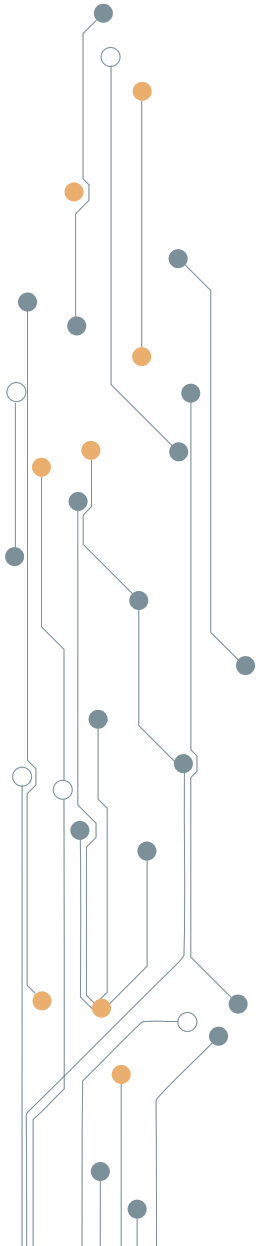




Gestión de excepciones

Índice



1 La jerarquía de excepciones	3
2 Los bloques try, catch y finally	8
3 Un try y varios catch	9
4 Anidar bloques try	11
5 Excepciones multicatch	12
6 Propagar excepciones: throws	13
7 Crear excepciones personalizadas	14

1. La jerarquía de excepciones

Una excepción es un problema que surge durante la ejecución de un programa. Cuando se produce una excepción el flujo normal del programa se interrumpe y el programa termina de forma anormal, por lo tanto estas situaciones deben ser gestionadas.

Una excepción puede ocurrir por muchas razones diferentes, por ejemplo algunos escenarios pueden ser los siguientes:

- Un usuario ha introducido datos no válidos.
- Un archivo que necesita ser abierto no se puede encontrar.
- Acceso a un elemento de un array con índice fuera de límites.
- Intento de acceder a un objeto con referencia null.
- Moldeo (conversión de tipos incorrecto).
- Una conexión de red se ha perdido en el proceso de comunicación
- La JVM se ha quedado sin memoria.

Un usuario ha introducido datos no válidos.

Un archivo que necesita ser abierto no se puede encontrar.

Acceso a un elemento de un array con índice fuera de límites.

Intento de acceder a un objeto con referencia null.

Moldeo (conversión de tipos incorrecto).

Una conexión de red se ha perdido en el proceso de comunicación

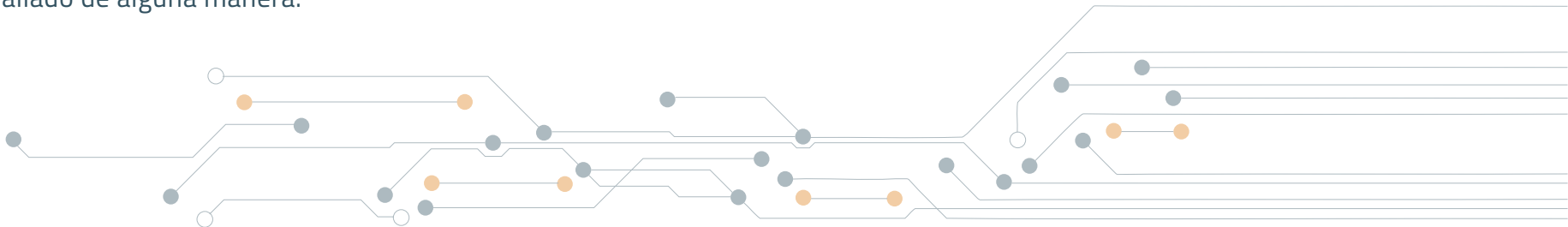
La JVM se ha quedado sin memoria.

Usuario

Programador

Recurso

Algunas de estas excepciones son causados por un error del usuario, otras por error del programador, y otras por recursos físicos que han fallado de alguna manera.



Sobre esta base se puede decir que en Java existen tres tipos de excepciones:

- **Errorres**, no son propiamente excepciones, porque surgen fuera del control del usuario o el programador. Los errores suelen ser ignorados en su código, ya que rara vez se puede hacer nada al respecto.

Por ejemplo, si se produce un desbordamiento de pila, surgirá un error.

