Java

JAVA

Qu'est-ce que le JDK, JRE et JVM ?

- **JDK (Java Development Kit) :** Un ensemble d'outils nécessaires pour développer des applications Java, y compris le compilateur `javac`, les bibliothèques standard et le JRE.
- **JRE (Java Runtime Environment) :** Fournit les bibliothèques, la JVM, et d'autres composants pour exécuter des applications écrites en Java. Il ne contient pas les outils de développement.
- **JVM (Java Virtual Machine) :** Composant de la JRE qui exécute les bytecodes Java. Il permet l'indépendance de la plateforme en interprétant le bytecode.

```
// Exemple d'utilisation basique du JDK
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Quelle est la différence entre `==` et `equals()` ?

- `==` **:** Utilisé pour comparer les références des objets pour vérifier s'ils pointent vers la même instance.
- `equals()` **:** Méthode utilisée pour comparer le contenu des objets.

```
String a = new String("test");
String b = new String("test");

System.out.println(a == b); // false, car différentes références
System.out.println(a.equals(b)); // true, car contenu identique
```

Qu'est-ce que l'immutabilité et comment créer une classe immuable ?

- **Immutabilité :** Une classe est immuable si, une fois créée, son état ne peut pas être modifié.
- **Créer une classe immuable :** Déclarer tous les champs `final`, fournir un constructeur pour initialiser les champs, et ne pas fournir de setters.

```
public final class ImmutableClass {
    private final String name;
    private final int age;

    public ImmutableClass(String name, int age) {
```

```
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

Quelle est la différence entre une Interface et une Classe Abstraite ?

- **Interface :** Ne peut contenir que des méthodes abstraites (Java 8 permet les méthodes par défaut) et des constantes. Une classe peut implémenter plusieurs interfaces.
- **Classe Abstraite :** Peut contenir des méthodes abstraites et des méthodes concrètes. Une classe peut étendre une seule classe abstraite.

```
interface Animal {
    void eat();
}

abstract class Mammal {
    abstract void move();
    void sleep() {
        System.out.println("Sleeping");
    }
}
```

Qu'est-ce qu'un `HashMap` et comment fonctionne-t-il ?

- `HashMap` **:** Une collection qui stocke les paires clé-valeur et permet un accès rapide basé sur le hachage de la clé. Utilise une table de hachage pour stocker les éléments.

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
System.out.println(map.get("one")); // 1
```

Expliquez le concept de Collection en Java.

- **Collection :** Cadre qui fournit des architectures pour stocker et manipuler des groupes d'objets. Principales interfaces : `List`, `Set`, `Queue`.

```
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
System.out.println(list);
```

Qu'est-ce que la gestion de la mémoire en Java ? Parlez du garbage collector.

- **Gestion de la mémoire :** Java gère automatiquement la mémoire via le garbage collector, qui libère la mémoire des objets non référencés.

```
// Exemple basique de mémoire gérée par le garbage collector
String str = new String("Hello");
// Après que str est réaffecté, "Hello" peut être éligible pour la collecte des ordures
str = null;
```

Expliquez le concept de multithreading et la différence entre `Runnable` et `Thread`.

- **Multithreading :** Exécution concurrente de plusieurs threads pour effectuer des tâches simultanément.
- `Runnable` **:** Interface implémentée pour définir la tâche à exécuter.
- `Thread` **:** Classe qui implémente `Runnable` et représente un thread.

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running");
    }
}

Thread thread = new Thread(new MyRunnable());
thread.start();
```

Qu'est-ce que le polymorphisme et comment est-il implémenté en Java ?

- **Polymorphisme :** Capacité d'un objet à prendre plusieurs formes. Implémenté via l'héritage (sous-typage) et les interfaces.

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
```

```
    }
}

Animal animal = new Dog();
animal.sound(); // Bark
```

Qu'est-ce que la sérialisation en Java ?

- **Sérialisation :** Processus de conversion d'un objet en un flux de bytes pour le stockage ou la transmission. `Serializable` interface est utilisée.

