1. **Problem**
   Selon l'introduction du cours, quels sont les 3 "mots" qui résument la POO ?

   (a) Ranger (organiser le code)

   (b) Protéger (éviter les erreurs)

   (c) Généraliser (réutiliser le code)

   (d) Optimiser (améliorer les performances)

   (e) Compiler (exécuter le code)

   Solution
   Les 3 piliers sont : **Ranger** (organiser le code en regroupant données et comportements), **Protéger** (encapsuler pour éviter les erreurs), **Généraliser** (héritage et polymorphisme pour réutiliser).

   (a) Correct.

   (b) Correct.

   (c) Correct.

   (d) Incorrect.

   (e) Incorrect.

2. **Problem**
   Analysez le code suivant :

```
Etudiant e1 = new Etudiant("Alice", 20);
Etudiant e2 = new Etudiant("Bob", 22);
Etudiant e3 = e1;
```

   Combien d'objets sont créés en mémoire ?

   (a) 2 objets

   (b) 0 objet

   (c) 3 objets

   (d) 1 objet

   Solution
   2 objets sont créés en mémoire (avec les deux `new`).

   `e3` contient une **référence** vers le même objet que `e1`. Ce n'est pas une copie de l'objet, c'est la même adresse mémoire.

   (a) Correct.

   (b) Incorrect.

   (c) Incorrect.

   (d) Incorrect.

3. **Problem**
   Soit le code suivant :

```
Etudiant e1 = new Etudiant("Alice", 20);
Etudiant e3 = e1;
e3.age = 25;
```

Quelle sera la valeur de `e1.age` après exécution ?

Solution
`e1.age` vaudra **25**. Puisque `e1` et `e3` pointent vers le même objet, modifier via `e3` modifie l'objet que `e1` référence aussi.

4. **Problem**
Que se passe-t-il quand on écrit `Personne jean = new Personne();` ?

   (a) Java crée une variable locale sans allocer de mémoire

   (b) Java alloue de la mémoire, appelle le constructeur, et retourne une référence

   (c) Java copie une instance existante de Personne

   (d) Java vérifie si un objet Personne existe déjà

   Solution
   Avec `new` : 1. Java alloue de la mémoire dans le tas (heap) 2. Le constructeur de la classe est appelé pour initialiser l'objet 3. Une référence vers cet espace mémoire est retournée et stockée dans la variable

   (a) Incorrect.

   (b) Correct.

   (c) Incorrect.

   (d) Incorrect.

5. **Problem**
Que se passe-t-il si vous définissez un constructeur `Personne(String nom)` sans définir de constructeur sans paramètres ?

   (a) Java crée automatiquement un constructeur sans paramètres

   (b) Une exception sera levée à l'exécution

   (c) Le code fonctionne normalement

   (d) `new Personne()` ne compilera plus

   Solution
   Dès que vous définissez un constructeur, le constructeur par défaut (sans paramètres) **disparaît** !

   Si vous avez besoin des deux, vous devez définir explicitement le constructeur sans paramètres.

   (a) Incorrect.

   (b) Incorrect.

   (c) Incorrect.

   (d) Correct.

6. **Problem**
Que se passe-t-il à l'exécution du code suivant ?

```
Etudiant e = null;
System.out.println(e.nom);
```

   (a) Erreur de compilation

   (b) Affiche une chaîne vide

   (c) Lance une NullPointerException

(d) Affiche “null”

Solution
Le programme va **planter** avec une `NullPointerException`.

Message d’erreur : `Cannot invoke "..." because "e" is null`

On ne peut pas appeler de méthode ou accéder à un attribut sur `null` car il n’y a pas d’objet.

(a) Incorrect.
(b) Incorrect.
(c) Correct.
(d) Incorrect.

7. **Problem**
Soit le code suivant :

```
String a = new String("Hello");
String b = new String("Hello");
String c = a;
```

Quelles expressions retournent `true` ?

(a) a.equals(c)
(b) a == b
(c) a.equals(b)
(d) a == c

Solution

(a) `a == b` **false** (deux objets différents en mémoire)
(b) `a.equals(b)` **true** (même contenu “Hello”)
(c) `a == c` **true** (même référence, même objet)
(d) `a.equals(c)` **true** (même contenu)

(a) Correct.
(b) Incorrect.
(c) Correct.
(d) Correct.

