Programmation Orientée Objet (Java)

Solutions midterm

Question 1

Qu'est-ce que le « casting » explicite ? Donnez un exemple et expliquez quand il est nécessaire.

Solution 1

Forcer le compilateur à traiter une variable comme si elle était d'un autre type.

Exemple de base :

```
double y = 2;
int x = (int) y;
```

Exemple avec des objets :

```
class Vehicule { }
class Voiture extends Vehicule { }
class Main {
  public static void main(String[] args) {
    Vehicule v = new Voiture();
    Voiture v2 = (Voiture) v;
  }
}
```

Remarque 1

Le compilateur ne vérifie **pas** que l'opération est correcte.

Question 2

Quel est le résultat ?

```
public class Main {
  void methode(Integer i) {
    System.out.println("Integer");
  }

  void methode(String s) {
    System.out.println("String");
  }

  public static void main(String[] args) {
    Main m = new Main();
    m.methode(null);
  }
}
```

Solution 2

Il y a une erreur de compilation : `null` est compatible avec n'importe quel type, en particulier `Integer` et `String`.

Il est donc impossible de savoir lequel utiliser et le compilateur indique

```
error: reference to methode is ambiguous m.methode(null);
                                          ^
both method methode(Integer) in Main and method methode(String)
```

Question 3

Écrivez une classe Livre avec les attributs titre (String), auteur (String) et nbPages (int). Rendez tous les attributs privés.

Solution 3

```
public class Livre {
  private String titre;
  private String auteur;
  private int nbPages;
}
```

Question 4

Pourquoi une méthode privée ne peut-elle pas être redéfinie ?

Solution 4

La méthode est privée et est donc « invisible » pour les classes en héritant.

Remarque 2

Il est possible de créer une méthode identique (du point de vue de la signature) dans la classe enfant, cela ne posera pas de problème.

Cependant indiquer `@Override` dans la classe fille indiquera *method does not override or implement a method from a supertype.*

Question 5

Expliquez ce qu'est un `NullPointerException` et donnez un exemple de code simple qui en provoquerait un.

Solution 5

Un `NullPointerException` est une erreur se produisant lorsque l'on essaie d'accéder à un attribut ou une méthode d'un objet dont la valeur est une référence à `null`.

Par exemple

```java
String s = null;
int l = s.length();
```

Remarque 3

Une réponse fréquente a été « lorsque l'on accède à un élément qui est null ». Ce n'est pas correct :

```java
String s = null;
String s2 = s1; // Accès à un élément égal à null.
```

Remarque 4

Dans « un attribut ou une méthode d'un objet dont la valeur », la partie « d'un objet dont la valeur » n'est pas équivalente à « d'une variable ».
Le code suivant va ainsi produire un telle exception sur quelque chose qui n'est pas une variable :

```java
public static String ex() {
  return null;
}

public static void main(String[] args) {
  ex().length();
}
```

Question 6

```java
public class Livre implements Comparable<Livre> {
  int nbPages;

  @Override
  public int compareTo(Livre o) {
    // Réponse ici
  }
}
```

Solution 6

```java
public class Livre implements Comparable<Livre> {
  int nbPages;

  @Override
  public int compareTo(Livre o) {
    return this.nbPages - o.nbPages;
  }
}
```

Remarque 5

On a vu en cours qu'il est important de penser au cas où l'argument est null. Cependant en lisant la documentation de l'interface il est écrit « Note that null is not an instance of any class, and e.compareTo(null) should throw a NullPointerException even though e.equals(null) returns false. ».

Il est donc toujours important de lire la documentation avant d'implémenter une méthode.

Question 7

Supposons qu'une classe Livre a un titre (`String`), un auteur (`String`) et un nombre de pages (nbPages) (`int`) a déjà été déclarée. Ajoutez à la classe Livre un constructeur qui initialise tous ses attributs.

Solution 7

```
public class Livre {
  private String titre;
  private String auteur;
  private int nbPages;

  public Livre(String titre, String auteur, int nbPages) {
    this.titre = titre;
    this.auteur = auteur;
    this.nbPages = nbPages;
  }
}
```

Remarque notation 1

Il s'agit d'une question basique sur la syntaxe.
Il s'agit (en gros) de la seule réponse valable et beaucoup trop de gens (même si c'est une minorité) n'ont pas eu la bonne réponse.
Pour ceux n'ayant la réussi cette question il est urgent de corriger ce point.

Question 8

On suppose qu'il existe une classe `Vehicule`. Créez une classe `Voiture` qui hérite de `Vehicule`. Ajoutez un attribut `nbPortes`.

Solution 8

```
public class Voiture extends Vehicule {
  private int nbPortes;
}
```

Remarque notation 2

Le seul point important ici était de savoir utiliser le mot-clef `extends`.

