
# Java Academy Xideral 2022

## Alumno: Moisés Vidal Hernández

## Maestro: Miguel Angel Rugerio

## Java Bootcamp Training

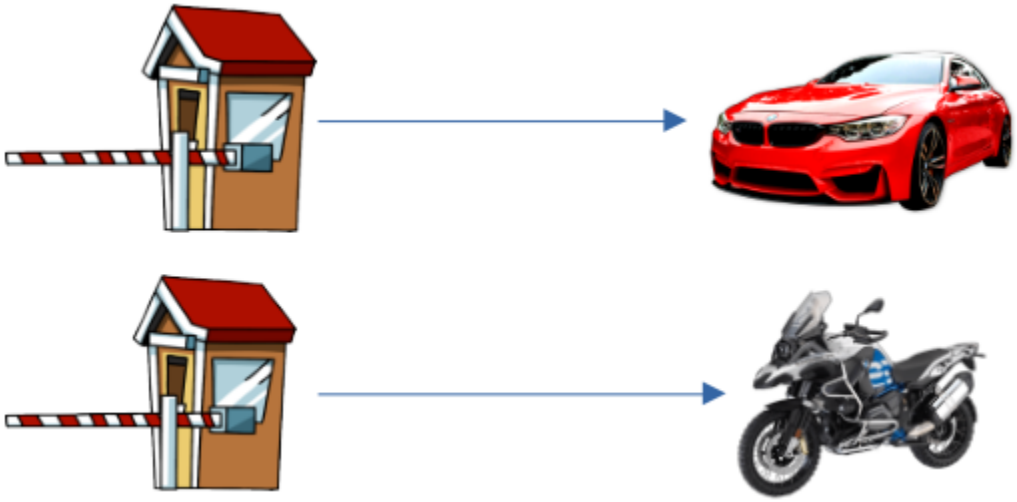
## Examen: Semana 2

## Diciembre 2 del 2022

## 1. Explica que es la inyección de dependencias

### Diagrama

Partiendo de la manera tradicional, la forma en como se hacía era instanciando la clase directamente para crear un objeto específico, justo como se muestra a continuación.

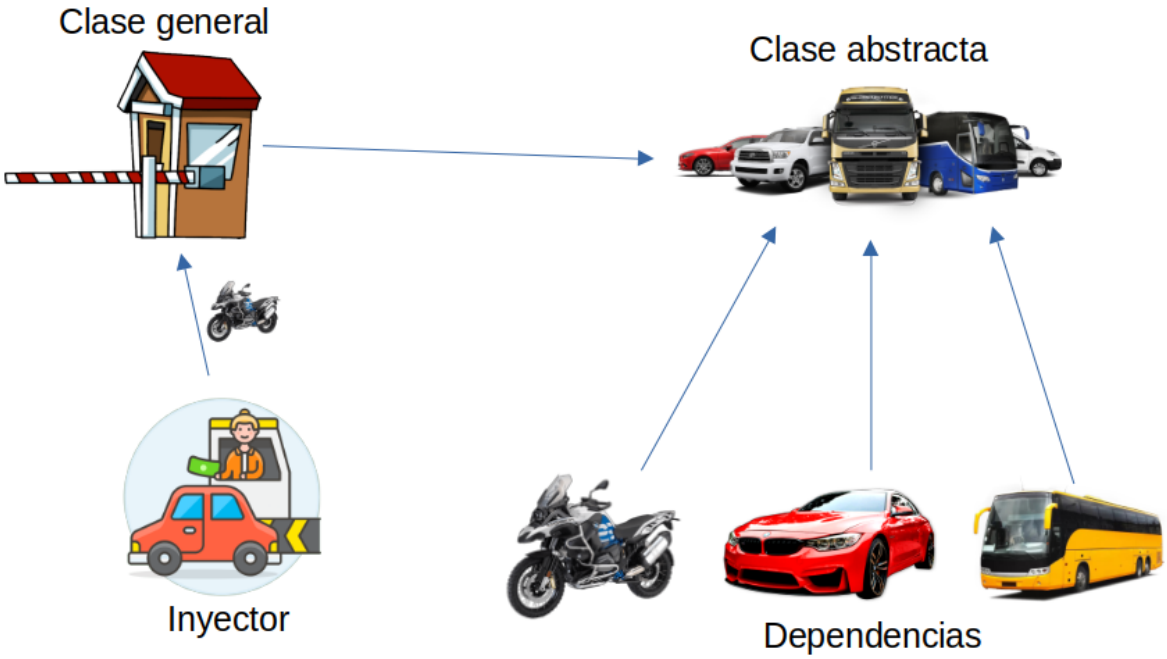


De esta manera podemos interpretar que para nuestra clase caseta, teníamos que crear dentro un objeto vehículo del tipo automóvil y motocicleta para así usarlos o manejar su información.

Esto no tiene nada de malo, sin embargo esta forma muestra que se está empleando un alto grado de acoplamiento, lo cual puede suponer un problema cuando se decidan agregar más vehículos como autobuses, camiones de carga o trailers, ya que nos veríamos obligados a crear más instancias de clase vehículo especificando el tipo de vehículo.

Para evitar este tipo de problemas en futuras actualizaciones o escalabilidad del proyecto, se decide usar el patrón de inyección de dependencias.
Ahora bien con el patrón de inyección de dependencias podemos simplificar todo esto y usar un nivel de abstracción avanzado, que nos ayude a reducir las instancias dentro de nuestra clase, con ayuda de una clase alterna que tiene el nombre de inyector, está clase asigna el tipo de objeto (en este caso un tipo de vehículo), según conforme se requiera y se solicite.

Para ser más claros a continuación se ejemplifica lo antes mencionado en el siguiente diagrama.




En base al diagrama antes presentado, tenemos una serie de elementos que componen el patrón de inyección de dependencias, donde:

### La clase general (clase Caseta)

- Es la clase donde se mostrará el tipo de vehículo, sus características (marca, modelo y color), así como la tarifa de peaje que se cobrará para ese vehículo.
- Es importante mencionar que dentro de esta clase no se define el tipo de vehículo.

### Clase abstracta

- Esta clase es una especie de modelo o molde, donde a partir de este se determinara el tipo de vehículo que pasara por la caseta de peaje.
- En esta clase tendremos una característica que todos los vehículos comparten, la cual es el número de ejes.

### Dependencias

- Estas son clases, que aplican herencia desde la clase abstracta, donde tomarán todos sus atributos y métodos abstractos, los cuales realizarán una sobreescritura (“Override”), para definir un comportamiento diferente según corresponda el tipo de vehiculo.
- Estas clases se definirán dentro de la clase alterna que sirve como inyector y así la clase general pueda ejecutar una acción específica,.

### Inyector

- El inyector es esa clase alterna o auxiliar que nos permite ir asignando el tipo de dependencia o clase específica (en este caso algún tipo vehículo), para que nuestra clase general pueda ejecutar una acción específica como lo es mostrar la información del vehículo y mostrar la tarifa a pagar.
- En este caso el inyector vendría siendo la persona o el escaner que cuenta el numero de ejes del vehículo.

## Código

Como lo explicamos anteriormente, tenemos una clase que sirve como modelo o molde para definir las dependencias (o clases hijas), esta clase lleva por nombre vehículo, la cual tiene un único atributo que comparte con todos los tipos de vehículos terrestres, el cual es el número de ejes, el cual para motivos de nuestro ejercicio puede ser un número entre 1 y 5.

```java
package com.inyecciondd.optimizado;

public abstract class Vehiculo {

    private int ejes;

    public Vehiculo(int ejes) {
        this.ejes = ejes;
    }

    public abstract void obtenerCuota();

    public int getEjes() {
        return ejes;
    }

    public void setEjes(int ejes) {
        this.ejes = ejes;
    }
}
```

Después tenemos las clases que aplican herencia desde la clase abstracta que lleva por nombre vehículo, estas clases podríamos decir que toman el rol de las dependencias y cada una de estas clases representa cada tipo de vehículo que pasará por la caseta de peaje.

