

# Orientación a Objetos I - Parte II

## Comunidad de objetos

### Competencias

- Comprender la sobrecarga de métodos.
- Comprender las variables de instancia, locales y de clase.

### Introducción

En este capítulo seguiremos viendo algunas definiciones de la Programación Orientada a Objetos, como por ejemplo la sobrecarga de métodos, el método `toString()`, las variables locales, las variables de instancia y las de clase. Además, veremos cómo se generan cada uno de estos conceptos, el contexto donde actúan y la forma en que podremos imprimir el estado de una instancia.

### Sobrecarga de métodos

Hay ocasiones en que necesitamos que un método reciba una lista de parámetros diferente. Cuando se presentan estos casos, lo que deberíamos hacer es sobrecargar un método, esto permite crear varias versiones (ya que se utiliza el mismo nombre de método, pero se cambian los parámetros). De esta forma podemos mantener el código limpio, ya que podemos usar el mismo nombre (que define exactamente lo que el método hace) como es el caso del constructor estándar, que no recibe parámetros y el constructor con parámetros, el cual es una sobrecarga del estándar.

### La sobrecarga de `toString()`

Hay un método muy útil que nos provee la clase padre y es `toString()`, este es un método que devuelve la representación de una instancia en forma de `String`. De esta forma, podemos imprimirla con el método `System.out.println()`.

## Ejercicio guiado: Vehículo

**Paso 1:** Crearemos una clase `Auto` con sus respectivos parámetros.

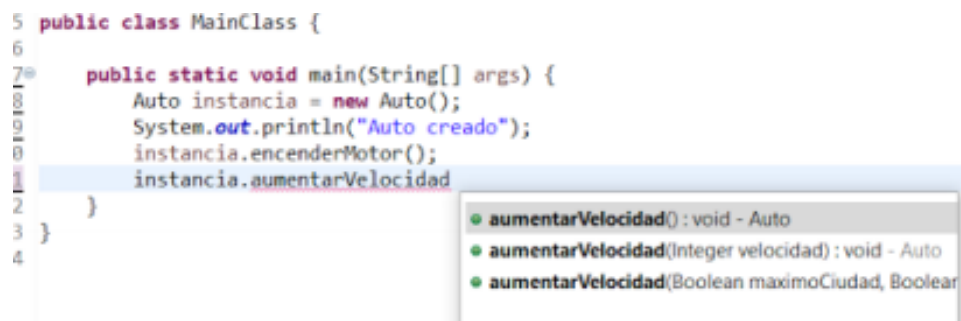
```
public class Auto(){  
    private String marca;  
    private String modelo;  
    private String color;  
    private int velocidadActual;  
    private boolean motorEncendido;  
}
```

**Paso 2:** Luego, crearemos el constructor respectivo.

```
public Auto(){  
}  
    public Auto(String marca, String modelo, String color, int  
velocidadActual, boolean motorEncendido){  
    this.marca = marca;  
    this.modelo = modelo;  
    this.color = color;  
    this.velocidadActual = velocidadActual;  
    this.motorEncendido = motorEncendido;  
}
```

**Paso 3:** Vamos a hacer una sobrecarga del método `aumentarVelocidad` para que, en caso de no recibir la velocidad por parámetro, aumente la velocidad en 10 y otra sobrecarga que reciba dos valores booleanos.

```
public void aumentarVelocidad(int velocidad){
    velocidadActual = velocidadActual + velocidad;
}
public void aumentarVelocidad(){
    velocidadActual = velocidadActual + 10;
}
public void aumentarVelocidad(boolean maximoCiudad, boolean
maximoCarretera){
    if(maximoCiudad) {
        velocidadActual = velocidadActual + 50;
    }
    if(maximoCarretera) {
        velocidadActual = velocidadActual + 100;
    }
}
```



```
5 public class MainClass {
6
7     public static void main(String[] args) {
8         Auto instancia = new Auto();
9         System.out.println("Auto creado");
10        instancia.encenderMotor();
11        instancia.aumentarVelocidad()
12    }
13 }
14
```