Question 9

Quelle est la différence entre `==` et la méthode `.equals()` pour comparer des objets ?

Solution 9

- `==` compare les références. Il indique si les deux éléments pointent vers le même emplacement en mémoire ;

- la méthode `equals` vise à comparer le contenu.

Cependant pour que `equals` se comporte ainsi il faut la redéfinir. Son implémentation par défaut (dans la classe `Object`) n'est qu'une comparaison des adresses.

Remarque notation 3

Dans le sujet il y a écrit `equals()` alors que c'est `equals(Object obj)`. La question reste cependant claire.

Remarque notation 4

Le paragraphe sur l'implémentation par défaut n'était pas obligatoire pour avoir tous les points.

Question 10

Quelle est la différence majeure entre une classe abstraite et une interface ?

Solution 10

Il y avait plusieurs réponses pouvant être acceptées :

1. - on ne peut hériter que d'une classe abstraite,
   - on peut hériter d'un nombre quelconque d'interfaces ;
2. - une classe abstraite est une relation « est un ». On a vu au dessus qu'une `Voiture` est un `Vehicule` et peut partager du code,
   - une interface est une idée de « peut faire ». On a vu en cours l'idée de « peut faire du bruit » (interface `PeutFaireDuBruit`).

Question 11

Expliquez ce qu'est une classe abstract. Peut-on instancier une classe abstract ?

Solution 11

- Une classe `abstract` décrit une relation « est un » ;
- C'est une base de code pour les classes qui l'étendront ;
- Elle peut contenir des attributs et des méthodes ;
- Les méthodes peuvent être abstraites ;
- Une classe qui n'est pas abstraite et qui hérite (directement ou non) d'une classe abstraite doit implémenter toutes les méthodes abstraites de cette classe qui n'ont pas déjà été implémentées.

On ne peut pas instancier une classe abstraite.

Remarque 6

Le dernier point est subtil : une classe abstraite qui étend une autre classe abstraite n'a pas besoin de tout implémenter.

```
abstract class A {
  abstract void f();
```

```
  abstract void g();
}

abstract class B extends A {
  @Override
  void g() {} // La classe est abstraite, je décide d'implémenter uniquement g().
}

class C extends B {

  @Override
  void f() { }  // La classe n'est pas abstraite, je dois implémenter f().
  // g a déjà été implémenté donc je peux ne pas le réimplémenter.
}
```

Remarque notation 5

Une réponse plus précise qu'à la question précédente était attendue.

Remarque notation 6

J'ai finalement donné tous les points lorsqu'une seule des réponses était présente.

Question 12

Quand devriez-vous choisir une interface plutôt qu'une classe abstraite ? Donnez deux raisons.

Solution 12

1. Lorsque l'on hérite déjà d'une classe ;
2. On cherche à décrire un contrat, un ensemble de capacité que doivent avoir des objets, liant plusieurs concepts sans lien de parenté (par exemple un humain et un avion).

Question 13

Quand devriez-vous choisir une classe abstraite plutôt qu'une interface ? Donnez deux raisons.

Solution 13

J'avais une autre idée qui n'a pas été discutée en cours donc « Partage de code entre classes fortement liées » est la seule solution attendue.

Question 14

On suppose qu'il existe une méthode `accelerer()` dans une classe `Vehicule` et on veut que cette méthode ait un comportement différent pour une voiture. Supposons qu'il existe une classe `Vehicule` et une classe `Voiture` qui en hérite. Redéfinissez la méthode `accelerer()` dans `Voiture` pour qu'elle affiche « La voiture accélère ».

Solution 14

```
public class Voiture extends Vehicule {
  @Override
  public void accelerer() {
    System.out.println("La voiture accélère");
  }
}
```

Remarque notation 7

Le `@Override` n'était pas nécessaire.

Remarque notation 8

Il s'agit encore de code basique, à savoir faire sans réfléchir.

Question 15

Supposons qu'une classe `Livre` a un titre (`String`), un auteur (`String`) et un nombre de pages (nbPages) (`int`) a déjà été déclarée. Ajoutez à la classe `Livre` des *getters* et *setters* pour tous les attributs.

Solution 15

```
public class Livre {
  private String titre;
  private String auteur;
  private int nbPages;

  public String getTitre() {
    return this.titre;
  }
  public String getAuteur() {
    return this.auteur;
  }
  public int getNbPages() {
    return this.nbPages;
  }

  public void setTitre(String titre) {
    this.titre = titre;
  }
  public void setAuteur(String auteur) {
    this.auteur = auteur;
  }
  public void setNbPages(int nbPages) {
    this.nbPages = nbPages;
  }
}
```

Question 16

Écrivez une interface `Volant` avec une méthode `decoller()` et une classe `Avion` qui l'implémente. L'avion décolle en affichant « viou ».

