Paradigmes de programmation

Table des matières

Introduction

Un paradigme est une façon d'approcher la programmation informatique en traitant d'une certaine manière les solutions aux problèmes.

Les paradigmes de programmation permettent de classer les langages de programmation en fonction de leurs caractéristiques. Un paradigme de programmation offre au développeur une manière de réfléchir à son programme et conditionne donc les solutions envisageables à un problème que le développeur doit résoudre. Ainsi, deux langages suivant deux paradigmes de programmation permettront de trouver deux solutions différentes à un même problème.

I La programmation impérative

Objectif

- Comprendre la programmation impérative.

Mise en situation

Il est courant de faire l'analogie entre un algorithme et une recette de cuisine, pour expliquer qu'un code est une série d'instructions à respecter pour obtenir un résultat donné. La programmation impérative est ce qui se rapproche le plus d'une recette de cuisine. Toutes les instructions sont exécutées les unes après les autres, et mènent à un résultat bien défini.

Programmation impérative Az Définition

Ce paradigme décrit les opérations d'un programme comme des séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme. L'état du programme est ce que le programme doit garder dans sa mémoire au cours de son exécution (numéro de ligne courant, valeur des variables, etc.).

Ainsi, avec ce paradigme, c'est au développeur d'écrire les instructions qu'il faut pour modifier l'état du programme de la bonne manière. Autrement dit, la programmation impérative se concentre sur la description de **comment** un programme doit travailler pour résoudre un problème.

💬 Remarque

C'est le paradigme de programmation le plus ancien et il est retrouvé dans les jeux d'instructions des processeurs et dans les langages les plus utilisés aujourd'hui.

Fondations Fondamental

Les langages impératifs comportent tous ces instructions de bases :

- **Assignation** (ou affectation) : permet de stocker en mémoire (dans une variable) le résultat d'une opération.
- **Condition** : permet d'exécuter un bloc d'instructions si une condition prédéterminée est réalisée.
- **Boucle** : permet de répéter un bloc d'instructions un nombre prédéfini de fois ou jusqu'à ce qu'une condition soit remplie.
- **Branchement** : permet à la séquence d'exécution d'être transférée ailleurs dans le programme (goto).
- **Séquence d'instructions** : désigne le fait d'exécuter, en séquence, plusieurs des instructions ci-dessus.

C

Exemple

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int a = 20;
    int b = 22;

    res = a + b;
    printf("Le résultat est %d\n", res);

    return 0;
}
```

Python

Exemple

```python
a = 20
b = 22

res = a + b
print("Le résultat est %d" % res)
```

PHP

Exemple

```php
<?php
    $a = 20;
    $b = 22;

    $res = $a + $a;
    echo 'Le résultat est $res';
?>
```

JavaScript

Exemple

```javascript
const a = 20
const b = 22

const res = a + b
console.log('Le résultat est ' + res)
```

À retenir

- La programmation impérative permet de donner des instructions précises à une machine et modifie son état global.
- Les langages les plus connus partagent un aspect impératif.

II Exercice

Question

[solution n°1 p. 28]

Avec Repl.it[1], exécuter le code JavaScript suivant :

```
const names = ['Alice', 'Bob', 'Charlie']
for (let i = 0; i < names.length; i++) {
  console.log(names[i][0])
}
```

Que fait-il ?

1. https://repl.it/

III Programmation déclarative

Objectif

- Comprendre les fondements de la programmation déclarative.

Mise en situation

Lorsque l'on programme, il est parfois nécessaire d'obtenir un résultat précis, indépendamment de l'état actuel du programme ou de la machine. C'est à dire que l'on souhaite que la machine réalise une tâche, sans avoir besoin de décrire les différentes opérations nécessaires pour y arriver.

Les langages de programmation **déclaratifs** permettent de décrire précisément le résultat que l'on souhaite, tout en laissant la gestion du "comment nous voulons l'obtenir" à l'ordinateur.

Programmation déclarative — Az Définition

C'est un paradigme qui consiste à créer des programmes sur la base de composants **indépendants** du contexte et sans état. Cette forme de programmation cherche à réduire les effets de bord en **décrivant la tâche** que le programme doit accomplir au lieu de décrire **comment** le programme doit accomplir cette tâche. Ainsi, on peut retrouver une correspondance claire entre ce type de langage et la logique mathématique.

SQL — Exemple

Les requêtes SQL de type SELECT sont déclaratives.