Principal.java | *Autobus.java

```java
package com.inyecciondd.optimizado;

public class Autobus extends Vehiculo{

    public Autobus(int ejes) {
        super(ejes);
    }

    @Override
    public void obtenerCuota() {
        System.out.println("Tipo: Autobus\nEjes: " + super.getEjes() + "\nPaga: $246 pesos");
    }
}
```

*Automovil.java

```java
package com.inyecciondd.optimizado;

public class Automovil extends Vehiculo {

    public Automovil(int ejes) {
        super(ejes);
    }

    @Override
    public void obtenerCuota() {
        System.out.println("Tipo: Automovil\nEjes: " + super.getEjes() + "\nPaga: $89 pesos");
    }
}
```

Motocicleta.java

```java
package com.inyecciondd.optimizado;

public class Motocicleta extends Vehiculo {

    public Motocicleta(int ejes) {
        super(ejes);
    }

    @Override
    public void obtenerCuota() {
        System.out.println("Tipo: Motocicleta\nPaga: $44 pesos");
    }

}
```

Posteriormente tenemos nuestra clase Caseta Peaje, la cual asume el rol de clase general, en donde esta misma tiene el papel de servir como contenedor de la información general de cada vehículo, así mismo dentro de esta clase existe el método cobrarPeaje(), que a su vez ejecuta el método obtenerCuota de cada clase que define el tipo de vehículo.

El método antes mencionado, nos mostrará en consola el precio del peaje de acuerdo al tipo del vehículo y sus características.

Principal.java  AutoBus.java  CasetaPeaje.java

```java
package com.inyecciondd.optimizado;

public class CasetaPeaje {

    private int noRegistro;
    private String marca;
    private String modelo;
    private String color;

    private Vehiculo vh;

    public CasetaPeaje(int noRegistro, String marca, String modelo, String color) {
        super();
        this.noRegistro = noRegistro;
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public String getModelo() {
        return modelo;
    }

    public void setModelo(String modelo) {
        this.modelo = modelo;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public void cobrarPeaje() {
        System.out.println(
                "\nRegistro " + noRegistro + "\nMarca: " + marca + "\nModelo: " + modelo + "\nColor: " + color);
        vh.obtenerCuota();
    }

    public Vehiculo getVehiculo() {
        return vh;
    }
```

Enseguida tenemos nuestra clase alterna que sirve como Inyector, ya que en esta clase por medio del método inyectarAuto() y con ayuda del número de ejes que se define en cada instancia de clase se define el tipo de vehículo a insertar o inyectar.

Principal.java | Autobus.java | CasetaPeaje.java | Inyector.java

```java
package com.inyeccionDD.optimizado;

public class Inyector {

    static Vehiculo moto = new Motocicleta(1);
    static Vehiculo bus = new Autobus(3);
    static Vehiculo auto = new Automovil(2);
    static Vehiculo camion = new Camion(5);

    static void inyectarAuto(CasetaPeaje cp, int ejes) {
        switch (ejes) {
        case 1:
            cp.setVehiculo(moto);
            break;
        case 2:
            cp.setVehiculo(auto);
            break;
        case 3:
            cp.setVehiculo(bus);
            break;
        default:
            cp.setVehiculo(camion);
            break;
        }
    }
}
```

Por último en nuestra clase principal, creamos nuestros objetos que por su nivel de abstracción y que todas heredan de la clase abstracta Vehículo, creamos su variable apuntando específicamente a un tipo de objeto, en este caso a cada tipo de vehiculo que pasara por la caseta.

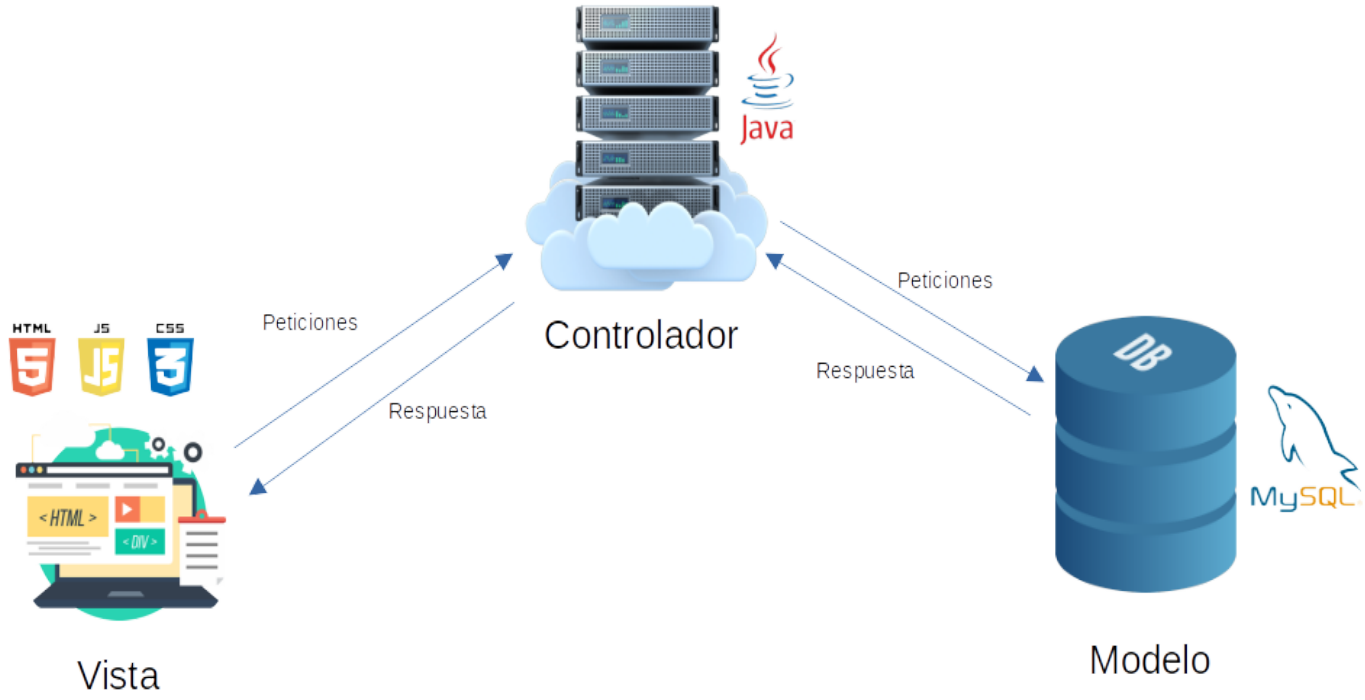
Principal.java | Autobus.java | CasetaPeaje.java | Inyector.java

```java
package com.inyecciondd.optimizado;

public class Principal {

    public static void main(String[] args) {

        CasetaPeaje c1 = new CasetaPeaje(1, "Yamaha", "Gixxer-250", "Azul");

        Inyector.inyectarAuto(c1, 1);

        c1.cobrarPeaje();

        CasetaPeaje c2 = new CasetaPeaje(2, "Volskwagen", "Jetta Clasico", "Blanco");

        Inyector.inyectarAuto(c2, 2);

        c2.cobrarPeaje();

        CasetaPeaje c3 = new CasetaPeaje(3, "Man", "Mand-113", "Rojo");

        Inyector.inyectarAuto(c3, 4);

        c3.cobrarPeaje();

        CasetaPeaje c4 = new CasetaPeaje(4, "Mercedez Bens", "Bus-line-2312", "Gris");

        Inyector.inyectarAuto(c4, 3);

        c4.cobrarPeaje();

    }

}
```

Finalmente nos produce un resultado como el siguiente

```
Registro 1
Marca: Yamaha
Modelo: Gixxer-250
Color: Azul
Tipo: Motocicleta
Paga: $44 pesos

Registro 2
Marca: Volskwagen
Modelo: Jetta Clasico
Color: Blanco
Tipo: Automovil
Ejes: 2
Paga: $89 pesos

Registro 3
Marca: Man
Modelo: Mand-113
Color: Rojo
Tipo: Camiom
Ejes: 5
Paga: $308 pesos

Registro 4
Marca: Mercedez Bens
Modelo: Bus-line-2312
Color: Gris
Tipo: Autobus
Ejes: 3
Paga: $246 pesos
```

## 2. Diagrama y explica una arquitectura web usando el patrón MVC




### Vista

- La vista es el lugar en el cual se muestra la información que el usuario desea consultar, en este caso específico se trata de una aplicación web a la cual se puede acceder mediante un navegador.
- En la mayoría de las aplicaciones web se suele presentar el contenido mediante archivos escritos en lenguaje de marcado (HTML), hojas de estilo (CSS) y algunas funciones o validaciones (Javascript / JS), los cuales tiene como objetivo mostrar una tabla, párrafos, listas, formularios, botones, etc. Al usuario final y que este interactúe con la información que se presenta o solicitar nueva.
- La vista solo puede interactuar con el controlador mediante peticiones HTTP, y que dentro de las más comunes se encuentran:
    - **GET**: Obtener información.
    - **POST**: Crear nueva información.
    - **PUT**: Actualizar o modificar información ya existente.
    - **DELETE**: Eliminar información.
- Una vez procesadas estas peticiones el controlador ejecuta ciertas acciones que permitan obtener la información que es solicitada desde la vista y de este modo retornar una respuesta..

