A. Investigación sobre las excepciones

1. ¿Qué diferencia hay entre las excepciones checked y unchecked en Java, y cómo afecta esto a la forma de escribir el código?

Las excepciones *checked* son las que heredan de `Exception` sin pasar por `RuntimeException` (por ejemplo `IOException` o `SQLException`). El compilador obliga a tratarlas: o se capturan con `try/catch` o se declaran con `throws` en el método.

Las excepciones *unchecked* son las que heredan de `RuntimeException` (por ejemplo `NullPointerException` o `IllegalArgumentException`) y los `Error`. El compilador no obliga a tratarlas.

Con las checked el código es más explícito y el compilador impide olvidar un error posible, pero se llena de bloques `try/catch` y de `throws` que se propagan por todas las firmas. Con las unchecked el código queda más limpio, pero es más fácil olvidarse de tratarlas porque el compilador no avisa.

2. ¿Qué criterio se debe usar para decidir si una excepción personalizada debe heredar de Exception o de RuntimeException?

El criterio principal es la recuperabilidad. Si quien llama al método puede razonablemente recuperarse del error (reintentar, avisar al usuario, usar otro recurso) y se quiere obligarle a tratarlo, la excepción debe heredar de `Exception` (checked). Si es un error de programación (argumento inválido, estado incorrecto) o quien llama no puede hacer nada útil, debe heredar de `RuntimeException` (unchecked).

3. ¿Qué ocurre si en un bloque try se lanza una excepción que no se captura en ningún catch y tampoco se declara con throws? ¿En qué casos eso es válido?

Si la excepción es checked, el código no compila, porque el compilador exige capturarla o declararla.

Si es unchecked, el código compila y en ejecución ocurre lo siguiente: se interrumpe el flujo normal en el punto donde se lanza; se ejecutan los bloques `finally` y se cierran los recursos del `try-with-resources`; la excepción sube por la pila de llamadas buscando un `catch` en los métodos que hicieron la llamada; y si nadie la captura, llega al hilo, que imprime el *stack trace* y termina. Si era el hilo `main`, el programa finaliza, salvo que queden otros hilos activos.

Esto es válido cuando la excepción es unchecked (`RuntimeException` o `Error`), o cuando algún método superior en la pila de llamadas la captura o la declara.

4. ¿Por qué en un proyecto profesional con Spring Boot estas excepciones se definirían como unchecked? (3 motivos)

1. **Evitar `throws` en cascada y el acoplamiento entre capas.** Con excepciones checked, un `SQLException` o `IOException` obligaría a declarar `throws` en el repositorio, el servicio y el controlador, y las capas superiores acabarían conociendo detalles de las inferiores. Con unchecked, las firmas de los métodos quedan limpias.
2. **Gestión de transacciones.** Con `@Transactional`, Spring hace *rollback* por defecto solo ante `RuntimeException` y `Error`. Las excepciones checked no revierten la transacción salvo que se configure expresamente con `rollbackFor`, lo que puede provocar errores difíciles de detectar.
3. **Tratamiento centralizado y compatibilidad con código funcional.** La mayoría de los errores no se pueden resolver en el punto donde ocurren, así que se gestionan en un único lugar con `@ControllerAdvice` y `@ExceptionHandler`, que devuelve la respuesta HTTP adecuada. Además, las excepciones checked no encajan con lambdas y *streams*, porque las interfaces funcionales no declaran `throws`. El propio Spring traduce las excepciones de persistencia a `DataAccessException`, que es unchecked.

Pregunta 5

Excepción elegida: `NegocioException`.

Análisis:

Un país que referencia un continente que no existe no es un fallo técnico: el fichero se ha leído bien y el DAO de continentes ha funcionado (simplemente no ha encontrado el código y ha devuelto `null`). Lo que ocurre es que los datos incumplen una regla del modelo: todo país debe pertenecer a un continente existente (integridad referencial).

- **`AccesoDatosException`**: se reserva para fallos técnicos al acceder a los datos (fichero inexistente, error de lectura o escritura). Aquí el acceso ha sido correcto.
- **`TecnicoException`**: es para errores técnicos generales, como el fallo al leer el `.properties`. Tampoco es el caso.
- **`NegocioException`**: expresa que la operación no se puede completar porque se viola una regla de negocio. El mensaje debe indicar el país y el código de continente, por ejemplo: *“El país 12 referencia el continente ‘XX’, que no existe”*.

Excepciones:

```
26              System.out.println("La operación debe ser un número del 1 al 5.'
27              return;
```

Console

<terminated> GestorContinentes (4) [Java Application] C:\Users\gonza\OneDrive\Documentos\Desktop\eclipse

```
Error técnico: No se encuentra /application.properties en el classpath
```

