ESTAS PREGUNTAS SON DESDE LA PÁGINA 45

1 . Which three are bad practices?

- Checking for an IOException and ensuring that the program can recover if one occurs.
- Checking for ArrayIndexOutOfBoundsException and ensuring that the program can recover if one occurs.
- Checking for FileNotFoundException to inform a user that a filename entered is not valid.
- Checking for Error and, if necessary, restarting the program to ensure that users are unaware problems.
- Checking for ArrayIndexOutOfBoundsException when iterating through an array to determine when all elements.-have been visited.

Explicación:

Las tres malas prácticas identificadas son:

1. **Checking for ArrayIndexOutOfBoundsException and ensuring that the program can recover if one occurs.**
   - **Mala práctica**. `ArrayIndexOutOfBoundsException` generalmente indica un error en la lógica del programa, como intentar acceder a una posición fuera de los límites de un array. Es mejor prevenir este tipo de excepciones usando comprobaciones de límites antes de acceder a los elementos del array.
2. **Checking for Error and, if necessary, restarting the program to ensure that users are unaware problems.**
   - Tratar de manejar estos errores reiniciando el programa no es recomendable, ya que `Error` no está destinado a ser manejado por aplicaciones normales. Los errores deben ser abordados y corregidos por desarrolladores o administradores del sistema.
3. **Checking for ArrayIndexOutOfBoundsException when iterating through an array to determine when all elements have been visited.**
   - Utilizar una excepción para controlar los límites de un array durante la iteración es una mala práctica. Es preferible usar bucles que verifiquen explícitamente los límites del array.

Las otras opciones (`IOException` y `FileNotFoundException`) representan buenas prácticas de manejo de excepciones y no se consideran malas prácticas.

2. What is the result?

```
Given
public static void main(String[] argos){
      int[][] array2D = {{0,1,2}, {3,4,5,6}};
      System.out.print(array2D[0].length + " ");
      System.out.print(array2D[1].getClass().isArray() + " ");
```

```
System.out.print(array2D[0][1]);
```

3false3

3false1

2false1

3true1 primero nos pide el tamaño del array en la posición [0] después con el metodo isArray que proviene de la clase Class nos devuelve true ya que el elemento en la posición [1] es un array y finalmente nos indica el índice 1 del array en la posición 0.

2true3

3. Which two statements are true?

- An interface CANNOT be extended by another interface.
- An abstract class can be extended by a concrete class.
- An abstract class CANNOT be extended by an abstract class.
- An interface can be extended by an abstract class.
- An abstract class can implement an interface.
- An abstract class can be extended by an interface.

explicación:

An interface CANNOT be extended by another interface. Falso. En Java, una interfaz puede extender otra interfaz. Esto se hace usando la palabra clave `extends`.

An abstract class can be extended by a concrete class. Verdadero. Una clase abstracta puede ser extendida por una clase concreta. La clase concreta debe proporcionar implementaciones para todos los métodos abstractos de la clase abstracta.

An abstract class CANNOT be extended by an abstract class. Falso. Una clase abstracta puede ser extendida por otra clase abstracta. La subclase abstracta puede optar por no implementar los métodos abstractos de la superclase.

An interface can be extended by an abstract class. Falso. Una interfaz no puede ser extendida por una clase abstracta. Sin embargo, una clase abstracta puede implementar una interfaz.

An abstract class can implement an interface. Verdadero. Una clase abstracta puede implementar una interfaz, pero no está obligada a implementar todos los métodos de la interfaz. La implementación completa de los métodos puede ser diferida a las subclases concretas.

An abstract class can be extended by an interface. Falso. Las interfaces no pueden extender clases abstractas o concretas. Las interfaces solo pueden extender otras interfaces.

4. What is the result?

```
Given:
class Alpha {String getType(){return "alpha";}}
class Beta extends Alpha {String getType(){return "beta";}}
public class Gamma extends Beta {String getType(){return "gamma"}
      public static void  main(String [] argos) {
            Gamma g1 =(Gamma) new Alpha();
            Gamma g2 =(Gamma) new Beta();
            System.out.print(g1.getType() + " "+ g2.getType());
      }
}
```

- Gamma gamma
- Beta beta
- Alpha beta
- Compilation fails

Casting Inapropiado: Intentar convertir instancias de `Alpha` y `Beta` a `Gamma` resulta en `ClassCastException`, haciendo que el código no compile correctamente.

