UD 5 – CONTROL DE ERRORES Y EXCEPCIONES

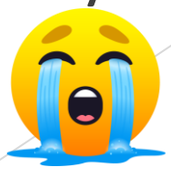
UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
Índice

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
1 - Introducción

- **1 – Introducción**
- Hasta ahora hemos “ignorado” los errores que se provocaban durante la ejecución de nuestros programas. Algunos son ya “viejos conocidos”:
  - ArrayIndexOutOfBoundsException
  - NullPointerException
  - InputMismatchException
  - ClassCastException

```
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:939)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
        at es.tuespiral.u5.p1.adivinaexcepciones.juego.Partida.preg
        at es.tuespiral.u5.p1.adivinaexcepciones.juego.Partida.inic
        at es.tuespiral.u5.p1.adivinaexcepciones.juego.JuegoAdivina
        at es.tuespiral.u5.p1.adivinaexcepciones.Aplicacion.main(Ap
Command execution failed.
```

- Todos provocaban que se cerrase nuestro programa y no podíamos hacer nada por impedirlo… ¡BASTA YA!



UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
1 - Introducción

- En los “Mundos de Yupi” todo sale bien y nunca habría problemas… Sin embargo, la vida real no es así y hay que aprender a tratar con los problemas.
- En este tema vamos a estudiar cómo Java permite detectar errores y cómo manejarlos correctamente.
- Para ello, vamos a crear una metáfora que nos sirva como referencia mental para digerir mejor todo esto:

PRESENTAMOS AL:

REPARTIDOR DE AMAZON

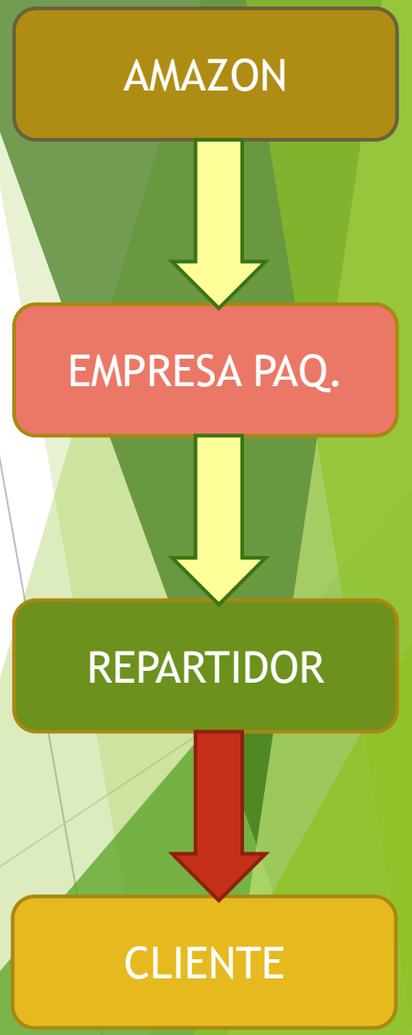
UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
1 - Introducción

- Un cliente compra un artículo en Amazon y se produce lo siguiente:
  - Una empresa de paquetería pasa a recoger el paquete que Amazon ha preparado.
  - La empresa transporta el paquete hasta la sede más próxima al domicilio del cliente.
  - Un repartidor recoge el paquete y lo lleva a casa del cliente **PERO NO ESTÁ EN CASA ¿QUÉ HACEMOS?**

Forzar la puerta y dejarle el paquete en la entrada

Esperar pacientemente hasta que vuelva

Llamar a Jeff Bezos y contarle lo sucedido



UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
1 - Introducción

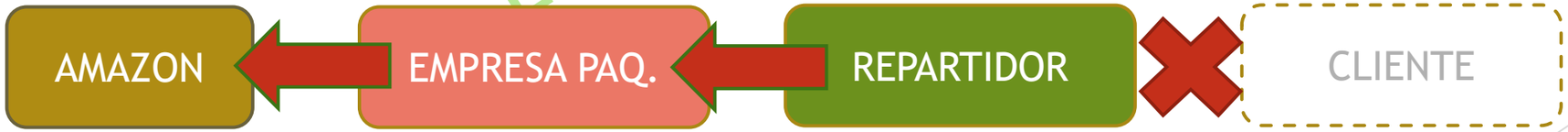
- La política más acertada para gestionar este “error” sería intentar arreglar el problema lo más localmente posible, entonces:

1. El repartidor llamaría al cliente para intentar dejárselo a un vecino/a o bien concertar otra hora de reparto.
2. Si el cliente no coge el teléfono entonces dejaría el problema en la sede más cercana.
3. Desde la sede intentarían contactar con el cliente para que recoja el paquete o se concierte una nueva entrega.
4. Si la sede no consigue localizar al cliente y realizar la entrega entonces tendría que devolver el paquete a Amazon

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
1 - Introducción

- Observa que este ejemplo contempla **dos posibles flujos** de procesos encadenados:
  - **El flujo normal o "exitoso"** que se consigue entregar el paquete.

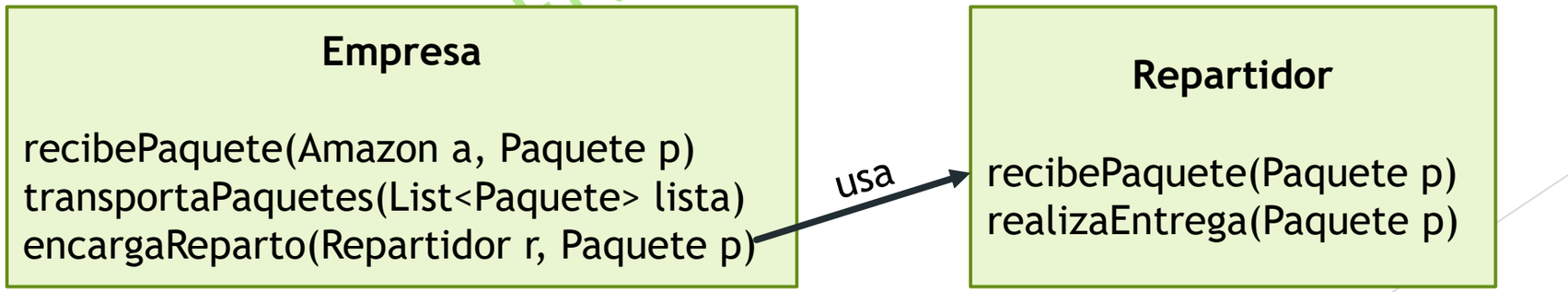
  - **El flujo de error o con fallo** que se produce cuando algún elemento de la cadena no consigue realizar correctamente su tarea.



