AP4B Final - Duration 1h30

Documents not allowed – paper dictionaries allowed

Each part must be returned on a separate copy !!!!!!

Part 1 : Java

Swap Methods (2 pts)

We consider the following 4 JAVA methods relating to the possibility of exchanging two integers. For each of them, give the result displayed by the call (on the right) and above all, justify your answer.

`public void swap1(int i, int j) {`
`  int k = i;`
`  i = j;`
`  j = k;`
`}`
	`int i = 1; int j = 2;`
	`swap1(i, j);`
	`System.out.println("i= "+ i + " j= "+ j);`

`public void swap2(Integer i, Integer j)`
`{`
`  Integer k = i;`
`  i = j;`
`  j = k;`
`}`
	`i = 1; j = 2;`
	`Integer I = i; Integer J = j;`
	`swap2(I, J);`
	`i = I.intValue(); j = J.intValue();`
	`System.out.println("i= "+ i + " j= "+ j);`

`public void swap3(A x, A y) {`
`  A z = x;`
`  x.a = y.a;`
`  y.a = z.a;`
`}`
	`A x = new A(); x.a = 1;`
	`A y = new A(); y.a = 2;`
	`swap3(x, y);`
	`System.out.println("x.a="+x.a+"y.a="+y.a);`

`public void swap4(int[] i, int[] j) {`
`  int k;`
`  k = i[0];`
`  i[0] = j[0];`
`  j[0] = k;`
`}`
	`int [] t1 = new int[1]; t1[0] = 1;`
	`int [] t2 = new int[1]; t2[0] = 2;`
	`swap4(t1, t2);`
	`System.out.println("t1="+t1[0]+"t2="+t2[0]);`

Lambdas (3,5 pts)

1. Write the Operation class (correctly) for this program to work:

```
Operation addition = (a, b) -> a + b;
Operation multiplication = (a, b) -> a * b;

System.out.println(addition.calculer(3, 5));
System.out.println(multiplication.calculer(3, 5));
```

2. Add an operator for the division by properly processing division by 0.
3. What does this code do and display ? :

```
List<Integer> list = Arrays.asList(1, 2, 3, 4);
list.stream().filter(n -> n % 2 == 0).forEach(System.out::println);
```

4. And this code (generated by AI) ? :

```
List<Produit> produits = Arrays.asList(
new Produit("PC", 1200),
new Produit("Clavier", 50),
new Produit("Téléphone", 600));

produits.stream().filter(p -> p.getPrix() > 100).map(p -> new Produit(p.getNom(),
p.getPrix() * 0.9)).sorted(Comparator.comparing(Produit::getPrix)).forEach(p ->
System.out.println(p.getNom() + " : " + p.getPrix()));

class Produit {
    String nom;
    double prix;
    Produit(String nom, double prix) {
        this.nom = nom;
        this.prix = prix;
    }
    public double getPrix() { return prix; }
    public String getNom() { return nom; }
}
```

Problem : Problem of the Dining Philosophers (**4.5 pts**)

In the problem of " dining philosophers ", there are five philosophers around a table and 5 forks placed on the table to the left and right of each philosopher. This is a typical resource allocation problem. Each philosopher has access to two ranges located on his left and right. One constraint is that a philosopher can only eat if he has grasped his two adjacent forks, left and right (one in each hand). When he has eaten, he puts down the 2 forks and then starts a discussion. When he has finished speaking, he decides to eat again. And the cycle repeats itself. We want to make a Java simulation program that simulates philosophers eating dinner by synchronized threads. Each philosopher must therefore try to get hold of the 2 forks, if he succeeds, he eats, puts down the two forks, and then discusses. And the eating-talking cycle resumes, endlessly. Note that all philosophers must eat fairly after a finite time.

In order to ensure proper operation, several solutions are considered for the choice of shared objects on which synchronization is carried out.

1) A first proposed solution can be to synchronize on the forks, each of which is shared between 2 philosophers. For example, each philosopher looks to see if his left fork is placed and therefore free, he grasps it if it is, otherwise he waits (on the fork monitor), once the left fork is in his hand, he proceeds in the same way with the right fork. How is this solution not correct? What is likely to happen? Give a specific example of unwanted operation.

2) A second solution proposed is to synchronize on a single shared object (let's take the Table) which manages access to the forks and memorizes the state of each fork (placed, taken). The 2 forks are set at the same time if they are free at the same time, otherwise you have to wait. The philosopher eats and then puts down the two forks at the same time.

