# Guía Java V

## ▼ Ej 1.- ¿Qué arroja? - Ciclos y flujo

```
public class Main {
    public static void main(String[] args) {
        String[] at = {"FINN", "JAKE"};
        for (int x=1; x<4; x++){
            for (String s : at){
                System.out.println(x + " "+ s);
                if(x==1){
                    break;
                }
            }
        }
    }
}
```

### Análisis

1. Se crea un arreglo de `String` con dos elementos.
2. El primer ciclo irá de `x=1` a `x=3` .
3. Dentro del primer ciclo hay un ciclo mejorado que itera el arreglo `at` .
4. Dentro de este segundo ciclo se imprime el valor `x` del primer ciclo concatenado al elemento de `at` seleccionado por el ciclo mejorado.
5. Después de imprimir, se evalúa si `x==1` , en caso de que sí, se ejecutar un `break` .

### Flujo

1. Primero se imprime `1 FINN` .
2. Se evalúa si x `x==1` , esto retorna `true` , por lo que se ejecuta el `break` y se interrumpe el ciclo mejorado.
3. Ahora `x=2` , se imprime `2 FINN` , `x==2` entonces no se ejecuta `break`
4. Siguiente elemento del arreglo, se imprime `2 JAKE` , no se ejecuta `break` y acaba ciclo mejorado.
5. Ahora `x=3` , se imprime `3 FINN` , no se ejecuta `break` .
6. Siguiente elemento, se imprime `3 JAKE` , no se ejecuta `break` y termina ciclo mejorado.
7. `x=4` entonces no se vuelve a entrar y termina programa

### Respuesta

`1 FINN 2 FINN 2 JAKE 3 FINN 3 JAKE`

## ▼ Ej 2.- ¿Qué 5 líneas son correctas? - Override y herencia

```
class Light{
    protected int lightsaber(int x){return 0;}
}
class Saber extends Light{
```

```
    private int lightsaber (int x){return 0;} //Incorrecto -> Se está haciendo Override
y no se puede reducir la visibilidad
    protected int lightsaber (long x){return 0;}//Correcto -> No se está haciendo Overr
ide, es otro método.
    private int lightsaber (long x){return 0;} //Correcto -> No se está sobreescribiend
o, se trata de otro método
    protected long lightsaber (int x){return 0;} //Incorrecto -> Se está haciendo Overr
ide, el retorno debe respetarse o hacer la covarianza
    protected long lightsaber (int x, int y){return 0;} //Correcto -> No se esta hacien
do Override,es otro método.
    public int lightsaber (int x){return 0;} //Correcto -> Amplia la visibilidad.
    protected long lightsaber (long x){return 0;} //Correcto -> No está haciendo Overri
de, es otro método
}
```

## ▼ Ej 3.- ¿Qué resultado arroja? - Constructores y sobrecarga

```
class Mouse{
    public int numTeeth;
    public int numWhiskers;
    public int weight;

    public Mouse (int weight){
        this(weight,16);
    }

    public Mouse (int weight, int numTeeth){
        this(weight, numTeeth, 6);
    }

    public Mouse (int weight, int numTeeth, int numWhiskers){
        this.weight = weight;
        this.numTeeth= numTeeth;
        this.numWhiskers = numWhiskers;
    }

    public void print (){
        System.out.println(weight + ""+ numTeeth+ ""+ numWhiskers);
    }

    public static void main (String[] args){
        Mouse mouse = new Mouse (15);
        mouse.print();
    }
}
```

### Análisis

- La clase `Mouse` cuenta con 3 atributos públicos, 3 constructores sobrecargados y el método print.
- En el método `main` se crea un objeto de `Mouse` pasando como parámetro `15` .

- El `Mouse` se crea invocando al constructor que recibe el atributo `weight` , este constructor invoca al constructor que recibe dos parámetros, entonces `weight = 15` y `numTeeth = 16` .
- El constructor de dos parámetros invoca al constructor de tres parámetros con `numWhiskers = 6` , finalmente este constructor asigna los atributos.

### Respuesta

Se imprime: `15 16 6`

---

## ▼ Ej 4.- ¿Cuál es la salida? - Herencia

```
class Arachnid {
    public String type = "a";

    public Arachnid(){
        System.out.println("arachnid");
    }
}

class Spider extends Arachnid{
    public Spider(){
        System.out.println("spider");
    }

    void run(){
        type = "s";
        System.out.println(this.type + " " + super.type);
    }

    public static void main(String[] args) {
        new Spider().run();
    }
}
```

### Análisis

- Clase `Arachnid` tiene un atributo `type` y un constructor sin parámetros.
- Clase `Spider` hereda de `Arachnid` , tiene un constructor sin parámetros y un método `run()` .

### Flujo

- Se crea un objeto spider, se imprime `arachnid` y posteriormente `spider` y se ejecuta el método `run()`
- En `run()` se asigna `"s"` al atributo `type` , que si bien no se ha definido en el objeto de `Spider` , se hereda de `Arachnid` .
- Se busca imprimir `type` utilizando `this` y `super` , sin embargo, `type` hace referencia al mismo atributo.

### Respuesta

**Se imprime** `arachnid spider s s`

---

## ▼ Ej 5.- Resultado - Decrementos

```
class Sheep {
    public static void main(String[] args) {
        int ov = 999;
        ov--;
        System.out.println(--ov);
        System.out.println(ov);
    }
}
```

### Análisis

1. Se declara la variable y se inicializa `ov=999` .
2. Se hace posdecremento, entonces `ov=998` .
3. Se hace predecremento, entonces `ov=997` y se imprime.
4. Se vuelve a imprimir `ov=997` .