## Controlador

- El controlador es aquel lugar en el cual se encuentra alojada toda la lógica del negocio, que no es otra cosa más que el código fuente que recibe las peticiones de la vista y a su vez hace peticiones al modelo.
- El controlador es a lo que llamamos “la nube”, que son aquellas computadoras gigantes que tienen el nombre de “servidores”, los cuales se encuentran en alguna ubicación geográfica de nuestro planeta.
- El controlador almacena toda la lógica o código hecho en un lenguaje de programación como Java, PHP, Python o JavaScript (NodeJS).
- El controlador es el intermediario entre la vista y el modelo, ya que el código que se encuentra dentro del controlador se encarga de procesar las peticiones hechas desde la vista, y hace nuevas peticiones hacia el modelo.
- El controlador al recibir las peticiones HTTP y procesarlas realiza nuevas peticiones al modelo pero de tal modo que el modelo pueda reconocerlas y le sea fácil retornar una respuesta, para el caso de la arquitectura de esta aplicación web usamos MySQL, por lo cual tendremos que pre procesar las peticiones que en este caso serían consultas y para atender las peticiones antes mencionadas serían las siguientes.
    - **SELECT * FROM alumnos**: para obtener los registros de una tabla con datos de alumnos.
    - **INSERT INTO alumnos (matricula ,nombre, edad, grado, grupo) VALUES 1, “Alex”, “21”, 1, “A”**: Para insertar nueva información o un registro a nuestro modelo.
    - **UPDATE alumnos SET name = “Alexander” WHERE matricula = 1**: para actualizar un registro, en este caso el nombre del alumno con la matrícula 1.
    - **DELETE alumnos WHERE matricula = 1**: Eliminar el registro correspondiente al alumno con la matrícula 1.
- Internamente el controlador al obtener una respuesta positiva con toda la información que se retorna desde el modelo, se suele enviar esta información a la vista mediante objetos en notación JSON, para que las tecnologías del frontend (HTML, CSS & JS) la procesen y la presenten al usuario final.
- En caso de que el modelo no retorna una respuesta válida o surja un error, el controlador válida y retorna un mensaje de error o advertencia a la vista.

## Modelo

- El modelo básicamente es donde se almacena toda nuestra información, que en este caso es nuestra base de datos.
- Para el caso de la arquitectura que presentamos, usamos como motor de base de datos a MySQL, por lo cual emplea el lenguaje de consultas SQL (Structure Query Language), entonces nuestra base de datos estará conformada por tablas y relaciones que las conectan entre sí.
- El modelo solamente interactúa con el controlador, este último le manda las peticiones (consultas) pre procesadas en lenguaje SQL, el modelo las recibe y las procesa,

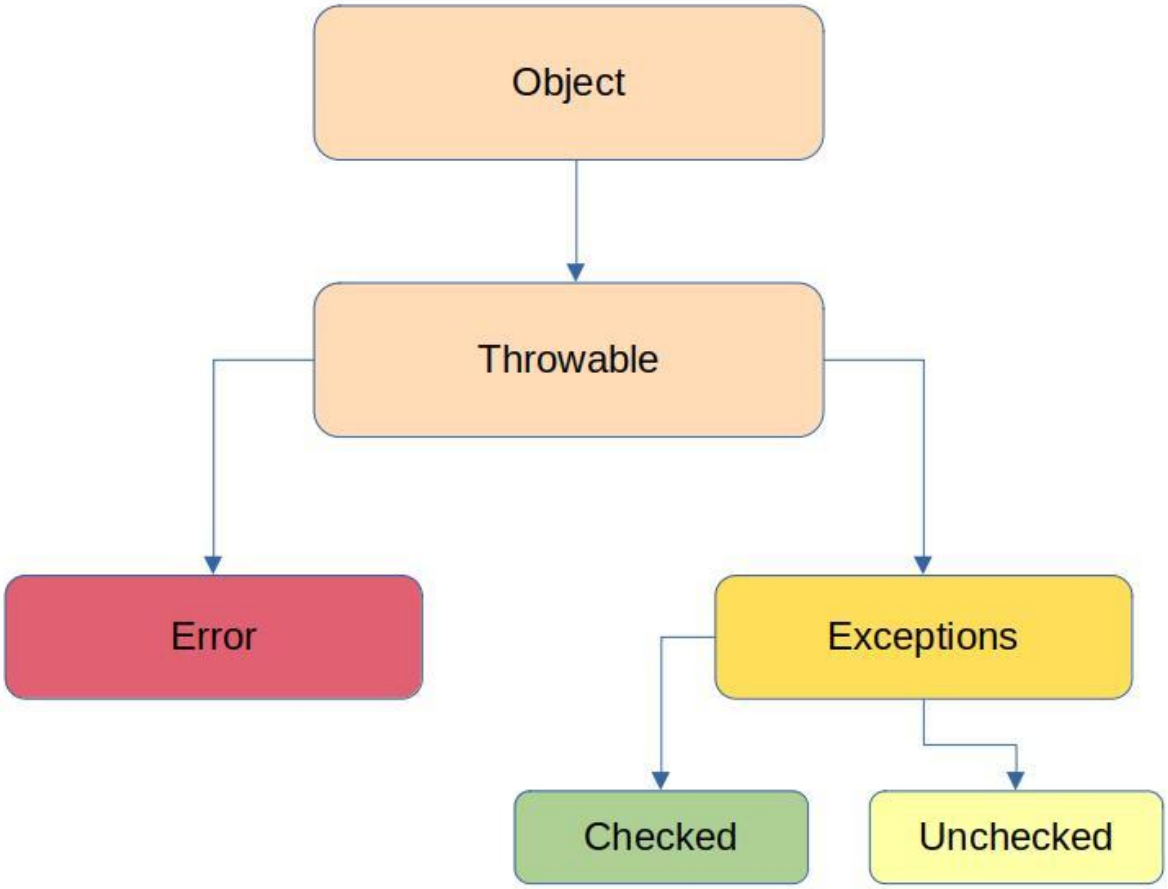
evalúa si estas peticiones son válidas y en el mejor de los casos retorna la información solicitada mediante las peticiones, caso contrario no retorna nada, alguna excepción o error (para el caso de las consultas que requiera insertar o actualizar información), para cualquiera de estos casos el controlador ya tiene una lógica y entonces tendrá claro que mandar a la vista.

## 3. Explica los diferentes tipos de excepciones que existen java

En Java existen 3 tipos de excepciones las cuales son:

- Error
- Runtime exception (ó unchecked)
- Checked exception.

La forma en que se organizan se representa en el siguiente diagrama.




Como podemos apreciar, las excepciones surgen a partir de un object y son lanzadas o creadas por medio de Throwable, de esta forma al ver cómo se organizan las excepciones, podemos describirlas a continuación.

### Error

Estas son aquellas excepciones de las cuales es muy difícil recuperar el estado o el comportamiento del programa, los errores son ocasionados por cuestiones ajenas al programador.

Algunos de los factores que pueden ocasionar excepciones de tipo error se deben a:

- Problemas con la conexión a la red
- No se tiene una conexión a una base de datos
- Se trata de abrir un archivo donde el dispositivo que lo almacena está dañado.

### Excepciones checked

Son aquellas excepciones que se pueden detectar al momento de compilar el programa, e incluso cuando se escribe el código, el propio IDE (si es que estamos usando uno), nos sugeriría realizar el manejo o tratamiento de dicha excepción.

El tratamiento de estas excepciones se puede hacer de 2 formas

- La primera y la más recomendable para evitar el desbordamiento de nuestro programa es, usar el contenedor try y que envuelva toda la lógica e instrucciones del bloque de código, posterior a esto usar el contenedor catch el cual captura la excepción y se le puede dar un tratamiento específico en este caso imprimir la excepción, de tal forma que nuestro código quedará justo como se muestra a continuación.

```
public static FileReader abrirArchivo() {
    FileReader fileR = null;
    try {
        fileR = new FileReader("C:\\Prueba\\archivo.txt");
    } catch (FileNotFoundException e) {
        System.out.println(e.toString());
    }
    return fileR;
}
```

- La segunda forma es mandar la excepción en la línea donde se declara el método con ayuda de la palabra reservada throws, de este modo el método no realizará el manejo de la excepción si no que le pasará la excepción hacia la función o método que la llame.

```
public static FileReader abrirArchivo() throws FileNotFoundException {
    FileReader fileR = null;
    fileR = new FileReader("C:\\Prueba\\archivo.txt");
    return fileR;
}
```

## Excepciones Unchecked

También conocidas como runtime exceptions, son aquellas excepciones que no se detectan al compilar el programa, si no que estas tienen presencia al momento de ejecutarlo.