Nuestros programas también tienen estos dos posibles flujos

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
2 – Flujo de ejecución de un programa

- **2 – Flujo de ejecución de un programa**
- Ahora tenemos que trasladar mentalmente los nombres “Cliente”, “Repartidor”, “Empresa”… a clases Java de las que podemos crear sus objetos respectivos.
- Esos objetos tendrán unos comportamientos (métodos) que serán utilizados por otros objetos, estableciéndose una relación de tipo “**cliente-servidor”, “utilizador-utilizado” o “llamador-llamado”**

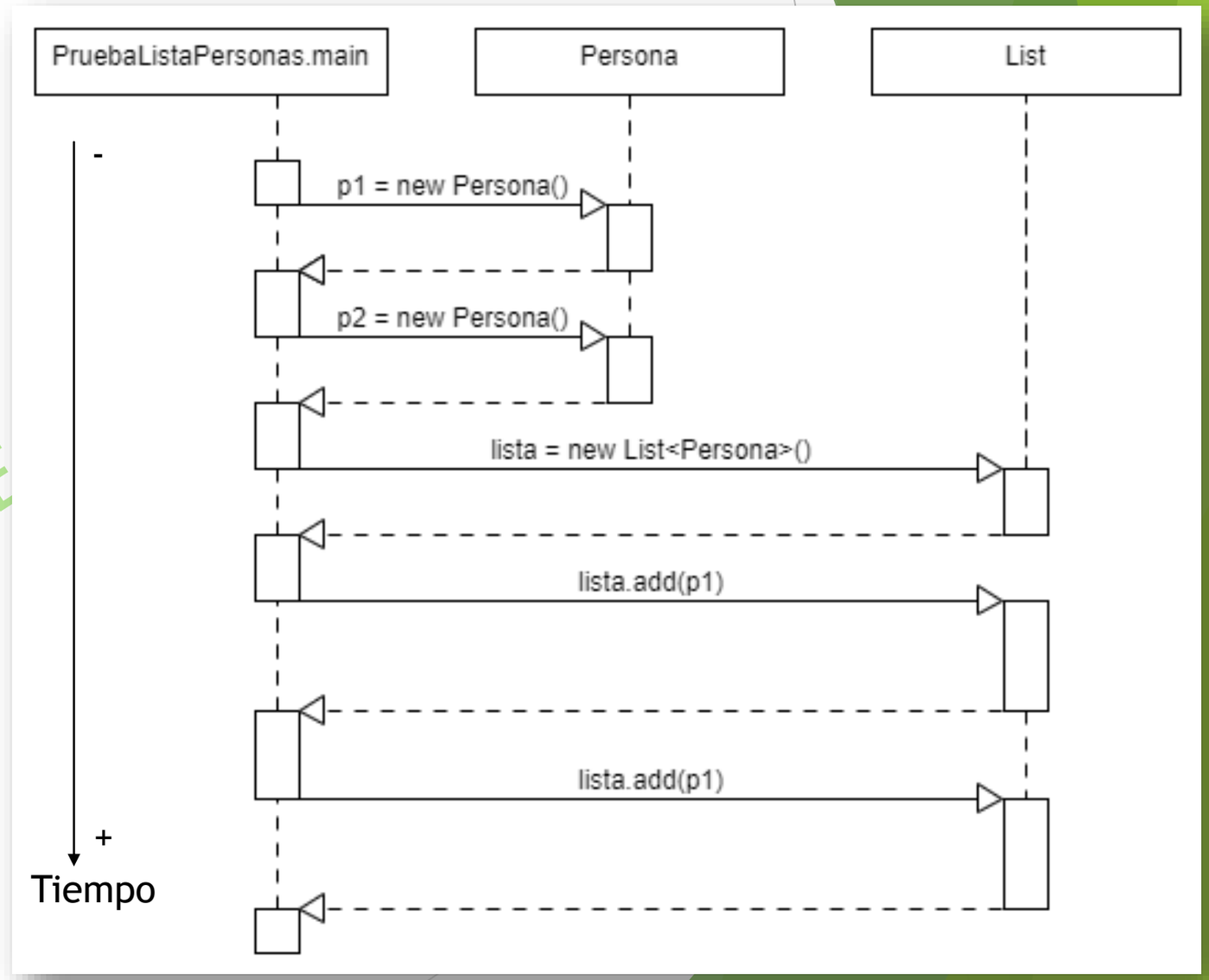


“Cliente”, “utilizador”, “llamador”

“Servidor”, “utilizado”, “llamado”

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
2 – Flujo de ejecución de un programa

- En una aplicación orientada a objetos, la secuencia de llamadas a métodos entre los distintos objetos crea un **“flujo de ejecución” o “flujo del programa”.**
- La imagen muestra una posible secuencia cronológica de llamadas de unos objetos a otros, reflejando la relación “cliente-servidor” entre los objetos.



UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
2 – Flujo de ejecución de un programa

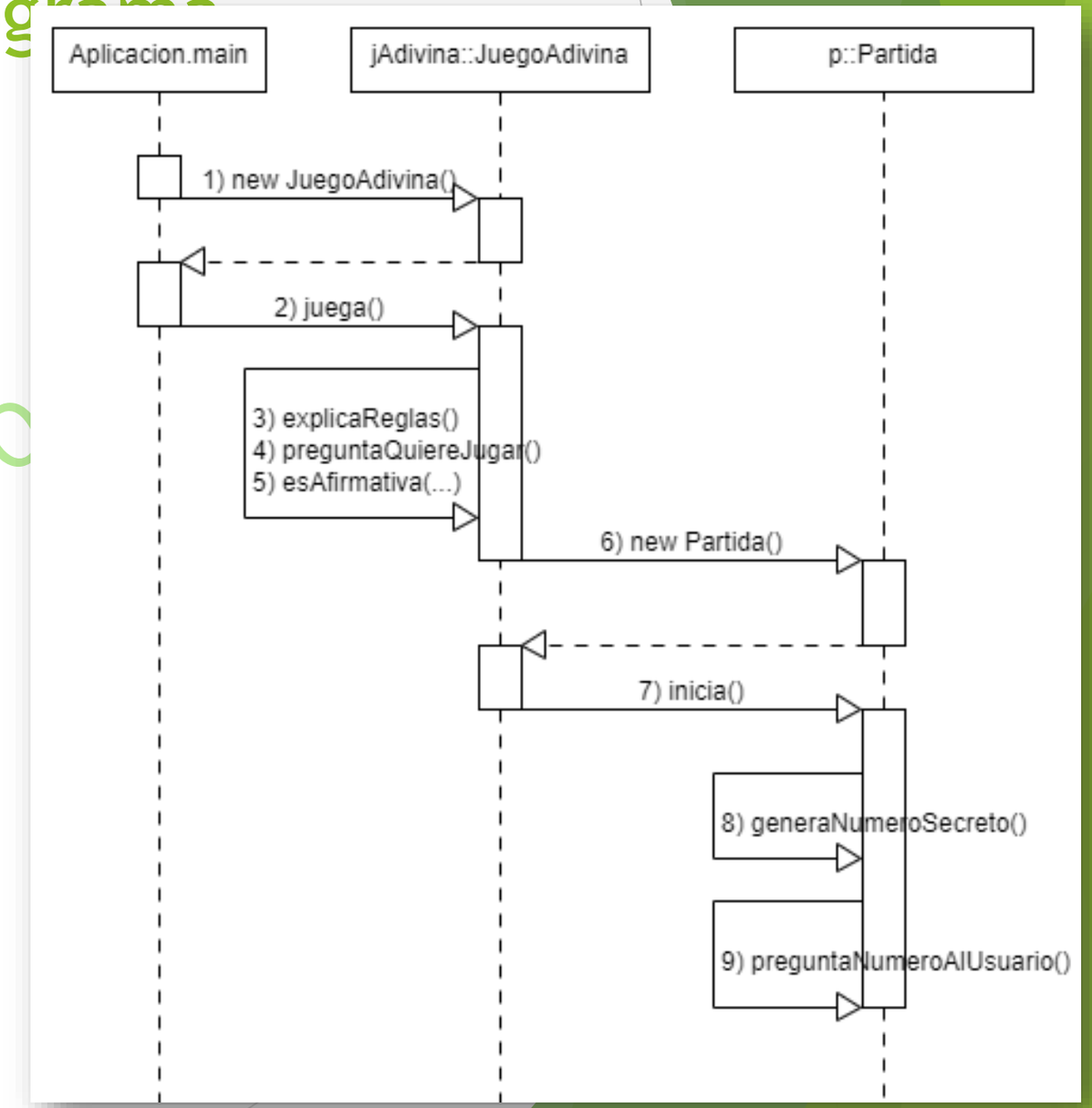
- Tenemos que comprender bien el concepto de **"flujo de ejecución"** para escribir correctamente un código que permita manejar los errores que puedan ocurrir al ejecutar un programa.
- Para profundizar más en este concepto vamos a verlo con código, para ello vamos a estudiar el código del: **juego Adivina**.

```
El juego consiste en que yo pienso un número entre 0 y 10 y tu
¿Quieres jugar? (S=Sí / N=No)
S
Dime un número entre 0 y 10
6
¡Uyy! Ese no es el número que he pensado
Te quedan 4 intentos
Dime un número entre 0 y 10
2
¡Uyy! Ese no es el número que he pensado
Te quedan 3 intentos
Dime un número entre 0 y 10
```