### Respuesta

**Se imprime** `997 997`

---

## ▼ Ej 6.- Resultado - Sobrecarga

```
class Overloading {
    public static void main(String[] args) {
        System.out.println(overload("a"));
        System.out.println(overload("a", "b"));
        System.out.println(overload("a", "b", "c"));
    }

    public static String overload(String s){
        return "1";
    }
    public static String overload(String... s){
        return "2";
    }
    public static String overload(Object o){
        return "3";
    }
    public static String overload(String s, String t){
        return "4";
    }
}
```

### Análisis

- Se tienen 4 métodos sobrecargados `overload()` .
- En el método `main()` invoca el método `overload(String s)` e imprime `"1"`
- Se invoca el método `overload(String s, String t)` e imprime `"4"`
- Los ***varargs*** indican que se esperan de `0` a `n` parámetros, o un `Array` .

- El método que recibe 3 parámetros retorna un 2, se imprime `"2"` .

### Respuesta

**Se imprime `1 4 2`**

---

## ▼ Ej 7.- Resultado - Casteo

```
class Base1 extends Base{
    public void test(){
        System.out.println("Base1");
    }
}

class Base2 extends Base{
    public void test(){
        System.out.println("Base2");
    }
}

class Test {
    public static void main(String[] args) {
        Base obj = new Base1();
        ((Base2) obj).test();
    }
}
```

### Análisis

- `Base1` y `Base2` son subclases de `Base` y ambas implementan el método `test()` .
- En el método `main()` se crea un objeto de tipo `Base1` en una variable de tipo `Base` .
- Se intenta hacer el casting a `Base2` pero no es posible pues no hay relación ascendente o descendente, ambas son hermanas y se produce un `ClassCastException`

### Respuesta

**`ClassCastException`**

---

## ▼ Ej 8.- Resultado - Asignación de variables y tipos

```
public class Fish {
    public static void main(String[] args) {
        int numFish = 4;
        String fishType= "Tuna";
        String anotherFish = numFish +1;
        System.out.println(anotherFish + " " + fishType);
        System.out.println(numFish + " " + 1);
    }
}
```

### Análisis

- Se crea el atributo `numFish = 4` , `fishType = "Tuna"` .
- Se intenta inicializar `anotherFish` con un `int` , entonces el código no compila.

### Respuesta

Error de compilación

## ▼ Ej 9.- Resultado - Sistemas numéricos

```
class MathFun {
    public static void main(String[] args) {
    int number1 = 0b0111;
    int number2 = 0111_000;

    System.out.println("Number1: "+number1);
    System.out.println("Number2: "+number1);
    }
}
```

### Análisis

- Se inicializa `number1 = 0b0111` , en decimal → `number1 = 7` .
- Se inicializa `number2 = 0111_000` , en octal → `number2 = 8*8*8 + 8*8*8*8 + 8*8*8*8*8`
- Se imprimen en ambas ocasiones `number1`

### Respuesta

`Number1: 7 Number2: 7`

## ▼ Ej 10.- Resultado - Scope

```
class Calculator {
    int num =100;

    public void calc(int num){
        this.num =num*10;
    }
    public void printNum(){
        System.out.println(num);
    }
    public static void main (String [] args){
        Calculator obj = new Calculator ();
        obj.calc(2);
        obj.printNum();
    }
}
```

### Análisis

- La clase `Calculator` tiene un atributo `num = 100` y dos métodos; `printNum()` y `calc()` .
- En el método `main()` se crea un objeto de tipo `Calculator` , posteriormente se invoca el método `calc(2)` , entonces el atributo `num = 10*2` .

### Respuesta

**Se imprime** `20`

## ▼ Ej 11.- ¿Qué aseveraciones son correctas? - Imports

```
class ImportExample {
    public static void main (String[] args){
        Random r = new Random();
        System.out.println(r.nextInt(10));
    }
}
```

- If you omit java.util import statements java compiles gives you an error
- java.lang and util.random are redundant
- you dont need to import java.lang

### Análisis

1. True → `Random` se encuentra en `java.util` .
2. False → No es redudante, cada uno tiene contenido diferente
3. True → `java.lang` es importada automáticamente.

### Respuesta

1,3

## ▼ Ej 12.- Resultado Incrementos

```
public class Main {
    public static void main(String[] args) {
    int var = 10;
    System.out.println(var++);
    System.out.println(++var);
}
```

### Análisis

- Se crea una variable `var = 10` .
- Se imprime `10` y posteriormente se incrementa una unidad, entonces `var = 11` .
- Se incrementa y se imprime `12`

### Respuesta

10, 12

## ▼ Ej 13.- Resultado - Ciclos

```
class MyTime {
    public static void main (String [] args){
        short mn = 11;
        short hr;
```

```
        short sg =0;
        for (hr=mn;hr>6;hr-=1){
            sg++;
        }
        System.out.println("sg="+sg);
    }
}
```

### Análisis

- Se crean las variables `mn = 11` , `hr` y `sg = 0` .
- El ciclo se lleva a cabo de `hr = 11` hasta `hr = 6` decrementando 1 unidad.
- Dentro del ciclo se incrementa `sg` , y al finalizar se imprime el valor de `sg` .

### Flujo

1. Primera iteración → `hr = 11` `sg = 1`
2. Segunda iteración → `hr = 10` `sg = 2`
3. Si se continua hasta `hr = 7` → `sg = 5`
4. Cuando `hr = 6` se rompe el ciclo y se imprime `sg = 5`

### Respuesta

5

---

## ▼ Ej 14.- ¿Cuáles son verdad? - Array y ArrayList

1. An ArrayList is mutable
2. An Array has a fixed size
3. An array is mutable
4. An array allows multiple dimensions
5. An arrayList is ordered
6. An array is ordered

### Análisis

1. True → `ArrayList` es mutable, puede ser modificado después de su creación.
2. True → Los `Array` tienen un tamaño fijo una vez que se define.
3. True → A pesar de que se tiene tamaño fijo, los valores almacenados pueden ser cambiados.
4. True → Se pueden tener arreglos de múltiples dimensiones
5. True → Un `ArrayList` es ordenado basados en como se agregaron elementos.
6. True → Un `Array` es ordenado según su índice.