```sql
CREATE TABLE users (
    id    INTEGER,
    name  VARCHAR(50),
    PRIMARY KEY(id)
);

INSERT INTO users (id, name)
VALUES (1, 'Luke');

INSERT INTO users (id, name)
VALUES (2, 'Anakin');

SELECT id, name
FROM users;
```

Ce code permet de déclarer à la machine que l'on souhaite créer, insérer puis récupérer les champs `id` et `name` de la table `users`, sans pour autant lui décrire **comment** y arriver. Le développeur ici ne s'intéresse pas au contexte ou à l'état du programme qui enverra le résultat.

Avec un langage impératif, le développeur aurait besoin d'écrire les instructions pour chercher parmi toutes les tables et trouver les bons résultats.

Programmation logique

Les langages **logiques** peuvent être considérés comme des langages déclaratifs, dans le sens où ils permettent de déclarer des vérités (prémisses) pour les mettre en relations et ainsi créer des programmes pour interroger ces vérités. Le développeur n'a pas à donner d'instructions précises mais doit cependant pouvoir formuler ses déclarations de la bonne manière (ce qui peut conduire à des effets de bord).

Prolog

Exemple

On déclare d'abord une base de faits (les vérités). Dans cet exemple, on indique qu'Alice habite en France et que Bob habite en Angleterre.

```
country(alice,france).
country(bob,angleterre).
```

On peut ensuite interroger cette base de fait, en demandant si Alice habite en France et si Bob habite en France.

```
| ?- country(alice,france).
yes.
| ?- country(bob,france).
no.
```

Il est aussi possible de déclarer une base de règles, c'est-à-dire des vérités générales. En effet, un développeur pourrait ajouter que deux personnes qui n'habitent pas le même pays ne sont pas voisins.

La manière avec laquelle le programme répond aux questions n'est pas importante et le développeur ne veut que le résultat.

À retenir

- La programmation déclarative sert à accomplir des tâches sans devoir exprimer comment accomplir ces tâches.
- Le développeur se concentre sur le **quoi** et pas le **comment**.
- La programmation logique peut être vu comme une forme de programmation déclarative.

IV Exercice

Ce code SQL crée un espace de stockage pour des utilisateurs ayant un e-mail et un mot de passe. L'exécuter dans un repl `SQLite`.

```sql
CREATE TABLE users (
  email       VARCHAR(50),
  pass_hash   VARCHAR(256),
  PRIMARY KEY(email)
);

INSERT INTO users (email, pass_hash) VALUES
('luke.skywalker@republic.ga', 'SECRET_PASS'),
('darth.vador@empire.ga', 'SECRET_PASS');

SELECT email FROM users;
```

Question 1

[solution n°2 p. 28]

Qu'affiche ce code lors de son exécution ?

Question 2

[solution n°3 p. 28]

En quoi ce code est déclaratif ?

V Programmation objet

Objectif

- Se familiariser avec la programmation objet.

Mise en situation

Il est parfois plus intéressant, ou plus efficace et intuitif, de représenter des concepts, idées ou objets du monde physique dans un programme.

La programmation orientée objet

La programmation impérative sépare le comportement d'un programme de son état, et la programmation déclarative ne présente pas d'état.

Cependant, nous disposons d'un troisième paradigme de programmation qui nous permet de mélanger le comportement d'un programme avec son/ses état(s).

En d'autres termes, il est possible de créer des briques qui représentent une entité bien précise, et qui **encapsulent** son état **et** son comportement propres.

Voiture Exemple

Un exemple typique est la voiture.

Un développeur a besoin d'intégrer l'idée de voiture dans son programme, cela peut être pour un jeu vidéo de course, une simulation du trafic routier ou tout autre programme ayant besoin du concept de voiture.

Il va utiliser un langage orienté objet pour créer une classe « Voiture » ; cette classe va permettre de stocker deux choses :

- L'état d'une voiture : le développeur stocke dans l'état des informations sur une voiture, comme sa couleur, sa marque ou son nombre de chevaux. L'état d'un objet est donc composé de variables et peut être amené à évoluer. Ces variables sont appelées des attributs. Un objet peut aussi contenir d'autres objets. Dans le cas présent il pourrait y avoir un tableau de quatre variables contenant des objets « Roue ».
- Le comportement : c'est un ensemble de fonctions (appelées méthodes) propres à toutes les voitures et qui permettent de savoir ce qu'un objet peut faire. Dans le cas de la classe « Voiture », il pourrait y avoir une méthode démarrer, accélérer ou arrêter.

Les méthodes ont accès à l'état de l'objet et peuvent donc interagir avec.

Classe et objet Remarque

En programmation orientée objet, une **classe** représente la structure d'un objet. On y indique à quoi devrait ressembler l'état et on y implémente les méthodes propres à l'objet. Un **objet** représente une **instance** de la classe, un élément particulier.

Par exemple, la classe « Voiture » nous dit qu'une voiture a une couleur, une marque, un nombre de chevaux et nous indique le comportement d'une voiture via ses méthodes. Un objet « Voiture » pourrait une voiture rouge de la marque Renault avec une puissance de 5 chevaux. Lorsque la méthode « accélérer » de cet objet en particulier est appelée, elle aura accès à l'état de l'objet et pourra ainsi modifier une variable accélération.

Java Exemple