- **Excepciones no comprobadas (unchecked exceptions)**, se producen en el momento de la ejecución, también se denominan excepciones de tiempo de ejecución. Suelen ser errores de programación, tales como errores de lógica o el uso indebido de una API.

Por ejemplo, si se define una array de tamaño 10 y se trata de acceder al undécimo elemento ocurre la excepción `ArrayIndexOutOfBoundsException`.

```
package com.juan.excepciones;
import java.io.File;
import java.io.FileReader;
public class PruebaExcepcionesChecked { public
static void main(String[] args) {
    File fichero=new File("c://fichero.txt");
    FileReader fr = new FileReader(fichero);

    }
}
```

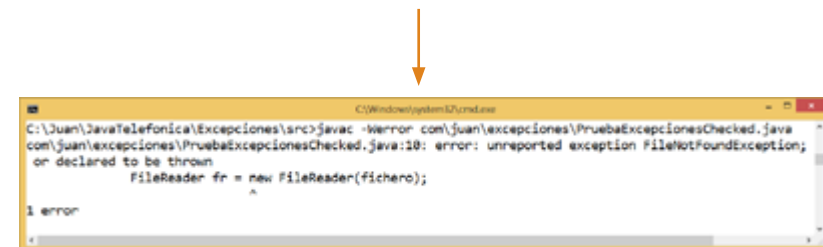


IMAGEN 8.1 EXCEPCIÓN NO COMPROBADA (CHECKED UNEXCEPTION).



- **Excepciones comprobadas (checked exceptions)**, una excepción que se produce en el momento de la compilación, estas también son llamados como excepciones de tiempo de compilación. Estas excepciones no pueden ser ignoradas en el momento de la compilación, el programador debe gestionar estas excepciones.

Por ejemplo, si se utiliza la clase `FileReader` en un programa para leer datos de un archivo, si el archivo especificado no existe, entonces se produce la excepción `FileNotFoundException`, y el compilador solicita al programador que gestione dicha excepción.

```
package com.juan.excepciones;
public class PruebaExcepcionesUnchecked {
    public static void main(String[] args) {
        int array[] = { 1, 2, 3, 4, 5, 6, 7, 8,
9, 10 };
        for (int i = 10; i >= 0; i--) {
            System.out.println(array[i]);
        }
    }
}
```

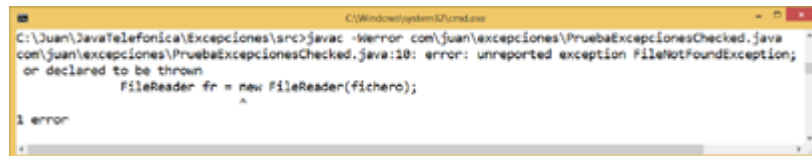


IMAGEN 8.1 EXCEPCIÓN NO COMPROBADA (CHECKED UNEXCEPTION).

Todo objeto de tipo excepción es una instancia de una clase que derive de `Throwable`. en la imagen siguiente se muestra un diagrama simplificado de la jerarquía de excepciones.

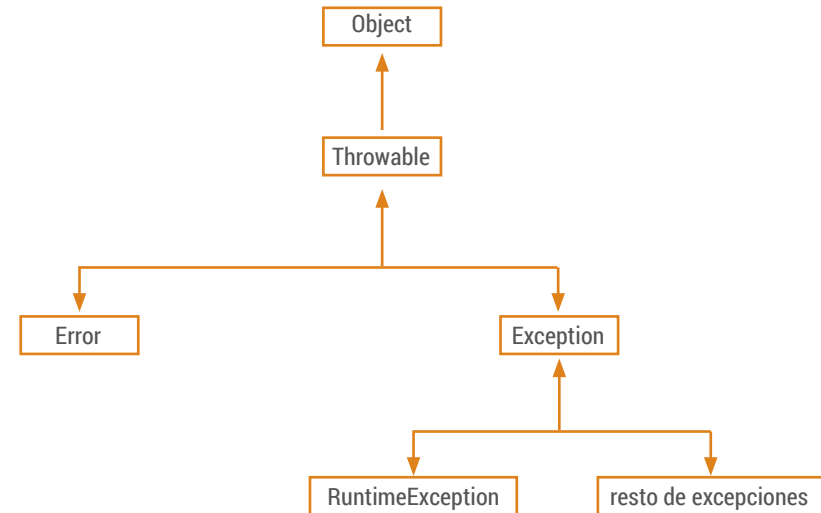


IMAGEN 8.3 JERARQUÍA DE EXCEPCIONES.