- Si estamos usando algún IDE, este no reconocerá ningún tipo de error ni tampoco el compilador, pero cuando la JVM ejecute el programa, la excepción se mostrará en consola.A continuación mostramos un ejemplo.

```
public static void main(String[] args) {
    int a = 5;
    int b = 10;
    double res = 0;

    while(a >= 0) {
        res = b / a;
        System.out.println("b: "+ b + "a: "+ a + " = "+ res);
        a--;
    }
}
```

- Hasta aquí podríamos decir que todo está normal, ya que no tenemos ningún error, pero ahora si lo ejecutamos pasa esto.

```
<terminated> Excepciones (2) [Java Application] /home/mvidal/.p2/pool/plugins/org.eclipse.justj.openjdk.hotsp
b: 10a: 5 = 2.0
b: 10a: 4 = 2.0
b: 10a: 3 = 3.0
b: 10a: 2 = 5.0
b: 10a: 1 = 10.0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at com.excepciones.v3.Excepciones.main(Excepciones.java:15)
```

- Podríamos decir que esta excepción es silenciosa, ya que en el código y en la compilación no tuvimos ningún error, pero al momento de ejecutarlo si tuvimos un problema.
- Ante esto podemos tratar esta excepción justo como lo describimos anteriormente en las excepciones checked con la ayuda de try catch o mandando la excepción en la declaración del método.
- Para este caso usaremos el try catch, ya que lo considero una buena práctica, entonces si surge una excepción le damos un tratamiento donde indicamos que se encontró una excepción y que esta nos muestre de qué tipo es.

```java
    public static void main(String[] args) {
        int a = 5;
        int b = 10;
        double res = 0;
        try {
            while (a >= 0) {
                res = b / a;
                System.out.println("b: " + b + " a: " + a + " = " + res);
                a--;
            }
        } catch (Exception e) {
            System.out.println("Excepcion encontrada: "+ e.toString());
        }
    }
```

Problems Javadoc Declaration Console ×

<terminated> Excepciones (2) [Java Application] /home/mvidal/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x8

```
b: 10 a: 5 = 2.0
b: 10 a: 4 = 2.0
b: 10 a: 3 = 3.0
b: 10 a: 2 = 5.0
b: 10 a: 1 = 10.0
Excepcion encontrada: java.lang.ArithmeticException: / by zero
```

## 4. Exponer un código aplicando excepciones con multicatch y try with resources.

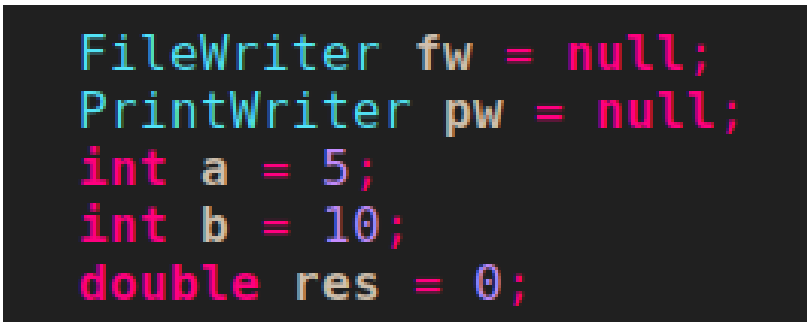
### Con multicatch

Partiendo con el primer ejemplo, tenemos un método main que tendrá algunas variables de referencia a archivos y algunos primitivos, el objetivo general de este programa es abrir o escribir dentro de un archivo, el resultado de una división entre 2 primitivos.

```java
public class Excepciones {

    public static void main(String[] args) {

        FileWriter fw = null;
        PrintWriter pw = null;
        int a = 5;
        int b = 10;
        double res = 0;

        try {
            fw = traerArchivo("C\\tatbajo\\prueba.txt");
            pw = new PrintWriter(fw);
            for (int i = 0; i < 6; i++) {
                res = b / a;
                System.out.println(a);
                pw.println("Resultado " + i + ": " + res);
                a--;
            }
        } catch (RutaInvalida | IOException | NullPointerException | ArithmeticException e) {
            System.out.println("Excepción encontrada: " + e.toString());
        } finally {
            if (fw != null) {
                try {
                    fw.close();
                } catch (Exception e2) {
                    System.out.println("Excepción al cerrar el archivo: " + e2.toString());
                }
            }
        }
    }

    public static FileWriter traerArchivo(String ruta) throws RutaInvalida, IOException {
        if (ruta.equals("")) {
            throw new RutaInvalida("La ruta esta vacia");
        } else if (ruta.contains("@") || ruta.contains("$") || ruta.contains("}") || ruta.contains("{")) {
            throw new RutaInvalida("La ruta no debe tener caracteres especiales");
        } else {
            // return new FileWriter(ruta);
            return null;
        }
    }
}
```

A continuación explicaremos cada parte del código y que es lo que estaría pasando en cada caso.

- Como lo mencionamos anteriormente tenemos esta parte donde declaramos la instancia a objetos de archivos y algunas variables de tipo primitivo.



- Posteriormente tenemos el inicio del bloque try, en donde indicamos que necesitamos abrir o crear un archivo mediante el método traer Archivo() el cual nos retorna un valor, seguido de esto inicializamos la escritura dentro del archivo que nos retorne el método antes mencionado.

```
try {
    fw = traerArchivo("C\\tatbajo\\prueba.txt");
    pw = new PrintWriter(fw);
```

- Aquí debemos hacer un paréntesis, ya que explicaremos el método traerArchivo(),en este mismo definimos que retornaremos un archivo, ya sea que lo creamos de cero o lo abrieramos de acuerdo con la ruta proporcionada, si la ruta está vacía o no existe arrojaremos una excepción, si la ruta contiene algún carácter especial también arrojaremos una nueva excepción, por último si no caemos en alguna de estas condiciones retornaremos el archivo o también podemos retornar null (esto lo haremos manualmente con fines demostrativos de la práctica).

```
public static FileWriter traerArchivo(String ruta) throws RutaInvalida, IOException {
    if (ruta.equals("") || ruta == null) {
        throw new RutaInvalida("La ruta esta vacia");
    } else if (ruta.contains("@") || ruta.contains("$") || ruta.contains("}") || ruta.contains("{")) {
        throw new RutaInvalida("La ruta no debe tener caracteres especiales");
    } else {
        // return new FileWriter(ruta);
        return null;
    }
}
```

- Después tenemos un ciclo for que inicia en 0 y llega hasta 5, dentro de este ciclo dentro de cada iteración se realiza la división entre 2 primitivos, inicialmente el primitivo a tiene un valor de 5 y el primitivo b tiene un valor de 10, seguido se imprime el resultado, después se escribe este resultado dentro del archivo que hemos abierto o creado y por último se hace un decremento al primitivo a.

```
for (int i = 0; i < 6; i++) {
    res = b / a;
    System.out.println(a);
    pw.println("Resultado " + i + ": " + res);
    a--;
}
```

- En seguida tenemos la sección del catch, donde realizaremos el manejo de excepciones con ayuda del multicatch, ya que varias excepciones pueden surgir en este programa, pero a la primera excepción encontrada el programa nos lo indicará.

```
} catch (RutaInvalida | IOException | NullPointerException | ArithmeticException e) {
    System.out.println("Excepción encontrada: " + e.toString());
```

- Por último tenemos la sección del finally, donde independientemente que surja una excepción o no, se ejecutará.