8. **Problem**
Quel code est le plus sûr pour comparer une variable `nom` (potentiellement null) avec la chaîne “admin” ?

(a) `nom.equals("admin")`
(b) `nom.compareTo("admin") == 0`
(c) `"admin".equals(nom)`
(d) `nom == "admin"`

Solution
Mettre la constante à gauche : `"admin".equals(nom)`

Ainsi, même si `nom` est `null`, pas de crash (`equals` gère `null` en paramètre).

L’autre syntaxe `nom.equals("admin")` provoque une **NullPointerException** si `nom` est null.

(a) Incorrect.

(b) Incorrect.

(c) Correct.

(d) Incorrect.

9. **Problem**
Quels sont les modificateurs d'accès en Java ?

(a) protected

(b) static

(c) final

(d) public

(e) private

(f) (default/package)

Solution
Les 4 modificateurs d'accès sont : `private`, `public`, `protected` et (default/package-private).

`static` et `final` ne sont **PAS** des modificateurs d'accès, mais des modificateurs de comportement.

(a) Correct.

(b) Incorrect.

(c) Incorrect.

(d) Correct.

(e) Correct.

(f) Correct.

10. **Problem**
Un attribut déclaré `private` est accessible depuis :

(a) La classe elle-même uniquement

(b) Partout dans le programme

(c) La classe et le même package

(d) La classe et ses sous-classes

Solution

Modificateur	Classe	Package	Sous-classe	Partout
`private`				

(a) Correct.

(b) Incorrect.

(c) Incorrect.

(d) Incorrect.

11. **Problem**
Un attribut déclaré `protected` est accessible depuis :

(a) Les classes du même package

(b) Les sous-classes (même dans d'autres packages)

(c) La classe elle-même

(d) N'importe quelle classe de n'importe quel package

Solution
`protected` donne accès à : - La classe elle-même - Les classes du même package - Les sous-classes (même dans d'autres packages)

C'est plus permissif que `private` mais moins que `public`.

(a) Incorrect.

(b) Correct.

(c) Correct.

(d) Incorrect.

12. **Problem**
À quoi sert l'annotation `@Override` ?

(a) Elle permet au compilateur de vérifier qu'on redéfinit bien une méthode du parent

(b) Elle rend la méthode publique

(c) Elle améliore les performances de la méthode

(d) Elle est obligatoire pour redéfinir une méthode

Solution
L'annotation `@Override` indique au compilateur qu'on redéfinit une méthode de la classe mère.

Avantages : - **Sécurité** : le compilateur vérifie que la méthode existe bien dans la classe mère (détecte les fautes de frappe) - **Lisibilité** : le code est plus clair pour le lecteur

(a) Correct.

(b) Incorrect.

(c) Incorrect.

(d) Incorrect.

13. **Problem**
Où doit se trouver l'appel à `super()` dans un constructeur de classe fille ?

(a) Après l'initialisation des attributs de la classe fille

(b) En dernière instruction du constructeur

(c) En première instruction du constructeur

(d) N'importe où dans le constructeur

Solution
`super()` doit être la **première instruction** du constructeur.

Le parent doit être construit avant l'enfant. Si vous oubliez `super()`, Java appelle automatiquement `super()` sans argument (ce qui échoue si le parent n'a pas de constructeur sans paramètres).

(a) Incorrect.

(b) Incorrect.

(c) Correct.

(d) Incorrect.

14. **Problem**
Soit `abstract class Forme { abstract double calculerSurface(); }`. Quel code est valide ?

(a) `Forme f = new Forme();`

(b) `Forme f = new Cercle(5.0);` (si Cercle extends Forme)

(c) `Forme f = new Forme() {};`

(d) Les trois sont valides

Solution
On ne peut **pas** instancier une classe abstraite avec `new Forme()`.

Mais on peut déclarer une variable de type `Forme` et l'initialiser avec une instance d'une classe concrète qui hérite de `Forme`.

(a) Incorrect.

(b) Correct.

(c) Incorrect.

(d) Incorrect.

15. **Problem**
En Java, une classe peut hériter de combien de classes ?