```java
public class Car {
    public String color;
    public String brand;
    private int horsepower;
    public int acceleration;
    public bool started;

    public Voiture(String color, String brand, int horsepower) {
        this.color = color;
        this.brand = brand;
        this.horsepower = horsepower;
        this.started = false;
        this.acceleration = 0;
    }

    public getHorsepower() {
        return this.horsepower;
    }

    public start() {
        this.started = true;
    }

    public accelerate() {
        this.acceleration = this.acceleration + 10;
    }

    public stop() {
        this.acceleration = 0;
        this.started = false;
    }

}
```

La première méthode « public Voiture » correspond au constructeur de la classe, c'est la méthode qui est appelée lorsqu'un objet est instancié.

```java
public static void main(String[] args) {
    Car myCar = new Car("red", "Renault", 5);
    System.out.println("La marque de ma voiture est" + myCar.brand);
    System.our.println("Elle a" + myCar.getHorsepower() + "chevaux");
}
```

Encapsulation

On remarque dans l'exemple précédent que les attributs ont pu être marqué publics ou privés. C'est une possibilité de Java qui permet d'intégrer une idée propre à la programmation objet : l'**encapsulation**.

C'est l'idée que certains éléments d'un objet ne peuvent être accessibles que par l'objet lui-même.

Dans l'exemple ci-dessus, l'attribut `chevaux` est à `private` et donc depuis l'extérieur de l'objet aucun élément n'y a accès. C'est pourquoi il y a une aussi une méthode `getChevaux` (qui elle est à `public`) qui renvoie la variable `chevaux`.

TypeScript Exemple

```typescript
class Car {
  public color: string;
  public brand: string;
  private horsepower: number;
  public acceleration: number;
  public started: boolean;

  public constructor(color: string, brand: string, horsepower: number) {
    this.color = color;
    this.brand = brand;
    this.horsepower = horsepower;
    this.acceleration = 0;
    this.started = false;
  }

  public getHorsepower() {
    return this.horsepower;
  }

  public start() {
    this.started = true;
  }


  public accelerate() {
    this.acceleration += 10;
  }


  public stop() {
    this.acceleration = 0;
    this.started = false;
  }
}
```

```typescript
let myCar = new Car("red", "Renault", 5);
console.log(`La marque de ma voiture est ${myCar.brand}`);
console.log(`Elle a ${myCar.getHorsepower()} chevaux`);
```

À retenir

- La programmation orientée objet permet de mélanger état et comportement d'un objet.
- Un objet représente un concept, une idée ou un élément du monde physique.
- La classe correspond à la structure générale de l'objet et la définition de ses méthodes, l'objet correspond à une instance de la classe.

VI Exercice

Sur repl.it[2], exécuter le code TypeScript suivant :