- Dentro del finally preguntamos si es que existe un archivo (ya sea que lo creamos o lo abrimos), si estos es afirmativo indicamos que se cierre el archivo, en caso contrario no se realiza nada.

```
} finally {
    if (fw != null) {
        try {
            fw.close();
        } catch (Exception e2) {
            System.out.println("Excepción al cerrar el archivo: " + e2.toString());
        }
    }
}
```

- Si ejecutamos este programa con indicando al método traerArchivo() que retorne null, obtendremos la siguiente salida que nos indica que no existe el objeto y por lo cual arroja una excepción encontrada de tipo NullPointerException.

```
43
44    public static FileWriter traerArchivo(String ruta) throws RutaInvalida, IOException {
45        if (ruta.equals("") || ruta == null) {
46            throw new RutaInvalida("La ruta esta vacia");
47        } else if (ruta.contains("@") || ruta.contains("$") || ruta.contains("}") || ruta.contains("{")) {
48            throw new RutaInvalida("La ruta no debe tener caracteres especiales");
49        } else {
50            // return new FileWriter(ruta);
51            return null;
52        }
53    }
54 }
55
```

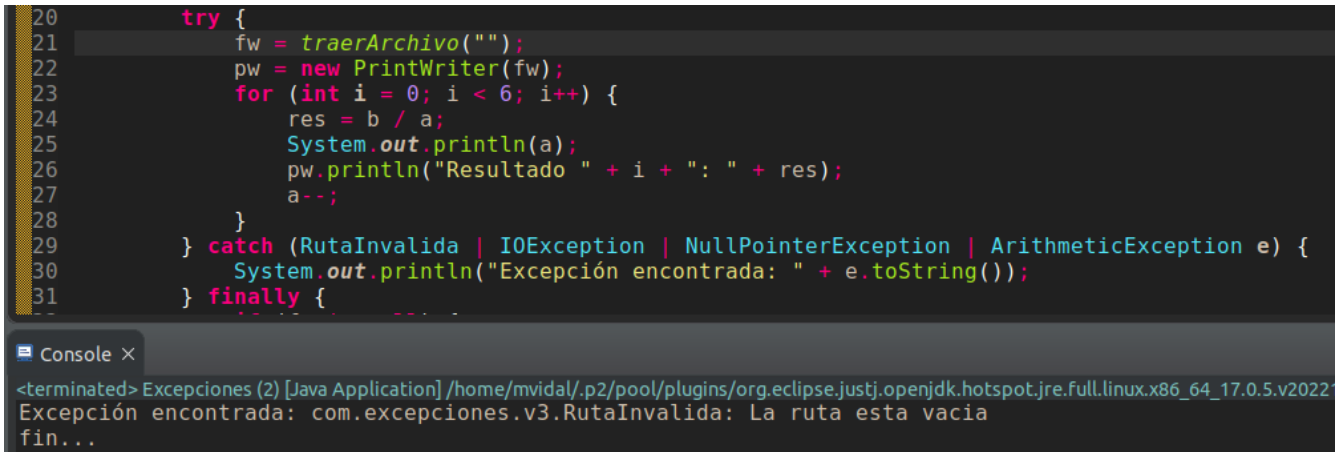
```
Console ×
<terminated> Excepciones (2) [Java Application] /home/mvidal/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_17.0.5.v20221102-0933/jre/bi
Excepción encontrada: java.lang.NullPointerException
fin...
```

- Si ejecutamos este programa con indicando al método traerArchivo() que retorne el archivo abierto o creado obtenemos la siguiente salida, que nos arroja una excepción de tipo ArithmeticException, ya que el archivo se creó pero la operación fue inválida dado que no se puede dividir un número entre 0 .

```
44    public static FileWriter traerArchivo(String ruta) throws RutaInvalida, IOException {
45        if (ruta.equals("") || ruta == null) {
46            throw new RutaInvalida("La ruta esta vacia");
47        } else if (ruta.contains("@") || ruta.contains("$") || ruta.contains("}") || ruta.contains("{")) {
48            throw new RutaInvalida("La ruta no debe tener caracteres especiales");
49        } else {
50            return new FileWriter(ruta);
51            //return null;
52        }
53    }
54 }
55
```

```
Console ×
<terminated> Excepciones (2) [Java Application] /home/mvidal/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_17.0.5.v20221102-0933/jre/bin/
5
4
3
2
1
Excepción encontrada: java.lang.ArithmeticException: / by zero
fin...
```

- Ahora si ejecutamos el programa mandando una ruta vacía, la salida retorna una excepción de tipo Ruta vacía.



```
20          try {
21              fw = traerArchivo("");
22              pw = new PrintWriter(fw);
23              for (int i = 0; i < 6; i++) {
24                  res = b / a;
25                  System.out.println(a);
26                  pw.println("Resultado " + i + ": " + res);
27                  a--;
28              }
29          } catch (RutaInvalida | IOException | NullPointerException | ArithmeticException e) {
30              System.out.println("Excepción encontrada: " + e.toString());
31          } finally {
```

```
Console ×
<terminated> Excepciones (2) [Java Application] /home/mvidal/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_17.0.5.v2022
Excepción encontrada: com.excepciones.v3.RutaInvalida: La ruta esta vacia
fin...
```

- Ahora si ejecutamos el programa, mandando un carácter especial en la ruta, la salida nos indicará que se encontró una excepción diciendo que no se permiten caracteres especiales en la ruta.

```
20          try {
21              fw = traerArchivo("C\\ta@bajo\\prueba.txt");
22              pw = new PrintWriter(fw);
23              for (int i = 0; i < 6; i++) {
24                  res = b / a;
25                  System.out.println(a);
26                  pw.println("Resultado " + i + ": " + res);
27                  a--;
28              }
29          } catch (RutaInvalida | IOException | NullPointerException | ArithmeticException e) {
30              System.out.println("Excepción encontrada: " + e.toString());
31          } finally {
32              if (fw != null) {
33                  try {
```

```
Console ×
<terminated> Excepciones (2) [Java Application] /home/mvidal/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_17.0.5.v2022
Excepción encontrada: com.excepciones.v3.RutaInvalida: La ruta no debe tener caracteres especiales
fin...
```

### Con try with resources

En este caso, el objetivo del programa es el mismo, solo que al inicio del try se agrega el tipo de recurso que estaremos utilizando, al hacer esto cuando el programa finalice el archivo se cerrará de manera automática, por lo cual ya no tendremos la necesidad de agregar un finally al final del try catch.

También es importante mencionar que estamos empleando el multi catching en este ejemplo.

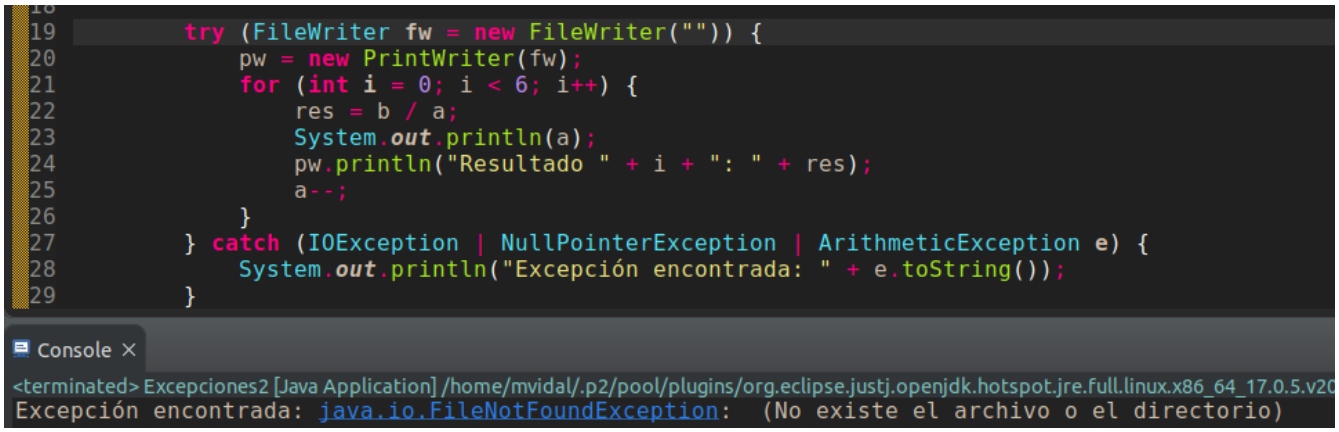
```
public class Excepciones2 {

    public static void main(String[] args) {

        PrintWriter pw = null;
        int a = 5;
        int b = 10;
        double res = 0;

        try (FileWriter fw = new FileWriter("C\\trabajo\\prueba.txt")) {
            pw = new PrintWriter(fw);
            for (int i = 0; i < 6; i++) {
                res = b / a;
                System.out.println(a);
                pw.println("Resultado " + i + ": " + res);
                a--;
            }
        } catch (IOException | NullPointerException | ArithmeticException e) {
            System.out.println("Excepción encontrada: " + e.toString());
        }
    }
}
```

- Si ejecutamos el programa especificando la ruta obtenemos la excepción de tipo ArtithmeticException.

```
19          try (FileWriter fw = new FileWriter("C\\trabajo\\prueba.txt")) {
20              pw = new PrintWriter(fw);
21              for (int i = 0; i < 6; i++) {
22                  res = b / a;
23                  System.out.println(a);
24                  pw.println("Resultado " + i + ": " + res);
25                  a--;
26              }
27          } catch (IOException | NullPointerException | ArithmeticException e) {
28              System.out.println("Excepción encontrada: " + e.toString());
29          }
```

```
Console ×
<terminated> Excepciones2 [Java Application] /home/mvidal/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_17.0.5.v20
Excepción encontrada: java.io.FileNotFoundException:  (No existe el archivo o el directorio)
```

- Si ejecutamos el programa, con una ruta vacía obtenemos el siguiente resultado, que nos arroja una excepción FileNotFound.