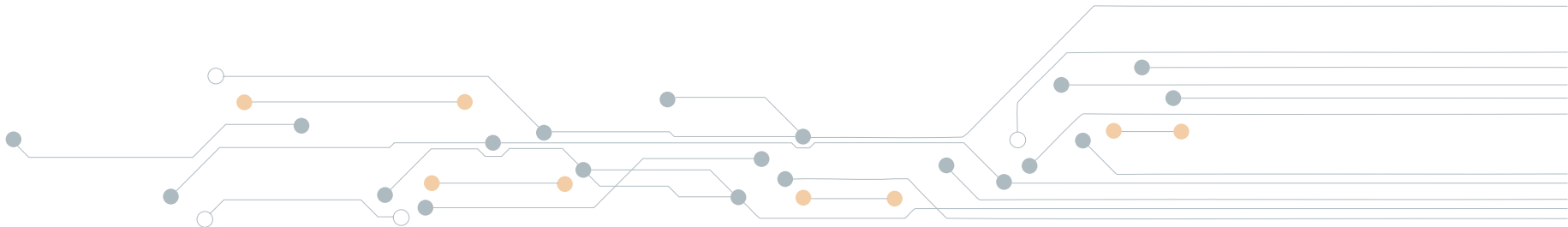
De Throwable heredan dos clases:

- **Error**, son los errores comentados anteriormente de errores internos de JVM y agotamiento de recursos dentro del sistema de ejecución de Java. Se podría simplemente avisar al usuario y salir del programa.
- **Exception**, que a su vez se divide en otras dos ramas:
- **RuntimeException**, estas son las que se producen cuando se comete un error de programación, en general provienen de conversiones erróneas, accesos a índices de arrays fuera de límites y accesos a punteros null.
- El resto de excepciones que no heredan de RuntimeException, pueden suceder por intentar leer más allá del límite de un archivo, intentar abrir una URL mal formada o intentar trabajar con un objeto de tipo Class con una cadena que no sea la identificación correcta de la clase.

Casi se puede afirmar que la excepción **RuntimeException** es **culpa del programador**.

Todas las **excepciones no comprobadas** (exception unchecked o excepciones en tiempo de ejecución) son objetos instanciados de clases que heredan de **Error** o **RuntimeException**.

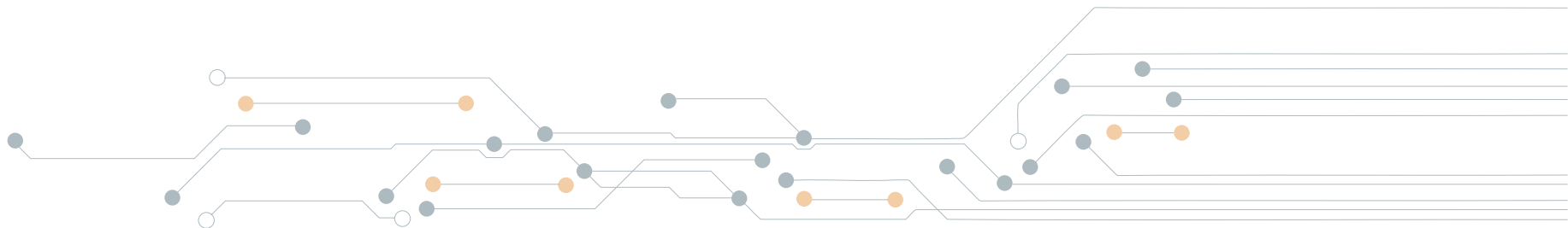
Todas las **excepciones comprobadas** (exception checked o excepciones en tiempo de compilación) son todas las que **heredan de Exception**, pero **no de RuntimeException**. El compilador es el aliado del programador advirtiéndole que estas excepciones deben ser tratadas y gestionadas.