```
class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    String name;
    int age;

    // Constructor, getters, setters
}

// Sérialisation
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("person.ser"));
out.writeObject(new Person("John", 30));
out.close();

// Désérialisation
ObjectInputStream in = new ObjectInputStream(new
FileInputStream("person.ser"));
Person person = (Person) in.readObject();
in.close();
```

Parlez de l'exception handling en Java.

- **Gestion des exceptions :** Mécanisme pour gérer les erreurs au moment de l'exécution. Utilise `try`, `catch`, `finally`, `throw`, et `throws`.

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero");
} finally {
    System.out.println("This block is always executed");
}
```

Qu'est-ce que le lambda expression en Java ?

- **Lambda expression :** Syntaxe pour écrire des méthodes anonymes concises. Introduit en Java 8.

```
List<String> list = Arrays.asList("a", "b", "c");
list.forEach(element -> System.out.println(element));
```

Qu'est-ce que le Stream API ?

- **Stream API :** Utilisé pour traiter des séquences d'éléments de manière fonctionnelle. Introduit en Java 8.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squares = numbers.stream()
                               .map(n -> n * n)
                               .collect(Collectors.toList());
System.out.println(squares);
```

Expliquez le fonctionnement des `synchronized` methods et blocks.

- **`synchronized` methods :** Garantit qu'un seul thread peut accéder à la méthode à la fois.
- **`synchronized` blocks :** Permet de synchroniser des blocs de code pour un contrôle plus fin.

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public void incrementBlock() {
        synchronized (this) {
            count++;
        }
    }
}
```

Qu'est-ce qu'une inner class en Java ?

- **Inner class :** Une classe définie à l'intérieur d'une autre classe. Types : Inner class, static nested class, local class, anonymous class.

```
class OuterClass {
    private int x = 10;

    class InnerClass {
        void display() {
            System.out.println("x = " + x);
        }
    }
}
```

```
OuterClass outer = new OuterClass();
OuterClass.InnerClass inner = outer.new InnerClass();
inner.display();
```

Qu'est-ce que la Programmation Orientée Objet (POO) ?

- **POO :** Paradigme de programmation basé sur les objets contenant des données et des méthodes. Concepts clés : encapsulation, héritage, polymorphisme, abstraction.

Expliquez les quatre principes fondamentaux de la POO.

- **Encapsulation :** Regroupement des données et des méthodes dans des classes.
- **Héritage :** Permet à une classe d'hériter des propriétés et méthodes d'une autre classe.
- **Polymorphisme :** Capacité d'un objet à prendre plusieurs formes.
- **Abstraction :** Simplification de la complexité en exposant uniquement les détails essentiels.

Quelle est la différence entre une exception checked et unchecked ?

- **Checked exceptions :** Doivent être déclarées dans la signature de méthode ou capturées. Ex : `IOException`.
- **Unchecked exceptions :** Sont des sous-classes de `RuntimeException` et ne nécessitent pas de déclaration. Ex : `NullPointerException`.

Qu'est-ce que le `final` keyword en Java ?

- **`final` :** Utilisé pour déclarer des constantes, empêcher l'héritage de classes, empêcher la redéfinition de méthodes.

```
final class Constants {
    static final int MAX_VALUE = 100;
}

// Ne peut pas être étendu
class Subclass extends

 Constants {} // Erreur de compilation
```

Qu'est-ce qu'une classe interne anonyme ?

- **Classe interne anonyme :** Classe sans nom déclarée et instanciée en même temps, souvent utilisée pour les callbacks ou les implémentations uniques.

```
Runnable runnable = new Runnable() {
    public void run() {
        System.out.println("Anonymous class example");
    }
```

```
};
new Thread(runnable).start();
```

Expliquez le concept de méthode générique.

- **Méthode générique :** Permet de créer des méthodes qui fonctionnent avec n'importe quel type de données.

```
public <T> void printArray(T[] array) {
    for (T element : array) {
        System.out.println(element);
    }
}
```

Comment fonctionnent les `static` variables et methods ?

- **Variables statiques :** Appartiennent à la classe plutôt qu'aux instances.
- **Méthodes statiques :** Peuvent être appelées sans instancier la classe.

```
class Example {
    static int count = 0;

    static void displayCount() {
        System.out.println("Count: " + count);
    }
}
Example.displayCount();
```

Qu'est-ce que le mot-clé `transient` ?