## 5. Explica los 4 propósitos del final con código.

### Propósito 1, “Definir variables locales como constantes”

Podemos definir variables con un scope o alcance local, y al anteponer la palabra reservada “final”, indicamos que esta variable no puede cambiar su valor, por lo tanto pasaría de ser una variable a una constante.

- A continuación tenemos un ejemplo muy claro, donde creamos una variable de tipo double y con la palabra final al inicio, esta variable lleva el nombre de PI.

```
5      public void calcularArea(double radio) {
6
7          final double PI = 3.1416;
8
9          PI = 4;
10
11         double area = (PI * radio) * 2;
12         System.out.println("Area: " + area);
13
14     }
15 }
16
```

- Por lo tanto podemos asimilar que la variable se convierte en la constante PI y por ende no puede modificar su valor, ya que esto se trata de hacer en la línea 9 pero el IDE no lo permite ya que esta variable pasa a ser inmutable.

### Propósito 2, “Definir atributos inmutables”

Este propósito, lo que hace básicamente es tomar el valor que se define como parámetro cuando se instala esta clase y en futuro si este se quiere modificar no puede hacerlo ya que el comportamiento de este atributo se vuelve inmutable.

```
5       private final double radio;
6       private final double PI = 3.1416;
7
8       public Circulo2(double radio) {
9           this.radio = radio;
10      }
11
12      public void calcularArea(double radio) {
13          double area = (PI * radio) * 2;
14          System.out.println("Area: " + area);
15      }
16
17
18      public double getRadio() {
19          return radio;
20      }
21
22
23      public double getPI() {
24          return PI;
25      }
26  }
27
```

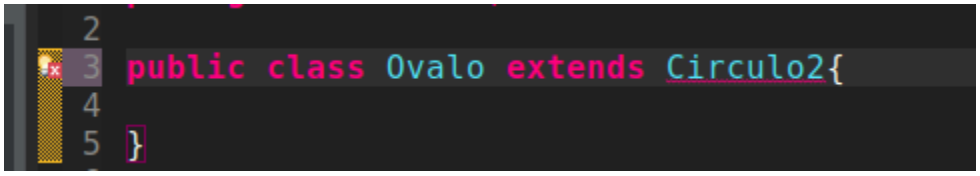
- El comportamiento es muy parecido al de las variables locales, solo que en este caso son variables para toda la clase y si por ejemplo quisiéramos crear el método setRadio no podremos hacerlo ya que no podemos cambiar su valor.

```
21
22      public void setRadio(double radio) {
23          this.radio = radio;
24      }
```

### Propósito 3, “Las clases final no pueden aplicar herencia”

Aquí si las clases están definidas como final, cuando se quiera hacer herencia a otra clase, la clase definida como final no permitirá hacerlo.

```java
public final class Circulo2 {

    private final double radio;
    private final double PI = 3.1416;

    public Circulo2(double radio) {
        this.radio = radio;
    }

    public void calcularArea(double radio) {
        double area = (PI * radio) * 2;
        System.out.println("Area: " + area);
    }


    public double getRadio() {
        return radio;
    }

    public void setRadio(double radio) {
        this.radio = radio;
    }

    public double getPI() {
        return PI;
    }
}
```




- Como recordatorio, no se puede aplicar final a las clases abstractas o interfaces, ya que estas sirven como moldes o padres que heredarán métodos o atributos a otras clases, además que no tendría sentido.

### Propósito 4, “Los métodos final no se pueden heredar, no se puede hacer Overriding”

Tomando en cuenta que tenemos definida la siguiente clase Circulo3, podemos notar que tenemos un método calcularArea() y un metodo final mostrarPI.

Circulo3.java

```java
package com.finalP;

public class Circulo3 {

    private final double radio;
    private final double PI = 3.1416;

    public Circulo3(double radio) {
        this.radio = radio;
    }

    public void calcularArea(double radio) {
        double area = (PI * radio) * 2;
        System.out.println("Area: " + area);
    }

    public final void mostrarPI() {
        System.out.println(PI);
    }
```

- Si aplicamos herencia, desde una clase Ovalo2, podemos heredar atributos y hacer un override del método calcularArea(), pero no podemos hacer esto con el método mostrarPI() ya que es un método definido como final.

```java
public class Ovalo2 extends Circulo3{

    private double diametro;

    public Ovalo2(double radio, double diametro
        super(radio);
        this.diametro = diametro;
    }

    @Override
    public void calcularArea(double radio) {
        double area = diametro * radio;
        System.out.println("Area: " + area);
    }

    @Override
    public void mostrarPI() {
        System.out.println(PI);
    }
```

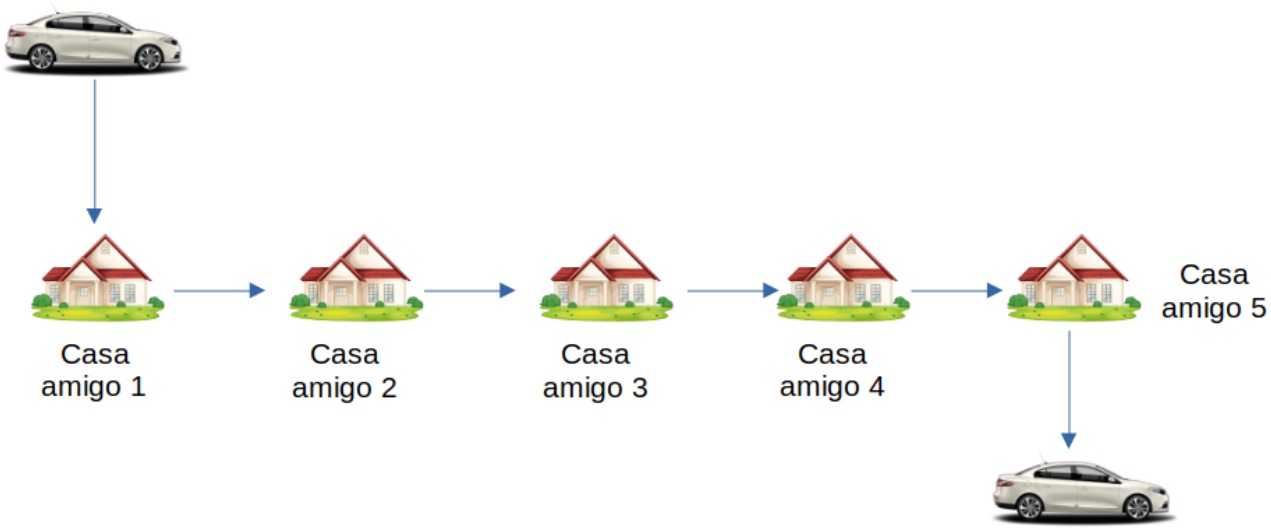
## 6. Diferencia entre un proceso sincrono y asincrono

Un proceso síncrono, es aquel en el que su resultado depende de que se cumpla una serie o secuencia de tareas que tiene que ejecutar 1 a la vez, mientras que un proceso asíncrono igual depende de una serie de tareas pero que se pueden ejecutar simultáneamente.

Si ponemos un ejemplo, puede ser el invitar a 5 amigos a una fiesta, de acuerdo a la respuesta que te den vas a saber cuanta comida y bebida preparar.