- `aumentarVelocidad()` : void - Auto
- `aumentarVelocidad(Integer velocidad)` : void - Auto
- `aumentarVelocidad(Boolean maximoCiudad, Boolean maximoCarretera)` : void - Auto

Imagen 1. Llamar al método "aumentarVelocidad" con sobrecarga.  
Fuente: Desafío Latam

**Paso 4:** Vamos a utilizarlo con la instancia de `Auto` al final del método `main` para probarlo:

```
Auto instanciaAuto = new Auto();
System.out.println("Auto creado");
instanciaAuto.aumentarVelocidad();
System.out.println(instanciaAuto.toString());
-----
Impresión en pantalla:

[Auto creado Modelo.Auto@15db9742]
```

Vemos que la instancia `Auto.toString()` es igual a "Modelo.Auto@15db9742" que es un valor que representa la instancia actual del Auto. Esto es muy difícil de traducir al español, así que vamos a crear una sobrecarga del método `toString` para que nos devuelva otro valor que podamos entender.

**Paso 5:** Al final de la clase Auto, hacemos clic derecho y elegimos la opción Source -> Generate toString(), tal como se muestra en la imagen a continuación.



Imagen 2. Generate toString().

Fuente: Desafío Latam.

En el cuadro de diálogo aparecerán las variables para generar y las seleccionamos todas. Se generará el siguiente código.

```
@Override
public String toString() {
    return "Auto [marca=" + marca + ", modelo=" + modelo + ", color=" + color + ",
    velocidadActual="
    + velocidadActual + ", motorEncendido=" + motorEncendido + "];"
}
```

**Paso 6:** Guardamos los cambios y al dar clic en "Play" se ve que el resultado de `System.out.println(instanciaAuto.toString())` cambió.

```
Auto creado
Auto [marca=null, modelo=null, color=null, velocidadActual=0, motorEncendido=false,
]
```

Se agregó la línea `@Override` por encima de la sobrecarga del método `toString()`. Esto es una anotación de sobrescritura, la diferencia entre esto y la sobrecarga es que este concepto se aplica cuando se quiere reutilizar métodos heredados de superclases.

## Las variables de instancia y las variables locales

Las variables de instancia son aquellas que se mantienen "almacenadas" dentro de las instancias. Las variables locales son aquellas que se declaran dentro de los métodos y, al terminar la ejecución del método, se descartan.

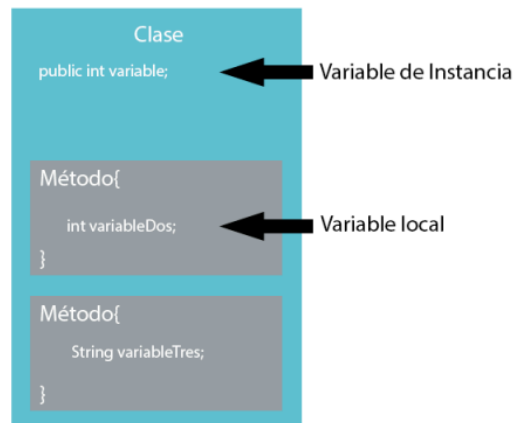


Imagen 3. Variables de instancia y variables locales.

Fuente: Desafío Latam.

En Java existen diferentes contextos o scopes. Como podemos ver, las variables de instancia están en el contexto de la instancia y las variables locales están en el contexto de los métodos.

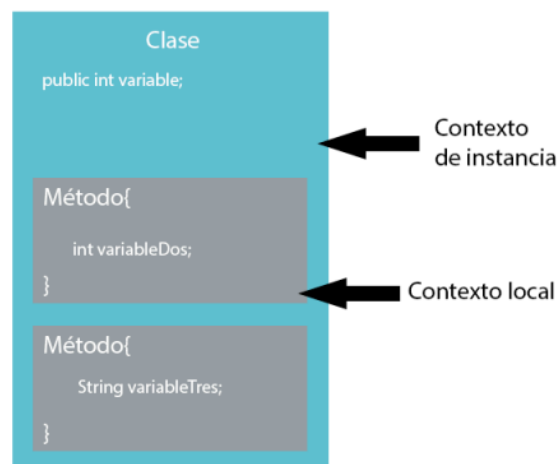


Imagen 4. Variables en el contexto.