Complete the Java simulation program below (Table class, Philosopher class, main()), where you find the dotted lines "**...**" ".

```
class Philosophe implements ...{
	int id;
	private Table table;

	public Philosophe(int id, Table table) {
		this.id = id;
		this.table = table;
		Thread t = new ... (this);
		t. ... ();
	}

	public void run() {
		while (true) {

			// Phase discussion
			System.out.println("I discuss " + id);

			// simulation d'un temps de traitement
			try { Thread.sleep ((int)(Math.random() * 1000)); }
			catch (InterruptedException e) {}

			System.out.println("I finish talking " + id);

			// Demander autorisation acces fourchettes G et D
			table.demanderFourchettesGD(...);

			// Phase manger
			System.out.println("I eat " + id);

			// simulation d'un temps de traitement
			try { Thread.sleep ((int)(Math.random() * 1000)); }
			catch (InterruptedException e) {}

			System.out.println("I finish eating " + id);

			// Liberer les 2 fourchettes
			table.reposerFourchettesGD(...);
		}
	}
}

class Table {
	private boolean[] fourchettePrise;

	public Table(int nbPhilo) {
		fourchettePrise = new boolean[nbPhilo];
		for (int i = 0; i < nbPhilo; i++) {
			fourchettePrise[i] = false;
		}
	}

	// Demander ressource
	public ... ... demanderFourchettesGD(int id) {
		// Determine id forks G et D (i.e. Left and Right)
		int idG = id;
		int idD = (id + 1) % fourchettePrise.length;// modulo the nb de forks

		//System.out.println("Id fourchettes " + idG + " " + idD);
	  try {
		while (fourchettePrise[idG] || fourchettePrise[idD]) {
			... . ... ();
```

```
                }
            }
            catch(InterruptedException e) {}

            fourchettePrise[idG] = true;
            fourchettePrise[idD] = true;
        }

        // Liberer ressource
        public ... ... reposerFourchettesGD(int id) {
                // Determiner id fourchettes G et D
                int idG = id;
                int idD = (id + 1) % fourchettePrise.length;
            fourchettePrise[idG] = false;
            fourchettePrise[idD] = false;
                ... . ... ();
        }
}

class PhiloDinant {
        public static void main(String args[]) {
                System.out.println("Philosophes dinant !!!");

                Table table = new Table(5);

                Philosophe p[] = new Philosophe[5];
                for (int i = 0; i < 5; i++) {
                        p[i] = new ... (i, table);
                }
        }
}
```

Part 2 : UML

Background

The Stranger **Things** series is coming to an end. Between fan expectations, production constraints and narrative coherence, the writing of the last script is a complex process that requires the intervention of several specialists.

This exercise proposes to model, using UML, an **automatic scenario generator** for the last episode of Stranger Things.

The screenplay is gradually built by several **specialized screenwriters**, each responsible for a specific type of information. All the writers work on **a shared script**.

Constraints

1. Only one screenwriter can edit the script at a time.
2. A screenwriter can only intervene:
    1. when the scenario is available,
    2. and when the script is waiting for an element corresponding to its specialty.
3. When a writer adds something:
    1. the script completes its text,
    2. it indicates the **next type of information expected**,
    3. then it waits for the next modification.

We have **5 screenwriters** :

1. A screenwriter who generates **names of main characters**.
2. A writer who generates **creature or antagonist names**.
3. A screenwriter who provides **emblematic locations**.
4. A screenwriter who provides **supernatural objects or powers**.
5. A screenwriter who describes the **destruction or consequences** caused by these powers.

Here is the general framework of a scenario. Items in square brackets [] must be generated by the corresponding writers.

In the final season of Stranger Things, ***[Hero Name]*** *and their friends return to face a new threat:* ***[Monster Name]****.The events take place in* ***[Location]****, where a mysterious* ***[Supernatural Object or Power]*** *begins to emerge. This power is capable of* ***[Destruction Description]****, putting the entire town at risk. With the help of* ***[Hero Name]****, the group fights back and manages to seal the rift. In the end,* ***[Hero Name]*** *realizes that the Upside Down is deeply connected to* ***[Location]****. At the end of the fights,* ***[Hero Name]*** *disappears. Is he/she dead?*

Question 1 – Competition issues (2 points)

The script is a shared resource modified by several screenwriters.

1. Identify potential **concurrency issues**.
2. Specify the **general mechanisms** for ensuring correct access to the scenario.

Question 2 – UML Class Diagram (4 points)

Make a **UML class diagram of** the Scenario Builder.

The diagram should show:

1. The class representing the scenario,
2. the screenwriters,
3. class relations,
4. abstract classes, specializations and relevant enumerations.

Question 3 – Scenario generation process (2 points)

Using a **suitable UML diagram**, describe the entire sequence of a scenario generation, from the first generated element to the final scenario.

You will specify:

1. the type of diagram chosen,
2. the role of the different actors.

Question 4 – Scenario states (2 points)

Using an **appropriate UML diagram**, represent:

1. the different possible states of the scenario,
2. the transitions between these states,
3. the events that trigger these transitions.

Question Bonus :

Propose changes to generate multiple scenarios in parallel