Contents

Manuel de Révision Complet - Analyste Programmeur

Version: 2026
Durée de révision: 2-3 heures
Objectif: Condenser 7 jours de préparation en un guide de révision rapide

---

TECHNIQUES MNÉMOTECHNIQUES GLOBALES

La méthode du PALAIS MENTAL

Associez chaque concept à un lieu familier (votre maison, école): - **Porte d'entrée** = SQL & BDD - **Salon** = POO & SOLID - **Cuisine** = UML - **Chambre 1** = Structures de données - **Chambre 2** = Algorithmes - **Balcon** = Backend & Networking - **Jardin** = Frontend

Techniques de mémorisation rapide

1. **Acronymes** - ACID, SOLID, OSI
2. **Rimes** - "TCP fiable, UDP rapide, comme un câble ou un rapide"
3. **Histoires** - Créer des mini-scénarios bancaires
4. **Visualisation** - Dessiner mentalement les diagrammes
5. **Répétition espacée** - Réviser 3x: maintenant, dans 1h, demain

---

BDD & SQL

Définitions Express

Base de données: Système organisé pour stocker, gérer et récupérer des données de manière structurée et efficace.

Transaction: Séquence d'opérations qui réussit complètement ou échoue complètement (tout ou rien).

ACID - À MÉMORISER ABSOLUMENT

Mnémonique: "**A**lice **C**rée **I**ntelligemment des **D**onnées"

Propriété	Définition	Exemple bancaire
Atomicity	Tout ou rien	Virement: débit ET crédit ensemble
Consistency	État valide → État valide	Solde jamais négatif si contrainte
Isolation	Transactions isolées	2 retraits simultanés sans conflit
Durability	Survit aux pannes	Données persistent après coupure

Code SQL Essentiel

```java
// Java
try {
    conn.setAutoCommit(false);

    // Débiter
    stmt1.executeUpdate("UPDATE comptes SET solde = solde - 500 WHERE id = 1");

    // Créditer
    stmt2.executeUpdate("UPDATE comptes SET solde = solde + 500 WHERE id = 2");

    conn.commit();
} catch (Exception e) {
    conn.rollback();
}
```

```python
# Python
try:
    cursor.execute("BEGIN TRANSACTION")

    # Débiter
    cursor.execute("UPDATE comptes SET solde = solde - 500 WHERE id = 1")

    # Créditer
    cursor.execute("UPDATE comptes SET solde = solde + 500 WHERE id = 2")

    cursor.execute("COMMIT")
except:
    cursor.execute("ROLLBACK")
```

Transaction complète (Java + SQL)

Les JOINs - Visualisation **Mnémonique**: "**I**NNER = **I**ntersection, **L**EFT = tout à **L**eft"

```
CLIENTS    COMPTES
  A ──────── 1
  B ──────── 2
  C          3

INNER JOIN → A-1, B-2 (seulement correspondances)
LEFT JOIN  → A-1, B-2, C-NULL (tous les clients)
```

```sql
-- INNER JOIN: Clients AVEC comptes
SELECT c.nom, a.solde
FROM clients c
INNER JOIN comptes a ON c.id = a.client_id;

-- LEFT JOIN: TOUS les clients
```

```sql
SELECT c.nom, a.solde
FROM clients c
LEFT JOIN comptes a ON c.id = a.client_id;
```

Normalisation (simplifié)

Mnémonique: "**1** valeur, **2** clé entière, **3** pas de transit"

- **1NF**: Valeurs atomiques + clé primaire
- **2NF**: 1NF + dépend de la CLÉ ENTIÈRE
- **3NF**: 2NF + pas de dépendance transitive

Index - Quand utiliser?

Mnémonique: "**W**HERE **J**OIN **O**RDER **F**K"

Créer index sur: - **W**HERE clauses fréquentes - **J**OIN colonnes - **O**RDER BY colonnes - **F**oreign **K**eys

Éviter sur: - Petites tables - Colonnes modifiées souvent

```sql
-- Index simple
CREATE INDEX idx_email ON clients(email);

-- Index composé
CREATE INDEX idx_date_montant ON transactions(date, montant);
```

POO & SOLID

Définitions Express

POO: Paradigme organisant le code en objets (données + comportements).

Classe: Modèle/blueprint pour créer des objets.

Objet: Instance concrète d'une classe.