5. Which five methods, inserted independently at line 5, will compile? (Choose five)

```
1 public class Blip{
2     protected int blipvert(int x){return 0
3 }
4 class Vert extends Blip{
5     //insert code here
6 }
```

- Private int blipvert(long x) { return 0; }
- Protected int blipvert(long x) { return 0; }
- Protected long blipvert(int x, int y) { return 0; }
- Public int blipvert(int x) { return 0; }
- Private int blipvert(int x) { return 0; }
- Protected long blipvert(int x) { return 0; }
- Protected long blipvert(long x) { return 0; }

Explicación:

```
private int blipvert(long x) { return 0; }
```

- **Compila**. Este método define una nueva sobrecarga del método `blipvert` en la subclase `Vert`. Aunque el método es `private`, lo cual significa que no puede ser sobrescrito, es válido para la sobrecarga. La visibilidad `private` no afecta la

posibilidad de sobrecargar métodos ya que la sobrecarga está relacionada con la firma del método (nombre y tipos de parámetros), no con la visibilidad.

```
protected int blipvert(long x) { return 0; }
```

- **Compila**. Este método es una sobrecarga válida del método `blipvert` heredado de `Blip`. La firma del método cambia porque el tipo del parámetro es diferente (de `int` a `long`). La visibilidad `protected` es válida para la subclase `Vert`.

```
protected long blipvert(int x, int y) { return 0; }
```

- **Compila**. Este método también es una sobrecarga válida porque el número de parámetros es diferente. La firma del método cambia porque se ha añadido un segundo parámetro (`int y`). La visibilidad `protected` es válida en la subclase `Vert`.

```
public int blipvert(int x) { return 0; }
```

- **Compila**. Cambiar la visibilidad de `protected` a `public` es permitido si la firma del método permanece igual (nombre del método y tipos de parámetros). En este caso, la firma es la misma y el cambio de visibilidad de `protected` a `public` es válido.

```
private int blipvert(int x) { return 0; }
```

- **No compila**. No se puede cambiar la visibilidad de un método heredado de `protected` a `private` en una subclase. En Java, no puedes reducir la visibilidad de un método en la subclase. La visibilidad `private` no permite que el método sea visible en la subclase y, por lo tanto, no puede ser una sobrecarga válida del método heredado.

```
protected long blipvert(int x) { return 0; }
```

- **Compila**. Este método cambia el tipo de retorno a `long`, pero la firma del método sigue siendo diferente porque el tipo de parámetro es el mismo. La visibilidad `protected` es adecuada en la subclase `Vert`.

```
protected long blipvert(long x) { return 0; }
```

- **Compila**. Cambiar el tipo de parámetro es una forma válida de sobrecargar el método. La visibilidad `protected` es válida para la subclase `Vert`, y la firma del método cambia porque el tipo del parámetro es diferente (de `int` a `long`).

6. Which two independently, will allow Sub to compile? (Choose two)

```
Given:
1. class Super {
2.   private int a;
```

```
3. protected Super(int a){this.a = a;}
4. }
…
11. class Sub extends Super{
12.   public Sub(int a){ super(a);}
13.   public Sub(){this.a = 5;}
14. }
```

- Change line 2 to: public int a;
- Change line 13 to: public Sub(){ super(5);}
- Change line 2 to: protected int a;
- Change line 13 to: public Sub(){ this(5);}
- Change line 13 to: public Sub(){ super(a);}

Change line 13 to: public Sub() { super(5); }

- **Válido:** Este cambio hace que el constructor sin parámetros de `Sub` llame al constructor `Super(int a)` con el valor `5`. Es válido porque `Super(int a)` es un constructor protegido y accesible desde `Sub`.

Change line 13 to: public Sub() { this(5); }

- **Válido:** Este cambio hace que el constructor sin parámetros de `Sub` llame al constructor `Sub(int a)` con el valor `5`. Dado que `Sub(int a)` llama a `Super(int a)`, esta opción es válida siempre que `Sub(int a)` esté definido en `Sub`.

Change line 2 to: public int a;

- **No válido:** Cambiar la visibilidad del campo `a` a `public` permite el acceso directo a `a` desde `Sub`, pero no resuelve el problema del constructor sin parámetros, por lo que no aborda el error de compilación.

Change line 2 to: protected int a;

- **No válido:** Cambiar la visibilidad del campo `a` a `protected` permite el acceso desde `Sub`, pero no resuelve el problema del constructor sin parámetros de `Sub`, por lo que no es una solución completa.