### Respuesta

**Todo es verdadero**

---

## ▼ Ej 15.- Resultado - Ciclos

```
public class MultiverseLoop {
    public static void main (String [] args){
        int negotiate = 9;

        do{
            System.out.println(negotiate);
        }while (--negotiate);
    }
}
```

### Análisis

- Se crea la variable `negotiate = 9` , hay un ciclo `do` pero la condición en `while()` no es un `boolean` .

### Respuesta

**Error de compilación.**

## ▼ Ej 16.- Resultado - Stream

```
class App {
    public static void main(String[] args) {
        Stream<Integer> nums = Stream.of(1,2,3,4,5);
        nums.filter(n -> n % 2 == 1);
        nums.forEach(p -> System.out.println(p));
    }
}
```

### Análisis

- Se crea un `Stream` de `Integer` con 5 elementos.
- El `Stream` es inmutable.
- Se utiliza el método `filter()` pasándole la lambda `n -> n%2==1` , entonces se aceptan únicamente números impares.
- El resultado de `filter()` no se almacena o encadena
- En el método `forEach()` se intenta imprimir cada elemento, sin embargo el `Stream` ya ha sido utilizado por `filter()` lo que marca el `Stream` como consumido.
- Una vez que se consume un `Stream` , este no puede ser reutilizado.

### Respuesta

`Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed`

### Alternativas

Encadenar las operaciones del `Stream` y no reutilizarlo

## ▼ Ej 17.- Pregunta - Cast

Suppose the declared type of x is a class, and the declared type of y is an interface. When is the assignment x = y; legal?

### Análisis

`x` → clase & `y` → interface

Entonces `x = y` cuando `y` contenga un objeto de tipo `x` y se realice el cast.

## ▼ Ej 18.- Pregunta - Types

When a byte is added to a char, what is the type of the result?

**`int` → Las operaciones entre enteros menores a `long` entregan un `int` .**

## ▼ Ej 19.- Pregunta - API JDBC

The standart application programing interface (API) for accesing databases in java?

**En Java 8, el estándar para acceder a bases de datos es JDBC (Java Database Connectivity). JDBC es una API que permite a las aplicaciones Java interactuar con bases de datos utilizando SQL. Aquí te doy un resumen de los componentes y características clave de JDBC:**

## ▼ Ej 20.- Pregunta - Packages

Which one of the following statements is true about using packages to organize your code in Java ?

- Packages allow you to limit access to classes, methods, or data from classes outside the package.

## ▼ Ej 21.- Pregunta - Inicialización y operaciones

Forma correcta de inicializar un boleano.

- boolean a = (3>6);

## ▼ Ej 23.- Pregunta - Exceptions

```
class Y{
    public static void main(String[] args) throws IOException {
        try {
            doSomething();
        }catch (RuntimeException exception){
            System.out.println(exception);
        }
    }

    static void doSomething() throws IOException {
        if (Math.random() > 0.5){
                throw new RuntimeException();
        }
    }
}
```

- Adding throws IOException to the main() method signature

## ▼ Ej 24.- Resultado - Herencia y sobreescritura

```
interface Interviewer {
    abstract int interviewConducted();
}

public class Manager implements Interviewer{
    int interviewConducted() {
        return 0;
    }
}
```

### Análisis

- Interfaz implementada adecuadamente.
- Los métodos en las interfaces implícitamente son `public abstract` , entonces hay problemas de compilación por reducción de visibilidad.

### Respuesta

**No compila**

## ▼ Ej 25.- Pregunta - Herencia y sobrecarga

```
class Arthropod {
    public void printName(double Input){
        System.out.println("Arth");
    }
}

class Spider extends Arthropod {
        public void printName(int input) {
            System.out.println("Spider");
        }

        public static void main(String[] args) {
            Spider spider = new Spider();
            spider.printName(4);
            spider.printName(9.0);
        }
}
```

### Análisis

- Se tiene la clase `Arthropod` con un método `printName()` .
- Se tiene la clase `Spider` que hereda de `Arthropod` con el método `printName()` .
- El método `printName()` de `Spider` no hace Override, no se tiene la misma firma.
- En el método `main()` se instancia un objeto `Spider` en variable de tipo `Spider` .
- Se invoca al método que recibe un `int` , se imprime `Spider` .
- Se invoca al método que recibe un `double` y es heredado del padre, se imprime `Arth`

### Respuesta

`Spider Arth`

## ▼ Ej 26.- Pregunta - ENUM

```
public class Main {
    public enum Days {Mon,Tue, Wed}
        public static void main(String[] args) {
            for (Days d:Days.values()) {
                Days[] d2 = Days.values();
                System.out.println(d2[2]);
            }
    }
}
```

### Análisis

Define un `enum Days` con 3 valores.

El método `values()` → Returns an array containing the constants of this enum type, in the order they're declared.

El método `main()` tiene un ciclo mejorado, donde itera un arreglo con los valores del `enum` y lo trabaja en un `d` de tipo `Days` . Entonces el ciclo se ejecuta 3 veces.

Dentro del ciclo, se crea un arreglo de tipo `Days` y se le asigna el array que retorna el método `values()` .

Se imprime el valor con índice 2 de `d2` → se imprime `Wed` .

Siguen dos iteraciones → finalmente se imprime `wed` `wed` .

### Respuesta

`wed wed wed`

## ▼ Ej 27.- Pregunta - Operaciones XOR

```
public class Main{
    public static void main(String[] args) {
        boolean x = true, z = true;
        int y = 20;
        x = (y!=10)^(z=false);
        System.out.println(x + " " + y + " "+ z);
    }
}
```

### Análisis

Se definen 2 booleanos `x = true` y `z = true` , y un entero `y = 20` .

