1. Diferencia entre excepciones *checked* y *unchecked* y su impacto en el código

La diferencia principal radica en **cuándo se verifica la excepción** y el **nivel de obligatoriedad** que impone el compilador:

- **Excepciones *Checked* (Verificadas):** Son aquellas que heredan de `Exception` (pero no de `RuntimeException`). El compilador de Java te **obliga** a gestionar el riesgo de que ocurran. Si un método puede lanzar una excepción checked, debes hacer una de dos cosas:

1. Capturarla usando un bloque `try-catch`.
2. Propagarla declarándola en la firma del método mediante la palabra clave `throws`.

- *Ejemplo:* `IOException`, `SQLException`.
- **Excepciones *Unchecked* (No verificadas):** Son aquellas que heredan de `RuntimeException` (o de `Error`). El compilador **no te obliga** a capturarlas ni a declararlas con `throws`. Ocurren generalmente por errores de lógica del programador o fallos irrecuperables en tiempo de ejecución.
- *Ejemplo:* `NullPointerException`, `IllegalArgumentException`, `IndexOutOfBoundsException`.

  Cómo afecta a la forma de escribir código:
  Las excepciones *checked* ensucian más la firma de los métodos y el flujo visual debido al "ruido" que generan las cláusulas `throws` acumuladas a lo largo de las capas de la aplicación (arquitectura "cebolla"), o debido a bloques `try-catch` repetitivos. Las *unchecked* permiten un código más limpio y legible, delegando la captura de errores a un componente centralizado.

---

2. Criterio para decidir si una excepción personalizada debe heredar de `Exception` o de `RuntimeException`

El criterio se basa en la **recuperabilidad del error** y la responsabilidad de quien invoca el método:

- **Heredar de `RuntimeException` (*Unchecked*):** Es la práctica recomendada para la gran mayoría de los casos modernos. Debes elegirla si:
- El error se debe a un **fallo de programación** (p. ej., se envió un parámetro inválido o un formato incorrecto que debió validarse antes).
- El error es **irrecuperable** en ese punto de la ejecución (p. ej., la base de datos está caída o un servicio externo crítico no responde). El cliente no puede hacer nada para solucionarlo al instante más que abortar la operación.
- **Heredar de `Exception` (*Checked*):** Úsala únicamente si:
- El error es una **consecuencia esperada y contingente** del entorno (no un bug) y el código que llama al método **puede y debe tomar una acción alternativa inmediata** para recuperarse del fallo.
- *Ejemplo clásico:* Intentar leer un archivo en un disco flexible; si no está, el programa puede pedirle al usuario de forma interactiva que inserte el disco para continuar.

---

3. Excepción no capturada ni declarada con `throws`

Si dentro de un bloque `try` ocurre una excepción que no es capturada por ningún `catch` asociado y el método **no** la declara con `throws`:

1. **El flujo normal del método se interrumpe inmediatamente.**
2. La excepción se propaga hacia arriba en la pila de llamadas (*call stack*), buscando un método previo que sí la capture.
3. Si nadie la captura en toda la pila, el hilo de ejecución (*thread*) muere. Si ocurre en el hilo principal (`main`), el programa completo finaliza abruptamente mostrando la traza del error en la consola.

¿En qué casos esto es válido (es decir, compila)?

Esto **solo es válido si la excepción lanzada es *Unchecked* (`RuntimeException`)**.

Si intentaras lanzar una excepción *checked* dentro de un método sin capturarla y sin poner `throws` en la firma, el código **no compilará**; el compilador de Java arrojará un error indicando que la excepción no ha sido gestionada.

---

4. ¿Por qué en proyectos profesionales de Spring Boot las excepciones se definen como *Unchecked*?

En el ecosistema de desarrollo moderno, y especialmente en **Spring Boot**, existe un consenso generalizado de preferir las excepciones *unchecked*. Si tu proyecto te propone usar *checked*, la arquitectura de Spring usualmente empuja a transformarlas a *unchecked* por tres motivos principales:

Motivo 1: Incompatibilidad con la programación funcional y Streams

Desde Java 8, el uso de expresiones Lambda y la API de Streams es un estándar de la industria. Las interfaces funcionales de Java (como `Predicate`, `Function` o `Consumer`) **no declaran excepciones checked** en sus firmas. Si intentas usar un método que lanza una excepción *checked* dentro de un Stream de Spring, el código no compilará a menos que añadas un molesto bloque `try-catch` dentro de la propia lambda. Convertirlas en *unchecked* elimina esta limitación arquitectónica.

Motivo 2: Gestión de Transacciones (`@Transactional`)

El mecanismo de gestión de transacciones por defecto de Spring (`@Transactional`) está diseñado bajo la premisa de las excepciones de tiempo de ejecución. **Spring solo realiza un *rollback* (marcha atrás) automático de la transacción si se lanza una `RuntimeException` o un `Error`.** Si tu código lanza una excepción *checked*, Spring asumirá que es un escenario de negocio controlable y **confirmará (*commit*) la transacción** a menos que configures explícitamente el atributo `rollbackFor = Exception.class`. Usar excepciones *unchecked* evita este grave riesgo de dejar datos corruptos o incompletos en la base de datos.

Motivo 3: Centralización de errores con `@ControllerAdvice` y limpieza de código

Spring Boot promueve el patrón de **Manejador Global de Excepciones** mediante las anotaciones `@ControllerAdvice` (o `@RestControllerAdvice`) y `@ExceptionHandler`. Este patrón captura los errores al vuelo al final del ciclo de vida de la petición HTTP para

transformarlos en respuestas legibles (como un JSON con código de estado 400 o 500). Si usaras excepciones *checked*, tendrías que arrastrar la declaración `throws MyException` a través de todas tus capas (Repository \(\rightarrow \) Service \(\rightarrow \) Controller), acoplando el código innecesariamente. Al hacerlas *unchecked*, las excepciones fluyen libremente hacia el manejador global sin alterar las firmas de tus métodos, manteniendo el código limpio, desacoplado y legible.