Change line 13 to: public Sub() { super(a); }

- **No válido:** Este cambio intenta pasar `a` al constructor de `Super`, pero `a` no está definido en el contexto del constructor sin parámetros de `Sub`, lo que provoca un error de compilación.

7. What is true about the class Wow?

```
public abstract class Wow{
     private int wow;
```

```
    public Wow(int wow){this.wow = wow;}
    public void wow(){}
    private void wowza(){}
}
```

- It compiles without error.
- It does not compile because an abstract class cannot have private methods
- It does not compile because an abstract class cannot have instance variables.
- It does not compile because an abstract class must have at least one abstract method.
- It does not compile because an abstract class must have a constructor with no arguments

`It compiles without error.`

- **Correcto:** La clase `Wow` es una clase abstracta que cumple con las reglas de Java. Aunque tiene métodos privados y un constructor con un argumento, estos elementos son válidos en una clase abstracta. La clase abstracta no necesita tener métodos abstractos ni constructores sin argumentos. Además, los métodos privados no afectan la capacidad de la clase para compilar, ya que son simplemente métodos internos que no se pueden acceder desde fuera de la clase.

`It does not compile because an abstract class cannot have private methods.`

- **Incorrecto:** Una clase abstracta puede tener métodos privados. Los métodos privados son accesibles sólo dentro de la clase en la que están definidos y no afectan la compilación de la clase abstracta.

`It does not compile because an abstract class cannot have instance variables.`

- **Incorrecto:** Las clases abstractas pueden tener variables de instancia. En este caso, la variable de instancia `wow` está correctamente definida y no causa problemas de compilación.

`It does not compile because an abstract class must have at least one abstract method.`

- **Incorrecto:** Aunque es una buena práctica que las clases abstractas contengan al menos un método abstracto, no es un requisito para la compilación. Una clase abstracta puede no tener métodos abstractos y aún así compilar correctamente.

`It does not compile because an abstract class must have a constructor with no arguments.`

- **Incorrecto:** No es necesario que una clase abstracta tenga un constructor sin argumentos. La clase abstracta `Wow` tiene un constructor con un argumento, lo cual es válido en Java.

8. What is the result?

```
class Atom{
      Atom(){System.out.print("atom ");}
}
class Rock extends Atom{
      Rock(String type){System.out.print(type);}
}
public class Mountain extends Rock{
      Mountain(){
            super("granite ");
            new Rock("granite ");
            }
            public static void main(String[] a){new Mountain();}
}
```

Compilation fails.

Atom granite.

Granite granite.

Atom granite granite.

An exception is thrown at runtime.

Atom granite atom granite.

Explicación. La salida será la combinación de las impresiones de ambos constructores:

1. `"atom "` por la primera llamada al constructor de `Atom` desde `Rock`.
2. `"granite "` por el constructor de `Rock` invocado por `super("granite ")`.
3. `"atom "` por la segunda llamada al constructor de `Atom` cuando se crea una nueva instancia de `Rock`.
4. `"granite "` por el constructor de `Rock` invocado con `new Rock("granite ")`

9. What is printed out when the program is executed?

```
public class MainMethod{
      void main(){
            System.out.println("one");
      }
      static void main(String args){
            System.out.println("two");
      }
      public static void main(String[] args){
            System.out.println("three");
      void mina(Object[] args){
            System.out.println("four");
      }
}
```

- one
- two
- three
- four
- There is no output.

Método `main()` sin parámetros (`void main()`)

- Este método no es el método `main` que Java busca para iniciar la ejecución del programa. Java no lo reconocerá como el punto de entrada del programa.

Método `main(String args)`

- Este es un método estático que acepta un solo argumento de tipo `String`. Aunque es un método válido, no es el método `main` que Java busca para iniciar la ejecución del programa. Además, este método no es llamado en el código proporcionado.

Método `main(String[] args)`

- Este es el método `main` estándar que Java busca para iniciar la ejecución del programa. Es el único método `main` que será ejecutado cuando el programa se inicie. Este método imprimirá `"three"`.

Método `mina(Object[] argos)`

- Este método tiene un nombre diferente y no es llamado desde el `main`, por lo que no se ejecutará.