```typescript
class User {
  private name: string;
  private age: number;

  public constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  public getName() {
    return this.name;
  }

  public getAge() {
    return this.age;
  }
}

let alice = new User('Alice', 25);
console.log(`${alice.getName()} a ${alice.getAge()} ans`);
```

Question

[solution n°4 p. 28]

Faire une méthode `birthday` qui affiche « Joyeux anniversaire [name] ! ».

2. https://repl.it/

VII Programmation fonctionnelle

Objectif

- Se familiariser avec la programmation fonctionnelle.

Mise en situation

Certains algorithmes, comme par exemple des calculs mathématiques, n'ont pas besoin de prendre en compte un état quelconque pour être réalisé. Dans ce cas, le résultat final ne dépend que des données d'entrée, sur lesquelles sont appliquées une fonction. C'est ce que l'on appelle la programmation fonctionnelle, qui se concentre donc sur l'utilisation de fonctions.

Programmation fonctionnelle

Az Définition

C'est un dérivé de la programmation déclarative, ainsi on n'y retrouve pas d'état. Les données ne peuvent être manipulées que par des évaluations de fonctions mathématiques.

Le manque d'état induit une absence d'opération d'affectation. Cela permet aussi d'éviter tout effet de bord et permet de voir le programme comme une véritable application mathématique. Si un bug apparaît, il est très simple à repérer car il se trouve **forcément** dans la fonction ne présentant pas le résultat voulu. En effet, il n'y a pas de variables partagées ou d'état global.

Autrement dit, un langage fonctionnel offre la possibilité de passer par des fonctions pour atteindre le résultat voulu. Ces fonctions renverront **toujours** le même résultat pour les mêmes données en entrées sans modifier l'état potentiel qui se trouve à l'extérieur de ces fonctions.

JavaScript

Exemple

Retourner les nombres pairs d'un tableau :

```
1 function even (tab) {
2   return tab.filter(nb => nb % 2 === 0)
3 }
4
5 const someArray = [33, 42, 12, 99, 103, 188]
6 const evenNumbers = even(someArray)
7 console.log(evenNumbers)
8
```

La fonction `filter` applique à chaque élément de `tab` la fonction anonyme « *`nb => nb % 2 === 0`* ». Dans ce contexte là, la fonction anonyme doit donner une condition. La fonction `filter` crée un nouveau tableau (et donc ne modifie pas `tab`), et y insère les éléments de `tab` répondant à la condition de la fonction passée à `filter`.

En programmation impérative il aurait fallu écrire les instructions nécessaires pour parcourir le tableau manuellement et vérifier la condition pour chaque élément. Cela aurait demandé à modifier `tab` directement ou écrire le code pour créer un nouveau tableau. On voit bien comment la version fonctionnelle est moins prompte aux erreurs et effets de bords.

Haskell Exemple

Haskell est un langage purement fonctionnel.

```
1 filter even [42, 55, 76, 77, 109]
```

- Les nombres entre crochets forment un tableau d'entiers.
- *Filter* crée un nouveau tableau contenant les éléments du premier en remplissant la condition donnée en premier argument, ici *even*, qui teste si un nombre est pair.

Remarque

Haskell est disponible sur Repl.it[3].

À retenir

- La programmation fonctionnelle fait fi des états dans un programme, il n'y a pas de partage de mémoire.
- Elle se concentre sur les fonctions, qui ne présentent pas d'effet de bord.
- Les fonctions ne dépendent que des données en entrée.

3. https://repl.it/

Exercice

[solution n°5 p. 29]

Le code suivant est à exécuter :

```
1 const a = ['Alice', 'Bob', 'Charlie']
2 const b = a.reduce((obj, k, i) => { obj[i] = k; return obj }, {})
3 console.log(b)
```

Que contient la constante « *b* » à la fin de l'exécution ?

- **A** ['Alice', 'Bob', 'Charlie']
- **B** { '0': 'Alice', '1': 'Bob', '2': 'Charlie' }
- **C** {'Alice': 0, 'Bob': 1, 'Charlie: 2}
- **D** Aucune de ces réponses.

IX Programmation événementielle

Objectif

- Se familiariser avec la programmation événementielle.

Mise en situation

Il est courant qu'un programme interagisse avec l'extérieur en échangeant des données en entrée et en sortie. Cependant il peut aussi échanger des événements, et déclencher des actions en fonctions d'autres actions extérieures. C'est le cas des sites Web dynamiques, qui réagissent pour changer l'état ou l'apparence de la page, par exemple lorsque l'on clique sur un bouton. La programmation événementielle se focalise ainsi sur l'exécution de code en fonction d'événements, plutôt qu'une simple exécution séquentielle des instructions.

Programmation événementielle Az Définition

Ce paradigme est fondé sur les événements qui se produisent dans le programme. En d'autres termes, l'exécution du programme sera déterminée par ce qui se produit à un instant donné.

On en voit l'intérêt dans des programmes à interface graphique, où l'utilisateur peut utiliser sa souris ou son clavier pour déclencher une action.

De tels programmes ont généralement besoin de détecter ces événements et d'exécuter immédiatement les actions associées.

Boucle asynchrone Az Définition

Une boucle asynchrone tourne en arrière-plan du programme, c'est-à-dire que sa présence est non-bloquante, ainsi elle n'empêche pas d'autres parties du programme de tourner.

HTML + JavaScript Exemple

Soit ce fichier HTML :