4 Piliers POO

Mnémonique: "**E**lle **H**érite **P**our **A**bstraire" (EHPA)

Pilier	Définition	Code
Encapsulation	Cacher les données	`private` + getters/setters
Héritage	Réutiliser du code	`extends`, `implements`
Polymorphisme	Même interface, comportements différents	Overriding
Abstraction	Montrer l'essentiel	`abstract`, `interface`

Code POO Simplifié

```java
// Java
class Compte {
    private double solde;

    public double getSolde() { return solde; }
```

```java
    public void deposer(double m) {
        if (m > 0) solde += m;
    }
}
```

```python
# Python
class Compte:
    def __init__(self):
        self.__solde = 0

    def get_solde(self):
        return self.__solde

    def deposer(self, m):
        if m > 0:
            self.__solde += m
```

Encapsulation

```java
// Java
abstract class Compte {
    protected double solde;
    abstract boolean retirer(double m);
}

class CompteEpargne extends Compte {
    boolean retirer(double m) {
        if (m <= solde) {
            solde -= m;
            return true;
        }
        return false;
    }
}
```

```python
# Python
class Compte:
    def __init__(self):
        self.solde = 0

    def retirer(self, m):
        raise NotImplementedError

class CompteEpargne(Compte):
    def retirer(self, m):
        if m <= self.solde:
            self.solde -= m
            return True
        return False
```

Héritage

SOLID - Principes

Mnémonique: “**S**uper **O**iseaux **L**ançant **I**ntelligemment des **D**arts”

Principe	Définition 1 phrase	Règle d’or
SRP	Une classe = une responsabilité	1 raison de changer
OCP	Ouvert extension, fermé modification	Ajouter, pas modifier
LSP	Sous-classes substituables	Respecter le contrat
ISP	Interfaces petites et ciblées	Pas de méthodes inutiles
DIP	Dépendre d’abstractions	Interface > classe concrète

SOLID Simplifié

```
//   MAUVAIS
class Employe {
    void calculerSalaire() {}
    void sauvegarderBDD() {}  // 2 responsabilités!
}

//   BON
class Employe {}
class SalaireService { void calculer(Employe e) {} }
class EmployeRepository { void sauvegarder(Employe e) {} }
```

S - Single Responsibility

```
//   BON - Extension par nouvelles classes
interface Forme { double aire(); }
class Rectangle implements Forme {
    public double aire() { return l * h; }
}
class Cercle implements Forme {
    public double aire() { return Math.PI * r * r; }
}
```

O - Open/Closed

```
//   BON - Dépendre de l'abstraction
interface Repository { void save(Object o); }

class Service {
    private Repository repo;  // Interface!
    Service(Repository repo) { this.repo = repo; }
}
```

D - Dependency Inversion

Design Patterns Essentiels

```
// Java - Thread-safe
class Config {
```

```java
    private Config() {}
    private static class Holder {
        static final Config INSTANCE = new Config();
    }
    public static Config get() { return Holder.INSTANCE; }
}
```

```python
# Python
class Config:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance
```

Singleton

```java
// Java
interface Paiement { void payer(double m); }
class PaiementCarte implements Paiement {
    public void payer(double m) {}
}

class FabriquePaiement {
    static Paiement creer(String type) {
        if (type.equals("CARTE")) return new PaiementCarte();
        // ...
    }
}
```

Factory

UML

Définitions Express

UML: Langage de modélisation graphique standardisé pour visualiser les systèmes logiciels.

Diagramme de classes: Montre structure statique (classes, attributs, relations).

Cas d'utilisation: Fonctionnalités du système et acteurs.

Les 3 Diagrammes Essentiels

Mnémonique: "**C**as **C**lasse **S**équence" (CCS)

1. **Diagramme de Cas d'utilisation** - Exigences fonctionnelles
2. **Diagramme de Classes** - Structure statique
3. **Diagramme de Séquence** - Interactions dynamiques

Diagramme de Classes

Symboles de visibilité

```
+ Public
- Privé
# Protégé
~ Package
```

Relations **Mnémonique**: “**A**gglo **C**ompose **H**érite”

```
Association:  ──────────  "utilise"
Agrégation:    ──────────  "a-un" (parties indépendantes)
Composition:   ──────────  "possède" (parties dépendantes)
Héritage:     ─────       "est-un" (triangle vers parent)
Réalisation:  - - -       "implémente" (pointillé)
```

Multiplicités

```
1      = Exactement un
0..1   = Zéro ou un
*      = Zéro ou plusieurs
1..*   = Un ou plusieurs
```