ESTA PREGUNTA Y LAS QUE SIGUEN SON DESDE LA PÁGINA 125

10. What is the result?

```
public class Test {
	public static void main(String[] args) {
		int b = 4;
		b--;
		System.out.println(--b);
		System.out.println(b);
	}
}
```

22 12 32 33

el operador de predecremento actua antes de que imprima la variable b

11. In Java the difference between throws and throw Is:

- Throws throws an exception and throw indicates the type of exception that the method.
- Throws is used in methods and throw in constructors.

- Throws indicates the type of exception that the method does not handle and throw an exception.

12. What is the result?

```
class Feline {
	public String type = "f ";
	public Feline() {
		System.out.println("feline ");
	}
}
public class Cougar extends Feline {
	public Cougar() {
		System.out.println("cougar ");
	}
	void go() {
		type = "c";
		System.out.println(this.type + super.type);
	}
	public static void main(String[] args) {
		new Cougar.go();
	}
}
```

Cougar c f.

Feline cougar c c.

Feline cougar c f.

Compilation fails.

Explicacion:

Instanciación de Cougar:

- Al crear una instancia de Cougar con new Cougar(), primero se ejecuta el constructor de la superclase Feline, que imprime "feline ".
- Luego se ejecuta el constructor de Cougar, que imprime "cougar "

Método go:

- Dentro del método go, type es asignado a "c". Esto afecta al atributo type de la instancia de Cougar porque type es una variable de instancia que se hereda de Feline.
- this.type ahora es "c", ya que type se ha modificado en la instancia de Cougar.
- super.type también es "c" porque super.type y this.type se refieren al mismo atributo en la instancia de Cougar. La asignación a this.type también modifica super.type ya que ambos apuntan al mismo atributo.

Impresión en go:

- El método go imprime this.type + super.type. Dado que ambos son "c", el resultado es "c c".

13. Which statement, when inserted into line " // TODO code application logic here", is valid in compilation time change?

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

asc = sc; sc = asc asc = (Object) sc; asc= sc.clone();

Debido a que la variable de referencia sc es padre de la variable asc.

14. What is the result?

```
public class Test {
	public static void main(String[] args) {
		int[][] array = {{0},{0,1},{0,2,4},{0,3,6,9},{0,4,8,12,16}};
		System.out.println(array[4][1]);
		System.out.println(array[1][4]);
	}
}
```

4 Null.

Null 4.

An IllegalArgumentException is thrown at run time.

4 An ArrayIndexOutOfBoundsException is thrown at run time.

En la Segunda impression en consola, se sale del tamaño de ese array, solo tiene 2 elementos y lo busca en el index 4

15. What is the result?

```
Import java.util.*;
public class App {
	public static void main(String[] args) {
		List p = new ArrayList();
		p.add(7);
		p.add(1);
		p.add(5);
		p.add(1);
		p.remove(1);
		System.out.println(p);
	}
}
```

[7, 1, 5, 1] [7, 5, 1] [7, 5] [7, 1]

Elimina el primer elemento que encuentra en la lista, de izquierda a derecha.

16. Which three lines will compile and output “Right on!”?

```
13. public class Speak {
14.   public static void main(String[] args) {
15.         Speak speakIT = new Tell();
16.         Tell tellIt = new Tell();
17.         speakIT.tellItLikeItIs();
18.         (Truth)speakIt.tellItLikeItIs();
19.         ((Truth)speakIt).tellItLikeItIs();
20.         tellIt.tellItLikeItIs();
21.         (Truth)tellIt.tellItLikeItIs();
22.         ((Truth)tellIt).tellItLikeItIs();
23.   }
24. }

class Tell extends Speak implements Truth {
      @Override
      public void tellItLikeItIs(){
            System.out.println(“Right on!”);
      }
}

Interface Truth {
      public void tellItLikeItIs();
```

Line 17 Line 18 Line 19 Line 20 Line 21 Line 22

- Línea 19 y 22: Las conversiones explícitas ((Truth)) se aplican a los objetos antes de llamar al método tellItLikeItIs().
- Línea 20: tellIt es un objeto de Tell, que tiene el método sobrescrito tellItLikeItIs().
- **Línea 18 y 21** no compilan porque la conversión (Truth) se aplica al resultado del método en lugar del objeto.
- **Línea 17** no compila porque speakIt es de tipo Speak, que no tiene el método tellItLikeItIs().