```html
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Example</title>
    <script src="index.js"></script>
  </head>
  <body>
    <button id="button">Click Me</button>
  </body>
</html>
```

Et ce fichier JavaScript utilisé dans le fichier HTML :

```
1 function eventHandler (event) {
2   console.log('Button has been clicked!')
3 }
4
5 const button = document.getElementById('button')
6 button.addEventListener('click', eventHandler)
7
```

La fonction « événement » `eventHandler` est appelée lorsque que le bouton est cliqué.

Multi-paradigme ⊕ Complément

La plupart des langages sont multi-paradigmes, c'est-à-dire qu'ils supportent simultanément les paradigmes impératif, fonctionnel, événementiel, etc.

La plupart ont néanmoins une dominante forte, par exemple le déclaratif pour la C ou l'objet pour Java.

JavaScript est un langage multi-paradigme. TypeScript est un sur-ensemble de JavaScript encore plus multi-paradigme car il complète les lacunes objet de JavaScript, tout en permettant de continuer à écrire du JavaScript.

À retenir

- La programmation événementielle utilise les événements pour déterminer l'exécution d'un programme.
- Un événement peut être une action de l'utilisateur.

X Exercice

HTML, CSS, JS — Créer un repl HTML, CSS, JavaScript et exécuter le code ci-après.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Example</title>
</head>
<body>

    <input type="text" id="text" />
    <p id="holder"></p>
</body>
<script>
    const text = document.getElementById("text");
    const holder = document.getElementById("holder");

    function updateHolder(event) {
        holder.innerText = text.value;
    }

    text.addEventListener("keyup", updateHolder);
</script>
</html>
```

Vous pouvez aussi copier le code HTML et JavaScript dans un fichier `exercice.html` et l'ouvrir avec un navigateur web.

Question 1

[solution n°6 p. 29]

Que se passe t-il lorsqu'on écrit dans le champ input ?

Question 2

[solution n°7 p. 29]

Pourquoi est-ce de la programmation événementielle ?

XI Essentiel

XII Quiz

Exercice 1 : Quiz - Culture

[solution n°8 p. 29]

Exercice

Qu'est-ce qu'un paradigme de programmation ?

- **A** Un langage de programmation
- **B** Une discipline du génie logiciel
- **C** Une façon d'approcher la programmation
- **D** Aucune de ces réponses

Exercice

Quel est le paradigme des jeux d'instructions des processeurs ?

- **A** Déclaratif
- **B** Événementiel
- **C** Fonctionnel
- **D** Impératif

Exercice

Cocher les paradigmes utilisables en JavaScript :

- **A** Fonctionnel
- **B** Événementiel
- **C** Impératif

Exercice

De quel paradigme peut-on dire que le fonctionnel dérive ?

- **A** Déclaratif

- **B** Impératif
- **C** Aucun

Exercice 6 : Quiz - Méthode

[solution n°9 p. 31]

Exercice

Dans quel paradigme est-il classique d'utiliser des boucles ?

- **A** Déclaratif
- **B** Impératif
- **C** Paradigme à boucle
- **D** Aucune de ces réponses

Exercice

Dans quel(s) paradigme(s) la modification de l'état du programme est-elle proscrite ?

- **A** Déclaratif
- **B** Impératif
- **C** Objet
- **D** Fonctionnel

Exercice

Parmi ces programmes, lesquels sont adaptés à la programmation événementielle ?