Exemple simplifié

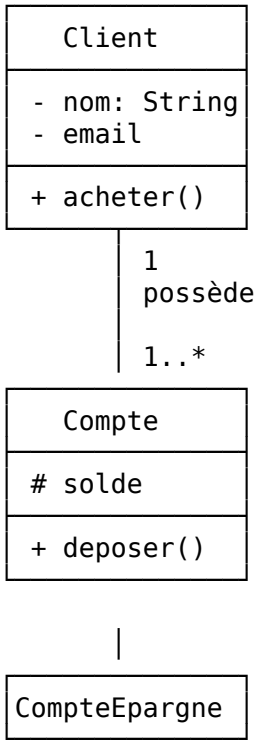
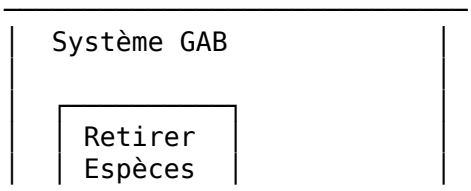
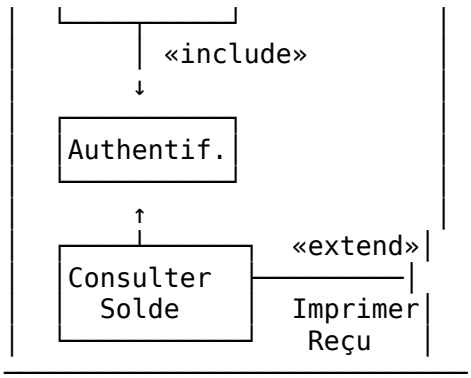


Diagramme de Cas d’Utilisation

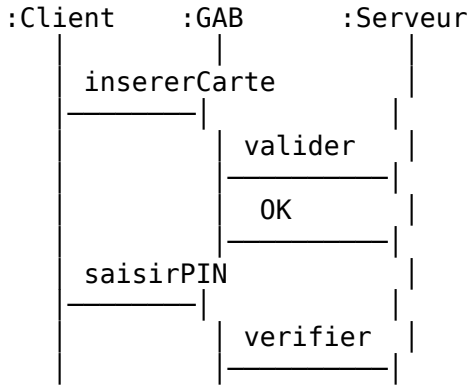
Mnémonique: “**I**NCLUDE obligatoire, **E**XTEND optionnel”





Règles: - **include**: Base → Inclus (obligatoire) - **extend**: Extension → Base (optionnel) - Acteurs HORS de la frontière

Diagramme de Séquence



Éléments: - Flèche pleine (———): Message synchrone - Flèche pointillée (– – –): Retour - Boîte d'activation: Objet actif - Fragment `alt`: Alternatives `[condition]`

STRUCTURES DE DONNÉES

Définitions Express

Structure de données: Façon d'organiser et stocker les données pour accès/modification efficaces.

Complexité temporelle: Mesure du temps d'exécution (Big O notation).

Tableau des Complexités

Mnémonique: "**A**ccès **R**echerche **I**nsert **S**upprime" (ARIS)

Structure	Accès	Recherche	Insert	Delete
Array	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)
LinkedList	O(n)	O(n)	O(1)*	O(1)*
HashMap	-	O(1)	O(1)	O(1)
BST	O(log n)	O(log n)	O(log n)	O(log n)

Stack (Pile) - LIFO

Mnémonique: "**L**ast **I**n, **F**irst **O**ut - comme une pile d'assiettes"

```java
// Java
class Stack {
    private int[] arr;
    private int top = -1;

    void push(int x) { arr[++top] = x; }
    int pop() { return arr[top--]; }
    int peek() { return arr[top]; }
    boolean isEmpty() { return top < 0; }
}
```

```python
# Python
stack = []
stack.append(5)  # push
x = stack.pop()  # pop
top = stack[-1] if stack else None  # peek
```

Usage bancaire: Historique d'annulation (undo)

Queue (File) - FIFO

Mnémonique: "**F**irst **I**n, **F**irst **O**ut - comme une file d'attente"

```java
// Java - Queue circulaire
class Queue {
    private int[] arr;
    private int front = 0, rear = -1, count = 0;

    void enqueue(int x) {
        rear = (rear + 1) % arr.length;
        arr[rear] = x;
        count++;
    }

    int dequeue() {
        int x = arr[front];
        front = (front + 1) % arr.length;
        count--;
        return x;
    }
}
```

```python
# Python
from collections import deque
q = deque()
q.append(5)     # enqueue
x = q.popleft() # dequeue
```