17. What is the result?

```
public class Bees {
      public static void main(String[] args){
            try{
                  new Bees().go();
            } catch (Exception e){
                  System.out.println(“thrown to main”);
            }
      }
      synchronized void go() throws InterruptedException{
            Thread t1 = new Thread();
            t1.start();
```

```
            System.out.print("1 ");
            t1.wait(5000);
            System.out.print("2 ");
      }
}
```

The program prints 1 then 2 after 5 seconds.

The program prints: 1 thrown to main.

The program prints: 1 2 thrown to main.

The program prints:1 then t1 waits for its notification.

Esto se debe a que `t1.wait(5000);` lanza una excepción `IllegalMonitorStateException` al no estar en un bloque sincronizado, y dicha excepción es capturada e imprime el mensaje "thrown to main".

18. Which three are valid?

```
class ClassA{}
class ClassB extends ClassA{}
class ClassC extends ClassA{}
And:
ClassA p0 = new ClassA();
ClassA p1 = new ClassB();
ClassA p2 = new ClassC();
ClassA p3 = new ClassB();
ClassA p4 = new ClassC();
```

p0 = p1; la variable p0 es clase A, padre de la clase B por lo que puede contener sus objetos.

p1 = p2;

p2 = p4;

p2 = (ClassC)p1;

p1 = (ClassB)p3;

p2 = (ClassC)p4; estas dos respuestas estan casteadas correctamente

19. Which three options correctly describe the relationship between the classes?

```
class Class1{String v1;}
class Class2{
      Class1 c1;
      String v2;
}
class Class3 {Class2 c1; String v3;}
```

Class2 has-a v3.

Class1 has-a v2.

Class2 has-a v2. --- `Class2` tiene un atributo de tipo `String` llamado `v2`.

Class3 has-a v1. --- `Class3` tiene un atributo de tipo `Class2` llamado `c1`, y `Class2` tiene un atributo de tipo `Class1` llamado `c1`, que a su vez tiene un atributo `String v1`.

Class2 has-a Class3.

Class2 has-a Class1. --- `Class2` tiene un atributo de tipo `Class1` llamado `c1`.

20. Which three implementations are valid?

```
interface SampleCloseable {
    public void close() throws java.io.IOException;
}
```

class Test implements SampleCloseable { public void close() throws java.io.IOException { // do something } }

- Esta implementación es válida porque la excepción `java.io.IOException` es exactamente la misma que se especifica en la interfaz `SampleCloseable`.

class Test implements SampleCloseable { public void close() throws Exception { // do something } }

class Test implements SampleCloseable { public void close() throws FileNotFoundException { // do something } }

- Esta implementación es válida porque `FileNotFoundException` es una subclase de `java.io.IOException`, lo que cumple con la regla de excepciones más específicas.

class Test extends SampleCloseable { public void close() throws java.io.IOException { // do something } }

class Test implements SampleCloseable { public void close() { // do something }}

- Esta implementación es válida porque no declara ninguna excepción, lo cual es permitido ya que no es necesario declarar excepciones si el método no lanza ninguna.

21. What is the result?

```
class MySort implements Comparator<Integer> {
    public int compare(Integer x, Integer y) {
        return y.compareTo(x);
    }
```

```
}

And the code fragment:
Integer[] primes = {2, 7, 5, 3};
MySort ms = new MySort();
Arrays.sort(primes, ms);
for (Integer p2 : primes) { System.out.print(p2 + " "); }
2 3 5 7

2 7 5 3

7 5 3 2
```

- Debido a que la implementación del método “`compare`” ordena los números de manera descendente, provoca que se imprima `7 5 3 2`.

```
Compilation fails
```

22. Which two possible outputs?

```
public class Main {
    public static void main(String[] args) throws Exception {
        doSomething();
    }

    private static void doSomething() throws Exception {
        System.out.println("Before if clause");
        if (Math.random() > 0.5) {
            throw new Exception();
        }
        System.out.println("After if clause");
    }
}
```

Before if clause Exception in thread "main" java.lang.Exception at Main.doSomething (Main.java:21) at Main.main (Main.java:15).

Before if clause Exception in thread "main" java.lang.Exception at Main.doSomething (Main.java:21) at Main.main (Main.java:15) After if clause.

Exception in thread "main" java.lang.Exception at Main.doSomething (Main.java:21) at Main.main (Main.java:15).

- Si el valor generado por “Math.random” es mayor a 0.5, se imprime "Before if clause" y se lanza una excepción, resultando en el mensaje de error "Exception in thread 'main' java.lang.Exception at Main.doSomething(Main.java:21) at Main.main(Main.java:15)", ya que la excepción interrumpe la ejecución antes de que se imprima "After if clause".