- **A** Une interface graphique de commande d'articles
- **B** Un logiciel de conduite de voiture autonome
- **C** Un dispositif médical qui surveille la tension cardiaque
- **D** Un programme de résolution d'équations

Exercice 10 : Quiz - Code

[solution n°10 p. 32]

Exercice

Cocher les deux paradigmes importants pour développer un jeu vidéo :

- **A** Fonctionnel
- **B** Événementiel
- **C** Objet
- **D** Déclaratif

Exercice

Quel paradigme serait à privilégier pour développer les pilotes graphiques nécessaires à un jeu vidéo ?

- **A** Impératif
- **B** Déclaratif
- **C** Objet
- **D** Fonctionnel

XIII Exercice : Défi final

On veut écrire un algorithme qui indique si on pourra sortir en vélo demain selon la météo du jour (niveau du soleil S, température T et niveau de pluie P) et si la personne est déjà sortie aujourd'hui. Les niveaux sont entre 0 et 100.

L'algorithme est en deux parties. D'abord il décide s'il fera beau demain puis utilise cette information pour décider de la sortie ou non le lendemain.

Question 1

[solution n°11 p. 33]

Par défaut, il ne fera pas beau demain. Voici les conditions (il faut juste que l'une d'entre elles soit vraie) pour qu'il fasse beau :

- P > 70
- S > 50 et P < 30
- P < 70 et T > 20
- S + P < 50
- T > 30

En revanche, si la somme des niveaux de soleil et de pluie est supérieure à 150, il ne fera pas beau le lendemain même si l'une des conditions ci-dessus est vraie.

Enfin, s'il fait beau demain, l'algorithme affiche « *Sortie demain* ». S'il ne fait pas beau, l'algorithme affiche « *Sortie demain* » seulement si la personne n'est pas sortie aujourd'hui. Sinon, il affiche « *Pas sortie demain* ».

Écrire un algorithme permettant de décider si la personne doit sortir le lendemain.

Indice :

L'algorithme doit préciser les entrées qu'il reçoit, ainsi que les conditions sur ces entrées. Par exemple, une entrée nommée n qui doit être supérieure à 0 sera notée :

```
Entrées:
    n, un entier supérieur à 0
```

Indice :

L'algorithme devra tester chacune des conditions une par une. Vous pouvez utiliser la structure suivante pour évaluer les conditions :

```
Si <condition> alors:
    ...
FinSi
```

Indice :

Il est utile d'utiliser une variable `beauTempsDemain` qui stocke l'information du temps du lendemain et qui est modifiée par la série de conditions, avec la syntaxe suivante :

```
Attribuer à beauTempsDemain la valeur Vrai
```

ou

```
Attribuer à beauTempsDemain la valeur Faux
```

Indice :

Si la somme des niveaux de soleil et de pluie est supérieur à 150, il ne fera pas beau le lendemain quoi qu'il arrive. Cette condition est donc à tester en dernier.

Question 2

[solution n°12 p. 33]

Voici le squelette de l'algorithme en JavaScript.

Compléter le code JavaScript de la première partie de l'algorithme.

```
const S = Number(prompt('Entrer le niveau de soleil actuel'))
const T = Number(prompt('Entrer la température actuelle'))
const P = Number(prompt('Entrer le niveau de pluie actuel'))
const sortieAujourdhui = prompt('Êtes-vous sorti aujourd\'hui? (O ou N)') === 'O'

// Première partie
let beauTempsDemain = false

if (P > 70) {
  beauTempsDemain = true
}
if (...) {
   ...
}
if (...) {
   ...
}
if (...) {
   ...
}
if (...) {
   ...
}

if (...) {
   ...
}


// Deuxième partie de l'algorithme
...
```

Indice :

L'opérateur « *et* » en Javascript s'écrit &&.

Question 3

[solution n°13 p. 34]

Peut-on réduire la complexité en temps de cette première partie ?

Indice :

Est-il nécessaire d'avoir un **if** pour chaque condition ?

Question 4

[solution n°14 p. 34]

Traduire la seconde partie de l'algorithme.

Indice :

```
1 if (...) {...} else {...}
```

Indice :

Pour afficher un texte on peut utiliser l'instruction :

```
1 console.log("toto")
```

Question 5

[solution n°15 p. 34]

Voici les conditions météorologiques d'aujourd'hui : S = 50, T = 17, P = 38. Sachant que je suis sorti aujourd'hui, est-ce que je devrais sortir demain ?