Fuente: Desafío Latam.

## Variables de clase

Dentro de la Programación Orientada a Objetos, así como existe un contexto de instancia, existe un contexto de clase (también conocido como estático). Este se caracteriza porque los valores se mantienen estáticos para todas las partes del programa, ya que no necesitan ser instanciados para ser utilizados, basta solamente con llamar al elemento en cuestión utilizando la clase que lo contiene.

Por ejemplo, si creamos una variable de clase dentro de la clase `Auto` y la nombramos `"pruebaEstatica"`, para usarla en otras partes del programa deberíamos llamarla como `"Auto.pruebaEstatica"`.

Esta nos devolvería el valor de la variable en cuestión, asimismo, cuando la llamamos desde otra parte del programa la variable tendrá ese valor, sin necesidad de crear una instancia de `Auto` para utilizar la variable.

Así como se pueden crear variables de clase, se pueden crear métodos de clase, que sirven por ejemplo para hacer cálculos genéricos y, si solamente queremos llamar al método una vez y luego no usaremos más la clase que contiene el método, vamos a tener que crear una instancia solo para eso, lo que hará que el código se vea sucio y se utilice memoria que podría ahorrarse. Lo mejor es crear métodos de clase (también conocidos como estáticos), para no tener que crear instancias de clases que utilizaremos solo para llamar a uno de sus métodos.

Otro ejemplo sería el que entrega Java con sus métodos de clases más útiles:

`Math.random()` ; : El cual genera un número decimal aleatorio entre 0 y 1.

Esto es posible en cualquier parte de la aplicación gracias al modificador de acceso `public` del método, si hubiera otro modificador también debe tenerse en cuenta en el contexto de clase.

## Ejercicio propuesto (1)

Dado el ejercicio guiado “Vehículo”, ahora debemos:

- Incorporar 3 nuevas variables: `patente` del tipo Boolean, `permisoCirculacion` del tipo Boolean y `revisionTecnica` del tipo int para indicar el año.
- Al interior de la misma clase `Auto`, creamos el método `circulacionCiudad` pasándole las variables como parámetros y definiendo lo siguiente:
  1. `revisionTecnica = 2021`.
  2. `permisoCirculacion = false`.
  3. `patente = true`.
- Luego, creamos otro método llamado también `circulacionCiudad` pero sin recibir ningún parámetro dentro de la clase `Auto`, y le decimos que si cumple con las condiciones de que la `revisionTecnica`, la `patente` y el `permisoCirculacion` estén correctas, se puede manejar tranquilamente por la ciudad.
- Imprimimos en pantalla la `instancia.circulacionCiudad` en la clase `Main`.

## Manejo de Excepciones

### Competencias

- Comprender las Excepciones de Java para optimizar nuestras aplicaciones.
- Aplicar Excepciones utilizando Try-Catch y “throw” para controlar y capturar las excepciones.

### Introducción

En este capítulo conoceremos todo sobre las excepciones: que son, para qué nos sirven y cómo manejarlas. En el mundo de la programación manejar las excepciones nos permite crear un sistema robusto y confiable donde la continuidad del software no se verá afectado por errores de programación como errores de los usuarios que lo utilizan. El control de las excepciones nos ayudará a comprender el uso de las sentencias Try-Catch porque las excepciones van de la mano con las palabras claves de Java.



## Excepciones en Java

Cuando hablamos de excepciones en Java (o cualquier lenguaje que las ocupe), hablamos de control de errores en ejecución del programa; esto quiere decir que las excepciones nos permiten controlar errores de usuarios o de programación. Java nos ofrece una cantidad de excepciones para utilizar, listas para su uso.