En la operación realizada se encuentra un `^` (XOR) → Diferentes → `true` , Iguales → `false`

`x = (true)^(false)` → `x = true`

Entonces al final `x = true` `y = 20` `z = false`

### Respuesta

`true 20 false`

## ▼ Ej 28.- Pregunta - Bloques de inicialización

```
class InitializacionOrder {
    static {add(2);}
    static void add(int num){
        System.out.println(num+"");
    }
    InitializacionOrder(){add(5);}
    static {add(4);}
    {add(6);}
    static {new InitializacionOrder();}
    {add(8);}
    public static void main(String[] args) {}
}
```

### Análisis

- Se tiene el método `add()` que imprime el `num` que recibe.
- Se ejecutan los bloques estáticos, entonces se imprime `2`
- Luego se ejecuta el bloque que imprime `4`
- Posteriormente se instancia un `InitializacionOrder()` , al crear la instancia, se ejecutan los bloques de inicialización de instancia, entonces se imprime `6` y `8` . Los bloques de inicialización de invocan antes que el constructor.
- Finalmente se invoca al constructor y se imprime `5`

### Respuesta

`2 4 6 8 5`

## ▼ Ej 29.- Pregunta - Pool String

```
public class Main {
    public static void main(String[] args) {
        String message1 = "Wham bam";
        String message2 = new String("Wham bam");
        if (message1!=message2){
            System.out.println("They dont match");
        }else {
            System.out.println("They match");
        }
    }
}
```

### Análisis

- Se crea una variable de tipo `String` con el mensaje `"Wham bam"`
- Se crea otra variable de tipo `String` a través del constructor con el mismo mensaje que el objeto anterior.
- Al usar el constructor, no se reutiliza el objeto asignado a `message1` , se tienen dos objetos independientes.
- `!=` para `String` verifica que la referencia es la misma, entonces el argumento del condicional es `true` .

- Se imprime `they dont match`

### Respuesta

`They dont match`

## ▼ Ej 30.- Pregunta - Excepciones

```
class Mouse{
    public String name;
    public void run(){
        System.out.println("1");
        try{
            System.out.println("2");
            name.toString();
            System.out.println("3");
        }catch(NullPointerException e){
            System.out.println("4");
            throw e;
        }
        System.out.println("5");
    }
    public static void main(String[] args) {
        Mouse jerry = new Mouse();
        jerry.run();
        System.out.println("6");
    }
}
```

### Análisis

- Se instancia un `Mouse` y se ejecuta el método `run()`
- Se imprime `1` y `2`, se intenta ejecutar el método `toString()` del atributo `name` pero este contiene un `null` .
- Se lanza la excepción `NullPointerException` , entonces en el catch se imprime `4` y se relanza la excepción
- Esta nueva excepción no es manejada y da fin al programa.

### Respuesta

`1 2 4 NullPointerException`

## ▼ Ej 31.- Pregunta - try with resources y conexión

```
public class Main {
    public static void main(String[] args) {
        try (Connection con = DriverManager.getConnection(url, uname, pwd)){
            Statement stmt =con.createStatement();
            System.out.print(stmt.executeUpdate("INSERT INTO User VALUES (500, 'Rames
h')"));
        }
    }
}
```

### Análisis

- `try (Connection con = DriverManager.getConnection(url, uname, pwd))`: Esto asegura que el `Connection` se cerrará automáticamente después de que el bloque `try` termine, incluso si ocurre una excepción.
- `executeUpdate` devuelve un `int` que indica el número de filas afectadas por la consulta. Esto se imprimirá en la consola.

### Respuesta

**Se imprime** `1`

---

## ▼ Ej 32.- Pregunta - Override

```
class MarvelClass{
    public static void main (String [] args){
        MarvelClass ab1, ab2, ab3;
        ab1 =new MarvelClass();
        ab2 = new MarvelMovieA();
        ab3 = new MarvelMovieB();
        System.out.println ("the profits are " + ab1.getHash()+ "," +
        ab2.getHash()+","+ab3.getHash());
    }

    public int getHash(){
        return 676000;
    }
}

class MarvelMovieA extends MarvelClass{
    public int getHash (){
        return 18330000;
    }
}
class MarvelMovieB extends MarvelClass {
    public int getHash(){
        return 27980000;
    }
}
```

### Análisis

- Se crean 3 variables de instancia de tipo `MarvelClass` .
- En ellas se apunta a 3 nuevos objetos, `ab1 -> MarvelClass` , `ab2 -> MarvelMovieA` y `ab3 -> MarvelMovieB`
- `MarvelMovieA` y `MarvelMovieB` heredan de `MarvelClass` .
- Se imprime `the profits are` seguido de el retorno de los métodos `getHash()` de cada objeto
- Para `ab1` → `return 676000;` , para `ab2` → `return 18330000;` y para `ab3` → `return 27980000;`

### Respuesta

**Se imprime** `the profits are 676000 18330000 27980000`

---

## ▼ Ej 33.- Pregunta - Arrays

```
class Song{
    public static void main (String [] args){
        String[] arr = {"DUHAST","FEEL","YELLOW","FIX YOU"};
        for (int i =0; i <= arr.length; i++){
            System.out.println(arr[i]);
        }
    }
}
```

### Análisis

- Se crea un arreglo se `String` con 4 elementos.
- Se utiliza un ciclo `for` que va de `i=0` hasta `i=4`
- En cada iteración se imprime el elemento que contiene.
- Se imprime hasta el índice 3 y para la iteración 4 se lanza la excepción por estar fuera del índice.

### Respuesta

`DUHAST FEEL YELLOW FIX YOU ArrayIndexOutOfBoundsException`

---

## ▼ Ej 34.- Pregunta - Ciclos

```
class Menu {
    public static void main(String[] args) {
        String[] breakfast = {"beans", "egg", "ham", "juice"};
        for (String rs : breakfast) {
            int dish = 2;
            while (dish < breakfast.length) {
                System.out.println(rs + "," + dish);
                dish++;
            }
        }
    }
}
```

### Análisis

- Se crea un arreglo de `Strings` con 4 elementos.
- Se utiliza un ciclo mejorado para iterar, dentro del ciclo se define un entero `dish = 2` . El ciclo mejorado realizará 4 iteraciones
- Dentro del ciclo se encuentra un ciclo `while` que se ejecutara siempre y cuando `dish` sea menor a `4` , entonces este ciclo se ejecuta 2 veces
- Dentro de `while` se imprime `rs` , que corresponde al elemento de `breakfast` , y el valor de `dish` .