Conclusion

En programmation, il existe différents paradigmes, c'est à dire différentes manière de dérouler un algorithme. Chaque paradigme permet ainsi de favoriser la résolution de problèmes bien spécifiques. Comme chaque langage implémente un ou plusieurs paradigmes, le choix d'un langage de programmation peut aussi se faire en fonction de son ou ses paradigmes. Tout langage de programmation n'est pas apte à résoudre tous les types de problèmes, c'est pourquoi il est courant d'en apprendre plusieurs.

Solutions des exercices

Solution n°1

[exercice p. 6]

Ce code affiche les initiales des noms du tableau `names` grâce à une boucle qui parcourt chaque élément du tableau.

C'est le développeur qui décide de la manière avec laquelle le résultat doit être calculé. Ici, il a choisi une boucle `for` et un compteur `i` pour parcourir le tableau. Il a aussi décidé d'utiliser la syntaxe `[]` plutôt qu'une fonction pré-définie pour récupérer l'initiale de chaque prénom.

Solution n°2

[exercice p. 9]

La liste des e-mails des utilisateurs gérés par la base de données.

Solution n°3

[exercice p. 9]

On ne fait que demander au système de gestion de base de données de stocker des informations pour les récupérer ultérieurement, cependant le code précédent n'explique en rien au système **comment** il doit procéder. On lui dit simplement : « récupère les e-mails des utilisateurs ».

Solution n°4

[exercice p. 13]

```
class User {
  private name: string;
  private age: number;

  public constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  public getName() {
    return this.name;
  }

  public getAge() {
    return this.age;
  }

  public birthday() {
    console.log(`Joyeux anniversaire ${this.name} !`);
  }
}

let alice = new User("Alice", 25);
console.log(`${alice.getName()} a ${alice.getAge()} ans`);
alice.birthday();
```

Solution n°5

[exercice p. 16]

Le code suivant est à exécuter :

```
1 const a = ['Alice', 'Bob', 'Charlie']
2 const b = a.reduce((obj, k, i) => { obj[i] = k; return obj }, {})
3 console.log(b)
```

Que contient la constante « *b* » à la fin de l'exécution ?

- **A** ['Alice', 'Bob', 'Charlie']
- **B** { '0': 'Alice', '1': 'Bob', '2': 'Charlie' }
- **C** {'Alice': 0, 'Bob': 1, 'Charlie: 2}
- **D** Aucune de ces réponses.

Ce code fonctionnel utilise la fonction `Array.reduce` qui prend en entrée une fonction ayant comme arguments un accumulateur `obj`, l'élément courant du tableau parcouru et son index. On voit bien ici l'utilisation de fonctions sans avoir à modifier des variables : ce code n'utilise que des constantes.

Solution n°6

[exercice p. 19]

Le paragraphe en dessous est mis à jour avec la nouvelle valeur du champ input.

Solution n°7

[exercice p. 19]

On a placé une brique logicielle qui s'occupe d'attendre qu'un événement se produise pour exécuter une fonction. Ici, l'événement se produit lorsqu'une touche a été pressée dans le champ input.

Solution n°8

[exercice p. 21]

Exercice

Qu'est-ce qu'un paradigme de programmation ?

- **A** Un langage de programmation
- **B** Une discipline du génie logiciel
- **C** Une façon d'approcher la programmation
- **D** Aucune de ces réponses

On retrouve les paradigmes dans les langages de programmation, ils permettent d'approcher la résolution de problèmes d'une certaine manière.

Exercice

Quel est le paradigme des jeux d'instructions des processeurs ?

- **A** Déclaratif
- **B** Événementiel
- **C** Fonctionnel
- **D** Impératif

Les instructions des processeurs sont exécutées les unes à la suite des autres, et modifient l'état global de la mémoire (numéro de ligne, valeur des registres, etc.) : ce sont des éléments de programmation impératives qui permettent de diriger les actions de l'ordinateur.

Exercice

Cocher les paradigmes utilisables en JavaScript :

- [x] **A** Fonctionnel
- [x] **B** Événementiel
- [x] **C** Impératif

JavaScript est multi-paradigmes et propose toutes ces approches.

Exercice

De quel paradigme peut-on dire que le fonctionnel dérive ?