Como todas las excepciones son subclases de **Throwable**, todas heredan los métodos de esta, y estos son:

Método	Descripción
Throwable fillStackTrace()	Devuelve un objeto Throwable que contiene el rastreo de la pila completo de la situación que produce la excepción.
String getLocalizedMessage()	Devuelve una descripción localizada del sitio en el que se produce la excepción.
String getMessage()	Devuelve la descripción de la excepción.
void PrintStackTrace()	Muestra el rastreo de la pila en una cadena.
void PrintStackTrace(PrintStream flujo)	Envía el rastreo de la pila a un flujo específico, por ejemplo un archivo.
void PrintStackTrace(PrintWriter flujo)	
String toString()	La cadena que devuelve contiene la descripción completa de la excepción. Es el que se invoca cuando el objeto Throwable es parámetro de funciones print o println.

Tabla 9.1: Métodos de Throwable.



2. Los bloques try, catch y finally

El control de excepciones se gestiona con la utilización de 5 palabras reservadas: **try**, **catch**, **finally**, **throw** y **throws**. Con **try** se define el bloque de sentencias en el que están las que pueden producir una excepción. Si se produce una excepción en este bloque se crea el objeto excepción y el flujo de ejecución de este bloque se interrumpe y salta al bloque de instrucciones del bloque **catch**. Las sentencias del bloque **finally** se ejecutan siempre, al finalizar el bloque try o al finalizar el bloque **catch**.

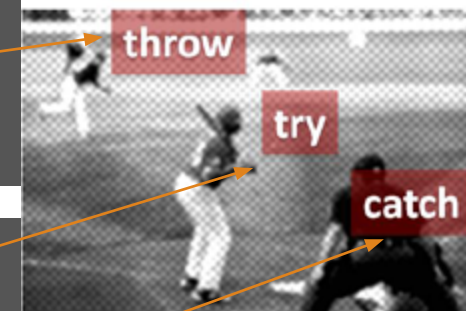
La estructura de utilización de los bloques try, catch y finally es el siguiente:

```
//sentencias
try{
    //sentencias con llamadas a funciones que lanzan
    excepciones
} catch(Exception e){
    //sentencias que con referencia "e" utilizan
    funcionalidades Throwable
}finally{
    //sentencias que se ejecutan siempre
}
```

throw se utiliza para asociar a las funciones la excepción que lanzarán cuando se produzcan unas condiciones determinadas. Las funciones "lanzadas" con **throw** son las que cuando son utilizadas deben ir en bloques try.

```
void funcionLanzadora(){
    //sentencias
    if(condicion==true)
        throw new
        UnaExcepcion("excepcion producida");
    //sentencias
}
```

```
void funcionGetionadora(){
    //sentencias
    try{
        funcionLanzadora();
    }catch(UnaExcepcion e){
        Sytem.out.println(e.getMessage());
    }
    finally{
        //sentencias que se ejecutan siempre
    }
}
```



throws es el mecanismo de propagar las excepciones, cuando una función “throws” (propaga) una excepción, se libera de gestionarla y pasa su gestión a quien la invoque.

```
public static void main(String[] args) throws  
InterruptedException {  
    //sentencias  
    Thread.sleep(5000); //Retraso de la ejecución de 5 segundos  
    //sentencias  
}
```



Thread.sleep lanza excepción InterruptedException, se propaga con throws

3. Un try y varios catch

En un bloque **try** se pueden generar una o más excepciones, por tanto se pueden encadenar bloques **catch**, con la precaución de ponerlos en orden tal que las excepciones que hereden de otras deberán estar en bloques catch superiores. Esto es así porque los bloques **catch** están unos a continuación de otros y sólo se accede a las sentencias de uno de ellos. Java va examinando cual es la clase de excepción del objeto generado y el primer catch que este parametrizado con esa clase es para el que se ejecutan sus sentencias.

```
try {  
    //sentencias  
}catch(ExceptionTipo1 e1)  
{  
    //sentencias catch  
}catch(ExceptionTipo1 e1){  
    //sentencias catch  
}catch(ExceptionTipo1 e1){  
    //sentencias catch  
}
```

De tal manera que si el primer **catch** gestiona objetos de clase **Exception**, y el siguiente por ejemplo **IOException**, siempre se ejecutaran las sentencias del bloque **Exception** y nunca las del bloque **IOException**. Por tanto hay que tener en cuenta la jerarquía de clases de las excepciones para construir el encadenamiento de los bloque catch.

```
try{
    archivo = new
    FileInputStream(strNombreArchivo);
    x = (byte) archivo.read();
}catch(IOException iex){
    iex.printStackTrace(); fallo = true;
}catch(FileNotFoundException f3x){ //No es correcto
    fex.printStackTrace();          fallo = true;
}
```