### Flujo

- **Se imprime** `rs -> beans` **y** `dish -> 2`
- **Se imprime** `rs -> beans` **y** `dish -> 3`

- **Se imprime `rs -> egg` y `dish -> 2`**
- **Se imprime `rs -> egg` y `dish -> 3`**
- **Se imprime `rs -> ham` y `dish -> 2`**
- **Se imprime `rs -> ham` y `dish -> 3`**
- **Se imprime `rs -> juice` y `dish -> 2`**
- **Se imprime `rs -> juice` y `dish -> 3`**

## ▼ Ej 35.- Pregunta - StringBuilder - StringBuffer

Which of the following statement are true:

- `StringBuilder` **es generalmente más rápido que** `StringBuffer`: Esto es cierto. La principal razón es que `StringBuilder` no está sincronizado, lo que significa que no tiene las sobrecargas asociadas con la sincronización que `StringBuffer` sí tiene. En entornos no concurrentes, `StringBuilder` tiende a ser más eficiente porque no necesita el overhead de las operaciones de sincronización.
- `StringBuffer` tiene métodos sincronizados, lo que garantiza que su estado interno sea seguro para el acceso concurrente desde múltiples hilos. Esto significa que puedes usar `StringBuffer` en aplicaciones multihilo donde múltiples hilos puedan estar manipulando el mismo objeto `StringBuffer` simultáneamente sin causar problemas de consistencia.
- Por otro lado, `StringBuilder` no tiene sincronización en sus métodos, lo que lo hace no seguro para el acceso concurrente. Si necesitas usar `StringBuilder` en un entorno multihilo, tendrás que manejar la sincronización tú mismo o usar algún mecanismo de sincronización externo para asegurar que el acceso sea seguro.

## ▼ Ej 36.- Pregunta - Cast - `equals()` - Override

```
class CustomKeys{
    Integer key;
    CustomKeys(Integer k){
        key = k;
    }
    public boolean equals(Object o){
        return ((CustomKeys)o).key==this.key;
    }
}
```

### Análisis

**El código tiene problemas, pues en el método `equals()` se asume que el `Object` que se recibe es de tipo `CustomKeys` y no se verifica. Además la evaluación entre `key` únicamente es valida si el objeto `Integer` tiene la misma dirección.**

## ▼ Ej 37.- Pregunta - Excepciones

The catch clause is of the type:

Throwable
Exception but NOT including RuntimeException
CheckedException
RunTimeException
Error

## ▼ Ej 38.- Pregunta -Loops

An enhanced for loop:

- also called for each, offers simple syntax to iterate through a collection but it can´t be used to delete elements of a collection

## ▼ Ej 39.- Pregunta - Herencia

which of the following methods may appear in class Y, which extends x ?
public void doSomething(int a, int b){...}

## ▼ Ej 40.- Pregunta -Pool Strings

```
public class Main {
    public static void main(String[] args) {
        String s1= "Java";
        String s2 = "java";
        if (s1.equalsIgnoreCase(s2)){
            System.out.println ("Equal");
        } else {
            System.out.println ("Not equal");
        }
    }
}
```

### Análisis

- Se crea `s1 = "Java"` y `s2 = "java"` y se evalúa `s1.equalsIgnoreCase(s2)` .
- El método `equalsIgnoreCase()` para `String` evalúa el contenido de la cadena ignorando el Case.
- Entonces la evaluación dentro del `if` resulta verdadera.

### Respuesta

**Se imprime** `Equal`

## ▼ Ej 41.- Pregunta - `CompareTo()`

```
class App {
    public static void main(String[] args) {
        String[] fruits = {"banana", "apple", "pears", "grapes"};
        // Ordenar el arreglo de frutas utilizando compareTo
        Arrays.sort(fruits, (a, b) -> a.compareTo(b));
        // Imprimir el arreglo de frutas ordenado
        for (String s : fruits) {
            System.out.println(""+s);
        }
    }
}
```

### Análisis

- Se inicializa el arreglo `fruits` con 4 elementos

- `sort()` es un método `static` perteneciente a la clase `Array` .
- `ComparteTo()` es un método de la clase `String` que compara dos cadenas alfabéticamente, retorna negativo si `a` es menor que `b` , `0` si son iguales y positivo si `a` es mayor que `b`
- `sort()` recibe el arreglo y un ***comparator,*** la expresión lambda `(a,b) -> a.compareTo(b)`
- En un ciclo mejorado se imprime cada elemento del arreglo.

### Respuesta

`apple banana grapes pears`

## ▼ Ej 42.- Pregunta - Excepciones

```
public class Main {
    public static void main(String[] args) {
        int[] countsofMoose = new int [3];
        System.out.println(countsofMoose[-1]);
    }
}
```

### Análisis

- Se instancia un arreglo de enteros `countsifMoose` con longitud de 3.
- Se intenta acceder al elemento con índice `-1`

### Respuesta

**Se lanza la excepción `arrayIndexOutOfBoundsException`**

## ▼ Ej 43.- Pregunta - Constructores e inicialización de atributos

```
class Salmon{
    int count;
    public void Salmon (){
        count = 4;
    }
    public static void main(String[] args) {
        Salmon s = new Salmon();
        System.out.println(s.count);
    }
}
```

### Análisis

- Se instancia la clase `Salmon`
- Se imprime el atributo `count`

### Respuesta

**Se imprime `0` , la variable `count` no está inicializada, los constructores no llevan retorno**

## ▼ Ej 44.- Pregunta - Inicialización

```
class Circuit {
    public static void main(String[] args) {
        runlap();
        int c1=c2;
        int c2 = v;
    }
    static void runlap(){
        System.out.println(v);
    }
    static int v;
}
```

### Análisis

- Se ejecuta el método `static` `runlap()` y se imprime `v = 0`
- `c2` no está definido