Solution 16

```
public interface Volant {
  void decoller();
}

public class Avion implements Volant {
  @Override
  public void decoller() {
    System.out.println("viou");
  }
}
```

Question 17

En TP nous avions la classe Cellule suivante :

```
class Cellule {
  int valeur;
  Cellule suivante;
}
```

Donnez une méthode récursive qui affiche les valeurs dans le sens inverse.

Solution 17

```
public void afficherInverse() {
  if (this.suivante != null)
    this.suivante.afficherInverse();
  System.out.println(this.valeur);
}
```

Remarque 7

Il n'était pas indiqué dans le sujet la signature et donc la version statique est également acceptée :

```
public static void afficherInverse(Cellule c) {
  if (c != null) {
    afficherInverse(c.suivante);
    System.out.println(c.valeur);
  }
}
```

Question 18

Comment s'assurer qu'une classe ne peut pas être instanciée ?

Solution 18

La solution attendue est « la rendre `abstract` ».
Une autre solution possible est de rendre le constructeur de la classe privé.

Remarque 8

Il y a beaucoup trop de gens qui ont répondu « la rendre `final` ».

Question 19

Qu'est-ce que l'*autoboxing* et l'*unboxing* ?

Solution 19

L'*autoboxing* est la transformation d'un type primitif (`int`, `char`...) en la classe associée (`Integer`, `Character`...) et l'*unboxing* l'inverse.

```
public static void f(int i) {}
public static void g(Integer i) {}

public static void main(String[] args) {
  f(Integer.valueOf(3)); // unboxing
  g(3); // autoboxing
}
```

Question 20

Si `Vehicule v = new Voiture();`, pourquoi ne pouvez-vous pas appeler une méthode spécifique à la classe `Voiture` (par exemple `ouvrirCoffre()`) directement sur la référence `v` ? Que devez-vous faire pour y parvenir ?

Solution 20

- Pourquoi ? Pour le compilateur `v` est un `Vehicule` et « oublie » que c'est une `Voiture`. Puisqu'une `Voiture` n'a pas de méthode `ouvrirCoffre()` il ne peut pas savoir que `v` a une telle méthode.
- Que faire ? Deux solutions :
  - `Voiture v2 = (Voiture) v;`
  - ```
    if (v instanceof Voiture v2)
        v2.ouvrirCoffre();
    ```

Remarque 9

J'ai vu trop de fois « changer le type de `v` en `Voiture` ».

Question 21

Ce code va-t-il compiler ? Si non, pourquoi ?

```
abstract class Animal {
    abstract void manger();
```
```

```
}

class Chien extends Animal {
    // Pas d'implémentation de manger()
}
```

Solution 21

Non, une classe qui hérite d'une classe abstraite doit implémenter les méthodes abstraites de ses parents ou être abstraite.

Question 22

Si une sous-classe a un constructeur, que doit faire ce constructeur en premier lieu (implicitement ou explicitement) ?

Solution 22

Il doit utiliser le constructeur du parent.
Un appel implicite est fait à `super()` s'il existe.
S'il n'existe pas, alors il doit exister un autre constructeur avec des arguments qui doit être appelé explicitement.

Remarque 10

S'il n'y a aucun constructeur accessible dans la classe mère on ne peut pas créer d'instance de la classe fille :

```
class A {
  private A() {}
}

class B extends A{
  public B() {
  }
}
```

indique « *Implicit super constructor `A()` is not visible. Must explicitly invoke another constructor* » alors qu'il n'y a aucun autre constructeur.

Question 23

Que va afficher le code suivant ?

```
class Vehicule {
  public Vehicule() {
    System.out.println("Vehicule");
  }
}

class Voiture extends Vehicule {
  public Voiture() {
```

```
    System.out.println("Voiture");
  }

  public static void main(String[] args) {
    new Voiture();
  }
}
```

Solution 23

On reste dans la même idée que pour la question précédente :
- on entre dans le constructeur de `Voiture` ;
- il utilise le constructeur de `Vehicule` ;
- le constructeur de `Vehicule` affiche « Vehicule » ;
- on retourne dans le constructeur de `Voiture` ;
- il affiche « Voiture » ;

Le résultat est donc

```
Vehicule
Voiture
```

Question 24

Est-ce une bonne pratique de trop utiliser `instanceof` ? Pourquoi ?

Solution 24

L'utilisation fréquente de `instanceof` est un signe d'un mauvais design : ce n'est pas à l'utilisateur de l'objet de choisir quoi faire en fonction du type de cet objet, l'utilisateur doit demander à l'objet de faire un travail.