Estudiamos el código
U5.P1.JuegoAdivina

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
2 - Flujo de ejecución de un programa

- El flujo de este programa sigue el siguiente diagrama de secuencia UML.
- Ahora vamos a **provocar un error en tiempo de ejecución** en el paso 9 porque vamos a poner la palabra “siete” cuando el programa nos pida un número.
- Estamos provocando un fallo porque el método nextInt() de Scanner no puede convertir “siete” a un número entero y **nos “lanza”** una excepción del tipo: **InputMismatchException**



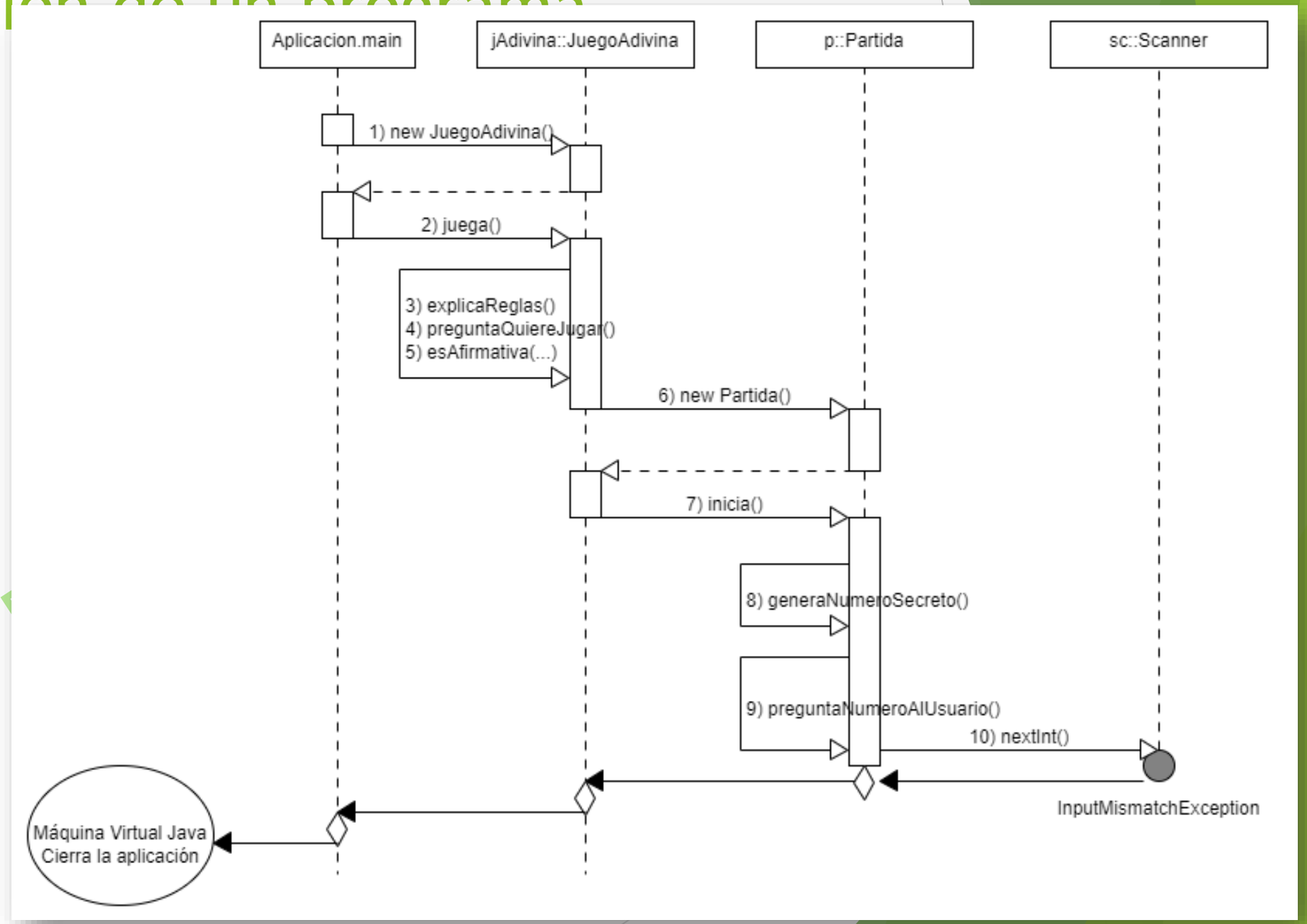
UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
2 – Flujo de ejecución de un programa

```
El juego consiste en que yo pienso un número entre 0 y 10 y tu tienes que adivinarlo en 5 intentos
¿Quieres jugar? (S=Sí / N=No)
S
Dime un número entre 0 y 10
siete
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:939)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
        at es.tuespiral.u5.p1.adivinaexcepciones.juego.Partida.preguntaNumeroAlUsuario(Partida.java:43)
        at es.tuespiral.u5.p1.adivinaexcepciones.juego.Partida.inicia(Partida.java:23)
        at es.tuespiral.u5.p1.adivinaexcepciones.juego.JuegoAdivina.juega(JuegoAdivina.java:18)
        at es.tuespiral.u5.p1.adivinaexcepciones.Aplicacion.main(Aplicacion.java:11)
Command execution failed.
```

- El mensaje resultante nos informa del tipo de error que se ha provocado (InputMismatchException) y en qué línea (939) de qué método (throwFor) de qué clase (Scanner) se provocó.
- A continuación nos muestra cómo ese error se va pasando "*como una patata caliente*" a las distintas clases clientes para ver si alguna puede solucionarlo.
- En caso de que el error llegue al *main* y nadie lo arregle, la máquina virtual java cierra nuestro programa.

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
2 - Flujo de ejecución de un programa

- Si vemos gráficamente qué está ocurriendo tendríamos algo así:



UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
2 – Flujo de ejecución de un programa

- Lo expresamos con palabras, cuando se produce la excepción lo que ocurre es:
  - **Se detiene el flujo normal de ejecución.**
  - **Se comienza con el flujo de error que va "deshaciendo la cadena de llamadas" que hizo el flujo normal de ejecución** para ver si algún método o alguna clase "sabe" arreglar ese error.
  - Si ninguna clase es capaz de manejar la excepción y ésta llega al **main** entonces la MVJ cierra el programa imprimiendo el texto de la excepción.
- Piensa en la metáfora del repartidor de Amazon. Si no localiza al cliente, lleva el paquete a la central de reparto a ver si ellos pueden arreglarlo… "**se va deshaciendo la cadena de pasos**"

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
3 – El bloque try…catch

- **3 – El bloque try…catch**
- Como en la vida misma, cuando nos encontramos con un problema podemos hacer dos cosas:
  - **Ignorarlo y asumir las consecuencias.** Eso es lo que hemos hecho hasta hoy en nuestros programas.
  - **Intentar arreglar el problema.** Es lo que vamos a aprender ahora.
- Java tiene una sintaxis que permite **intentar (try)** la ejecución de un trozo de código que sea "*potencialmente conflictivo*" porque pueda "*lanzar una excepción*". Es el **bloque try {…}**

```
try {
  // Código
  // potencialmente
  // conflictivo
}
```