Las excepciones más utilizadas en programación son las de tipo aritméticas, de validaciones de nulos y de archivos para controlar los errores. Las excepciones tienen jerarquía entre ellas. A continuación mostramos una imagen que ilustra tal orden:

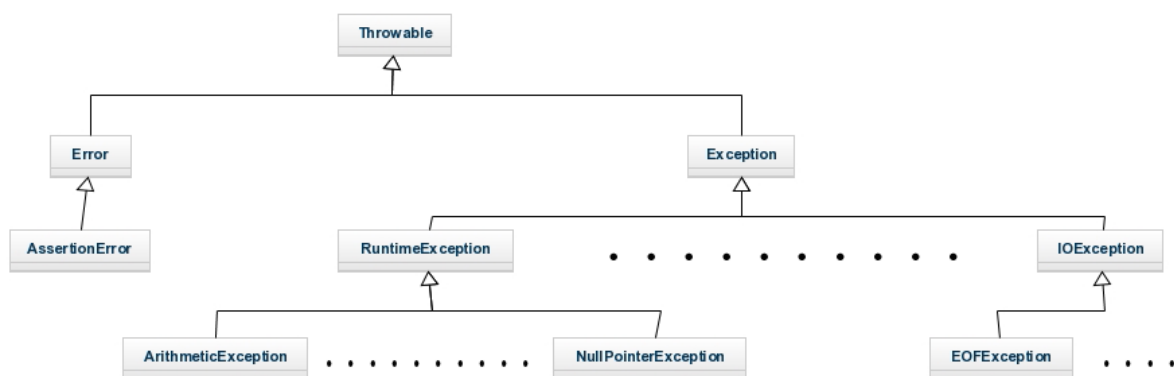


Imagen 5. Jerarquía en las Excepciones  
Fuente: Desafío Latam.

## Throwable

Es una clase base que representa todo lo que Java puede “lanzar”, de hecho la palabra *throwable* se puede considerar en el español como “lanzar o arrojar”. A su vez esta clase:

- Almacena un mensaje (variable de instancia de tipo `String`) que podemos utilizar para detallar qué error se produjo.
- Puede ser una causa, también de tipo `Throwable`, que permite representar el error que causó este error.

Tenemos dos grandes clases que nos permiten controlar errores y excepciones :

## Error

Esta clase nos indica que el error es a nivel de hardware y no aplicativo, uno de los errores más comunes es el error de memoria. Por ejemplo, cuando un proceso toma más del tiempo habitual en ejecutarse y colapsa la memoria del programa.

*Ejemplos: Memoria agotada, error interno de la JVM...*

## Exception

Por otro lado, tenemos todas las excepciones que podemos controlar a nivel de usuario y programación, con este tipo de Exception manejamos la mayor cantidad de errores controlables del software.

Exception y sus subclases indican situaciones que una aplicación debería tratar de forma razonable.

Los dos tipos principales de excepciones son:

- **RuntimeException:** Errores del programador, como una división por cero o el acceso fuera de los límites de un array.
- **IOException:** Errores que no puede evitar el programador, generalmente relacionados con la entrada/salida del programa.

Las excepciones que nos ofrece Java tiene métodos listos para su uso, algunos son:

1. **getMessage():** Este método nos permite mostrar el error y dónde está pasando, esto quiere decir la línea de la clase de Java o el tipo de error.
2. **printTrace():** Este método nos permite mostrar el error con más detalle, sin embargo utilizarlo en todo el código nos provoca un mayor uso de la memoria de la JVM, lo cual es recomendable utilizarlo cuando hay un error en particular que no se logra detectar.

## Excepciones personalizadas

Java nos ofrece su clase `Exception` para su uso, pero también nos ofrece heredar de esta clase para crear nuestras propias clases con nuestros propios métodos. Esto nos permite controlar las excepciones según funcionalidad, evento o lógica en el programa.

```
public class MiExcepcion extends Exception {  
  
    public MiExcepcion(String arg) {  
        super(arg);  
    }  
  
    public String validaNulo(String arg) {  
        String mensaje = "";  
        if(arg == null) {  
            mensaje= "campo nulo";  
        }  
        return mensaje;  
    }  
  
}
```

En este ejemplo tenemos el uso de Herencia y la clase `MiExcepcion` que hereda de `Exception`, sobrescribimos el constructor y creamos un método llamado “`validaNulo`”, el cual valida si el campo es nulo.