Remarque 11

Il y a parfois des cas où `instanceof` reste une bonne solution. Par exemple `Soignable` vu en cours.

Remarque 12

Le nom de la question était « pourquoi c'est pas bien `instanceof` ? » alors répondre que c'est une bonne pratique…

Question 25

Quel avertissement donnera le compilateur ? Avec vos mots.

```
class Enfant {
  static void methode() {
    System.out.println("Enfant");
  }
}
```

```java
public class Main {
  public static void main(String[] args) {
    Enfant e = new Enfant();
    e.methode();
  }
}
```

Solution 25

La réponse est « *The static method methode() from the type Enfant should be accessed in a static way* ». L'idée est « Pourquoi créer un `Enfant` pour appeler une méthode statique ? ».

Remarque 13

Ce n'est pas une erreur mais bien un avertissement.

Question 26

Donnez deux usages distincts du mot-clef `super`.

Solution 26

1. Appeller un constructeur de la classe mère
2. Accéder à une méthode ou un attribut de la classe mère

Question 27

Supposons qu'il existe une classe abstraite `Forme` ayant une méthode abstraite `calculerAire()`. Créez deux classes, `Cercle` et `Rectangle`, qui héritent de `Forme` et implémentent `calculerAire()`.

Solution 27

```java
public class Cercle extends Forme {
  private double rayon;

  public Cercle(double rayon) {
    this.rayon = rayon;
  }

  @Override
  public double calculerAire() {
    return Math.PI * rayon * rayon; // ou 3.14 * rayon * rayon;
  }
}

public class Rectangle extends Forme {
  private double largeur;
  private double longueur;

  public Rectangle(double largeur, double longueur) {
    this.largeur = largeur;
```

```
    this.longueur = longueur;
  }

  @Override
  public double calculerAire() {
    return largeur * longueur;
  }
}
```

Remarque 14

Plusieurs personnes ont utilisé `x**2` pour le carré de `x`. Ce n'est pas une syntaxe valide en Java.

Il faut utiliser `pow(double a, double b)` ou `x * x`.

Remarque notation 9

La note ne prend pas en compte si les formules pour calculer les aires sont correctes.

Remarque notation 10

Il n'était pas indiqué le type de retour de `calculerAire()` ni celui des attributs mais ce ne pouvaient qu'être des `int`, `double`, `float`, `Integer`, `Double` ou `Float`.

Question 28

Écrivez un constructeur pour une classe `Etudiant` qui prend un nom et un âge en paramètres.

Solution 28

```
public class Etudiant {
  private String nom;
  private int age;

  public Etudiant(String nom, int age) {
    this.nom = nom;
    this.age = age;
  }
}
```

Question 29

Qu'affichera le code suivant ?

```
  class A {
    public void process() {
        System.out.println("A");
    }
}

class B extends A {
    public void process() {
```

```
        System.out.println("B");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new B();
        a.process();
    }
}
```

Solution 29

La variable est une référence à un objet de type `A` mais ce qui est réellement présent en mémoire est un `B`. L'appel est fait sur l'objet et donc `B` est affiché.

Question 30

Dans quel ordre les constructeurs sont-ils appelés lors de la création d'un objet d'une sous-classe ?

Solution 30

D'abord les parents puis les enfants.

Remarque 15

Cette question était très ambiguë : c'est le constructeur du fils qui est appelé mais la première chose qu'il fait est d'utiliser celui du parent.

Question 31

Quelle est la différence entre une variable d'instance et une variable de classe (statique) ?

Solution 31

Une variable de classe est partagée par toutes les instances alors que ce n'est pas le cas pour les variables d'instance.

On a vu cela en cours pour la gestion des identifiants.

Question 32

Donnez un exemple de problème résolu par l'utilisation de génériques en Java.

Solution 32

On a vu en cours comment créer une classe pour représenter une liste d'entiers et on s'est demandé comment créer une liste de `String` et on a vu qu'il s'agissait d'un énorme copier-coller.

Une solution aurait été de créer une liste de `Object`. Cependant cette méthode implique une perte de la sécurité liée au typage. Par exemple avec ce code :

```
class Liste {
  private Object[] tab;
  private int taille;
  private int capacite;

  public Liste() {
    this.tab = new Object[10];
    this.taille = 0;
    this.capacite = 10;
  }

  public void ajoute(Object o) {
    // Version simplifiée, il manque la gestion de la capacité
    this.tab[this.taille] = o;
    this.taille++;
  }
}
```

on peut faire

```
Liste l = new Liste();
l.ajoute(3);
l.ajoute("a");
```

et on n'a donc aucun contrôle sur le type des objets dans la liste.

Question 33

Créez une classe abstraite `Forme` avec une méthode abstraite `calculerAire()`.