(a) Deux classes maximum

(b) Aucune, l'héritage n'existe pas en Java

(c) Une seule classe

(d) Autant de classes que souhaité

Solution
En Java, une classe ne peut hériter que d'**une seule classe** (héritage simple).

C'est pour éviter le "problème du diamant" qui existe en C++ avec l'héritage multiple.

Pour "simuler" l'héritage multiple, on utilise les **interfaces**.

(a) Incorrect.

(b) Incorrect.

(c) Correct.

(d) Incorrect.

16. **Problem**
Pour chaque affirmation, dites si elle est vraie :

(a) "Une Voiture est un Moteur" Héritage

(b) "Un Maître a un Animal" Composition

(c) "Un Rectangle a une Forme" Composition

(d) "Un Chien est un Animal" Héritage

(e) "Une Voiture a un Moteur" Composition

Solution

- "Un Chien **est un** Animal" **Héritage**
- "Une Voiture **a un** Moteur" **Composition**
- "Une Voiture **est un** Moteur" Faux ! (n'a pas de sens)

- “Un Maître **a un** Animal” **Composition**
- “Un Rectangle **est une** Forme” (pas “a une”) **Héritage**

(a) Incorrect.

(b) Correct.

(c) Incorrect.

(d) Correct.

(e) Correct.

17. **Problem**
Comment choisir entre héritage et composition ?

(a) Utiliser l’héritage pour les attributs, la composition pour les méthodes

(b) Toujours utiliser la composition, c’est plus moderne

(c) Si “X est un Y” Héritage ; Si “X a un Y” Composition

(d) Toujours utiliser l’héritage, c’est plus simple

Solution
La règle d’or : - Si on peut dire “X **est un** Y” (relation d’identité) **Héritage** - Si on peut dire “X **a un** Y” (relation de possession) **Composition**

Exemple : “Une Voiture est un Moteur” sonne faux, donc pas d’héritage. “Une Voiture a un Moteur” sonne juste, donc composition.

(a) Incorrect.

(b) Incorrect.

(c) Correct.

(d) Incorrect.

18. **Problem**
Quel mot-clé utilise-t-on pour implémenter une interface ?

(a) extends

(b) uses

(c) implements

(d) inherits

Solution
On utilise `implements` pour implémenter une interface :

```
class SiegeAuto implements ArticleLouable { ... }
```

On utilise `extends` pour hériter d’une classe.

(a) Incorrect.

(b) Incorrect.

(c) Correct.

(d) Incorrect.

19. **Problem**
Une classe peut-elle implémenter plusieurs interfaces ?

(a) Oui, mais seulement si les interfaces n’ont pas de méthodes en commun

(b) Non, une seule interface par classe

(c) Oui, autant qu'elle veut

(d) Maximum 2 interfaces

Solution
Oui, une classe peut implémenter **autant d'interfaces qu'elle veut** :

```
class Canard implements Volant, Nageur, Marcheur { ... }
```

C'est la solution de Java au problème de l'héritage multiple : on hérite d'une classe mais on implémente plusieurs interfaces.

(a) Incorrect.

(b) Incorrect.

(c) Correct.

(d) Incorrect.

20. **Problem**
Quel est l'ordre correct pour hériter d'une classe ET implémenter des interfaces ?

(a) class X implements I1, I2 extends Y

(b) L'ordre n'a pas d'importance

(c) class X extends Y implements I1, I2

Solution
L'ordre est obligatoire : `extends` AVANT `implements`.

```
class SiegeAuto extends Accessoire implements ArticleLouable, Transportable {
    // ...
}
```

(a) Incorrect.

(b) Incorrect.

(c) Correct.

21. **Problem**
Que retourne `bob instanceof Carnivore` si `Humain implements Omnivore` et `Omnivore extends Carnivore` ?

(a) true

(b) false

(c) Erreur de compilation

(d) NullPointerException

Solution
`true` !

Si `Omnivore extends Carnivore` et `Humain implements Omnivore`, alors un `Humain` est aussi un `Carnivore` (par transitivité).

`instanceof` vérifie toute la chaîne d'héritage/implémentation.

(a) Correct.

(b) Incorrect.

(c) Incorrect.

(d) Incorrect.