### Respuesta

**Error de compilación**

---

## ▼ Ej 45.- Pregunta - Procedencia de operaciones

```
class Foo {
    public static void main(String[] args) {
        int a = 10;
        long b = 20;
        short c = 30;
        System.out.println(++a + b++ c);
    }
}
```

### Análisis

- Entre `b++` y `c` no existe una operación

### Respuesta

**Error de compilación**

---

## ▼ Ej 46.- Pregunta - Varargs

```
public class Shop{
    public static void main(String[] args) {
        new Shop().go("welcome",1);
        new Shop().go("welcome", "to", 2);
    }
    public void go (String... y, int x){
        System.out.print(y[y.length-1]+"");
    }
}
```

### Análisis

- Los varargs deben ir al final de la lista de parámetros

### Respuesta

**Error de compilación**

## ▼ Ej 47.- Pregunta - Constructores

```
class Plant {
    Plant() {
        System.out.println("plant");
    }
}

class Tree extends Plant {
    Tree(String type) {
        System.out.println(type);
    }
}

class Forest extends Tree {
    Forest() {
        super("leaves");
        new Tree("leaves");
    }
    public static void main(String[] args) {
        new Forest();
    }
}
```

### Análisis

- Se instancia un objeto `Forest`
- El constructor de `Forest` invoca al constructor de su padre con argumento `"leaves"`
- Se invoca al constructor de `Tree` , este invoca el constructor de `Plant` .
- Se imprime `"plant"` , se imprime `"leaves"` .
- A continuación se instancia un nuevo `Tree` dentro del constructor de `Forest` .
- Se invoca el constructor de `Tree` con argumento `"leaves"` .
- Se invoca el constructor de `Plant` , se imprime `"plant"` y `"leaves"`

### Respuesta

**Se imprime** `plant leaves plant leaves`

## ▼ Ej 48.- Pregunta - String `.intern()`

```
class Test {
    public static void main(String[] args) {
        String s1 = "hello";
```

```
        String s2 = new String ("hello");
        s2=s2.intern(); // el intern() asigna el mismo hash conforme a la cadena
        System.out.println(s1==s2);
    }
}
```

### Análisis

- Se instancias dos `String` , ambos diferentes debido al modo de creación.
- `intern()` crea una copia de la cadena y la añade al pool de strings, en caso de que ya exista no se crea un objeto nuevo y se reutiliza el existente en el pool.
- Entonces `s2.intern()` crea una copia de la cadena y la intenta añadir al pool de strings
- La cadena `"hello"` ya se encuentra en el pool, por lo que no se crea un nuevo objeto y a `s2` se le asigna el objeto referido por `s1` .
- Se compara la referencia de `s1` y `s2`

### Respuesta

**Se imprime** `true`

---

## ▼ Ej 49.- Pregunta - Simulador `.getClass()`

```
class SampleClass{
    public static void main(String[] args) {
        AnotherSampleClass asc =new AnotherSampleClass ();
        SampleClass sc = new SampleClass();
        //sc = asc;
        //TODO CODE
    }
}
class AnotherSampleClass extends SampleClass {}
```

## ▼ Ej 50.- Pregunta - Flujo

```
public class Main {
    public static void main(String[] args) {
        int a= 10;
        int b =37;
        int z= 0;
        int w= 0;

        if (a==b){
            z=3;
        }else if(a>b){
            z=6;
        }
        w=10*z;
        System.out.println(z);
```

```
    }
}
```

### Análisis

- Se instancian variables tipo `int` , se evalúa si `a = 10` es igual a `b = 37` → `false` y no se entra al `if`
- Se evalúa `else if` con `a = 10` mayor que `b = 37` → `false`
- Se realiza la operación `w = 10 * 0`

### Respuesta

**Se imprime `0`**

## ▼ Ej 51.- Pregunta - Inception `java.lang.StackOverflowError`

```
public class Main{
    public static void main(String[] args) {
        Course c = new Course();
        c.name="java";
        System.out.println(c.name);
    }
}

class Course {
    String name;
    Course(){
        Course c = new Course();
        c.name="Oracle";
    }
}
```

### Análisis

- Se instancia un `Course` en el método `main()`
- El constructor de `Course` instancia un nuevo `Course` en una variable local
- Se llama nuevamente al constructor y el ciclo se repite.

### Respuesta

`Exception in thread "main" java.lang.StackOverflowError`

## ▼ Ej 52.- Pregunta - Inicialización

```
public class Main{
    public static void main(String[] args) {
        String a;
        System.out.println(a.toString());
    }
}
```

### Análisis

- Se crea una variable local `String a`
- Se intenta imprimir `a.toString()`

### Respuesta

**Las variables no se inicializan automáticamente, entonces error de compilación →** `java: variable a might not have been initialized`

---

## ▼ Ej 53.- Pregunta - polimorfismo en `+`

```
public class Main{
    public static void main(String[] args) {
        System.out.println(2+3+5);
        System.out.println("+"+2+3+5);
        System.out.println(2+3+5+"+");
        //System.out.println(2+3+5+"+"+100+11);
    }
}
```

### Análisis

- Se realiza la suma `2+3+5 = 10` y se imprime.
- Al tener un `String` el operador `+` toma la función de concatenar los valores, entonces se imprime `+235`
- Las operaciones se realizan de izquierda a derecha, entonces se suman los números y posteriormente se concatena.