- `transient` : Indique qu'un champ ne doit pas être sérialisé.

```
class Person implements Serializable {
    transient int age;
    String name;
}
```

Quelle est la différence entre `String`, `StringBuilder` et `StringBuffer` ?

- `String` : Immutable.
- `StringBuilder` : Mutable, non thread-safe.
- `StringBuffer` : Mutable, thread-safe.

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb.toString());
```

Qu'est-ce qu'un singleton et comment l'implémenter en Java ?

- **Singleton :** Patron de conception qui restreint l'instanciation d'une classe à une seule instance.

```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

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

- **Autoboxing :** Conversion automatique de types primitifs en objets wrapper.
- **Unboxing :** Conversion automatique des objets wrapper en types primitifs.

```
Integer obj = 10; // Autoboxing
int num = obj; // Unboxing
```

Expliquez les différences entre `Comparable` et `Comparator`.

- `Comparable` : Interface utilisée pour l'ordre naturel d'une classe.
- `Comparator` : Interface utilisée pour définir un ordre externe.

```
class Person implements Comparable<Person> {
    String name;

    public int compareTo(Person other) {
        return this.name.compareTo(other.name);
    }
}

class AgeComparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

Qu'est-ce que le `try-with-resources` statement ?

- `try-with-resources` : Déclaration de `try` qui garantit que chaque ressource sera fermée à la fin du bloc.

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Comment implémenter la sécurité des threads en Java ?

- **Sécurité des threads :** Utilisation de `synchronized`, `volatile`, classes `concurrent`, et autres mécanismes pour garantir la sécurité des données partagées entre threads.

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public synchronized int getCount() {
        return count;
    }
}
```

Qu'est-ce qu'une interface fonctionnelle ?

- **Interface fonctionnelle :** Interface avec une seule méthode abstraite, utilisée principalement dans les expressions lambda et les références de méthode.

```
@FunctionalInterface
interface MyFunctionalInterface {
    void execute();
}
```

Qu'est-ce que l'inférence de type en Java ?

- **Inférence de type :** Capacité du compilateur à déduire automatiquement le type de données.

```
List<String> list = new ArrayList<>(); // Le compilateur infère <String>
```

Expliquez le concept de capture de wildcard.

- **Capture de wildcard :** Technique utilisée pour résoudre les problèmes liés aux génériques avec des wildcards.

```
public <T> void printList(List<T> list) {
    for (T element : list) {
        System.out.println(element);
    }
}
```

Qu'est-ce qu'un enum et comment l'utiliser ?

- **Enum :** Type spécial de classe pour représenter un ensemble fixe de constantes.

```
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

Day day = Day.MONDAY;
```

Qu'est-ce que la Reflection API en Java ?

- **Reflection API :** Permet d'examiner ou de modifier le comportement des classes, interfaces, et objets à l'exécution.

```
Class<?> clazz = Class.forName("com.example.MyClass");
Method method = clazz.getMethod("myMethod");
method.invoke(clazz.newInstance());
```

Parlez de la `ClassLoader` et ses types.

- `ClassLoader` **:** Charge les classes dynamiquement en Java. Types principaux : Bootstrap, Extension, System/Application.

```
ClassLoader classLoader = MyClass.class.getClassLoader();
```

Qu'est-ce qu'une `WeakReference` en Java ?

- `WeakReference` **:** Référence qui permet au garbage collector de récupérer l'objet référencé si aucune référence forte n'existe.

```
WeakReference<MyObject> weakRef = new WeakReference<>(new MyObject());
```

Qu'est-ce que l'instruction `assert` et comment l'utiliser ?

- `assert` **:** Utilisée pour tester des expressions et lever une erreur si l'expression est fausse.

```
int x = -1;
assert x > 0 : "x must be positive";
```

Expliquez le concept de Proxies dynamiques en Java.

- **Proxies dynamiques :** Classes générées au moment de l'exécution qui implémentent une ou plusieurs interfaces spécifiées.

```
InvocationHandler handler = new MyInvocationHandler();
MyInterface proxy = (MyInterface) Proxy.newProxyInstance(
    MyInterface.class.getClassLoader(),
    new Class<?>[] { MyInterface.class },
    handler
);
```

Comment fonctionne le `Volatile` keyword en Java ?