Usage bancaire: File de transactions à traiter

Linked List

```java
// Java
class Node {
```

```java
    int data;
    Node next;
}

class LinkedList {
    Node head;

    void insertHead(int val) {
        Node n = new Node();
        n.data = val;
        n.next = head;
        head = n;
    }

    boolean delete(int val) {
        if (head.data == val) {
            head = head.next;
            return true;
        }
        Node curr = head;
        while (curr.next != null) {
            if (curr.next.data == val) {
                curr.next = curr.next.next;
                return true;
            }
            curr = curr.next;
        }
        return false;
    }
}
```

```python
# Python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert_head(self, val):
        n = Node(val)
        n.next = self.head
        self.head = n
```

Hash Table

Mnémonique: “**H**ash pour **R**apide **R**echerche” (O(1))

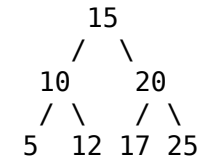
```java
// Java
import java.util.HashMap;
HashMap<String, Integer> map = new HashMap<>();
map.put("Alice", 95);     // O(1)
int score = map.get("Alice"); // O(1)
boolean exists = map.containsKey("Alice"); // O(1)
```

```python
# Python
map = {}
map["Alice"] = 95      # O(1)
score = map.get("Alice", 0)  # O(1)
exists = "Alice" in map      # O(1)
```

Usage bancaire: Recherche rapide de compte par numéro

Binary Search Tree (BST)

Propriété: Gauche < Racine < Droite




```java
// Java
class TreeNode {
    int data;
    TreeNode left, right;
}

TreeNode insert(TreeNode root, int val) {
    if (root == null) return new TreeNode(val);
    if (val < root.data)
        root.left = insert(root.left, val);
    else
        root.right = insert(root.right, val);
    return root;
}

TreeNode search(TreeNode root, int val) {
    if (root == null || root.data == val) return root;
    if (val < root.data) return search(root.left, val);
    return search(root.right, val);
}
```

Traversées **Mnémonique**: "**IN**order = **IN**térieur (donne ordre trié)"

```java
// Inorder: Gauche-Racine-Droite → Ordre trié
void inorder(TreeNode n) {
    if (n == null) return;
    inorder(n.left);
    print(n.data);
    inorder(n.right);
}

// Preorder: Racine-Gauche-Droite
void preorder(TreeNode n) {
    if (n == null) return;
    print(n.data);
    preorder(n.left);
    preorder(n.right);
}
```

```java
// Postorder: Gauche-Droite-Racine
void postorder(TreeNode n) {
    if (n == null) return;
    postorder(n.left);
    postorder(n.right);
    print(n.data);
}
```

ALGORITHMES

Définitions Express

Algorithme: Séquence d'instructions précises et finies qui résout un problème.

Big O: Mesure de la performance (vitesse).

Big O - Ordre croissant

Mnémonique: "1 **L**og **N**otre **N**ombre **E**xponentiellement"

```
O(1) < O(log n) < O(n) < O(n log n) < O(n²) < O(2^n)
|      |          |      |             |       |
Constant  Log   Linéaire QuickSort  Bubble  Exponentiel
```

Binary Search

PRÉREQUIS: Tableau TRIÉ

```java
// Java
int binarySearch(int[] arr, int target) {
    int lo = 0, hi = arr.length - 1;

    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;  // Éviter overflow

        if (arr[mid] == target) return mid;
        if (arr[mid] < target) lo = mid + 1;
        else hi = mid - 1;
    }
    return -1;
}
```

```python
# Python
def binary_search(arr, target):
    lo, hi = 0, len(arr) - 1

    while lo <= hi:
        mid = lo + (hi - lo) // 2

        if arr[mid] == target:
            return mid
        if arr[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1
```

Complexité: O(log n)

Algorithmes de Tri

```java
// Java
void bubbleSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = 0; j < arr.length - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```python
# Python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Bubble Sort - O(n²)

```java
// Java
void quickSort(int[] arr, int lo, int hi) {
    if (lo < hi) {
        int p = partition(arr, lo, hi);
        quickSort(arr, lo, p - 1);
        quickSort(arr, p + 1, hi);
    }
}

int partition(int[] arr, int lo, int hi) {
    int pivot = arr[hi];
    int i = lo - 1;

    for (int j = lo; j < hi; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, hi);
    return i + 1;
}
```