Before if clause After if clause

- Si el valor generado por “Math.random” es menor o igual a 0.5, la condición del `if` no se cumple, por lo que se imprimen ambas líneas: "Before if clause" seguido de "After if clause".

23. What is the result?

```
class MyKeys {
    Integer key;

    MyKeys(Integer k) {
        key = k;
    }

    public boolean equals(Object o) {
        return ((MyKeys) o).key == this.key;
    }
}

public class Main {
    public static void main(String[] args) {
        Map<MyKeys, String> m = new HashMap<>();
        MyKeys m1 = new MyKeys(1);
        MyKeys m2 = new MyKeys(2);
        MyKeys m3 = new MyKeys(1);
        MyKeys m4 = new MyKeys(new Integer(2));

        m.put(m1, "car");
        m.put(m2, "boat");
        m.put(m3, "plane");
        m.put(m4, "bus");

        System.out.print(m.size());
    }
}
```

2
3
4

- La respuesta es `4` porque, aunque la clase `MyKeys` sobrescribe el método `equals`, no sobrescribe el método `hashCode`. En Java, un `HashMap` utiliza tanto `equals` como `hashCode` para determinar la unicidad de las claves. Dado que `MyKeys` no sobrescribe `hashCode`, los objetos `m1`, `m2`, `m3` y `m4` se consideran diferentes incluso si tienen el mismo valor de `key`. Por lo tanto, cada llamada a `m.put` agrega una nueva entrada al mapa, resultando en cuatro entradas distintas.

Compilation fails.

24. What is the result?

```
public static void main(String[] args) {
    String color = "Red";
```

```
    switch (color) {
        case "Red":
            System.out.println("Found Red");
        case "Blue":
            System.out.println("Found Blue");
        case "White":
            System.out.println("Found White");
            break;
        default:
            System.out.println("Found Default");
    }
}
```

Found Red.
Found Red Found Blue.
Found Red Found Blue Found White.

- La salida del programa es "Found Red", "Found Blue", "Found White". Esto debido a que, en Java, cuando no se utiliza la instrucción `break` al final de un caso en un `switch`, la ejecución continúa en el siguiente caso, sin importar si la condición se cumple o no.

Found Red Found Blue Found White Found Default.

25. Which two statements are true?

An abstract class can implement an interface.

- Una clase abstracta en Java puede implementar una o más interfaces. Esto se debe a que una clase abstracta puede proporcionar implementaciones para algunos o todos los métodos de la interfaz, o puede dejar que las subclases concreten esos métodos.

An abstract class can be extended by an interface.

An interface CANNOT be extended by another interface.

An interface can be extended by an abstract class.

An abstract class can be extended by a concrete class.

- Una clase concreta puede extender una clase abstracta. Esto significa que la clase concreta debe proporcionar implementaciones para todos los métodos abstractos declarados en la clase abstracta.

An abstract class CANNOT be extended by an abstract class
26. The pom.xml file is the configuration file For:

Ant

Gradle

Maven

- Maven utiliza el archivo `pom.xml` (Project Object Model) para gestionar la configuración del proyecto, las dependencias, las tareas de construcción y otros aspectos del ciclo de vida del proyecto.

27. The SINGLETON pattern allows:

Have a single instance of a class and this instance cannot be used by other classes.
Having a single instance of a class, while allowing all classes have access to that instance.

- El patrón Singleton asegura que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia.
-

Having a single instance of a class that can only be accessed by the first method that calls it.

28. What is the result?

```
class X {
    static void m(int i) {
        i += 7;
    }
    public static void main(String[] args) {
        int i = 12;
        m(i);
        System.out.println(i);
    }
}
```

7
12

- La salida del programa es 12 porque en Java los parámetros primitivos (como `int`) se pasan por valor, no por referencia. Esto significa que cuando se llama al método `m(i)`, se pasa una copia del valor de `i`. Cualquier cambio hecho a la copia dentro del método `m` no afecta al valor original de `i` en el método `main`. Por lo tanto, cuando se imprime `i` en el método `main`, su valor sigue siendo 12.

19
Compilation fails.

An exception is thrown at run time

29. Which statement is true?

```
class ClassA {
    public int numberOfInstances;
```

```
    protected ClassA(int numberOfInstances) {
        this.numberOfInstances = numberOfInstances;
    }
}

public class ExtendedA extends ClassA {
    private ExtendedA(int numberOfInstances) {
        super(numberOfInstances);
    }
    public static void main(String[] args) {
        ExtendedA ext = new ExtendedA(420);
        System.out.print(ext.numberOfInstances);
    }
}
```

420 is the output.