### Respuesta

`10 + 235 10+`

---

## ▼ Ej 54.- Pregunta - Flujo

```
public class Main {
    public static void main(String[] args) {
        int a = 2;
        int b = 2;
        if (a==b)
            System.out.println("Here1");
        if (a!=b)
            System.out.println("here2");
        if (a>=b)
            System.out.println("Here3");
    }
}
```

### Análisis

- Se compara si `a = 2` es igual a `b = 2` → `true` y se imprime `"Here1"` .
- Se compara si `a` es diferente de `b` → `falso` .
- Se compara si `a` es mayor o igual a `b` → `true` y se imprime `"Here3"`

### Respuesta

**Se imprime** `Here1 Here3`

## ▼ Ej 55.- Pregunta - Métodos

```
public class Main extends count {
    public static void main(String[] args) {
    int a = 7;
    System.out.println(count(a,6));
    }
}
class Count {
    int count(int x, int y){return x+y;}
}
```

### Análisis

- `Main` hereda de `Count` y no hay Override.
- En el método `main()` se crea una variable local `int a = 7`
- Se intenta imprimir el retorno del método `count()`

### Respuesta

**Error de compilación → `count()` es un método de instancia**

## ▼ Ej 56.- Pregunta - Sobrecarga del método `main()`

```
class Trips{
    void main(){
        System.out.println("Mountain");
    }
    static void main (String args){
        System.out.println("BEACH");
    }
    public static void main (String [] args){
        System.out.println("magic town");
    }
    void mina(Object[] args){
        System.out.println("city");
    }
}
```

### Análisis

- La case `Trips` cuenta con el método `main` sobrecargado

### Respuesta

**Se imprime** `magic town`

## ▼ Ej 57.- Pregunta - Procedencia de operaciones

```
public class Main{
    public static void main(String[] args) {
        int a=0;
        System.out.println(a++ +2);
        System.out.println(a);
    }
}
```

### Análisis

- Se define una variable local `int a = 0`
- Se intenta imprimir la operación `a++ +2` → `0 +2` → `2` y `a = 1`
- Se imprime `a = 1`

### Respuesta

**Se imprime `2 1`**

---

## ▼ Ej 58.- Pregunta - ArrayList

```
public class Main{
    public static void main(String[] args) {
        List<E> p =new ArrayList<>();
        p.add(2);
        p.add(1);
        p.add(7);
        p.add(4);
    }
}
```

### Análisis

- Se crea un `ArrayList` y se le agregan elementos `Integer`

### Respuesta

**Se espera que la lista contenga elementos de tipo `E` → Error de compilación**

---

## ▼ Ej 59.- Pregunta - Palabras reservadas

```
public class Car{
    private void accelerate(){
        System.out.println("car acelerating");
    }
    private void break(){
        System.out.println("car breaking");
    }
    public void control (boolean faster){
        if(faster==true)
            accelerate();
        else
```

```
            break();
    }
    public static void main (String[] args){
        Car car = new Car();
        car.control(false);
    }
}
```

### Respuesta

`break` es una palabra reservada → Error de compilación

## ▼ Ej 60.- Pregunta - Sobrecarga y selección de métodos

```
class App {
    App() {
        System.out.println("1");
    }
    App(Integer num) {
        System.out.println("3");
    }
    App(Object num) {
        System.out.println("4");
    }
    App(int num1, int num2, int num3) {
        System.out.println("5");
    }
    public static void main(String[] args) {
        new App(100);
        new App(100L);
    }
}
```

### Análisis

- Se crea un nuevo `App` y como argumento se pasa un `int 100`
- Se utiliza el constructor `App(Integer num)` por el autoboxing y se imprime 3
- Se crea un nuevo `App` y como argumento se pasa un `long 100L`
- Se utiliza el constructor `App(Object num)` por el autoboxing y se imprime 4

### Respuesta

**Se imprime** `3 4`

## ▼ Ej 61.- Pregunta - operador ternario

```
class App {
    public static void main(String[] args) {
        int i=42;
        String s = (i<40)?"life":(i>50)?"universe":"everething";
        System.out.println(s);
```

```
    }
}
```

### Análisis

- Se define una variable local `int i = 42` .
- Se asigna una variable local `String s` con `life` en caso que `i = 42` sea menor a `40` , entonces no se asigna, ahora se evalúa si `i = 42` es mayor `50` , entonces se asigna `"everething"`

### Resultado

**Se imprime `everething`**

---

## ▼ Ej 62.- Pregunta - `NumberFormat()` - Excepciones

```
class App {
    App(){
        System.out.println("1");
    }
    App(int num){
        System.out.println("2");
    }
    App(Integer num){
        System.out.println("3");
    }
    App(Object num){
        System.out.println("4");
    }
    public static void main(String[] args) {
        String[] sa = {"333.6789","234.111"};
        NumberFormat inf= NumberFormat.getInstance();
        inf.setMaximumFractionDigits(2);
        for(String s:sa){
            System.out.println(inf.parse(s));
        }
    }
}
```

### Análisis

- Se instancia un arreglo de `String` con 2 valores.
- Se define una variable de tipo `NumberFormat` y se obtiene una instancia de su clase con `getInstance()` .
- `java.text.Format` → `java.text.NumberFormat`
- `NumberFormat` es una clase abstracta, provee una interfaz o métodos para dar formato y convertir números.
- El método `setMaximumFractionDigits(2)` establece el máximo número de dígitos permitidos en la parte decimal de un número y no retorna nada.
- Se entra en un ciclo mejorado recorriendo el arreglo `sa` .
- En cada iteración se imprime el retorno del método `parse(s)` de la clase `NumberFormat` .
- El método `parse(String source)` retorna un `Number` y convierte texto en un número.

- `parse()` puede lanzar la excepción checked `ParseException`
- No se está manejando la excepción

### Respuesta

**Error de compilación, no se está manejando la excepción**

### Caso alternativo

Si la excepción se maneja, se imprimen los números tal cuales están en el arreglo.

Para hacer valido el formato definido se utiliza el método `format(Number)` , entonces la línea que imprime queda `inf.format(inf.parse(s))` y así se hace valido el número de dígitos redondeando hacia arriba cuando es más de `0.5` → se imprime `333.68 234.11` .

## ▼ Ej 63.- Pregunta - Pool String

```
class Y{
    public static void main(String[] args) {
        String s1 = "OCAJP";
        String s2 = "OCAJP" + "";
        System.out.println(s1 == s2);
    }
}
```

### Análisis

- Se instancia un `String s1` con la cadena `"OCAJP"`
- Se instancia un `String s2` con la cadena `"OCAJP"` concatenada a la cadena vacía `""` , entonces el resultado de la concatenación es `"OCAJP"` , cadena que ya se encuentra en el pool de `String` , por lo tanto `s2` toma la referencia del objeto en `s1`
- Se compara si ambas cadenas son la misma.