```python
# Python
def quick_sort(arr, lo, hi):
    if lo < hi:
```

```
        p = partition(arr, lo, hi)
        quick_sort(arr, lo, p - 1)
        quick_sort(arr, p + 1, hi)

def partition(arr, lo, hi):
    pivot = arr[hi]
    i = lo - 1

    for j in range(lo, hi):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[hi] = arr[hi], arr[i + 1]
    return i + 1
```

Quick Sort - O(n log n) moyen

Comparaison Tri

Mnémonique: "**B**ubble **S**low, **Q**uick **F**ast, **M**erge **S**table"

Tri	Meilleur	Moyen	Pire	Stable?
Bubble	O(n)	O(n^2)	O(n^2)	Oui
Selection	O(n^2)	O(n^2)	O(n^2)	Non
Insertion	O(n)	O(n^2)	O(n^2)	Oui
Quick	O(n log n)	O(n log n)	O(n^2)	Non
Merge	O(n log n)	O(n log n)	O(n log n)	Oui

Parcours de Graphes

BFS (Breadth-First Search) **Mnémonique**: "**B**FS = **B**roader = **Q**ueue = Niveau par niveau"

```java
// Java
void BFS(Map<Integer, List<Integer>> graph, int start) {
    Set<Integer> visited = new HashSet<>();
    Queue<Integer> q = new LinkedList<>();

    visited.add(start);
    q.add(start);

    while (!q.isEmpty()) {
        int node = q.poll();
        System.out.print(node + " ");

        for (int neighbor : graph.get(node)) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                q.add(neighbor);
            }
        }
    }
}
```

```python
# Python
from collections import deque
```

```python
def bfs(graph, start):
    visited = {start}
    q = deque([start])

    while q:
        node = q.popleft()
        print(node, end=" ")

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                q.append(neighbor)
```

Complexité: O(V + E)
Usage: Plus court chemin (non pondéré)

DFS (Depth-First Search) **Mnémonique**: "**D**FS = **D**eeper = **S**tack/Récursion = Profondeur"

```java
// Java
void DFS(Map<Integer, List<Integer>> graph, int node, Set<Integer> visited) {
    visited.add(node);
    System.out.print(node + " ");

    for (int neighbor : graph.get(node)) {
        if (!visited.contains(neighbor)) {
            DFS(graph, neighbor, visited);
        }
    }
}
```

```python
# Python
def dfs(graph, node, visited):
    visited.add(node)
    print(node, end=" ")