Solution 33

```
public abstract class Forme {
  public abstract double calculerAire();
}
```

Remarque notation 11

Encore une fois le type de retour n'était pas indiqué donc `int`, `double` et `float` sont autorisés.

Question 34

Qu'est-ce que l'opérateur `instanceof` et quand l'utilise-t-on ?

Solution 34

C'est un opérateur qui prend en argument une référence à un objet et indique s'il s'agit d'une instance d'une classe ou d'une de ses sous-classes ou qu'elle implémente une interface.

```
class A { }
```

```java
class B extends A { }

public class Main {
  public static void main(String[] args) {
    B val = new B();
    System.out.println(val instanceof A); // true
    System.out.println(val instanceof B); // true
  }
}
```

Il peut être utilisé comme une protection contre les *cast* incorrects.

Question 35

En TP nous avons vu les arbres binaires de recherche. Quelles sont les propriétés de ces arbres ?

Solution 35

- dans chaque nœud il y a une valeur ;
- il y a au plus deux fils :
  - un potentiel sous-arbre gauche ne contenant que des valeurs inférieures à sa valeur,
  - un potentiel sous-arbre droit ne contenant que des valeurs supérieures à sa valeur.

Remarque 16

Il y a eu plusieurs erreurs apparues plusieurs fois :

1. « Le nœud à gauche a une valeur plus petite. ». Ce n'est pas correct car cette définition indique que le fils droit du fils gauche de la racine pourrait avoir une valeur plus grande que celle de la racine.
2. « la racine a une feuille gauche… ». Ce n'est pas forcement une feuille.

Question 36

Créez une enum nommée `JourDeLaSemaine` avec les jours de `Lundi` à `Dimanche`. Chaque variant devra pouvoir se décrire.

Solution 36

Il y en a plusieurs qui pouvaient être acceptées :

```java
enum JourDeLaSemaine {
  LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE;

  public String description() {
    switch (this) {
      case LUNDI: return "lundi";
      case MARDI: return "mardi";
      case MERCREDI: return "mercredi";
      case JEUDI: return "jeudi";
      case VENDREDI: return "vendredi";
      case SAMEDI: return "samedi";
```

```
      case DIMANCHE: return "dimanche";
      default: return "";
    }
  }
}
```

ou alors

```
enum JourDeLaSemaine {
  LUNDI { public String description() { return "lundi"; } },
  MARDI { public String description() { return "mardi"; } },
  MERCREDI { public String description() { return "mercredi"; } },
  JEUDI { public String description() { return "jeudi"; } },
  VENDREDI { public String description() { return "vendredi"; } },
  SAMEDI { public String description() { return "samedi"; } },
  DIMANCHE { public String description() { return "dimanche"; } };

  public abstract String description();
}
```

Question 37

Écrivez une classe `Manager` qui hérite d'une classe `Employe`.

Solution 37

```
public class Manager extends Employe {}
```

Question 38

Supposons qu'une classe `Livre` a un titre (`String`), un auteur (`String`) et un nombre de pages (nbPages) (`int`) a déjà été déclarée. Écrivez une méthode `toString()` dans cette classe qui retourne une chaîne de caractères décrivant le livre.

Solution 38

```
public class Livre {
  @Override
  public String toString() {
    return "Livre [titre=" + this.titre + ", auteur=" + this.auteur + ", nbPages="
        + this.nbPages + "]";
  }
}
```

Remarque notation 12

C'est la version générée par mon IDE, le texte n'influe pas sur la note.

Question 39

Créez une classe `Vehicule` avec un attribut vitesse et une méthode `accelerer()`.

Solution 39

```
public class Vehicule {
  private int vitesse;

  public void accelerer() {
    this.vitesse++;
  }
}
```

Question 40

Supposons qu'il existe une classe abstraite `Forme` contenant une méthode abstraite `calculeAire()` ainsi que des classes `Cercle` et `Rectangle` qui l'étendent. Écrivez une méthode qui prend un tableau de `Forme` (`Forme[] formes`) et qui affiche l'aire de chaque forme, quelle que soit sa nature (cercle, rectangle).

Solution 40

```
public void afficherAires(Forme[] formes) {
  for (Forme f : formes) {
    System.out.println("L'aire est : " + f.calculerAire());
  }
}
```

Remarque 17

On pourrait ajouter le test `if (f != null)`.

Question 41

Écrivez la syntaxe pour déclarer une classe simple nommée `Voiture`.

Solution 41

```
public class Voiture { }
```

Question 42

Créez une classe `Calculatrice` avec des méthodes `static` pour additionner, soustraire, multiplier et diviser deux nombres.