### Respuesta

**Se imprime `true`**

## ▼ Ej 64.- Pregunta - Switch

```
class Y{
    public static void main(String[] args) {
        int score = 60;
        switch (score) {
            default:
                System.out.println("Not a valid score");
            case score < 70:
                System.out.println("Failed");
                break;
            case score >= 70:
                System.out.println("Passed");
                break;
```

```
    }
}
```

### Análisis

- Los casos del switch no están definidos, además de que esperan un valor boolean.

### Respuesta

**Error de compilación**

## ▼ Ej 65.- Pregunta - Procedencia de operaciones unarias

```
class Y{
    public static void main(String[] args) {
        int a = 100;
        System.out.println(-a++);
    }
}
```

### Análisis

- Se pretende imprimir `-a++` , se utiliza a y se hace negativo, posteriormente de incrementa

### Respuesta

**Se imprime** `-100`

### Alternativas

`System.out.println(-++a);`

`-` **(operador unario negativo)**: Después de que `a` ha sido incrementado, se aplica el operador unario negativo, lo que invierte el signo del valor de `a`.

## ▼ Ej 66.- Pregunta - Switch

```
class Y{
    public static void main(String[] args) {
        byte var = 100;
        switch(var) {
            case 100:
                System.out.println("var is 100");
                break;
            case 200:
                System.out.println("var is 200");
                break;
            default:
                System.out.println("In default");
        }
    }
}
```

### Análisis

Un switch que recibe un `byte = 100`

Hay definido un caso que excede el valor máximo de los byte.

> Recordar que el literal value es un Int

### Respuesta

Error de compilación `java: incompatible types: possible lossy conversion from int to byte`

## ▼ Ej 67.- Pregunta - Herencia casteo

```
class Y{
        public static void main(String[] args) {
            A obj1 = new A();
            B obj2 = (B)obj1;
            obj2.print();
        }
    }

class A {
        public void print(){
            System.out.println("A");
        }
    }
    class B extends A {
        public void print(){
            System.out.println("B");
        }
}
```

### Análisis

- Se crea un objeto `obj1` de tipo `A` en una variable de tipo `A` .
- Se crea una variable `obj2` de tipo `B` y se le intenta pasar la referencia `obj1` haciendo un cast de `B`

### Respuesta

`ClassCastException` el objeto nació como `A`

## ▼ Ej 68.- Pregunta - Switch

```
class Y{
    public static void main(String[] args) {
        String fruit = "mango";

        switch (fruit) {
            default:
                System.out.println("ANY FRUIT WILL DO");
            case "apple":
                System.out.println("APPLE");
            case "mango":
                System.out.println("MANGO");
            case "banana":
```

```
                System.out.println("BANANA");
            break;
        }
    }
}
```

### Análisis

- Se define la variable `String fruit = "mango"` .
- Únicamente hay un `break` al final.

### Respuesta

**Se imprime `MANGO BANANA` , no importa la posición de `default`**

## ▼ Ej 69.- Pregunta - Constructores y herencia

```
abstract class Animal {
    private String name;
    Animal(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}

class Dog extends Animal {
    private String breed;
    Dog(String breed) {
        this.breed = breed;
    }
    Dog(String name, String breed) {
        super(name);
        this.breed = breed;
    }
    public String getBreed() {
        return breed;
    }
}

class Test {
        public static void main(String[] args) {
        Dog dog1 = new Dog("Beagle");
        Dog dog2 = new Dog("Bubbly", "Poodle");
        System.out.println(dog1.getName() + ":" + dog1.getBreed() +
        ":" +
        dog2.getName() + ":" + dog2.getBreed());
    }
}
```

### Análisis

- Se instancia un `Dog` utilizando el constructor de 1 parámetro.
- El constructor de `Dog` llama a `super()` ,pero no existe este constructor en `Animal`

### Respuesta

**Error de compilación**

## ▼ Ej 70.- Pregunta - NumberFormat

```
public class Main {
    public static void main(String[] args) throws ParseException {
        String[]sa = {"333.6789","234.111"};
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        for (String s: sa) {
            System.out.println(nf.parse(s));
        }
    }
}
```

### Análisis

- Se instancia un arreglo de `String` de 2 lugares.
- Se obtiene la instancia de un `NumberFormat`
- Se establece el máximo número de dígitos decimales que se muestran.
- En un ciclo mejorado se imprime el método `parse()` pasando cada elemento del arreglo `sa`
- El método `parse()` únicamente convierte, no aplica formato → `Format()`
- `parse()` lanza una checked exception, entonces se informa en el `main` con `throws`

### Repuesta

**Se imprimen los números tal cual** `33.6789` `234.111`

### Alternativas

**Revisar Ej 62**

## ▼ Ej 71.- Pregunta - Queue ArrayDeque

```
import java.util.ArrayDeque;
import java.util.Queue;

public class Main{
    public static void main(String[] args){
        Queue<String> products = new ArrayDeque<String>();
        products.add("p1");
        products.add("p2");
        products.add("p3");
        System.out.println(products.peek());
        System.out.println(products.poll());
```

```
        System.out.println("");
        products.forEach(s -> {System.out.println(s);});
    }
}
```

### Análisis

- Se crea un `ArrayDeque` para `String` .
- Se añaden 3 cadenas.
- El método `peek()` retorna el head de la queue (primer elemento), pero no la elimina.
- El método `poll()` retorna el head de la queue, y la remueve del Array.

### Respuesta

**Se imprime** `p1 p1 p2 p3`

## ▼ Ej 72.- Pregunta - Procedencia de operaciones

```
public class Main {
    public static void main(String[] args){
        System.out.println(2+3+5);
        System.out.println("+"+2+3*5);
    }
}
```

### Análisis

- Se realiza la operación `2+3+5`
- Se realiza la segunda operación `"+" concatenando 2 concatenando 15`

### Respuesta

**Se imprime** `10 + 215`