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
3 – El bloque try…catch

- Dado que la jerga utilizada es “lanzar una excepción” (throw an exception), cuando queremos manejar la excepción **primero hay que “cogerla” o “capturarla” (catch).**
- Esto hace que el **bloque try** se acompañe de un **bloque catch** que capture la excepción y la maneje. La sintaxis completa es:


```
try {
  // Código
  // potencialmente
  // conflictivo
} catch(TipoExcepción e) {
  // Código que maneja
  // la excepción
}
```

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
3 – El bloque try…catch

- Volvamos a nuestro **juego Adivina** para ver cómo podemos **manejar la excepción** que hemos provocado.
- La solución que deberíamos dar a este problema es indicarle al usuario que se ha equivocado y que lo vuelva a intentar.
- Primera versión:

Retocamos el código y lo probamos

```
private int preguntaNumeroAlUsuario() {
    Scanner sc = new Scanner(System.in);
    try {
        System.out.println("Dime un número entre 0 y "+LIMITE_SUPERIOR);
        return sc.nextInt();
    }catch(InputMismatchException e) {
        System.out.println("¡Oups! Te has equivocado");
        System.out.println("Dime un número entre 0 y "+LIMITE_SUPERIOR);
        return sc.nextInt();
    }
}
```

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
3 – El bloque try…catch

- Sin embargo, si el usuario se equivoca dos veces seguidas se provocará otra excepción en el **bloque catch** y nuestro programa se cerrará de nuevo. Vamos a mejorar nuestro código con la segunda versión:

```
private int preguntaNumeroAlUsuario() {
    while(true) {
        try {
            Scanner sc = new Scanner(System.in);
            System.out.println("Dime un número entre 0 y "+LIMITE_SUPERIOR);
            return sc.nextInt();
        }catch(InputMismatchException e) {
            System.out.println("¡Oups! Te has equivocado al escribir");
        }
    }
}
```

Retocamos el código y lo probamos

Realiza el ejercicio 1 del boletín

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
3 – El bloque try…catch

- Es importante recalcar que, en cuanto se produce la excepción, se **interrumpe el flujo normal** de ejecución y **NO se ejecutan las instrucciones siguientes que hubiera**. Observa la siguiente modificación:

```
private int preguntaNumeroAlUsuario() {
    while(true) {
        try {
            Scanner sc = new Scanner(System.in);
            System.out.println("Dime un número entre 0 y "+LIMITE_SUPERIOR);
            int numero = sc.nextInt();
            System.out.println("Has introducido el número: "+numero);
            return numero;
        }catch(InputMismatchException e) {
            System.out.println("¡Oups! Te has equivocado al escribir");
        }
    }
}
```

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
3 – El bloque try…catch

- Si forzamos el error vemos que la frase *"Has introducido el número: XX"* no aparece cuando se provoca la excepción:

```
¿Quieres jugar? (S=Sí / N=No)
S
Dime un número entre 0 y 10
5
Has introducido el número: 5
¡Uyy! Ese no es el número que he pensado
Te quedan 4 intentos
Dime un número entre 0 y 10
cinco
¡Oups! Te has equivocado al escribir
Dime un número entre 0 y 10
```

Sigue el flujo normal

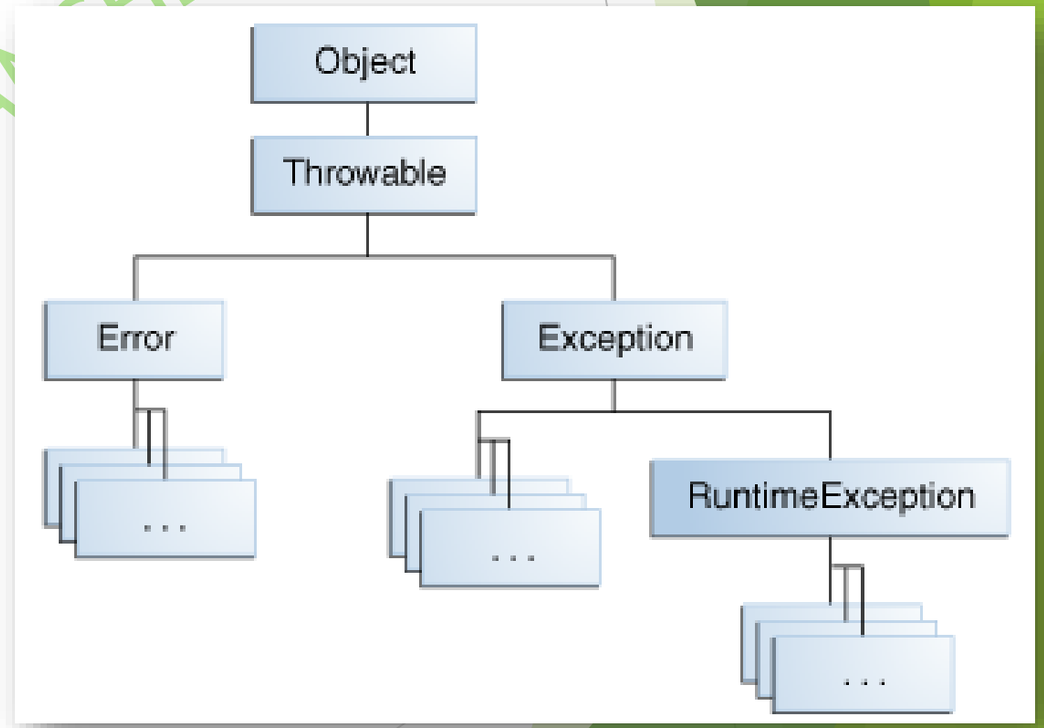
Se interrumpe el flujo normal

No aparece el mensaje *"Has introducido el número…"*

Realiza el ejercicio 2 del boletín

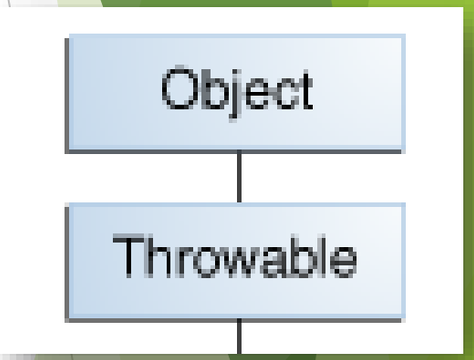
UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
4 – La jerarquía de “cosas lanzables”

- **4 – La jerarquía de “cosas lanzables”**
- Como no podía ser de otra manera, en Java las excepciones no son más que CLASES, que permiten crear su correspondientes OBJETOS.
- La clase **Throwable** (“lanzable”) es la superclases de todos los posibles errores y excepciones que puedan ocurrir.
- Esta clase sobrescribe el método **toString()** y nos devuelve el nombre de la clase de excepción/error que se ha producido. Por ejemplo: *java.util.InputMismatchException*



UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
4 – La jerarquía de “cosas lanzables”

- Además, la clase **Throwable** presenta dos métodos que se usan frecuentemente:
  - **String getMessage()**: este método devuelve el mensaje que escribiera el programador al crear y lanzar la excepción. Como no es obligatorio escribir un mensaje, la cadena devuelta puede ser nula.
  - **void printStackTrace()**: este método imprime en la salida de errores del sistema la “traza” o lista de llamadas a métodos que hay en la pila de ejecución del programa.




```
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:939)
        at java.base/java.util.Scanner.next(Scanner.java:1594)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
        at es.tuespiral.u5.p1.adivinaexcepciones.juego.Partida.preg
        at es.tuespiral.u5.p1.adivinaexcepciones.juego.Partida.inic
        at es.tuespiral.u5.p1.adivinaexcepciones.juego.JuegoAdivina
        at es.tuespiral.u5.p1.adivinaexcepciones.Aplicacion.main(Ap
Command execution failed.
```

