aggiunge numero di porta, controllo errori).

- Arriva al Network Layer (IP decide dove spedire il pacchetto).
- Infine al Data Link / Physical Layer (il pacchetto viaggia fisicamente su cavi o onde radio).

Nota: Ogni livello aggiunge "informazioni di controllo" chiamate **header** per garantire che il pacchetto arrivi a destinazione correttamente.

10 Concetti di Informatica di Base

10.1 Bit e Byte

- **Bit:** la più piccola unità di informazione, può essere 0 o 1.
- **Byte:** gruppo di 8 bit, usato per rappresentare caratteri o piccoli numeri.
- **Multipli comuni:**
  - 1 KB (kilobyte) = 1024 B
  - 1 MB (megabyte) = 1024 KB
  - 1 GB (gigabyte) = 1024 MB
  - 1 TB (terabyte) = 1024 GB

10.2 Rappresentazione dei dati

- **Numeri interi:** rappresentati in binario, es. 5 = 00000101.
- **Numeri decimali e floating point:** rappresentano numeri con la virgola (IEEE 754).
- **Caratteri:** codificati tramite standard come ASCII o UTF-8.

10.3 Sistema binario e logica

- I computer lavorano in **base 2**: tutto è 0 e 1.
- Operazioni logiche fondamentali:
  - AND, OR, NOT, XOR
  - Usate per confronto, calcoli e controllo del flusso.

10.4 Memoria e archiviazione

- **RAM (Random Access Memory):** memoria volatile, usata per dati temporanei e processi in esecuzione.
- **ROM (Read Only Memory):** memoria non volatile, contiene dati permanenti come firmware.
- **Storage:** hard disk, SSD, memorie esterne. Conservano dati a lungo termine.

10.5 CPU e elaborazione

- **CPU (Central Processing Unit):** il "cervello" del computer che esegue istruzioni.
- **Ciclo di fetch-decode-execute:** la CPU prende istruzioni dalla memoria, le interpreta e le esegue.

10.6 Reti e protocolli base

- **IP address:** identificativo unico di un dispositivo in rete.
- **MAC address:** identificatore unico della scheda di rete.
- **DNS:** sistema che traduce nomi di dominio in indirizzi IP.

10.7 File e filesystem

- I dati sono organizzati in file.
- I file sono organizzati in cartelle/directories.
- I filesystem gestiscono come i dati sono memorizzati e recuperati.

10.8 Unità di misura delle prestazioni

- **Hz:** cicli al secondo, misura velocità della CPU.
- **Ops/sec:** operazioni al secondo, utile per misurare performance di calcolo.

10.9 Indirizzi IP

Breve spiegazione: Un indirizzo IP (Internet Protocol) è un identificativo numerico unico assegnato a ogni dispositivo connesso a una rete. Serve per far "parlare" i computer tra loro, permettendo l'invio e la ricezione di dati.

10.9.1 Dettagli tecnici e informazioni complete

- **Versioni principali:**
  - **IPv4:** indirizzi a 32 bit, scritti in notazione decimale puntata (es. 192.168.0.1). Permette circa 4,3 miliardi di indirizzi.
  - **IPv6:** indirizzi a 128 bit, scritti in esadecimale separata da due punti (es. 2001:0db8:85a3::8a2e:0370:7334). Serve per risolvere il problema della scarsità di IPv4.
- **Tipi di indirizzi IP:**
  - **Statici:** assegnati manualmente e non cambiano.
  - **Dinamici:** assegnati automaticamente tramite DHCP, possono cambiare nel tempo.
  - **Pubblici:** visibili su Internet, unici globalmente.
  - **Privati:** usati in reti locali, non visibili su Internet (es. 192.168.x.x, 10.x.x.x).
  - **Multicast:** per inviare pacchetti a un gruppo di destinatari.
  - **Broadcast:** per inviare pacchetti a tutti i dispositivi di una rete locale.
- **Struttura di un indirizzo IPv4:**
  - Composto da 4 ottetti (8 bit ciascuno), separati da punti.
  - Esempio: 192.168.1.5 → ogni ottetto è un numero da 0 a 255.
  - Può essere diviso in **network part** e **host part** usando la subnet mask (es. 255.255.255.0).
- **Subnetting:** Permette di dividere una rete in sotto-reti più piccole per ottimizzare l'uso degli indirizzi IP e migliorare la sicurezza/gestione della rete. Esempio: 192.168.1.0/24 → 256 indirizzi totali (0-255), di cui molti riservati.

- **IPv6 caratteristiche aggiuntive:**
  - Indirizzi enormi, praticamente illimitati.
  - Supporta autoconfigurazione (SLAAC).
  - Migliore gestione del routing e supporto nativo per multicast.
  - Notazione abbreviata: zeri consecutivi possono essere compressi (::).
- **Funzione pratica:** Gli indirizzi IP permettono a router e switch di instradare pacchetti fino al dispositivo corretto, sia in una rete locale che su Internet. Senza indirizzi IP un pacchetto non saprebbe a chi consegnare i dati.
- **Esempi concreti:**
  - PC in una rete domestica: 192.168.1.5
  - Router con IP pubblico: 85.12.45.67
  - IPv6 di un server moderno: 2001:0db8:85a3:0000:0000:8a2e:0370:7334
- **Strumenti utili:**
  - `ping` → verifica se un IP è raggiungibile.
  - `traceroute / tracert` → mostra il percorso dei pacchetti verso un IP.
  - `ipconfig / ifconfig` → mostra l'indirizzo IP del dispositivo.