## Try-Catch

El uso de estas palabras nos permite relacionarlas con las excepciones ya que su uso siempre va ligado a este concepto.

### Try

Dentro de este bloque escribiremos todo el código que debiese funcionar sin problemas, podemos hacer llamadas a otras clases, validaciones y escribir todo el código que debiese funcionar.

Esta palabra va acompañada con el uso de llaves de apertura y cerrado, y siempre se utiliza con la otra palabra reservada de Java Catch.

### Catch

En esta cláusula es donde lanzamos las posibles excepciones del bloque de código dentro del **try**. Pueden existir tantos **catch** como Exception se quiera controlar, esto quiere decir que para utilizar Catch debemos siempre utilizar el nombre de la Excepción dentro del paréntesis, al lado de la palabra, y, a continuación, abrir y cerrar bloque de código con el uso de llaves como se muestra en la imagen a continuación.

Para ejemplificar un poco, usaremos una división por cero la cual, al ejecutarse, el código podría llegar a fallar. Si se dan las condiciones para que esta porción de código falle, se va a ejecutar otra porción de código (**catch**) para seguir ejecutando el programa y controlar este error.

Ejemplo: División por 0

```
try {  
    int total = 3 / 0;  
}catch(Exception excepcion) {  
    int total = 0;  
}
```

Se puede utilizar más de un catch a la vez, por ejemplo, si sabemos qué excepciones podría arrojar un bloque `try`, podemos agregar un `catch` con cada subclase de `Exception` que consideremos necesaria.

```
try {
    Scanner sc = new Scanner(System.in);
    String variable = sc.nextLine();
    if(variable.isEmpty()){
        variable = null;
    }
    int total = 3 / Integer.parseInt(variable);
} catch (NullPointerException ex1) {
    System.out.println("No se puede dividir por un valor nulo.");
    int total = 0;
} catch (NumberFormatException ex2) {
    System.out.println("El valor de variable no es un número.");
    int total = 0;
} catch (Exception ex3){
    System.out.println("Error inesperado: "+ex3.getMessage());
    int total = 0;
}
```

En este caso, el bloque `catch` a ejecutar dependerá del valor que ingrese el usuario dentro de la variable. Si el usuario no ingresa nada, el valor será `null` y se ejecutará el `catch` con `NullPointerException` y si, por ejemplo, ingresa una letra o un valor no numérico, se arrojará `NumberFormatException` y en cualquier otro caso, se arrojará el `catch` con `Exception` con una subclase de `Exception`.

## La cláusula finally

En ocasiones, nos interesa ejecutar un fragmento de código independientemente de si se produce o no una excepción, si agregamos un **Finally** en el bloque Try-Catch, este se ejecutará siempre, ya que es independiente si se ejecuta el **try** o el **catch**. Se utiliza por ejemplo cuando se debe cerrar un fichero que estemos manipulando o cerrar una conexión de base de datos para liberar recursos.

Ejemplo de una conexión a una fuente de datos usando la cláusula finally:

```
Connection cn = null;
try {
    cn = fuenteDeDatos.getConnection();
} catch (Exception e) {
    System.err.out("Ha ocurrido un error");
} finally {
    cn.close();
}
```

Otro ejemplo con código:

```
public void validaEdad(String arg) {
    String mensaje ="prueba"
    try {
        if((Integer.parseInt(arg)) >=18)
        {
            System.out.println("Edad es mayor a 18 y un número" +
mensaje);
        }
    }
    catch (NumberFormatException e) {
        System.out.println(e.getMessage() + mensaje);
    }
}
```

- Entrará al Catch siempre y cuando el parámetro sea algo distinto de un número.
- El código `Integer.parseInt(arg)` convierte un String a un int.
- Si lo ingresado es algo distinto a número, lanza una excepción de tipo `NumberFormatException`.