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
4 – La jerarquía de “cosas lanzables”

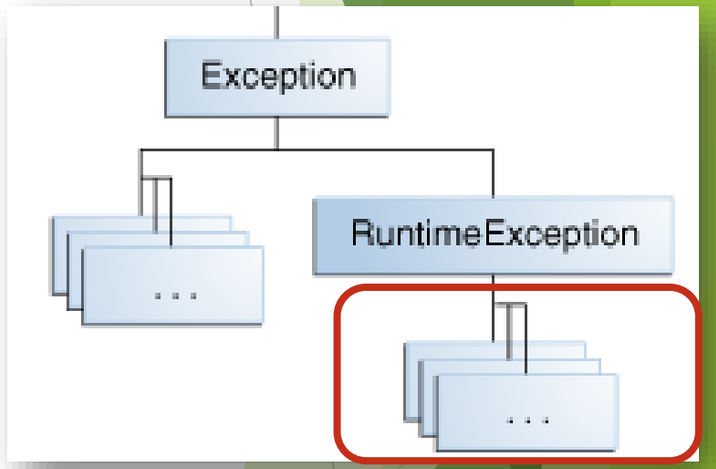
- La clase **Error** es una subclase de **Throwable** que representa a los problemas que **NO SE PUEDEN ARREGLAR** en tiempo de ejecución. Son situaciones que teóricamente no deberían ocurrir normalmente como:
  - Error en la máquina virtual java: VirtualMachineError
  - No queda memoria libre: OutOfMemoryError
  - Desbordamiento de la pila de ejecución: StackOverflowError
  - …
- Estos errores no son previsibles ni recuperables. Simplemente tenemos que cruzar los dedos para que no pasen, **NUNCA debemos intentar capturarlos con un try…catch**.

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
4 – La jerarquía de “cosas lanzables”

- La clase **Exception** también es una subclase de Throwable y representa los problemas que una aplicación **PODRÍA INTENTAR ARREGLAR** en tiempo de ejecución. Aquí nos encontramos algunas “viejas conocidas” y muchas otras nuevas:
  - ArrayIndexOutOfBoundsException, InputMismatchException, Arithmetic-Exception, NullPointerException, NumberFormatException…
  - SQLException, FileNotFoundException, XMLParseException…
- Dependiendo de si estos problemas son previsibles o no previsibles, Java los separa en dos ramas de la jerarquía de clases:
  - **Problemas no previsibles**: clases que descienden de **RuntimeException**
  - **Problemas previsibles**: clases que descienden **directamente de Exception.**
- Veamos cómo hay que actuar en cada caso.

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
5 – Manejando una “unchecked exception”

- **5 – Manejando una “unchecked exception”**
- Las excepciones que descienden de **RuntimeException** representan problemas no previsibles, así que el lenguaje **NO TE OBLIGA A CAPTURARLAS Y MANEJARLAS**. Por eso se les llama “unchecked exceptions” (excepciones no verificadas).
- Ante una “unchecked exception” tenemos dos opciones a la hora de escribir nuestro código:
  - Ignorarla y que se propague hasta el main y que se nos cierre el programa.
  - Añadir código para capturarlas y manejarlas.

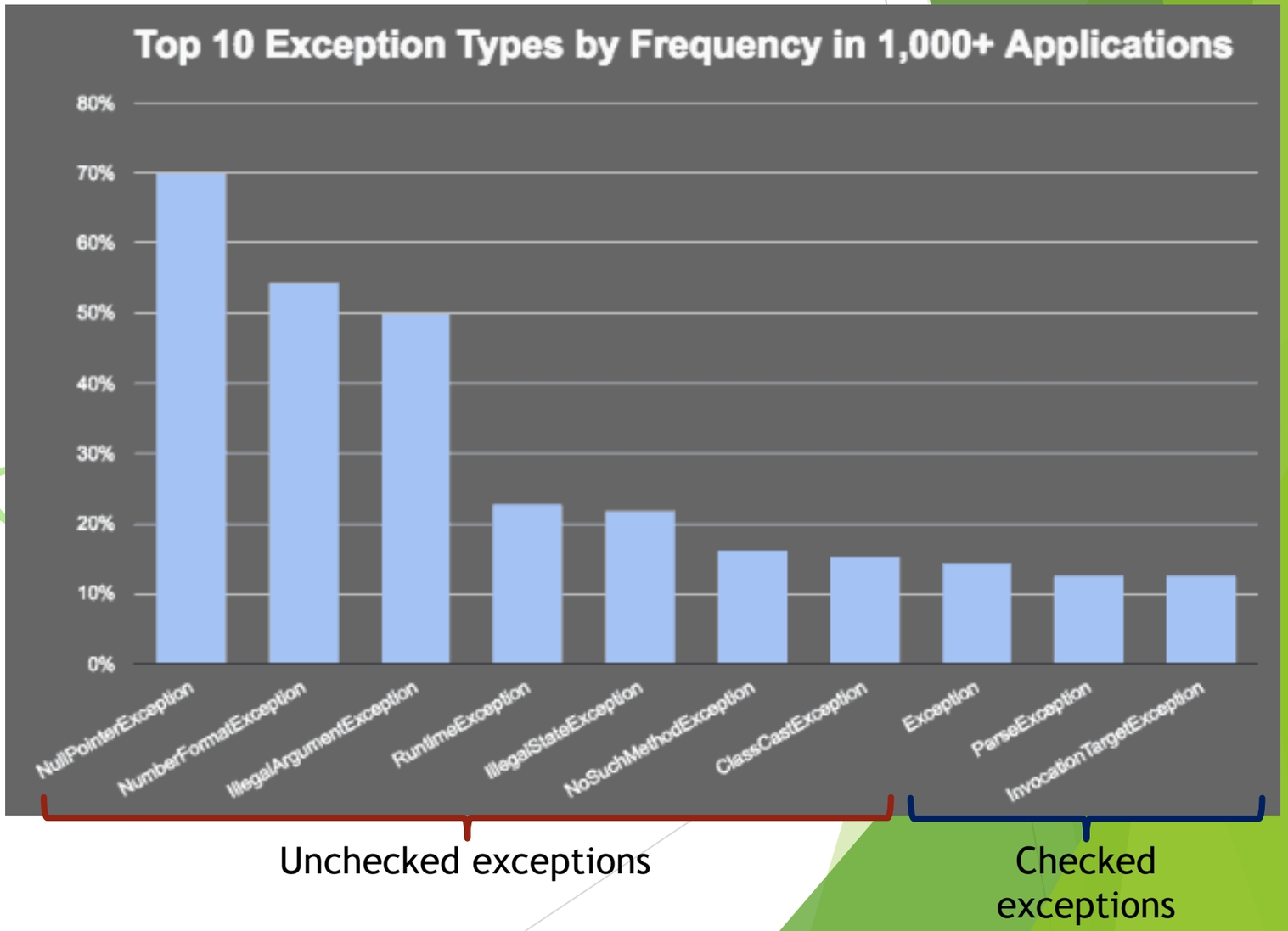


UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
5 – Manejando una “unchecked exception”

- Estas excepciones **se producen cuando hay descuidos o pequeños fallos en el código** (NullPointerException ClassCastException, IndexOutOfBoundsException…). Estas excepciones realmente **son “amigas” del programador** porque nos indican qué partes del código no hacen bien su trabajo y hay que modificar.
- **Normalmente NO nos interesa capturarlas.** De este modo el programa fallará en la fase de prueba y podremos arreglar el código antes de darlo por bueno.
- Aunque no las capturemos, tenemos que conocer el significado de algunas de las excepciones “unchecked”, ya que aparecen frecuentemente.