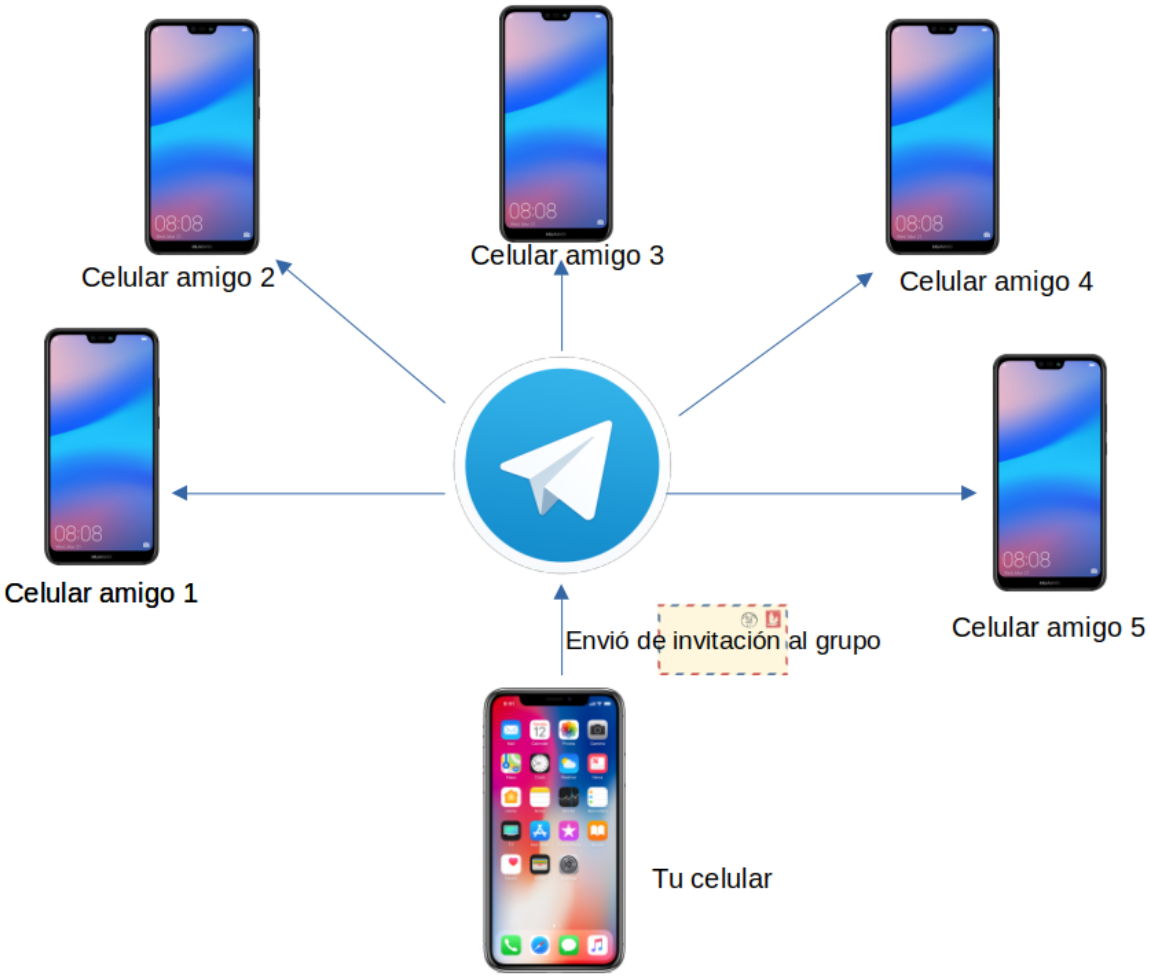
### Invitación de manera síncrona

Viéndolo de manera síncrona, tomarías tu auto e irías a casa de cada uno de tus amigos, esto supondría un consumo alto en combustible, tiempo y energía.



## Invitación de manera asíncrona

Si lo quisiéramos ver de manera asíncrona todo sería tan fácil, como tomar tu smartphone y crear un grupo de telegram, en el cual agregues a tus amigos y les mandes la invitación, de esta forma invitas a todos al mismo tiempo y ahorras combustible, tiempo y energía, ya que no requieres salir de casa para invitarlos.



## 7. Realiza un ejercicio (código) usando lambas

### Lambdas aplicadas a una FUNCTIONALINTERFACE

Partiendo de que usaremos este principio, debemos mencionar que tenemos una clase llamada Vehículo, esta clase tiene propiedades generales de un vehículo, las cuales toma como atributos, tiene su constructor y sus getters & setters.

```java
public class Vehiculo {

    private String marca;
    private String modelo;
    private String color;
    private int ejes;

    public Vehiculo (String marca, String modelo, String color, int ejes) {
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
        this.ejes = ejes;
    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public String getModelo() {
        return modelo;
    }

    public void setModelo(String modelo) {
        this.modelo = modelo;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public int getEjes() {
        return ejes;
    }

    public void setEjes(int ejes) {
        this.ejes = ejes;
    }

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

Posteriormente tenemos una interface funcional llamada PredicadoVehiculo, esta misma tiene un método abstracto que retornara un true o false, dicho método lleva por nombre ejecutar, este método nos servirá cuando usemos lambdas ya que definiremos su lógica al momento de crear lambdas.

Principal.java | PredicadoVehiculo.java | Vehiculo.java

```java
package com.lambdas;

@FunctionalInterface
public interface PredicadoVehiculo {
    abstract boolean ejecutar(Vehiculo vh);
}
```

Ahora pasaremos a nuestra clase principal, la cual es una clase tiene el método main y por la cual correrá nuestro programa.

```java
public static void main(String[] args) {

    List<Vehiculo> listaVehiculos = new ArrayList<>();
    listaVehiculos.add(new Vehiculo("Dodge", "Charger 400", "Negro", -2));
    listaVehiculos.add(new Vehiculo("Volskwagen", "Gol", "Rojo", 2));
    listaVehiculos.add(new Vehiculo("Scannia", "K360C B4x2NB", "Azul", 3));
    listaVehiculos.add(new Vehiculo("Ford", "Fiesta 2012", "Verde", 2));
    listaVehiculos.add(new Vehiculo("Yamaha", "R6", "Azul c Gris", 1));
    listaVehiculos.add(new Vehiculo("Chevrolet", "Silverado", "Blanco", 2));
    listaVehiculos.add(new Vehiculo("Kenworth", "T680", "Amarrillo", 4));
    listaVehiculos.add(new Vehiculo("Man", "MAN TGS 33.540", "Negro", 0));
    listaVehiculos.add(new Vehiculo("Kawasaki", "Ninja 400", "Verde", 1));
    listaVehiculos.add(new Vehiculo("Ford", "Lobo 2021", "Negra", 2));

    System.out.println("\n/ / / VEHICULOS PESADOS / / /");
    PredicadoVehiculo pv = v -> v.getEjes() > 2;
    mostrarVehiculo(listaVehiculos, pv);

    System.out.println("\n/ / / VEHICULOS LIGEROS / / /");
    mostrarVehiculo(listaVehiculos, pvh -> pvh.getEjes() > 0 && pvh.getEjes() <= 2);

    System.out.println("\n/ / / VEHICULOS CON ERRORES AL ESCANEAR / / /");
    PredicadoVehiculo pv0 = v -> v.getEjes() <= 0;
    mostrarVehiculo(listaVehiculos, pv0);

    System.out.println("\n/ / / VEHICULOS DE COLOR NEGRO / / /");
    mostrarVehiculo(listaVehiculos, v1 -> v1.getColor().equals("Negro") || v1.getColor().equals("Negra"));

    System.out.println("\n---- LAMBDAS CON GENERICS ----");

    Vehiculo vehiculo1 = new Vehiculo("Ford", "Mustang", "Naranja", 2);

    Function<Vehiculo, String> f1 = vhf -> vhf.getColor();
    System.out.println("Vehiculo 1 es de color: " + f1.apply(vehiculo1));

    BiConsumer<String, Integer> concatStringInt = (a, b) -> System.out.println(a + " " + b);
    concatStringInt.accept("No. Ejes: ", vehiculo1.getEjes());

    BiPredicate<Vehiculo, Vehiculo> vehiculoPesado = (vhc1, vhc2) -> vhc1.getEjes() > vhc2.getEjes();
    System.out.println("Más pesado que un camion: "
            + vehiculoPesado.test(vehiculo1, new Vehiculo("Kenworth", "T680", "Amarrillo", 4)));

    UnaryOperator<String> obtenerMarcaModelo = v -> v.concat(vehiculo1.getModelo());
    System.out.println("Del modelo y marca: " + obtenerMarcaModelo.apply(vehiculo1.getMarca()+" "));
}

public static void mostrarVehiculo(List<Vehiculo> lvh, PredicadoVehiculo pvh) {
    for (Vehiculo vh : lvh) {
        if (pvh.ejecutar(vh))
            System.out.println(vh.toString());
    }
}
```

A continuación explicaremos de manera detallada cada parte de esta clase.

- En primer lugar tenemos la definición de una lista de vehículos, la cual llenamos creando objetos de tipo Vehiculo.