Este tipo de control nos permite manejar nuestras aplicaciones con errores controlados y así evitar que el programa se detenga y no se pueda utilizar.

### Importante

Cabe mencionar que cada variable que se escriba dentro de los bloques de llave de apertura y cerrado, solo será vista y podrá utilizarse ahí. Se recomienda que al utilizar variables se declaren antes del bloque try-catch.

### Throw

Las excepciones son errores durante la ejecución de un código y en la mayoría de los casos nos muestran la línea de código donde se originó el problema. Para lanzar un objeto de tipo Exception se necesita utilizar la palabra reservada de Java "Throw", esta nos permite lanzar exception en ejecución y capturar la ejecución de un objeto.

Cuando se lanza una excepción:

1. Se sale inmediatamente del bloque de código actual.
2. Si el bloque tiene asociada una cláusula catch adecuada para el tipo de la excepción generada, se ejecuta el cuerpo de la cláusula catch.
3. Si no, se sale inmediatamente del bloque (o método) dentro del cual está el bloque en el que se produjo la excepción y se busca una cláusula catch apropiada.
4. El proceso continúa hasta llegar al método main de la aplicación. Si ahí tampoco existe una cláusula catch adecuada, la máquina virtual Java finaliza su ejecución con un mensaje de error.

```
public void validaEdad(String arg) {  
    try {  
        if((Integer.parseInt(arg)) >=18)  
        {  
            System.out.println("Edad es mayor a 18");  
        }  
    }  
    catch (NumberFormatException e) {  
        throw new NumberFormatException(e.getMessage());  
    }  
}
```

## Ejercicio Guiado: Validaciones con Try-Catch

Crear un método que nos permita dividir dos parámetros de tipo String y retornar el resultado de la división.

### Explicación

**Paso 1:** Crear la variable local de método resultado.

**Paso 2:** Realizar la conversión de datos de String a Int dentro del bloque try.

**Paso 3:** Utilizar el primer catch con la Exception `NumberFormatException`, esta controlará la excepción cuando uno de los dos parámetros sea algo distinto a un número.

```
java.lang.NumberFormatException: Formato de número incorrecto :For  
input string: "dos"
```

**Paso 4:** Utilizar el segundo Catch con la Exception `ArithmeticException`. Esta controlará la Exception cuando uno de los dos parámetros sea cero, dando la división de una excepción de tipo aritmética, por ejemplo 0/2.

```
java.lang.ArithmeticException: Error en aritmetico : / by zero
```

**Paso 5:** Si no entra a ninguna Exception, retornar el resultado.



```
public static void main (String [] args) {
    division("22","0");
}
public static int division(String valorUno, String valorDos) {
    int resultado = 0;
    try {
        int uno = Integer.parseInt(valorUno);
        int dos = Integer.parseInt(valorDos);
        resultado = uno/dos;
    }
    catch (NumberFormatException e) {
        //se lanza cuando el parámetro sea distinto a una numero
        throw new NumberFormatException("Formato de número
incorrecto :" + e.getMessage());
    }
    catch (ArithmeticException e) {
        // se lanzará cuando el parámetro sea un cero
        throw new ArithmeticException("Error en aritmética :
" +e.getMessage());
    }
    return resultado;
}
```

## Ejercicio Propuesto (2) - Wurlitzer

Crear un programa que permita reproducir una canción según el número de la canción dentro del wurlitzer.

Para esto se entrega un ArrayList con la lista de canciones ya cargadas, se necesita:

- Validar que el número de canción de tipo string sea mayor o igual a 1, si lo es, retornar la canción.
- Controlar todas las Exception posibles.