Solution 42

```
public class Calculatrice {
  public static int additionner(int a, int b) {
    return a + b;
  }

  public static int soustraire(int a, int b) {
    return a - b;
```

```
  }

  public static int multiplier(int a, int b) {
    return a * b;
  }

  public static double diviser(double a, double b) {
    return a / b;
  }
}
```

Remarque 18

Pour la division il faudrait ajouter

```
if (b == 0)
  return Double.NaN;
```

mais `Double.NaN` n'a pas été vu en cours.

Remarque 19

On a ici un exemple où on pourrait ajouter un constructeur privé : `private Calculatrice() {}` pour empêcher l'utilisateur de créer une instance de `Calculatrice`.

Question 43

Sachant que `String` est une classe `final`, ces deux codes font-ils la même chose ? Si non donnez un exemple.

```
class C1<T extends String> {
  T val;
}

class C2 {
  String val;
}
```

Solution 43

Puisque `String` est `final` la seule valeur possible pour `T` est `String`.
Les deux classes sont donc équivalentes.

Question 44

Que se passera-t-il à l'exécution ?

```
Object o = new Integer(5);
String s = (String) o;
```

Solution 44

On est dans le cas d'un *cast* incorrect. C'est une `ClassCastException` qui est levée.

Remarque 20

Il y a eu plusieurs fausses réponses :
- La valeur de `s` est `"5"` : pour cela il faut faire `String.valueOf(o);` ;
- Il y a une erreur à la compilation : un *cast* de `Object` vers `Integer` n'est pas vérifié car `Integer` est une sous-classe de `Object`.

Question 45

En TP nous avions la classe `Cellule` suivante :

```
class Cellule {
  int valeur;
  Cellule suivante;
}
```

Donnez une méthode itérative qui retourne la somme des valeurs dans la liste.

Solution 45

```
class Cellule {
  int valeur;
  Cellule suivante;

  public static int somme(Cellule c) {
    int total = 0;
    Cellule actuelle = c;

    while (actuelle != null) {
      total += actuelle.valeur;
      actuelle = actuelle.suivante;
    }

    return total;
  }
}
```

ou alors

```
class Cellule {
  int valeur;
  Cellule suivante;

  public int somme() {
    int total = 0;
    Cellule actuelle = this;

    while (actuelle != null) {
      total += actuelle.valeur;
      actuelle = actuelle.suivante;
    }
```

```
    return total;
  }
}
```

Remarque notation 13

- Certains ont donné une solution récursive qui n'est pas acceptée.
- Certains testent `while (actuelle.suivante != null)` qui fait que la dernière valeur n'est pas affichée, et le cas où `c` est `null` n'est pas géré.

Question 46

```
public class Point {
  private int x;
  private int y;

  @Override
  public boolean equals(Object obj) {
      // ... à compléter
  }
}
```

Solution 46

```
@Override
public boolean equals(Object obj) {
  // obj peut ne pas être un Point !
  if (obj instanceof Point other)
    return this.x == other.x && this.y == other.y;
  return false;
}
```

Remarque 21

La partie `if (obj instanceof Point other)` teste si `obj` n'est pas `null` (voir cours).

Remarque 22

Vous avez peut-être vu que lorsque vous réimplementez `equals(Object obj)` il faut réimplémenter `int hashCode()`. C'est lié à la propriété suivante : « Si deux éléments sont égaux, alors l'appel à `hashCode()` sur ces deux éléments doit donner le même résultat. ».
Si on ne réimplemente pas `hashCode()` alors il pourra y avoir des problèmes avec certaines structures comme les tables de hachage, mais c'est hors programme.

Question 47

Qu'est-ce que l'annotation `@Override` et pourquoi est-il recommandé de l'utiliser ?

Solution 47

C'est une annotation qui signale au compilateur que la méthode qui suit doit redéfinir une méthode d'une super-classe ou implémenter une méthode d'une interface.

Ainsi en cas de faute de frappe lorsque l'on essaie de redéfinir une méthode le compilateur nous indiquera que ce n'est pas ce que l'on est en train de faire.

Remarque 23

Certains ont indiqué que c'est obligatoire pour redéfinir une méthode. Ce n'est pas le cas.

Question 48

À quoi servent les « getters » et les « setters » (accesseurs et mutateurs) ?

Solution 48

Ils permettent un accès contrôlé (lecture pour les *getters*, écriture pour les *setters*) aux attributs inaccessibles d'une classe.

Remarque 24

Ce n'est pas forcement un accès aux attributs `private` mais un accès aux attributs inaccessibles. Par exemple on peut avoir un getter pour un attribut `protected`.

Question 49

En TP nous avons vu les arbres binaires de recherche. Quels attributs y avait-il dans les nœuds ?

Solution 49

- une valeur ;
- un fils gauche ;
- un fils droit.