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
5 – Manejando una “unchecked exception”

- Como vemos en el gráfico las unchecked son más frecuentes que las checked.
- Veamos dos de ellas:
  - **IllegalArgumentException:** se lanza por un método que ha recibido un parámetro no válido. Ejemplo: una llamada a un método como *divide(10, 0)* podría lanzar esta excepción para indicar que el 0 no vale como divisor.



UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
5 – Manejando una “unchecked exception”

- **IllegalStateException:** se lanza por un método que no puede ejecutarse correctamente porque el objeto tiene un estado inválido o incoherente. Ejemplo: si tenemos un objeto de la clase Bombilla pero que está fundida y se le llama al método *encender()* podría lanzar esta excepción.

- Si nos ocurre alguna de estas situaciones deberíamos crear y lanzar una excepción. Veamos la sintaxis:


```
public void encender() {
    if (fundida) {
        throw new IllegalStateException("La bombilla está fundida, no se puede entender");
    }
    else {
```

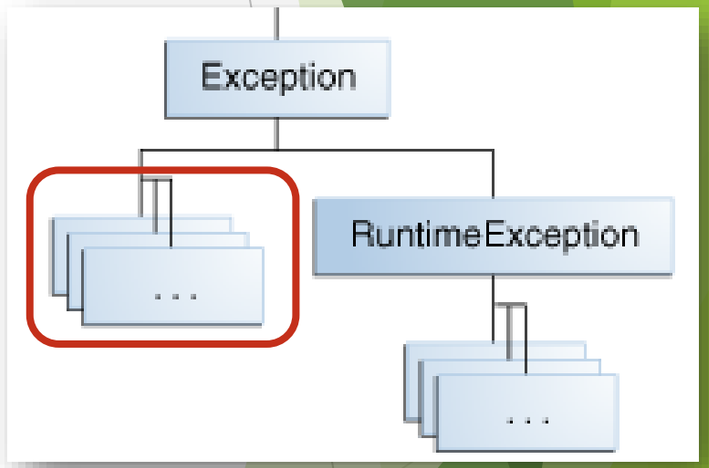
Una excepción se crea con un **new** al igual que todos los objetos y se lanza con un **throw**

El mensaje “La bombilla está fundida…” es opcional.

Realiza los ejercicios 3 y 4 del boletín

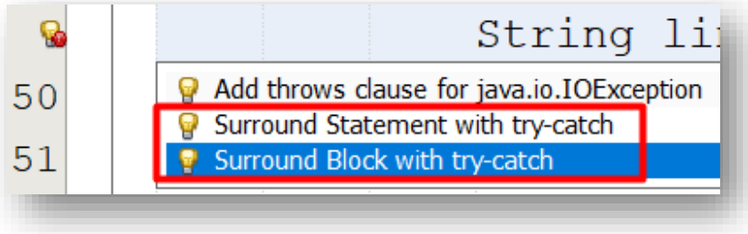
UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
6 – Manejando una “checked exception”

- **6 – Manejando una “checked exception”**
- Las clases que descienden directamente de **Exception** representan problemas previsibles. Ejemplos:
  - Intentamos abrir una conexión a una base de datos pero no lo conseguimos -> se lanza una SQLException.
  - Intentamos abrir un fichero pero no existe o ha sido borrado -> se lanza una FileNotFoundException.
  - Intentamos analizar/parsear un documento XML pero hay fallos sintácticos -> se lanza una XMLParseException.
- Java considera que **estos problemas son previsibles y nos obliga a hacer algo, no podemos ignorarlos**. Por eso se le llaman “checked exceptions” (excepciones verificadas).



UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
6 – Manejando una “checked exception”

- Ante una “checked exception” tenemos dos opciones:
  - Añadir código para capturarlas y manejarlas.
  - Permitir que la excepción se propague.
- **Capturar y manejar**
  - Se realiza con un try… catch como con las unchecked exceptions.
  - Sin embargo, ahora el **compilador nos marcará como errónea** una instrucción que pueda lanzar una checked exception y no esté “dentro” de un bloque try…catch.
  - Como siempre el IDE nos ofrece su ayuda:



Abrimos y estudiamos el código de U5.P2.FicheroCheckedException

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
6 – Manejando una “checked exception”

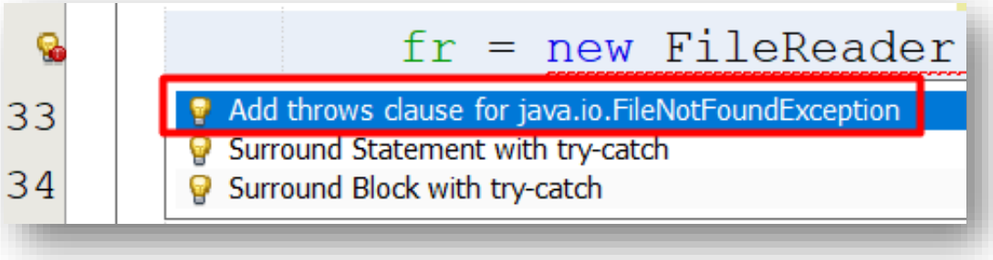
- Este código sigue una política de gestión de excepciones entre la clase “cliente” (PruebaFicheroCheckedException) y la clase “servidora” (Fichero):
  - La clase “servidora” captura todas la checked exception, ofreciendo simplemente un mensaje de error.
  - Esto hace que el código de la clase “cliente” sea más sencillo porque no tiene que poner try…catch.
  - Sin embargo, la clase “cliente” no tiene información del resultado de la ejecución de un método. No tiene forma de saber si el método ha hecho su trabajo o se ha provocado una excepción de forma “silenciosa” y solo se ha escrito un mensaje de error en la pantalla.
- Podríamos llamar a esta política de gestión como de “**ocultación de excepciones**”.

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
6 – Manejando una “checked exception”

- **Permitir que la excepción se propague**
  - En este caso NO usamos un bloque un try… catch, pero tenemos que “marcar la firma del método” para indicar que ese método es potencialmente peligroso.
  - Simplemente, tenemos que añadir la coletilla **throws** seguida de la excepción que se lance. Ejemplo:

```
public void abrir() throws FileNotFoundException{
```

  - De nuevo el IDE es nuestra ayuda:



Abrimos y estudiamos el código de U5.P3.FicheroThrowsCheckedException

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
6 – Manejando una “checked exception”

- La nueva política de gestión de excepciones entre la clase “cliente” (PruebaFicheroThrowsCheckedException) y la clase “servidora” (Fichero) sería así:
  - La clase “servidora” lanza a la clase “cliente” TODAS las checked exceptions ocurran. Esto simplifica el código de la clase “servidora” ya que no trata NINGUNA excepción.
  - Sin embargo, complica el código de la clase “cliente” porque tendrá que capturarlas y manejarlas.
  - Ahora la clase “cliente” tiene TODA la información (quizás demasiada) sobre el resultado de la ejecución de un método y podrá tomar las medidas que considere oportunas para poder redirigir el flujo de ejecución según corresponda.
- Podríamos llamar a esta política de gestión como de “**exhibición de excepciones**”.