- La salida del programa es `420` porque cuando se crea una instancia de `ExtendedA` con el valor `420`, el constructor de `ExtendedA` llama al constructor de `ClassA` con `super(numberOfInstances)`. Esto asigna el valor 420 al campo `numberOfInstances` en `ClassA`.

An exception is thrown at runtime.

All constructors must be declared public.

Constructors CANNOT use the private modifier.

Constructors CANNOT use the protected modifier.

30. Which is true?

```
5. class Building {}
6. public class Barn extends Building {
7.     public static void main(String[] args) {
8.     Building build1 = new Building();
9.     Barn barn1 = new Barn();
10.    Barn barn2 = (Barn) build1;
11.    Object obj1 = (Object) build1;
12.    String str1 = (String) build1;
13.    Building build2 = (Building) barn1;
14.    }
15. }
```

If line 10 is removed, the compilation succeeds.

If line 11 is removed, the compilation succeeds.

If line 12 is removed, the compilation succeeds.

- Esto causará un error de compilación porque no se puede convertir un Building a String.

If line 13 is removed, the compilation succeeds.

More than one line must be removed for compilation to succeed.

31. What is the result?*

```
public static void main(String[] args) {
	int[][] array2D = {{0, 1, 2}, {3, 4, 5, 6}};
	System.out.print(array2D[0].length + " ");
	System.out.print(array2D[1].getClass().isArray() + " ");
	System.out.println(array2D[0][1]);

}
```

3false1

2true3

2false3

3true1

-La longitud (length) de array2D[0] es 3 porque tiene tres elementos. El método .getClass() obtiene el objeto Class que representa int[], y isArray() comprueba si este objeto representa un arreglo, lo cual es cierto, por lo que imprime true. array2D[0][1] accede al segundo elemento del primer arreglo ({0, 1, 2}), que es 1.

3false3

2true1

2false1

32. What is the DTO pattern used for?

To implement the data access layer

To exchange data between processes

Está diseñado específicamente para transferir datos entre diferentes partes de una aplicación, a menudo entre el cliente y el servidor o diferentes capas de una aplicación.

To implement the presentation layer

33. What is the result if the integer value is 33?

```
public static void main(String[] args) {
	if (value >= 0) {
		if (value != 0) {
			System.out.print("the ");
		} else {
			System.out.print("quick ");
```

```
            }
            if (value < 10) {
                  System.out.print("brown ");
            }
            if (value > 30) {
                  System.out.print("fox ");
            } else if (value < 50) {
                  System.out.print("jumps ");
            } else if (value < 10) {
                  System.out.print("over");
            } else {
                  System.out.print("the ");
            }
            if (value > 10) {
                  System.out.print("lazy");
            } else {
                  System.out.print("dog ");
            }
            System.out.print("... ");
      }
}
```

The fox jump lazy ?

The fox lazy ?

Quick fox over lazy ?

The fox lazy…

- value es 33.
- Condición if (value >= 0): Se cumple porque 33 es mayor que 0.
- Condición if (value != 0): Se cumple porque 33 no es 0, por lo que se imprime "the ".
- Condición if (value < 10): No se cumple porque 33 no es menor que 10, no se imprime nada.
- Condición if (value > 30): Se cumple porque 33 es mayor que 30, por lo que se imprime "fox ".
- Condición if (value > 10): Se cumple porque 33 es mayor que 10, por lo que se imprime "lazy".
- Finalmente, se imprime "..."

34. What is the result?

```
import java.text.*;
public class Align {
      public static void main(String[] args) throws ParseException {
      String[] sa = {"111.234", "222.5678"};
      NumberFormat nf = NumberFormat.getInstance();
      nf.setMaximumFractionDigits(3);
      for (String s : sa) {System.out.println(nf.parse(s));}
      }
}
```

111.234 222.567

111.234 222.568

111.234 222.5678

-El método parse en NumberFormat no modifica los números al imprimirlos; simplemente los convierte de cadena a número. La configuración de setMaximumFractionDigits(3) afecta solo la parte de formateo, no el parseo. Por lo tanto, los números se imprimen tal cual son parseados.

An exception is thrown at runtime.

