Guía Java

Thursday, December 5, 2024 4:49 PM

1. Excepciones de Java permitida*
   - org.springframework.web.client.RestClientException
   - No conozco la respuesta
   - java.lang.NumberFormatException
   - com.bbva.apx.exception.db.NoResultException
   - com.bbva.elara.utility.interbackend.cics.exceptions.BusinessException

//////////

2. Which three are bad practices?

   - Checking for ArrayIndex0utOfBoundsException 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 ArrayIndex0utOfBoundsException when iterating through an array to determine when all elements.-have been visited.

Checking for ArrayIndexOutOfBoundsException to "recover" (1).

Using exceptions (ArrayIndexOutOfBoundsException) for array iteration (4).

Catching Error and restarting the program (3).

//////

3. Indica a JUnit que la propiedad que usa esta anotación es una simulación y, por lo tanto, se inicializa como tal y es susceptible de ser inyectada por @InjectMocks.
   - Mockito
   - Mock
   - Inject
   - InjectMock

Aspecto	Mock	Mockito
Definición	Concepto general de un objeto simulado.	Framework específico para mocking en Java.
Uso	Se puede implementar manualmente o con una herramienta.	Es una herramienta lista para usar.
Funcionalidad	Simula el comportamiento de objetos.	Facilita la creación, configuración y verificación de mocks.
Popularidad	General en pruebas unitarias.	Amplia aceptación en Java (compatible con JUnit, TestNG, etc.).

//////////////////

```
Given

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.print(array2D[0][1]);
}
What is the result?
```

- 3false3
- 3false1
- 2false1
- 3true1
- 2true3

- **array2D[0].length devuelve la longitud de la primera fila (3).**
- **array2D[1].getClass().isArray() verifica si la segunda fila es un arreglo (true).**
- **array2D[0][1] obtiene el valor de la posición 1 en la primera fila (1).**

/////

5. Which two statments 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.

Una clase abstracta puede ser extendet por otra clase concreta

Una clase abstracta puede implementar una interface.

//////////

6. Which five methods, inserted independiently 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; }

Un método protegido (protected) puede ser sobrescrito en la clase hija con igual o mayor nivel de acceso (protected o public).

Para sobrescritura, el tipo de retorno debe ser igual o compatible (covariante) con el del método padre.

Protected long blipvert(int x) { return 0; }
- Este método tiene un tipo de retorno long, que **no coincide** con el tipo de retorno del método de la clase base (int).
- Esto provoca un error de compilación porque no es válido para sobrescritura.
- **No compila**

//////////////

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

- 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)

Tiene un atributo privado a.
Al ser private, no es accesible directamente desde clases derivadas.

Tiene un constructor protegido que inicializa a:

- **Acceso al campo a desde Sub:**

- La línea this.a = 5; es inválida porque a es private en Super.
- Para resolver esto, habría que cambiar la visibilidad de a o ajustar cómo se inicializa en Sub.

- **Falta de llamada al constructor padre en el constructor sin parámetros de Sub:**
  - En Java, cada constructor de una subclase debe llamar explícita o implícitamente a un constructor de la superclase. Como Super no tiene un constructor sin parámetros, el constructor por defecto no funcionará.

Estas opciones permiten que la clase Sub compile porque resuelven los problemas relacionados con el acceso al atributo a y la llamada al constructor de la superclase.

/////////

8. Whats 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.

Una clase abstracta puede tener metodos privados.

Una clase abstracta puede tener variables de instancoa.

No es necesario una clase abstracta tenga un metodo abstracto, ni un constructor.

//////

9. 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(); }
}
```

- "atom ": Se imprime al llamar al constructor de Atom cuando se ejecuta super("granite ") en el constructor de Mountain.
- "granite ": Se imprime al finalizar el constructor de Rock cuando se ejecuta super("granite ").
- "atom ": Se imprime al crear una nueva instancia de Rock dentro del constructor de Mountain.
- "granite ": Se imprime al finalizar el constructor de Rock para esa nueva instancia.

////////

10. What is printed out when the program is excuted?

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

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

- **JVM ejecuta únicamente el método que tiene la firma válida como punto de entrada (public static void main(String[] args)).**
- **Los otros métodos no son considerados, y el método mina no se llama.**

//////////////////////////

11. What is the result?

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

- Cougar c f.
- Feline cougar c f.
- Feline cougar c c.
- Compilation fails.

- **Se crea una nueva instancia de Cougar:**
  - **Primero, se ejecuta el constructor de Feline, que imprime "feline ".**
  - **Luego, se ejecuta el constructor de Cougar, que imprime "cougar ".**
- **El método go es llamado:**
  - **Cambia type a "c ".**
  - **Imprime el valor de this.type + super.type. Dado que ambos apuntan al mismo atributo (type), el resultado es "c c ".**

//

```
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[] args) {
                Gamma g1 = new Alpha();
                Gamma g2 = new Beta();
                System.out.println(g1.getType() + " " + g2.getType());
        }
}
```

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

- **Gamma g1 = new Alpha();**
  - Esto no es válido porque Alpha no es una subclase de Gamma. En Java, no se puede asignar una instancia de una clase base (padre) a una referencia de una clase derivada (hija).
  - Este error hace que la compilación falle.
- **Gamma g2 = new Beta();**
  - Similar al primer caso, Beta es una clase base respecto a Gamma, por lo que no se puede asignar una instancia de Beta a una referencia de tipo Gamma.

/////

```
import java.util.*;
public class MyScan {
        public static void main(String[] args) {
                String in = "1 a 10 . 100 1000";
                Scanner s = new Scanner(in);
                int accum = 0;
                for (int x = 0; x < 4; x++) {
                        accum += s.nextInt();
                }
                System.out.println(accum);
        }
}
```

- 11
- 11
- 1111
- An exception is thrown at runtime

- **Primera iteración (x = 0)**:
  El Scanner lee el primer token ("1") y lo suma a accum.
  accum = 0 + 1 = 1.
- **Segunda iteración (x = 1)**:
  El Scanner intenta leer el segundo token ("a"). Aquí ocurre un problema: "a" no es un entero, por lo que el método nextInt() lanza una excepción de tipo **InputMismatchException**.

El Scanner en Java es una herramienta que permite analizar un flujo de texto, y funciona dividiendo el texto en **tokens**. En este caso, el String que proporcionas ("1 a 10 . 100 1000") es tratado como una secuencia de tokens que el Scanner puede procesar uno por uno.

//////////////

```
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.

Llamada a go():

- Desde main, se crea una nueva instancia de la clase Bees y se llama al método go().
- Este método está **sincronizado** (synchronized), lo que significa que requiere un bloqueo en el objeto actual para ejecutarse.

2. Creación y arranque del hilo (Thread t1):
- En el método go(), se crea un nuevo hilo t1 con

- El método wait() está diseñado para ser llamado sobre el monitor del objeto sincronizado (en este caso, debería ser el objeto actual, es decir, this, no el hilo t1).
- El uso de wait() sobre un objeto que no está sincronizado (en este caso, el hilo t1) genera una excepción en tiempo de ejecución: **IllegalMonitorStateException**.
- Esto hace que el programa salga del método go() y entre al bloque catch en el main.

  The program prints: 1 thrown to main.