- **`volatile` :** Indique qu'une variable peut être modifiée de façon imprévisible, garantissant la visibilité de sa valeur entre les threads.

```
class SharedObject {
    private volatile boolean flag = false;

    public void setFlag(boolean flag) {
        this.flag = flag;
    }

    public boolean getFlag() {
        return flag;
    }
}
```

Qu'est-ce que le pattern de conception `Factory` ?

- **Factory :** Patron de conception qui fournit une méthode pour créer des objets sans spécifier leur classe concrète.

```
class ShapeFactory {
    public Shape getShape(String shapeType) {
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        }
        return null;
    }
}
```

Quelle est la différence entre le pattern `Factory` et `Abstract Factory` ?

- **Factory :** Crée des objets de types spécifiques.
- **Abstract Factory :** Fournit une interface pour créer des familles d'objets liés ou dépendants.

```
interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

class WinFactory implements GUIFactory {
    public Button createButton() {
        return new WinButton();
    }

    public Checkbox createCheckbox() {
        return new WinCheckbox();
    }
}
```

Expliquez le pattern de conception `Observer`.

- **Observer :** Patron de conception où un objet (observé) notifie ses dépendants (observateurs) de tout changement d'état.

```
interface Observer {
    void update();
}

class ConcreteObserver implements Observer {
    public void update() {
        System.out.println("State updated");
    }
}

class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void attach(Observer observer) {
        observers.add(observer);
    }

    public void notifyAllObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }
}
```

Qu'est-ce que la méthode `clone()` en Java et comment l'utiliser ?

- `clone()` : Crée et renvoie une copie de l'objet actuel.

```
class Person implements Cloneable {
    String name;

    public Person clone() throws CloneNotSupportedException {
        return (Person) super.clone();
    }
}

Person p1 = new Person();
Person p2 = p1.clone();
```

Qu'est-ce que le `Fork/Join Framework` en Java ?

- **Fork/Join Framework :** Utilisé pour le traitement parallèle. Divise les tâches en sous-tâ

ches (fork) et combine les résultats (join).

```
class MyTask extends RecursiveTask<Integer> {
    protected Integer compute() {
        // Logic for dividing and conquering tasks
    }
}

ForkJoinPool pool = new ForkJoinPool();
pool.invoke(new MyTask());
```

Qu'est-ce que le `Executors` framework en Java ?

- `Executors` **framework :** Fournit un pool de threads et des méthodes pour gérer la soumission et l'exécution de tâches.

```
ExecutorService executor = Executors.newFixedThreadPool(10);
executor.submit(() -> {
    System.out.println("Task executed");
});
executor.shutdown();
```

Comment gérer les Deadlocks en Java ?

- **Gestion des Deadlocks :** Éviter les deadlocks en acquérant les verrous dans un ordre cohérent, utilisant des délais d'attente ou en détectant les cycles dans le graphe d'attente.

Qu'est-ce que l'annotation `@FunctionalInterface` ?

- `@FunctionalInterface` : Annotation utilisée pour indiquer qu'une interface est une interface fonctionnelle avec une seule méthode abstraite.

```java
@FunctionalInterface
interface MyFunctionalInterface {
    void execute();
}
```

Expliquez la différence entre `Optional` et `Nullable`.

- **`Optional`** : Conteneur pour un objet qui peut être ou ne pas être présent. Préfère à `null` pour éviter les NullPointerExceptions.
- **`Nullable`** : Annotation indiquant qu'une variable peut être `null`.

```java
Optional<String> optionalString = Optional.of("Hello");
optionalString.ifPresent(System.out::println);
```

Qu'est-ce que le `CompletableFuture` en Java ?

- **`CompletableFuture`** : Classe utilisée pour représenter une tâche asynchrone qui peut être complétée manuellement.

```java
CompletableFuture<String> future = CompletableFuture.supplyAsync(() ->
"Hello");
future.thenAccept(result -> System.out.println(result));
```

Quelle est la différence entre `Collection` et `Collections` ?

- **`Collection`** : Interface racine pour les structures de données.
- **`Collections`** : Classe utilitaire pour manipuler des collections.

```java
List<String> list = new ArrayList<>(Arrays.asList("a", "b", "c"));
Collections.sort(list);
System.out.println(list);
```

Qu'est-ce que le pattern de conception `Decorator` ?