```
8     public static void main(String[] args) {
9
10        List<Vehiculo> listaVehiculos = new ArrayList<>();
11        listaVehiculos.add(new Vehiculo("Dodge", "Charger 400", "Negro", -2));
12        listaVehiculos.add(new Vehiculo("Volskwagen", "Gol", "Rojo", 2));
13        listaVehiculos.add(new Vehiculo("Scannia", "K360C B4x2NB", "Azul", 3));
14        listaVehiculos.add(new Vehiculo("Ford", "Fiesta 2012", "Verde", 2));
15        listaVehiculos.add(new Vehiculo("Yamaha", "R6", "Azul c Gris", 1));
16        listaVehiculos.add(new Vehiculo("Chevrolet", "Silverado", "Blanco", 2));
17        listaVehiculos.add(new Vehiculo("Kenworth", "T680", "Amarrillo", 4));
18        listaVehiculos.add(new Vehiculo("Man", "MAN TGS 33.540", "Negro", 0));
19        listaVehiculos.add(new Vehiculo("Kawasaki", "Ninja 400", "Verde", 1));
20        listaVehiculos.add(new Vehiculo("Ford", "Lobo 2021", "Negra", 2));
```

- En segundo lugar, abriendo un paréntesis para explicar el método mostrarVehiculo, en este mismo tenemos definido que no retornamos nada, recibimos una lista de Vehiculos y un predicado personalizado que es el de PredicadoVehiculo, que no es otra cosa que una instancia de la interface funcional que definimos previamente.
- Dentro del método indicamos que recorreremos la lista de vehículos mediante un ciclo for each, en cada iteración preguntamos si cada objeto dentro de la lista tiene información si este es el caso lo imprimimos.

```
54    public static void mostrarVehiculo(List<Vehiculo> lvh, PredicadoVehiculo pvh) {
55        for (Vehiculo vh : lvh) {
56            if (pvh.ejecutar(vh))
57                System.out.println(vh.toString());
58        }
59    }
```

- Regresando al método main, después de declarar la lista de vehículos, tenemos la primer lambda, en donde condicionamos que si el número de ejes es mayor o igual a 2 determinamos serán vehículos pesados, enseguida tenemos el método mostrarVehiculos() que muestra los aquellos vehículos que cumplen esa acción.

```
21
22        System.out.println("\n/ / / VEHICULOS PESADOS / / /");
23        PredicadoVehiculo pv = v -> v.getEjes() > 2;
24        mostrarVehiculo(listaVehiculos, pv);
```

- Después tenemos, la siguiente lambda que hace lo contrario de la anterior, ya que si los vehículos tienen 2 ejes o menos serán vehículos ligeros .

```
26          System.out.println("\n/ / / VEHICULOS LIGEROS / / /");
27          mostrarVehiculo(listaVehiculos, pvh -> pvh.getEjes() > 0 && pvh.getEjes() <= 2);
28
```

- En la siguiente lambda preguntamos que si los vehículos tienen como ejes el número 0 o un valor negativo, serán considerados vehículos con errores al escanear en la caseta del peaje.

```
29          System.out.println("\n/ / / VEHICULOS CON ERRORES AL ESCANEAR / / /");
30          PredicadoVehiculo pv0 = v -> v.getEjes() <= 0;
31          mostrarVehiculo(listaVehiculos, pv0);
```

- Por ultimo tenemos una lambda, que pregunta si los vehiculos tiene definido como color la palabra “Negro” o “Negra”, seran considerados como vehiculos de color negro.

```
33          System.out.println("\n/ / / VEHICULOS DE COLOR NEGRO / / /");
34          mostrarVehiculo(listaVehiculos, v1 -> v1.getColor().equals("Negro") || v1.getColor().equals("Negra"));
35
```

- Ahora si ejecutamos el programa hasta este punto obtendremos el siguiente resultado:

```
/ / / VEHICULOS PESADOS / / /
Vehiculo [marca=Scannia, modelo=K360C B4x2NB, color=Azul, ejes=3]
Vehiculo [marca=Kenworth, modelo=T680, color=Amarrillo, ejes=4]

/ / / VEHICULOS LIGEROS / / /
Vehiculo [marca=Volskwagen, modelo=Gol, color=Rojo, ejes=2]
Vehiculo [marca=Ford, modelo=Fiesta 2012, color=Verde, ejes=2]
Vehiculo [marca=Yamaha, modelo=R6, color=Azul c Gris, ejes=1]
Vehiculo [marca=Chevrolet, modelo=Silverado, color=Blanco, ejes=2]
Vehiculo [marca=Kawasaki, modelo=Ninja 400, color=Verde, ejes=1]
Vehiculo [marca=Ford, modelo=Lobo 2021, color=Negra, ejes=2]

/ / / VEHICULOS CON ERRORES AL ESCANEAR / / /
Vehiculo [marca=Dodge, modelo=Charger 400, color=Negro, ejes=-2]
Vehiculo [marca=Man, modelo=MAN TGS 33.540, color=Negro, ejes=0]

/ / / VEHICULOS DE COLOR NEGRO / / /
Vehiculo [marca=Dodge, modelo=Charger 400, color=Negro, ejes=-2]
Vehiculo [marca=Man, modelo=MAN TGS 33.540, color=Negro, ejes=0]
Vehiculo [marca=Ford, modelo=Lobo 2021, color=Negra, ejes=2]
```

## Lambdas aplicadas a GENERICS

En esta forma de aplicar lambdas, ya no es necesario crear predicados, ya que las api de java.util, nos proporciona una serie de interfaces ya pre diseñadas, por decirlo así.

Entonces una vez comprendido este principio, podemos explicar cada lambda y parte de nuestro código.

```
36          System.out.println("\n---- LAMBDAS CON GENERICS ----");
37
38          Vehiculo vehiculo1 = new Vehiculo("Ford", "Mustang", "Naranja", 2);
39
40          Function<Vehiculo, String> f1 = vhf -> vhf.getColor();
41          System.out.println("Vehiculo 1 es de color: " + f1.apply(vehiculo1));
42
43          BiConsumer<String, Integer> concatStringInt = (a, b) -> System.out.println(a + " " + b);
44          concatStringInt.accept("No. Ejes: ", vehiculo1.getEjes());
45
46          BiPredicate<Vehiculo, Vehiculo> vehiculoPesado = (vhc1, vhc2) -> vhc1.getEjes() > vhc2.getEjes();
47          System.out.println("Más pesado que un camion: "
48                  + vehiculoPesado.test(vehiculo1, new Vehiculo("Kenworth", "T680", "Amarrillo", 4)));
49
50          UnaryOperator<String> obtenerMarcaModelo = v -> v.concat(vehiculo1.getModelo());
51          System.out.println("Del modelo y marca: " + obtenerMarcaModelo.apply(vehiculo1.getMarca()+" "));
```

- Primero creamos nuestra variable de referencia con la instancia a una clase (un objeto), de tipo vehículo, en el cual le especificamos sus atributos.

```
36          System.out.println("\n---- LAMBDAS CON GENERICS ----");
37
38          Vehiculo vehiculo1 = new Vehiculo("Ford", "Mustang", "Naranja", 2);
```

- Creamos nuestra primera lambda con ayuda del generic, Function el cual recibe un objetivo Vehículo y un String, dentro de esta lambda concatenamos el un mensaje y el color de nuestro vehículo.

```
40          Function<Vehiculo, String> f1 = vhf -> vhf.getColor();
41          System.out.println("Vehiculo 1 es de color: " + f1.apply(vehiculo1));
```

- En la segunda lambda usamos un BiConsumer, el cual recibe un String y un Integer, los cuales nos ayudan a concatenar el mensaje “No. Ejes: “ y el numero de llantas de nuestro objeto vehiculo.

```
43          BiConsumer<String, Integer> concatStringInt = (a, b) -> System.out.println(a + " " + b);
44          concatStringInt.accept("No. Ejes: ", vehiculo1.getEjes());
```

- En la tercera lambda usamos un Bipredicate, el cual recibe 2 objetos de tipo vehículo, dentro de esta lambda comparemos que si nuestro objeto de tipo vehículo es más pesado que otro objeto de tipo vehículo pero con más ejes.

```
46          BiPredicate<Vehiculo, Vehiculo> vehiculoPesado = (vhc1, vhc2) -> vhc1.getEjes() > vhc2.getEjes();
47          System.out.println("Más pesado que un camion: "
48                  + vehiculoPesado.test(vehiculo1, new Vehiculo("Kenworth", "T680", "Amarrillo", 4)));
```

- En nuestra última lambda, tenemos al generic UnaryOperator, el cual recibe un String, en esta lambda concatenamos el mensaje: “De la marca y modelo” con la marca y el modelo de nuestro objeto de tipo vehículo .

```
50          UnaryOperator<String> obtenerMarcaModelo = v -> v.concat(vehiculo1.getModelo());
51          System.out.println("Del modelo y marca: " + obtenerMarcaModelo.apply(vehiculo1.getMarca()+" "));
```

- Finalmente obtenemos el siguiente resultado:

```
---- LAMBDAS CON GENERICS ----
Vehiculo 1 es de color: Naranja
No. Ejes:  2
Más pesado que un camion: false
Del modelo y marca: Ford Mustang
```