Question 50

Quelle est la principale utilité de l'héritage en programmation orientée objet ?

Solution 50

- réutilisation de code ;
- polymorphisme (voir question 40).

Question 51

Ce code va-t-il compiler ? Si non, pourquoi ?

```java
final class MaClasse {
    // ...
}

class MaSousClasse extends MaClasse {
    // ...
}
```

Solution 51

`MaClasse` est finale donc personne ne peut en hériter. Le code ne compile donc pas.

Question 52

Comment peut-on appeler un autre constructeur de la même classe depuis un constructeur ?

Solution 52

En utilisant `this()`.

Remarque 25

Une réponse plus correcte aurait été « En utilisant `this` suivi d'un certain nombre d'arguments entre parenthèses » mais il n'y avait pas de confusion possible en regardant les autres propositions.

Question 53

Considérez `class B extends A`. Quelle est la relation correcte ?

Solution 53

« B est un A » : c'est la définition de l'héritage.

Question 54

Dans un constructeur de sous-classe, l'appel à `super()` doit être…

Solution 54

L'idée était « Optionnel, il peut être omis. » mais la question n'était pas assez précise : s'il n'y a pas de constructeur sans argument ou qu'il n'est pas accessible, alors la question n'a aucun sens.

Question 55

En Java, une classe peut hériter de combien de classes ?

Solution 55

Une seule (question de cours).

Question 56

Laquelle de ces déclarations est correcte pour une méthode `main` ?

Solution 56

`public static void main(String[] args).`

Remarque 26

Cette fonction a été utilisée beaucoup de fois dans les questions précédentes…

Remarque 27

J'ai vu « `String[] args` peut être supprimé pour des questions de performance ». C'est faux.

Question 57

Laquelle de ces relations est un bon exemple d'héritage ?

Solution 57

Une `Voiture` est un `Véhicule`.

Question 58

Le *downcasting* (sous-typage) de `Object` à `String` est-il sûr sans vérification ?

Solution 58

On a déjà vu que non (`ClassCastException`, `instanceof`).

Question 59

Le modificateur `protected` rend un membre accessible…

Solution 59

Dans la classe, le package, et les sous-classes (même hors du package).

Question 60

Lequel de ces mots-clés est utilisé pour créer une instance d'une classe ?

Solution 60

`new`

Question 61

Lequel de ces scénarios illustre le polymorphisme ?

Solution 61

Le polymorphisme permet de traiter des objets de types différents de manière uniforme, via leur super-type commun. Un tableau de type `Animal` contenant des objets `Chien` et `Chat`.

Remarque 28

On a déjà fait cela avec le tableau de `Forme`.

Question 62

Où sont stockées les variables locales (déclarées à l'intérieur d'une méthode) ?

Solution 62

La pile.
Les primitives et références sont sur la pile, les objects dans le tas.

Question 63

Peut-on utiliser `super.super.methode()` pour appeler la méthode du grand-parent ?

Solution 63

`super.super` n'existe pas.

Question 64

Pour qu'une classe utilise une interface, quel mot-clé doit-elle utiliser ?

Solution 64

`implements`.

Question 65

Qu'est-ce qu'un constructeur ?

Solution 65

Une méthode spéciale utilisée pour initialiser un objet.

Question 66

Qu'est-ce qu'une classe en Java ?

Solution 66

Un modèle ou un plan pour créer des objets.

Question 67

Qu'est-ce qu'une méthode abstract ?

Solution 67

Une méthode qui n'a pas de corps d'implémentation.

Question 68

Qu'est-ce que la redéfinition de méthode (*method overriding*) ?

Solution 68

Fournir une implémentation spécifique pour une méthode qui est déjà définie dans sa superclasse.

Remarque 29

Créer une autre méthode ayant le même nom mais des arguments différents est la surcharge (*overloading*).

Question 69

Qu'est-ce que la surcharge de constructeur (constructor overloading) ?

Solution 69

Définir plusieurs constructeurs dans une classe, chacun avec une liste de paramètres différente.

Question 70

Que fait le mot-clé `super` ?

Solution 70

Il est utilisé pour appeler le constructeur ou les membres de la superclasse.

Question 71

Que fait le mot-clé `this` ?

Solution 71

Il fait référence à l'instance actuelle de l'objet.

Question 72

Que se passe-t-il si le constructeur de la superclasse n'est pas appelé explicitement dans le constructeur de la sous-classe ?

Solution 72

Deux réponses à donner :
- Le compilateur insère un appel implicite à `super()`.
- Une erreur de compilation se produit si la superclasse n'a pas de constructeur sans argument.

Question 73

Quel est l'objectif principal d'une interface ?

Solution 73

Définir un contrat de comportement pour les classes qui l'implémentent.