    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
```

Complexité: O(V + E)
Usage: Détection de cycles, composantes connexes

BACKEND & NETWORKING

Définitions Express

Backend: Partie serveur gérant logique métier, données, authentification.

API REST: Interface utilisant HTTP pour communication client-serveur.

Networking: Protocoles et technologies pour communication entre systèmes.

REST API

Méthodes HTTP **Mnémonique**: "**G**irl **P**uts **P**ink **P**ants **D**aily"

Méthode	Action	Exemple	Idempotent?
GET	Lire	GET /comptes/123	Oui
POST	Créer	POST /comptes	Non
PUT	Remplacer	PUT /comptes/123	Oui
PATCH	Modifier	PATCH /comptes/123	Non
DELETE	Supprimer	DELETE /comptes/123	Oui

Codes de Statut **Mnémonique**: “**2**00 **S**uper, **4**00 **C**lient **E**rror, **5**00 **S**erveur **E**rror”

Code	Signification	Quand
200	OK	GET/PUT/PATCH réussi
201	Created	POST réussi
204	No Content	DELETE réussi
400	Bad Request	Données invalides
401	Unauthorized	Non authentifié
403	Forbidden	Pas autorisé
404	Not Found	Ressource inexistante
500	Internal Error	Erreur serveur

Modèle OSI - 7 Couches

Mnémonique: “**P**lease **D**o **N**ot **T**hrow **S**ausage **P**izza **A**way”

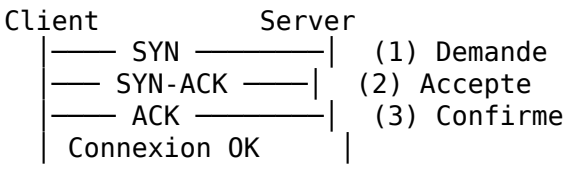
#	Couche	Protocoles	Équipement
7	**A**pplication	HTTP, HTTPS, FTP, DNS	-
6	**P**résentation	SSL/TLS, JPEG	-
5	**S**ession	NetBIOS, RPC	-
4	**T**ransport	TCP, UDP	-
3	**N**etwork	IP, ICMP	Routeur
2	**D**ata Link	Ethernet	Switch
1	**P**hysique	Câbles, Wi-Fi	Hub

TCP vs UDP

Mnémonique: “**TCP** = **T**rusty **C**onnection **P**erfect, **UDP** = **U**ltra **D**irect **P**acket”

TCP	UDP
Connexion établie	Pas de connexion
Fiable (garanties)	Pas de garantie
Plus lent	Plus rapide
Web, Email, FTP	Streaming, VoIP, DNS

TCP 3-Way Handshake



HTTP vs HTTPS

HTTP	HTTPS
Port 80	Port 443
Non chiffré	Chiffré (SSL/TLS)
Vulnérable	Protégé

Ports Courants

Mnémonique: “**22** SSH, **80** HTTP, **443** HTTPS, **3306** MySQL”

Port	Service
20/21	FTP
22	SSH
25	SMTP (email)
53	DNS
80	HTTP
443	HTTPS
3306	MySQL
5432	PostgreSQL

FRONTEND

Définitions Express

Frontend: Partie client avec laquelle l’utilisateur interagit (HTML, CSS, JS).

DOM: Représentation en arbre d’un document HTML manipulable via JavaScript.

Event: Action détectable (click, submit) déclenchant du code.

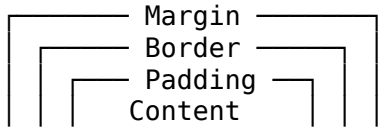
HTML5 - Balises Sémantiques

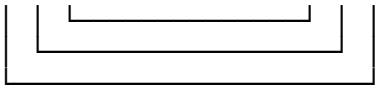
Mnémonique: “**H**eader **N**av **M**ain **S**ection **A**rticle **A**side **F**ooter”

Balise	Usage
`<header>`	En-tête
`<nav>`	Navigation
`<main>`	Contenu principal
`<section>`	Section thématique
`<article>`	Contenu autonome
`<aside>`	Contenu annexe
`<footer>`	Pied de page

CSS - Box Model

Mnémonique: “**M**aman **B**orde **P**etit **C**afé” (MBPC)





```css
.box {
    width: 300px;      /* Content */
    padding: 20px;     /* Espace interne */
    border: 2px solid; /* Bordure */
    margin: 15px;      /* Espace externe */
}
```

CSS - Flexbox

```css
.container {
    display: flex;
    justify-content: space-between;  /* Horizontal */
    align-items: center;             /* Vertical */
    gap: 20px;                       /* Espace */
}
```

Responsive - Media Queries

```css
/* Mobile first */
.container { width: 100%; }

/* Tablette (768px+) */
@media (min-width: 768px) {
    .container { width: 750px; }
}

/* Desktop (1024px+) */
@media (min-width: 1024px) {
    .container { width: 960px; }
}
```

JavaScript - DOM

```javascript
// Par ID
const el = document.getElementById('id');

// Par classe (premier)
const el = document.querySelector('.classe');

// Par classe (tous)
const els = document.querySelectorAll('.classe');
```

Sélection

```javascript
// Contenu
el.textContent = 'Texte';
el.innerHTML = '<b>HTML</b>';

// Classes
```

```
el.classList.add('actif');
el.classList.remove('inactif');
el.classList.toggle('visible');

// Style
el.style.color = 'red';

// Créer et ajouter
const div = document.createElement('div');
div.textContent = 'Nouveau';
parent.appendChild(div);

// Supprimer
el.remove();
```

Manipulation

```
// Click
btn.addEventListener('click', function(e) {
    console.log('Cliqué');
});

// Submit formulaire
form.addEventListener('submit', function(e) {
    e.preventDefault();  // Empêcher rechargement
    const val = input.value;
    // Traiter...
});

// Input (temps réel)
input.addEventListener('input', function(e) {
    console.log(e.target.value);
});
```

Événements

Fetch API

```
// GET
async function charger() {
    try {
        const response = await fetch('/api/comptes');
        if (!response.ok) throw new Error('Erreur ' + response.status);
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}

// POST
async function creer(data) {
    try {
        const response = await fetch('/api/comptes', {
            method: 'POST',
```