4. Anidar bloques try

Un bloque try se puede anidar dentro de otro. en esta situación hay que tener en cuenta la regla siguiente:

Una excepción generada en el bloque try interno que no se capture en una sentencia catch asociada a ese bloque se propaga a bloque try externo.

```
public class PruebaExcepcionesAnidadas {
    public static void main(String[] args) {
        int numerador[] = { 6, 5, 8, 24, 120, 345, 654};
        int denominador[] = {2, 1, 0, 3, 0, 5};

        try{ //externo
            for( int i=0; i<numerador.length; i++){
                try{ //anidado
                    System.out.println(numerador[i] + "/" +
denominador[i] +
                                "= "+numerador[i]/denominador[i]);
                }
                catch (ArithmeticException ex){
                    System.out.println(ex.getMessage());
                }
            }
        }catch(ArrayIndexOutOfBoundsException ex){
            System.out.println(ex.getMessage());
            System.out.println("Error fatal: programa terminado");
        }
    }
}
```

```
C:\Windows\system32\cmd.exe
C:\Juan\JavaTelefonica\Excepciones\bin>java -cp . com.juan.excepciones.PruebaExcepcionesAnidadas
6/2= 3
5/1= 5
8/3= 8
java.lang.ArithmeticException: / by zero
345/0= 69
6
Error fatal: programa terminado
```

IMAGEN 8.6 TRYs ANIDADOS.

5. Excepciones multicatch

Cuando se tienen que tratar varias excepciones y el código para todas ellas es el mismo se puede agrupar utilizando esta funcionalidad de multicatch, que consiste en poner las excepciones a tratar separadas unas de otras por el carácter "|".

```
catch( <excepcion_01> | <excepcion_02> | <excepcion_n> {
    //sentencias
}
```

Ninguna de las excepciones en la lista puede ser subclase de otra que este en la misma lista, si es así el compilador produce error de excepción ya tratada.

```
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Scanner;
public class PruebExcepcionesMultiCatch {
    public static void main(String a[]) {
        PruebExcepcionesMultiCatch prueba =
            new PruebExcepcionesMultiCatch();
        prueba.probar(1);
        prueba.probar(2);
    }
    public void probar(int i) {
        Scanner teclado = new Scanner(System.in);
        System.out.println("Escribe algo:");
        String str = teclado.nextLine();
        try {
            if (i == 1) Integer.parseInt(str);
```

```
else new URL(str);
        /*catch (NumberFormatException nfe){
            * System.out.println(
            * "NumberFormatException: "+nfe.
getMessage());
            * }
            * catch (MalformedURLException mue){
            * System.out.println(
            * "MalformedURLException: "+mue.
getMessage());
            * }
            * catch (Exception ex){
            * System.out.println("Exception... "+ex.
getMessage());
            * }*/
        catch (NumberFormatException | MalformedURLException
e) {
            System.out.println(e.getMessage());
        }
    }
}
```

6. Propagar excepciones: throws

Tal y como se menciona anteriormente con throws se propagan excepciones, en realidad lo que se está haciendo es ignorar el tratamiento de una excepción dentro del bloque de sentencias de la función que utilice throws.

```
<acceso> <tipo> <identificador_funcion> throws
<excepción>{
    //se ignora el tratamiento de <excepción>
    //no tiene que utilizarse try con catch para <excepción>
}
```

La excepción será tratada por el código en el que se llame a la función.

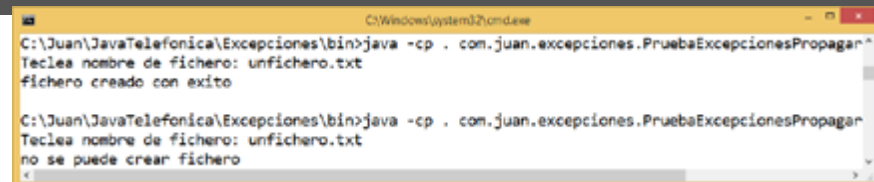
Si la función main throws una excepción, esta será ignorada en su tratamiento.