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
6 – Manejando una “checked exception”

COMPARATIVA	OCULTACIÓN DE EXCEPCIONES	EXHIBICIÓN DE EXCEPCIONES
Ventajas	Facilita la escritura del código de la clase “cliente” ya que no tendrá que escribir los bloques try…catch.	La clase “cliente” conoce el resultado de la ejecución de un método, permitiendo la redirección del flujo de ejecución si fuera necesario.
Inconvenientes	Oculta a la clase “cliente” el resultado de la ejecución de un método. Así que la clase “cliente” no sabe si la ejecución fue bien o mal.	Dificulta la escritura del código de la clase “cliente” que tendrá que escribir código para tratar las excepciones.
¿Cuándo se usa?	Se debe capturar una excepción solo si se le puede dar una **solución completa** al problema y, además, la clase “cliente” **no necesita ser informada**.	Se debe lanzar la excepción cuando no se pueda dar una **solución completa** al problema o bien la clase **“cliente” deba conocer** que se ha producido un problema.
Ejemplo de uso	Conexión con un servicio web que suele estar saturado. Se hacen hasta 3 intentos de conexión (3 excepciones capturadas) antes de desistir y lanzar una excepción informando a la clase cliente.	Los fallos provocados por el usuario (el nombre del fichero no existe, la contraseña es incorrecta…) necesitan ser informados a la clase cliente para posibilitar que el usuario rectifique y pueda intentarlo de nuevo.

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
7 – Manejando varias “checked exceptions”

- **7 – Manejando varias “checked exceptions”**
- Cuando en un método se pueden producir varias “checked exceptions” tenemos las dos opciones de siempre: “lanzar o capturar” sin embargo, tenemos que hacer algunas matizaciones:
- **Lanzar varias checked exceptions**
  - Podemos ponerlas una detrás de otra en la firma del método:

```
public void abreImprimeContenido()
                    throws FileNotFoundException, IOException {
```

  - Pero también podemos “agrupar” varias excepciones en alguna de sus superclases:

```
public void abreImprimeContenido() throws IOException {
```


UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
7 – Manejando varias “checked exceptions”

- El siguiente método es la fusión entre “abre” e “imprimeContenido”:

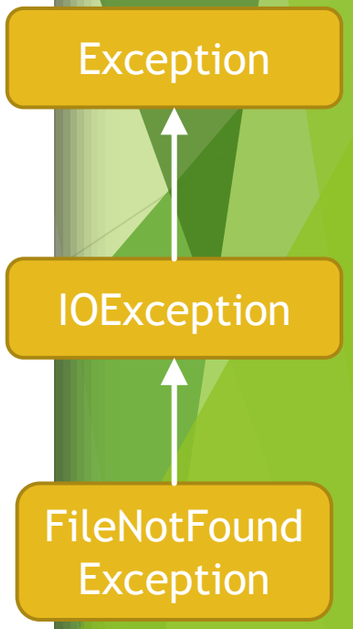
```
public void abreImprimeContenido() throws FileNotFoundException, IOException {

    fichero = new File(nombreFichero);
    fr = new FileReader(fichero);
    br = new BufferedReader(fr);
    abierto = true;

    if (abierto) {
        String linea = br.readLine();
        while (linea != null) {
            System.out.println(linea);
            linea = br.readLine();
        }
    } else {
        throw new IllegalStateException("El fichero no está abierto");
    }
}
```

O bien, las “agrupamos” en

```
throws IOException {
```



UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
7 – Manejando varias “checked exceptions”

- Esta técnica de agrupar excepciones se usa cuando:
  - Se lanzan más de 2 o 3 excepciones distintas en un mismo método, ya que afecta demasiado la programación de las clases “clientes” creando un código más denso, difícil de leer y de mantener.
  - O bien si no se quiere dar un nivel excesivo de detalle a la clase “cliente”, “agrupando” varias excepciones en una sola, más general.
- **Capturar varias checked exceptions**
  - Cuando decidimos capturar varias excepciones lo podemos hacer:
    - Individualmente pero siempre “en orden”.
    - O agrupándolas en una de sus superclases.
  - Veamos cómo funciona:

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
7 – Manejando varias “checked exceptions”

- Captura individual “en orden”:

Cuando se produce una excepción, se detiene el flujo normal y se va **probando EN ORDEN si la excepción es compatible** con alguno de los bloques catch.

Una excepción es compatible con un bloque catch si la clase de la excepción **coincide con la clase del bloque catch o bien es una subclase de ella**

```
public void abreImprimeContenido() {
    try {
        fichero = new File(nombreFichero);
        fr = new FileReader(fichero);
        br = new BufferedReader(fr);
        String linea = br.readLine();
        while (linea != null) {
            System.out.println(linea);
            linea = br.readLine();
        }
    } catch (FileNotFoundException e) {
        System.out.println("No se encuentra el fichero "+nombreFichero);
    } catch (IOException ioe) {
        System.out.println("Error al leer del fichero "+nombreFichero);
    }
}
```

Puede lanzar una FileNotFoundException o una IOException

1º

2º

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
7 – Manejando varias “checked exceptions”

- Si nos equivocamos y ponemos una excepción “más general” primero y después alguna “más concreta” el IDE nos dará un error:

```
public void abreImprimeContenido() {
    try {
        fichero = new File(nombreFichero);
        fr = new FileReader(fichero);
        br = new BufferedReader(fr);
        String linea = br.readLine();
        while (linea != null) {
            System.out.println(linea);
            linea = br.readLine();
        }
    } catch (IOException ioe) {
        System.out.println("Error al leer del fichero "+nombreFichero);

    } catch (FileNotFoundException e) {
        System.out.println("No se encuentra el fichero "+nombreFichero);
    }
}
```

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
7 – Manejando varias “checked exceptions”

- La otra posibilidad es “capturarlas en grupo” usando **una superclase común a las posibles excepciones** que se puedan lanzar:

```
public void abreImprimeContenido() {
    try {
        fichero = new File(nombreFichero);
        fr = new FileReader(fichero);
        br = new BufferedReader(fr);
        String linea = br.readLine();
        while (linea != null) {
            System.out.println(linea);
            linea = br.readLine();
        }
    } catch (IOException ioe) {
        System.out.println("Error al abrir o leer del fichero "+nombreFichero);
    }
}
```

Si se lanza una FileNotFoundException será capturada porque IOException es su superclase

Realiza los ejercicios 5 y 6 del boletín

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
8 – El bloque finally

- **8 – El bloque finally**
- El bloque finally (finalmente, para finalizar) acompaña a un bloque try…catch y se ejecuta SIEMPRE, se produzca o no la excepción.
- Su sintaxis es así:

```
try {
    // Código que lanza excepciones
} catch (Exception e) {
    // Código para la excepción
} finally {
    // Código que se ejecuta siempre
}
```

Se podría leer como:

Intentamos ejecutar {
    // Código del try
} si algo va mal hacemos {
    // Código del catch
} y para finalizar hacemos esto {
    // Código del finally
}

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
8 – El bloque finally

- El bloque *finally* se usa mucho en la **gestión de recursos** (ficheros, bases de datos, sesiones con servicios web…) ya que cuando los utilizamos normalmente se sigue la siguiente secuencia de pasos:
  - **Abrir** el recurso (abrir un fichero, conectar con una BD)
  - **Utilizar** el recurso (pedirle datos, realizar operaciones…)
  - **Cerrar** el recurso (cerrar el fichero, cerrar sesión…)
- Si al abrir o utilizar el recurso se produce una excepción y no lo cerramos entonces ese recurso **puede quedarse bloqueado** y no podrá ser utilizado por otros programas o hilos de programación.
- El bloque **finally permite cerrar un recurso**, se produzca o no una excepción, garantizando que no se queda inutilizable.