Question 74

Quel est le modificateur d'accès qui rend un membre accessible uniquement à l'intérieur de sa propre classe ?

Solution 74

`private`.

Question 75

Quel mot-clé est utilisé pour qu'une classe hérite d'une autre classe ?

Solution 75

`extends`.

Question 76

Quel opérateur est utilisé pour vérifier le type d'un objet à l'exécution ?

Solution 76

`instanceof`.

Question 77

Quelle est la classe parente de toutes les classes en Java ?

Solution 77

`Object`.

Question 78

Quelle est la valeur par défaut d'une référence d'objet (comme `String`...) ?

Solution 78

`null`.

Question 79

Quelle est la valeur par défaut d'une variable d'instance de type `boolean` ?

Solution 79

C'est `false`.

Question 80

Quelle est la visibilité d'un membre de classe déclaré sans modificateur d'accès (par défaut) ?

Solution 80

Accessible uniquement au sein du même package.

Question 81

Quelle est la visibilité par défaut d'une méthode dans une interface ?

Solution 81

`public`.

Question 82

Si aucun constructeur n'est défini dans une classe, que se passe-t-il ?

Solution 82

Java fournit un constructeur par défaut sans arguments.

Question 83

Dans quel(s) cas « x » sera affiché ?

```
public static void foo(Object o) {
  if (o instanceof Animal a)
      a.parle("x");
}
```

Solution 83

```
foo(new Animal(...)).
```

Question 84

Un tableau est-il un objet en Java ?

Solution 84

Oui.

Il a juste une syntaxe un peu particulière.

Question 85

Une classe déclarée `final` peut-elle être étendue (héritée) ?

Solution 85

Non

Question 86

Une classe peut-elle hériter d'une interface ?

Solution 86

Non.

Une classe implémente une interface.

Question 87

Une classe peut-elle implémenter plusieurs interfaces ?

Solution 87

Oui, mais seulement si les interfaces n'ont pas de méthodes ayant le même nom, les mêmes arguments et un type de retour différent.

Si une classe implémente deux interfaces ayant une fonction de même nom, les mêmes arguments mais un type de retour différent, comment savoir laquelle utiliser ?

Question 88

Une classe peut-elle à la fois hériter d'une autre classe et implémenter une interface ?

Solution 88

Oui, la syntaxe est `class A extends B implements I`.

Question 89

Une classe qui contient au moins une méthode `abstract` doit être déclarée…

Solution 89

`abstract`.

Question 90

Une classe qui hérite d'une classe abstraite doit…

Solution 90

Implémenter toutes les méthodes abstraites de sa superclasse, ou être elle-même déclarée abstraite.

Question 91

Une enum peut-elle hériter d'une autre classe ?

Solution 91

Non.

Un enum hérite déjà de `Enum` et il n'est pas possible d'hériter de plusieurs classes.

Question 92

Une enum peut-elle implémenter une interface ?

Solution 92

Oui.

Question 93

Une interface en Java peut contenir…

Solution 93

Uniquement des méthodes abstraites et des constantes.

Remarque 30

Cette réponse n'est plus correcte depuis bien longtemps mais elle correspond à celle du cours. Les autres réponses sont incorrectes.

Question 94

Une interface peut-elle hériter d'une autre interface ?

Solution 94

Oui, avec le mot-clef `extends`.

Question 95

Une méthode déclarée `final` dans une superclasse peut-elle être redéfinie dans une sous-classe ?

Solution 95

Non, cela provoque une erreur de compilation.

Question 96

Une méthode déclarée `static` peut :

Solution 96

Être appelée sans créer d'instance de la classe.

Question 97

Une méthode `final` peut-elle être `private` ?

Solution 97

Oui, même si cela ne sert à rien, puisque `private` implique `final`.

Question 98

Une méthode `static` peut-elle être `abstract` ?

Solution 98

Non.

- `static` : l'implémentation est liée à la classe ;
- `abstract` : l'implémentation est faite par une sous-classe.

Question 99

Une variable déclarée `final` signifie que :

Solution 99

Sa valeur ne peut pas être modifiée après initialisation.

Remarque 31

Dans le cas des objets, la variable est une référence. La déclarer `final` indique que la variable ne va pas référencer un autre objet mais l'objet peut changer :

```
class C {
  protected int x;

  public C(int x) {
    this.x = x;
  }

  public void setX(int x) {
    this.x = x;
  }
}

class Main {
  public static void main(String[] args) {
    final C c = new C(3);
    System.out.println(c.x); // Affiche 3
    c.setX(2);
    System.out.println(c.x); // Affiche 2
  }
}
```

Question 100

À quoi sert une énumération (`enum`) en Java ?

Solution 100

À définir un type qui représente un ensemble fixe de constantes.