- **A** Déclaratif
- **B** Impératif
- **C** Aucun

En programmation déclarative, on ne cherche pas à diriger la machine dans le but d'obtenir un résultat, on ne s'intéresse qu'à la **description** du résultat. Cependant, un langage déclaratif n'est pas forcément fonctionnel.

Solution n°9

[exercice p. 22]

Exercice

Dans quel paradigme est-il classique d'utiliser des boucles ?

- **A** Déclaratif
- **B** Impératif
- **C** Paradigme à boucle Ce paradigme n'existe pas.
- **D** Aucune de ces réponses

Les boucles permettent de contrôler le flux de l'exécution du programme : elles indiquent au programme où aller, et quand. En d'autres termes, elles indiquent **comment** résoudre un problème.

Exercice

Dans quel(s) paradigme(s) la modification de l'état du programme est-elle proscrite ?

- **A** Déclaratif
- **B** Impératif
- **C** Objet
- **D** Fonctionnel

Dans ces paradigmes, les variables sont immuables (constantes).

Exercice

Parmi ces programmes, lesquels sont adaptés à la programmation événementielle ?

- **A** Une interface graphique de commande d'articles
- **B** Un logiciel de conduite de voiture autonome

C Un dispositif médical qui surveille la tension cardiaque

D Un programme de résolution d'équations

Globalement, un développeur utilisera des méthodes de programmation événementielle lorsqu'il aura besoin que son programme réagisse à une entrée spécifique.

Solution n°10

[exercice p. 23]

Exercice

Cocher les deux paradigmes importants pour développer un jeu vidéo :

A Fonctionnel

B Événementiel

C Objet

D Déclaratif

Un jeu vidéo aura besoin d'une interface graphique (ou tout autre interface utilisée par le joueur) et aura probablement besoin de créer des objets ayant leurs propres état et comportement (joueurs, monstres, inventaire, etc.).

Exercice

Quel paradigme serait à privilégier pour développer les pilotes graphiques nécessaires à un jeu vidéo ?

A Impératif

B Déclaratif

C Objet

D Fonctionnel

Ce type de programmation bas niveau (proche du matériel) requiert de donner des instructions précises à l'ordinateur (envoyer telle donnée, lire à tel endroit de la mémoire, etc).

Solution n°11

[exercice p. 24]

```
Algo sortie_demain

Entrées:
    S, un entier entre 0 et 100
    T, un entier supérieur à -273
    P, un entier entre 0 et 100
    SortieAujourdhui, un booléen

Attribuer à beauTempsDemain la valeur Faux

Si P > 70 alors:
    Attribuer à beauTempsDemain la valeur Vrai
FinSi
Si S > 50 et T > 15 alors:
    Attribuer à beauTempsDemain la valeur Vrai
FinSi
Si P < 70 et T > 20 alors:
    Attribuer à beauTempsDemain la valeur Vrai
FinSi
Si S + P < 50 alors:
    Attribuer à beauTempsDemain la valeur Vrai
FinSi
Si T > 30 alors:
    Attribuer à beauTempsDemain la valeur Vrai
FinSi
Si S + P > 150 alors:
    Attribuer à beauTempsDemain la valeur Faux
FinSi

Si beauTempsDemain alors:
    Afficher "Sortie demain"
Sinon
    Si non sortieAujourdhui alors:
        Afficher "Sortie demain"
    Sinon:
        Afficher "Pas sortie demain"
```

Solution n°12

[exercice p. 25]

```
let beauTempsDemain = false

if (P > 70) {
  beauTempsDemain = true
}
if (S > 50 && T > 15) {
  beauTempsDemain = true
}
if (P < 70 && T > 20) {
  beauTempsDemain = true
}
if (S + P  < 50) {
  beauTempsDemain = true
}
if (T > 30) {
  beauTempsDemain = true
}

if (S + P > 150) {
```

```
  beauTempsDemain = false
}
```

Solution n°13

[exercice p. 25]

On pourrait factoriser les conditions avec des « ou » (`||` en JS) afin de n'avoir qu'un seul `if`.

Solution n°14

[exercice p. 26]

```
if (beauTempsDemain){
  console.log('Sortie demain')
} else {
  if (sortieAujourdhui) {
      console.log('Pas sortie demain')
  } else {
      console.log('Sortie demain')
  }
}
```

Solution n°15

[exercice p. 26]

Pas de sortie demain.