UD 5 - CONTR
8 - El bloque

- Ejemplo:

```
public void abreImprimeCierra() {
    try {
        fichero = new File(nombreFichero);
        fr = new FileReader(fichero);
        br = new BufferedReader(fr);
        String linea = br.readLine();
        while (linea != null) {
            System.out.println(linea);
            linea = br.readLine();
        }
    } catch(IOException e) {
        System.out.println("Se produjo un error al abrir o leer del archivo");
    } finally {
        try {
            if (br != null)
                br.close();
            if (fr != null)
                fr.close();
        } catch(IOException e) {
            System.out.println("Se produjo un error al cerrar el archivo");
        }
    }
}
```

Realiza el ejercicio 7 del boletín

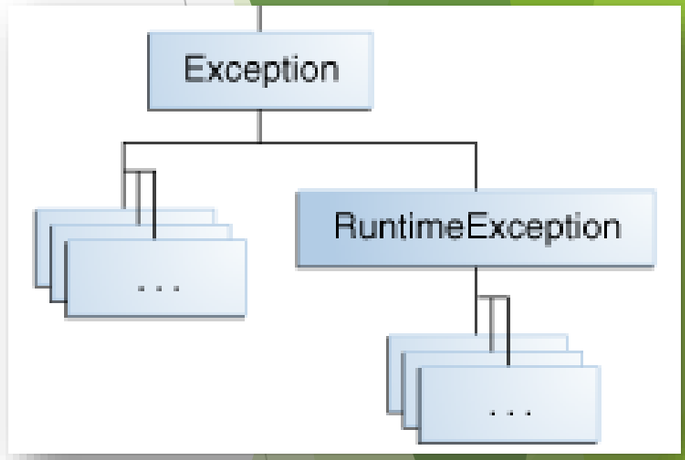
Si dentro del **finally** se lanzan excepciones tendremos que hacer lo de siempre: capturar o dejar que se propaguen

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
9 – Excepciones “personalizadas”

- **9 – Excepciones “personalizadas”**
- Como programadores podemos crear, lanzar y capturar **nuestras propias excepciones**. Esta técnica se usa frecuentemente en las siguientes situaciones:
  - **Necesitamos representar fielmente los problemas asociados al negocio de nuestra aplicación** y las excepciones que aporta el lenguaje no son suficientemente descriptivas. Ejemplos de excepciones que podríamos crear: ClienteNoEncontradoException, LoginException, CreditoInsuficienteException…
  - **Queremos una excepción más general que agrupe a otras más concretas**. De este modo simplificamos la gestión de excepciones a la clase “cliente” y ocultamos detalles e información que para esta clase no son relevantes. Ej: ClienteException (engloba varias)

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
9 – Excepciones “personalizadas”

- Para crear una excepción “personalizada” simplemente creamos una clase y la “extendemos” de:
  - **Exception:** si la consideramos previsible y debe ser comprobada por el compilador.
  - **RuntimeException**: si la consideramos imprevisible o bien se corresponde con un error irrecuperable.
- Ejemplo de creación de una excepción:




```
public class LoginException extends Exception {
    public LoginException(String mensaje) {
        super(mensaje);
    }
}
```

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
9 – Excepciones “personalizadas”

- Ejemplo de uso:

```
public class Login {
    public Usuario intentaLogin(String usuario, String password)
                    throws LoginException {

        if (existeUsuario(usuario) && coincidePassword(usuario, password))
            return obtieneUsuario(usuario);
        else
            throw new LoginException("Usuario no existe o clave incorrecta");
    }
```

- La creación de excepciones personalizadas es una práctica habitual, sobre todo para representar los problemas asociados al negocio que estamos modelando.

Realiza el ejercicio 8 del boletín

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
10 – Debate abierto. Buenas y malas prácticas

- **10 – Debate abierto. Buenas y malas prácticas**
- Antes de que C++ popularizara el uso de las excepciones los métodos **devolvían un "error code"**, es decir, un número entero que representaba qué había ocurrido dentro del método. Ejemplo:
  - 0 -> Si todo fue bien
  - 1 -> Error al abrir el fichero
  - 2 -> Error al escribir en el fichero
  - …

Esto limita el rango del parámetro de salida de un método, ya que si devuelve un error no podrá devolver un resultado que se haya calculado en el interior del método

Además obliga a la clase cliente a escribir un código con muchos if o un switch para controlar los posibles errores:

```
int error = método();
switch(error) {
   case 0: … break;
   case 1: … break;
   case 2: … break;
}
```

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
10 – Debate abierto. Buenas y malas prácticas

- **Las excepciones vienen a "limpiar" ese código denso y poco legible**, separando el código productivo (bloques try y finally) del código que gestiona los problemas (catch). Además, aporta un mecanismo más expresivo y seguro que los "error code".
- Aún así el código asociado a las excepciones también puede llegar a ser denso e ilegible y esto **genera un debate** en la comunidad de desarrolladores que aún no se ha cerrado:
  - Hay consenso en que las **"unchecked exceptions" son buenas** porque hacen bien su trabajo sin ensuciar el código.
  - Pero con las **"checked exceptions" hay mucha controversia** porque pueden complicar el código considerablemente si no se adopta una buenas política de gestión de las excepciones desde el principio.

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
10 – Debate abierto. Buenas y malas prácticas

- Dado que en C++ todas las excepciones son “unchecked”, cuando un programador de este lenguaje pasa a un proyecto en Java tiende a cometer el siguiente error: crear todas las excepciones del proyecto extendiendo de RuntimeException.

- **Este debate no se resolverá nunca** porque en el fondo de la controversia hay algo subjetivo: “¿qué le parece al programador el código resultante? ¿feo o bonito? ¿sucio o limpio? ¿útil o inútil?”

- Personalmente, **estoy a favor de las excepciones** pero para que no se nos vuelvan en nuestra contra tenemos que tenerlas presentes desde el comienzo del proyecto, aplicando **buenas prácticas** y evitando las **malas prácticas**. Veamos algunas:

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
10 – Debate abierto. Buenas y malas prácticas

- **Malas prácticas**:
  - **Colgar todas las excepciones del proyecto de RuntimeException** puede ser cómodo y nos puede ayudar a programar más rápido pero estamos **alimentando un monstruo** que nos la hará pagar cuando mantengamos el código. El problema reside en que los métodos se convierten en "una caja de sorpresas desagradables" para las clases clientes. Al no tener las excepciones documentadas en la firma del método, el compilador no nos puede ayudar y tendremos que descubrir los errores "en tiempo de ejecución" (que es más costoso).
  - **Tragarse una excepción**, esto se produce cuando una excepción se captura pero el código del **bloque catch no hace nada**. Esto silencia al compilador que nos "dejará tranquilos" pero oculta los errores y los hace más difícil de localizar y depurar.

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
10 – Debate abierto. Buenas y malas prácticas

- **Buenas prácticas**:
  - **Detección temprana.** Cuanto más tarde se detecte un problema más difícil de arreglar se vuelve. Esto se vuelve muy importante para gestionar la introducción de datos al sistema por parte del usuario.
  - **Excepciones personalizadas significativas.** Se debe crear excepciones personalizadas para cada proyecto de forma que representen problemas que sean **significativos para la clase cliente**. Ejemplo: es más significativo lanzar una LoginException que una SQLException.
  - Si una clase es **hija de Exception** debe representar un problema **recuperable**. Si no fuera así ensuciaríamos el código para nada.
  - Si una clase es **hija de RuntimeException** debe representar un problema **irrecuperable o muy poco probable**. Si no fuera así perderíamos la opción de añadir código que lo arregle y el problema acabaría apareciendo más tarde.

UD 5 – CONTROL DE ERRORES Y EXCEPCIONES
10 – Debate abierto. Buenas y malas prácticas

- **Solo capturaremos una excepción en alguno de estos casos**:
  - Si el método puede arreglar el problema completamente y no es necesario informar a la clase cliente de este problema.
  - Si el método necesita hacer un arreglo parcial del problema se captura la excepción pero tendrá que informar a la clase cliente lanzando una nueva excepción.
  - Si queremos ofrecer a la clase cliente una excepción más general que agrupe a otras más concretas. Entonces capturaremos todas la excepciones concretas para lanzar la excepción más general.