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class PruebaExcepcionesPropagar {
    public static void main(String [] args) throws
    IOException{
        Scanner teclado =new Scanner(System.in);
        System.out.print ("Teclea nombre de fichero: ");

        String nombre= teclado.nextLine();
        fichero(nombre);
    }

    static void fichero(String nombre) throws IOException{
        File file = new File(nombre);
        if(file.createNewFile()) System.out.
        println("fichero creado con éxito");
        else System.out.println("no se puede crear
        fichero");
    }
}
```



```
C:\Windows\system32\cmd.exe
C:\Juan\JavaTelefonica\Excepciones\bin>java -cp . com.juan.excepciones.PruebaExcepcionesPropagar
Teclea nombre de fichero: unfichero.txt
fichero creado con éxito

C:\Juan\JavaTelefonica\Excepciones\bin>java -cp . com.juan.excepciones.PruebaExcepcionesPropagar
Teclea nombre de fichero: unfichero.txt
no se puede crear fichero
```

IMAGEN 8.7 PROPAGAR EXCEPCIONES CON THROWS.

7. Crear excepciones personalizadas

Para crear una excepción personalizada basta con crear una clase que herede de **Throwable**, normalmente de la clase **Exception** o sus subclases, no necesita ningún requisito más como clase, ya que hereda los métodos de **Throwable**.

Se deberán generar los constructores que se considere necesario, así cuando se cree el objeto instanciado de esta clase, se puede tener información sobre la causa que produjo la excepción.

Normalmente se sobrepasa la función **toString** para que cuando se utilice el objeto de esta excepción en `print` o `println`, el mensaje este adaptado a esta excepción.

La función que lanza una de estas excepciones, cuando se produzcan las condiciones para las que fue creada, debe crear el objeto con la sentencia **throw** seguida del operador `new` con una de las formas de constructor implementado en la clase de excepción personalizada.

Además la función que crea los objetos de la excepción tiene que propagar dicha excepción con `throws`.

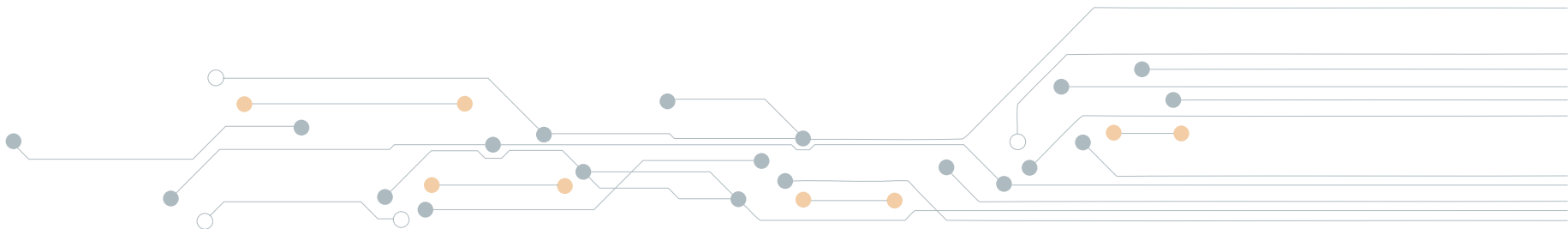
```
if(<condición>) throw new <excepción>(<lista_parametros>);
```

```
import java.util.Random;

public class PruebaExcepcionesPersonalizadas {
    public static void main(String[] args) {
        Clase p = new Clase();
        try {
            for (int i = 1; i <= 5; i++)
                p.ejecutaAlgo();
        } catch (MiExcepcion miex) {
            System.out.println(miex);
            System.out.println(miex.getMessage());
        }
    }
}
```

```
class Clase {  
    public void ejecutaAlgo() throws MiExcepcion {  
        Random aleatorio = new Random();  
        int numero = aleatorio.nextInt(5);  
        System.out.println("el numero generado es "  
+ numero);  
        if (numero == 0)  
            throw new MiExcepcion("se ha generado  
un cero");  
        if (numero == 4)  
            throw new MiExcepcion("se ha generado  
un cuatro");  
    }  
}  
class MiExcepcion extends Exception {  
    private String mensaje;
```

```
    public MiExcepcion(String mensaje) {  
        this.mensaje = mensaje;  
    }  
    public String getMessage() {  
        // TODO Auto-generated method stub  
        return "Se ha producido MiExcepcion: " +  
mensaje;  
    }  
    public String toString() {  
        // TODO Auto-generated method stub  
        return "Se ha producido MiExcepcion: " +  
mensaje;  
    }  
}
```



Telefonica

EDUCACIÓN DIGITAL