ArrayList de canciones

```
ArrayList<String> canciones = new ArrayList<String>();  
canciones.add("Yellow ledbetter");  
canciones.add("Echoes");  
canciones.add("Tu Sangre");  
canciones.add("Miño");  
canciones.add("La voz de los 80");  
canciones.add("Mira niñita");
```

## Soluciones ejercicios propuestos

### Solución ejercicio propuesto (1)

**Paso 1:** Creamos las variables nuevas y generamos el constructor y `toString()` respectivo.

```
public class Auto {
    private String marca;
    private String modelo;
    private String color;
    private int velocidadActual;
    private boolean motorEncendido;
    private boolean permisoCirculacion;
    private boolean patente;
    private int revisionTecnica;

    public Auto(String marca, String modelo, String color, int
velocidadActual, boolean motorEncendido,
        boolean permisoCirculacion, boolean patente, int revisionTecnica)
    {
        super();
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.velocidadActual = velocidadActual;
        this.motorEncendido = motorEncendido;
        this.permisoCirculacion = permisoCirculacion;
        this.patente = patente;
        this.revisionTecnica = revisionTecnica;
    }

    @Override
    public String toString() {
        return "Auto [marca=" + marca + ", modelo=" + modelo + ", color=" +
color + ", velocidadActual=" + velocidadActual
            + ", motorEncendido=" + motorEncendido + ",
permisoCirculacion=" + permisoCirculacion + ", patente="
            + patente + ", revisionTecnica=" + revisionTecnica + "];"
    }
}
```

**Paso 2:** Creamos el método `circulacionCiudad` y le pasamos por año el 2021, luego al permiso le colocamos `false` (no lo ha sacado) y a la patente `true` (tiene patente al día). Luego generamos con un `if` todos los requisitos que debe cumplir para circular por la calle.

```
public void circulacionCiudad(int revisionTecnica) {  
    revisionTecnica = 2021;  
}  
public void circulacionCiudad(boolean permisoCirculacion ,boolean  
patente) {  
    permisoCirculacion = false;  
    patente = true;  
}  
  
public void circulacionCiudad() {  
    if(revisionTecnica >= 2020 && permisoCirculacion == true && patente  
== true) {  
        System.out.println("El auto tiene derecho a circular");  
    } else {  
        System.out.println("El auto no tiene derecho a circular");  
    }  
}  
}
```

**Paso 3:** Imprimimos la instancia de `circulacionCiudad` lo cual nos debería dar por resultado lo siguiente:

```
public class Main {  
  
    public static void main(String[] args) {  
        Auto instancia = new Auto(null, null, null, 0, false, false, false,  
0);  
        System.out.println("Auto Creado");  
        System.out.println(instancia.toString());  
        instancia.circulacionCiudad();  
    }  
}  
-----  
Impresión en pantalla:  
["El auto no tiene derecho a circular"]
```

## Solución ejercicio propuesto (2)

```
public static void main (String [] args) {
    reproductor("1");
}
public static String reproductor(String numero) {
    ArrayList<String> canciones = new ArrayList<String>();
    canciones.add("Yellow ledbetter");
    canciones.add("Echoes");
    canciones.add("Tu Sangre");
    canciones.add("Miño");
    canciones.add("La voz de los 80");
    canciones.add("Mira niñita");
    String cancion = "";
    try {
        int indiceCancion = Integer.parseInt(numero);
        if(indiceCancion >=1) {
            cancion = canciones.get(indiceCancion);
        }
    }
    catch (IndexOutOfBoundsException e) {
        // se lanzara cuando el indice de la cancion es mayor
al tamaño
        // del arreglo
        throw new IndexOutOfBoundsException("Indice fuera de
rango " + e.getMessage());
    }
    catch (NumberFormatException e) {
        // se lanza cuando el parámetro sea distinto a una
numero
        throw new NumberFormatException("Formato de número
incorrecto :" + e.getMessage());
    }
    return cancion;
}
```

- Declarar el método con el parámetro de entrada.
- Declarar una variable local llamada canción.
- Pasamos el parámetro de String a int.
  - Si es algo distinto del número lanza:

```
java.lang.NumberFormatException: Formato de número incorrecto :For  
input string: "prueba"
```

- Validar que sea mayor o igual a 1.
- Retornar la canción con el método get y el número ingresado por parámetro, le asignamos la canción del índice.
  - Si el índice es mayor al índice del ArrayList lanza:

```
java.lang.NumberFormatException: Error en la operacion Index: 17, Size: 6
```