- **Decorator :** Patron de conception qui permet d'ajouter dynamiquement des responsabilités à un objet.

```java
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}
```

```
class RedShapeDecorator implements Shape {
    private Shape decoratedShape;

    public RedShapeDecorator(Shape decoratedShape) {
        this.decoratedShape = decoratedShape;
    }

    public void draw() {
        decoratedShape.draw();
        System.out.println("Coloring Red");
    }
}
```

Expliquez le fonctionnement de `TreeMap` et `TreeSet` en Java.

- **`TreeMap`** : Implémente `Map` et trie les clés selon leur ordre naturel ou un `Comparator` fourni.
- **`TreeSet`** : Implémente `Set` et trie les éléments selon leur ordre naturel ou un `Comparator` fourni.

```
TreeMap<Integer, String> treeMap = new TreeMap<>();
treeMap.put(3, "Three");
treeMap.put(1, "One");
treeMap.put(2, "Two");

TreeSet<String> treeSet = new TreeSet<>();
treeSet.add("Banana");
treeSet.add("Apple");
treeSet.add("Cherry");

System.out.println(treeMap);
System.out.println(treeSet);
```

Qu'est-ce que le `Stream.collect()` en Java ?

- **`Stream.collect()`** : Méthode terminale utilisée pour transformer un Stream en une autre forme, comme une collection.

```
List<String> list = Arrays.asList("a", "b", "c");
List<String> upperCaseList = list.stream()
                                 .map(String::toUpperCase)
                                 .collect(Collectors.toList());
System.out.println(upperCaseList);
```

Comment fonctionne le Pattern Matching pour les `instanceof` en Java ?

- **Pattern Matching pour `instanceof`** : Simplifie les tests de type et le casting.

```
Object obj = "Hello";
if (obj instanceof String str) {
    System.out.println(str.toUpperCase());
}
```

Quelle est la différence entre `CountDownLatch` et `CyclicBarrier` ?

- **`CountDownLatch` :** Permet à un ou plusieurs threads d'attendre jusqu'à ce qu'un ensemble d'opérations se termine.
- **`CyclicBarrier` :** Permet à un groupe de threads d'attendre l'un l'autre jusqu'à ce qu'ils atteignent une barrière commune, réutilisable.

```
CountDownLatch latch = new CountDownLatch(3);
Runnable task = () -> {
    // Task logic
    latch.countDown();
};

CyclicBarrier barrier = new CyclicBarrier(3, () ->
System.out.println("Barrier reached"));
Runnable barrierTask = () -> {
    // Task logic
    barrier.await();
};
```

Expliquez le concept de Method References en Java.

- **Method References :** Syntaxe simplifiée pour appeler des méthodes existantes par leurs noms.

```
List<String> list = Arrays.asList("a", "b", "c");
list.forEach(System.out::println);
```

Qu'est-ce que le `ReentrantLock` en Java et comment l'utiliser ?

- **`ReentrantLock` :** Implémentation de `Lock` qui permet de verrouiller de manière réentrante, offrant plus de contrôle que `synchronized`.

```
ReentrantLock lock = new ReentrantLock();

lock.lock();
try {
    // Critical section
} finally {
    lock.unlock();
}
```

Qu'est-ce que l'annotation `@Override` en Java et pourquoi l'utiliser ?

- `@Override` : Indique qu'une méthode redéfinit une méthode d'une superclasse, permettant au compilateur de vérifier les erreurs.

```
class Parent {
    void display() {
        System.out.println("Parent");
    }
}

class Child extends Parent {
    @Override
    void display() {
        System.out.println("Child");
    }
}
```

Qu'est-ce que le Double Brace Initialization en Java ?

- **Double Brace Initialization :** Utilisation des blocs d'initialisation pour ajouter des éléments à une collection lors de la création.

```
List<String> list = new ArrayList<String>() {{
    add("a");
    add("b");
    add("c");
}};
System.out.println(list);
```

Expliquez le concept de Memory Consistency Errors en Java.

- **Memory Consistency Errors :** Se produisent lorsque plusieurs threads accèdent à des données partagées sans synchronisation appropriée, entraînant des valeurs incohérentes.