```
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(data)
        });
        const result = await response.json();
        return result;
    } catch (error) {
        console.error(error);
    }
}
```

Sécurité

XSS (Cross-Site Scripting) Prévention:

```
//   DANGEREUX
el.innerHTML = userInput;

//   SÛR
el.textContent = userInput;  // Échappe automatiquement
```

CSRF (Cross-Site Request Forgery) Prévention:

```
// Token CSRF dans requêtes
const token = document.querySelector('meta[name="csrf-token"]').content;

fetch('/api/action', {
    method: 'POST',
    headers: { 'X-CSRF-Token': token },
    body: JSON.stringify(data)
});
```

```
//   DANGEREUX - localStorage vulnérable XSS
localStorage.setItem('token', 'secret');

//   PRÉFÉRÉ - Cookies HttpOnly (côté serveur)
// Set-Cookie: token=...; HttpOnly; Secure; SameSite=Strict
```

Stockage sécurisé

TECHNIQUES MNÉMOTECHNIQUES AVANCÉES

1. Acronymes Essentiels

Mnémonique global: “**AC**tive **SO**ft **OP**tion”

- **ACID**: Atomicity, Consistency, Isolation, Durability
- **SOLID**: SRP, OCP, LSP, ISP, DIP
- **OSI**: Please Do Not Throw Sausage Pizza Away
- **HTTP**: GET, POST, PUT, PATCH, DELETE
- **DOM**: Document Object Model

2. Histoires Mnémotechniques

Histoire ACID (Banque)

```
Alice (Atomicity) va à la banque.
Elle Crée (Consistency) un virement.
Isolément (Isolation), personne n'interfère.
Durablement (Durability), c'est enregistré forever.
```

Histoire SOLID (Construction)

```
Un architecte (S)ingle construit une maison.
Ouverte (O)pen pour extensions futures.
Les fondations (L)iskov sont solides.
Interfaces (I)SP petites et précises.
Dépendances (D)IP abstraites, pas concrètes.
```

3. Visualisation Spatiale

Palais mental: Votre maison

- **Entrée** = SQL & ACID
- **Salon** = POO (4 canapés = 4 piliers)
- **Cuisine** = Algorithmes (mixeur = tri, frigo = stockage)
- **Chambre 1** = Structures (armoire = stack, tiroirs = array)
- **Chambre 2** = UML (miroir = diagrammes)
- **Balcon** = Backend (vue extérieure = API)
- **Jardin** = Frontend (fleurs = HTML, arrosage = CSS, soleil = JS)

4. Associations Visuelles

Structures de données

- **Stack** = Pile d'assiettes
- **Queue** = File d'attente
- **Tree** = Arbre généalogique
- **HashMap** = Dictionnaire
- **LinkedList** = Train

Protocoles

- **TCP** = Lettre recommandée (fiable)
- **UDP** = Carte postale (rapide, pas garanti)
- **HTTP** = Conversation normale
- **HTTPS** = Conversation chiffrée

5. Règle des 3R

Pour chaque concept, appliquez: 1. **Révision** - Lire maintenant 2. **Répétition** - Relire dans 1 heure 3. **Rappel** - Tester sans notes dans 24h

6. Méthode Feynman (Simplification)

Expliquez chaque concept comme à un enfant de 10 ans: - **Base de données** = "Classeur magique qui ne perd jamais rien" - **POO** = "Boîtes avec instructions à l'intérieur" - **Algorithme** = "Recette de cuisine pour ordinateur" - **API** = "Serveur de restaurant qui prend les commandes"

7. Associations Chiffrées

Codes HTTP

- **200** = 2 yeux OK
- **404** = 4 coins, 0 trouvé, 4 directions = perdu
- **500** = 5 doigts cassés = serveur cassé

Ports

- **80** = 8 ressemble à lunettes = HTTP (voir)
- **443** = 4+4+3=11 lettres dans HTTPS
- **3306** = 33 âge moyen + 06 juin = MySQL birthday

8. Phrases Mémo

Complexités

```
"1 LOG Noir N'a Rien E²xagéré"
O(1) < O(log n) < O(n) < O(n²) < O(2^n)
```

JOINs SQL

```
"INNER INtersection, LEFT tout LEft, RIGHT tout Right"
```

Traversées BST

```
"INorder = INtérieur trié
PREorder = PREfixe racine
POSTorder = POSTpone racine"
```

9. Cartes Mentales Rapides

SQL