35. What is the result?

```
11. class Person {
12.   String name = "No name";
13.   public Person (String nm) { name = nm; }
14. }
15.
16. class Employee extends Person {
17.   String empID = "0000";
18.   public Employee (String id) { empID ‟id; }
19. }
20.
21. public class EmployeeTest {
22.   public static void main(String[] args) {
23.         Employee e = new Employee("4321");
24.         System.out.println(e.empID);
25.   }
26. }
```

4321.

0000.

An exception is thrown at runtime.

Compilation fails because of an error in line 18.

El constructor de Employee tiene un carácter no válido justo antes de id.

36. What is the result?

Given

```
public class SuperTest {
      public static void main(String[] args) {
      //statement1
      //statement2
      //statement3
      }
}
class Shape {
      public Shape() {
```

```
            System.out.println("Shape: constructor");
      }
      public void foo() {
            System.out.println("Shape: foo");
      }
}
class Square extends Shape {
      public Square() {
            super();
      }
      public Square(String label) {
            System.out.println("Square: constructor");
      }
      public void foo() {
            super.foo();
      }
      public void foo(String label) {
            System.out.println("Square: foo");
      }
}
```

What should be the statement1, statement2 and statement3, be respectively, in order to produce the result?

Shape: constructor
Shape:foo
Square: foo

Square square = new Square ("bar"); square.foo ("bar"); square.foo();

Square square = new Square ("bar"); square.foo ("bar"); square.foo ("bar");

Square square = new Square (); square.foo (); square.foo(bar);

Square square = new Square (); square.foo (); square.foo("bar");

- new Square(); crea una instancia de Square. Se llama al constructor de Square, que a su vez llama al constructor de Shape. Esto produce la línea Shape: constructor.
- square.foo(); llama al método foo() en la clase Square, que a su vez llama a super.foo(). Este método imprime Shape: foo antes de devolver el control al método foo() de Square.
- square.foo("bar"); llama al método foo(String label) en la clase Square, que imprime Square: foo.

Square square = new Square (); square.foo (); square.foo ();

37. Which code fragment is illegal?*

Class Base1 { abstract class Abs1 { } }

Abstract class Abs2 { void doit() { } }

class Base2 { abstract class Abs3 extends Base2 { } }

class Base3 { abstract int var1 = 89; }

-Este código es ilegal. No puedes tener una variable abstract en una clase. Las variables no pueden ser declaradas como abstractas en Java. Una variable abstracta no está permitida porque abstract se utiliza para métodos que no tienen una implementación, y las variables deben ser inicializadas.

38. What is the result?

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

Result: 10 Result: 30

Result: 25 Result: 10

Result: 235 Result: 215

-En la primera línea, los números se concatenan como cadenas sin paréntesis. En la segunda línea, se realiza primero la multiplicación debido a la precedencia de operadores.

Result: 215 Result: 215

Compilation fails.

39. What is the result?

```
public class MyStuff {
      String name;
      MyStuff(String n) { name = n;}
      public static void main(String[] args) {
            MyStuff m1 = new MyStuff("guitar");
            MyStuff m2 = new MyStuff("tv");
            System.out.println(m2.equals(m1));
      }
      public boolean equals(Object o) {
      MyStuff m = (MyStuff) o;
      if (m.name != null) {return true;}
      return false;
      }
}
```

The output is true and MyStuff fulfills the Object.equals() contract

The output is false and MyStuff fulfills the Object.equals() contract

The output is true and MyStuff does NOT fulfill the Object.equals() contract.

-El método equals de MyStuff simplemente devuelve true si m.name no es null, lo cual no asegura el cumplimiento de las propiedades simétrica y transitiva. Por ejemplo, m1.equals(m2) puede ser true, pero m2.equals(m1) puede no ser necesariamente true si m.name no se evalúa de la misma manera.

The output is false and MyStuff does NOT fulfill the Object.equals() contract

40. Which one is valid as a replacement for foo?*

```
public static void main(String[] args) {
	Boolean b1 = true;
	Boolean b2 = false;
	int i = 0;
	while (foo) {
	// El cuerpo del bucle while está vacío
	}
}
```

b1.compareTo(b2)

i = 1

i == 2? -1:0

foo.equals("bar")

-Para que un bucle while se ejecute, la condición dentro de los paréntesis debe ser de tipo booleano. Entre las opciones dadas, solo foo.equals("bar") devuelve un valor booleano (verdadero o falso).