```
SQL
├── ACID (4 propriétés)
├── JOINs (3 types)
├── Normalisation (3 formes)
└── Index (performance)
```

POO

```
POO
├── 4 Piliers (EHPA)
├── SOLID (5 principes)
└── Patterns (Singleton, Factory, etc.)
```

10. Technique Pomodoro pour Révision

Session de 25 minutes: - 10 min: Lire section - 10 min: Faire exercices mentaux - 5 min: Réciter sans notes

Pause 5 minutes: Marcher, visualiser

Répéter 4 fois = 2 heures de révision intensive

CHECKLIST FINALE DE RÉVISION

SQL & BDD

- ☐ Réciter ACID avec exemples
- ☐ Dessiner diagramme Venn des JOINs
- ☐ Expliquer 1NF, 2NF, 3NF
- ☐ Écrire transaction complète

POO & SOLID

- ☐ Nommer les 4 piliers POO
- ☐ Réciter les 5 principes SOLID
- ☐ Coder un Singleton
- ☐ Expliquer Factory pattern

UML

- ☐ Dessiner diagramme de classes avec relations
- ☐ Différencier include vs extend
- ☐ Dessiner séquence avec alt

Structures

- ☐ Citer complexités de 6 structures
- ☐ Coder Stack et Queue
- ☐ Implémenter BST insert/search
- ☐ Expliquer les 3 traversées

Algorithmes

- ☐ Coder binary search
- ☐ Coder quick sort
- ☐ Différencier BFS et DFS
- ☐ Comparer complexités de tri

Backend & Network

- ☐ Réciter 5 méthodes HTTP
- ☐ Mémoriser 8 codes de statut
- ☐ Réciter OSI avec mnémonique
- ☐ Comparer TCP vs UDP
- ☐ Lister 8 ports courants

Frontend

- ☐ Lister 7 balises sémantiques
- ☐ Dessiner Box Model
- ☐ Sélectionner éléments DOM (5 méthodes)
- ☐ Coder Fetch GET et POST
- ☐ Expliquer XSS et CSRF

PLAN DE RÉVISION 2-3 HEURES

Heure 1: Fondamentaux (SQL, POO, UML)

- **0-20 min**: SQL (ACID, JOINs, normalisation)
- **20-40 min**: POO (4 piliers, SOLID)
- **40-60 min**: UML (3 diagrammes essentiels)

Heure 2: Structures & Algorithmes

- **0-30 min**: Structures (Stack, Queue, HashMap, BST)
- **30-60 min**: Algorithmes (Binary Search, Tri, BFS/DFS)

Heure 3: Technologies Web

- **0-20 min**: Backend (REST, Codes HTTP)
- **20-40 min**: Networking (OSI, TCP/UDP, Ports)
- **40-60 min**: Frontend (HTML, CSS, JS, Sécurité)

Session Express (90 min)

Si moins de temps, prioriser: 1. **SQL ACID** (10 min) 2. **SOLID** (10 min) 3. **Diagramme de classes** (10 min) 4. **Complexités structures** (15 min) 5. **Binary Search + Quick Sort** (15 min) 6. **Codes HTTP + OSI** (15 min) 7. **DOM + Fetch** (15 min)

CONSEILS DE DERNIÈRE MINUTE

Avant l'examen

1. Dormir 7-8 heures
2. Petit-déjeuner léger
3. Relire mnémoniques seulement
4. Arriver 15 min en avance
5. Respirer profondément 3x

Pendant l'examen

1. Lire TOUTES les questions d'abord
2. Commencer par les plus faciles
3. Dessiner pour visualiser (UML, graphes)
4. Vérifier la syntaxe du code
5. Relire les réponses si temps

En cas de trou de mémoire

1. Utiliser les mnémoniques
2. Visualiser le palais mental
3. Reconstruire logiquement
4. Passer et revenir plus tard
5. Rester calme, respirer

MESSAGE FINAL

Vous avez maintenant tous les outils essentiels!

Les 3 piliers du succès

1. **Compréhension** - Pas de mémorisation aveugle
2. **Pratique** - Coder sur papier
3. **Confiance** - Vous êtes prêt(e)!

Rappels importants

- La qualité > quantité
- Dessinez les diagrammes
- Expliquez à voix haute
- Faites des pauses
- Croyez en vous!

En cas de difficulté

> “Je ne connais pas tout, mais je comprends les fondamentaux et je peux raisonner logiquement.”

Bonne chance! Vous allez réussir!