ACADEMIA JAVA

Luis Wonen Olvera Vasquez
Examen Semana 2

Lugar:Ciudad de México
Fecha: 02/12/2022

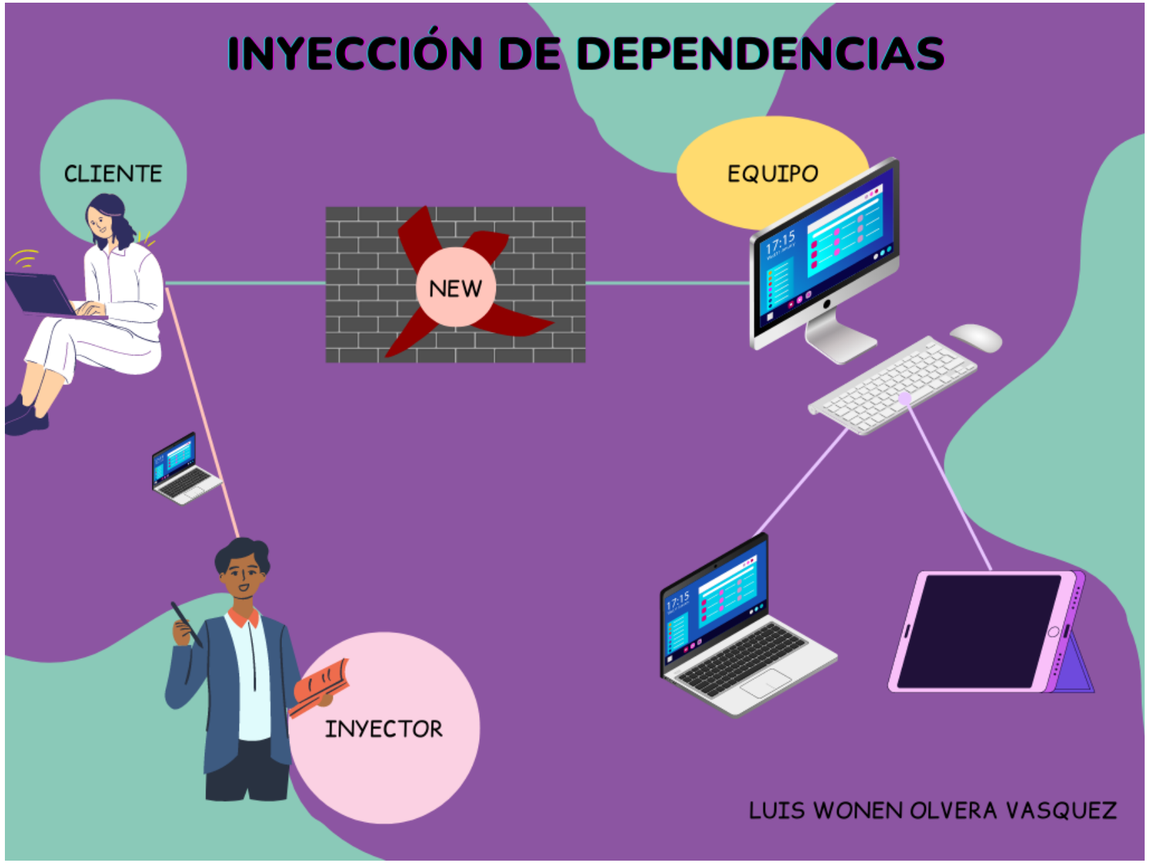
1.- Diagrama y codifica que es inyección de dependencias

Inyección de dependencia: es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos. Esos objetos cumplen contratos que necesitan nuestras clases para poder funcionar (de ahí el concepto de dependencia).

Nuestras clases no crean los objetos que necesitan, sino que se los suministra otra clase 'Inyector' que inyectará la implementación deseada a nuestro contrato.

Podemos explicarlo ayudándonos de diagrama anterior, normalmente el Cliente sería el encargado de asignar, pero no realizamos esto ya que nos llevaría a un alto acoplamiento, es por esto que se delega la tarea al Inyector.

Lo que conocería el cliente hasta el momento es que se le será asignado un equipo electrónico que puede ser una laptop o una tableta, solo el inyector será quien sepa el tipo de equipo se le asignará al cliente.



1.1- Diagrama y codifica que es inyección de dependencias

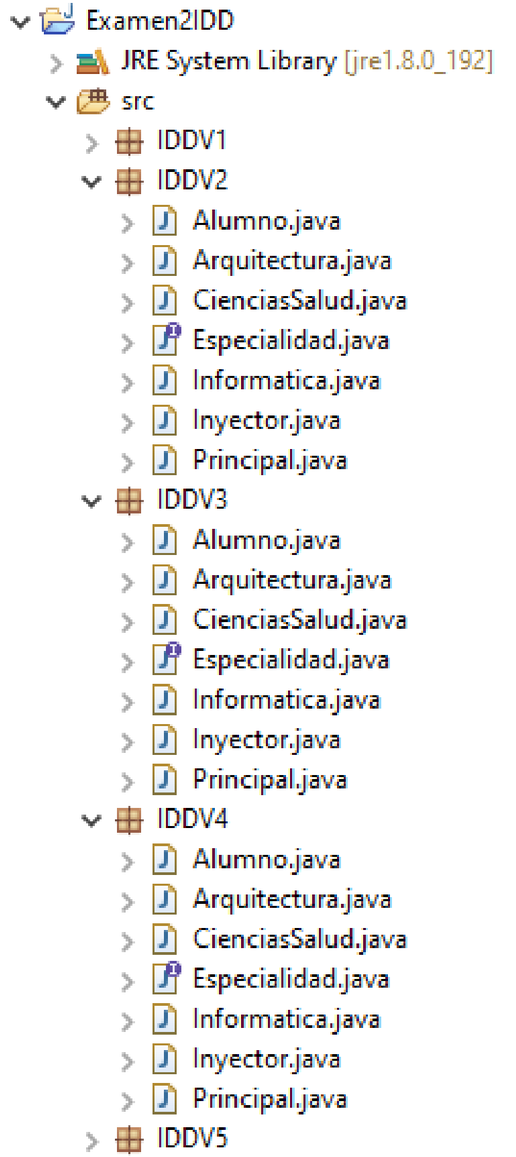
En la carpeta Examenes/Semana 2, encontraremos el programa **Examen2IDD**, en el cual tendremos 5 paquetes: IDDV1,IDDV2, IDDV3, IDDV4 y IDDV5.

El objetivo general del programa es demostrar la inyección de dependencias, por lo que tomaremos de ejemplo un alumno que tiene un nombre y grupo. Al alumno se le asignará una especialidad en las cuales entra arquitectura, informática y ciencias de la salud. Un factor importante a contemplar es que el alumno no es el responsable de escoger la especialidad, el alumno solo sabrá que tendrá una especialidad. El inyector, será quien tenga la responsabilidad de asignarle su especialidad al alumno.

Dentro del paquete IDDV1 no ocupamos la inyección de dependencias ya que el objetivo es mostrar como funcionaria si el alumno fuera quien se asignara la especialidad de igual forma mostrando el alto acoplamiento.

Paquete IDDV2, en este paquete ocupamos la inyección de dependencias por una variable, por lo tanto haremos uso de una interface que tendrá solamente el método asignar de esta manera el alumno tendrá un método void asignar pero el inyector será quien se encargue de asignar la especialidad.

```java
package IDDV2;

public class Inyector {

    //De esta forma el inyector es quien asigna la especialidad
    static void InyEsp(Alumno alu) {

        alu.esp = new CienciasSalud(" salud");
    }
}
```



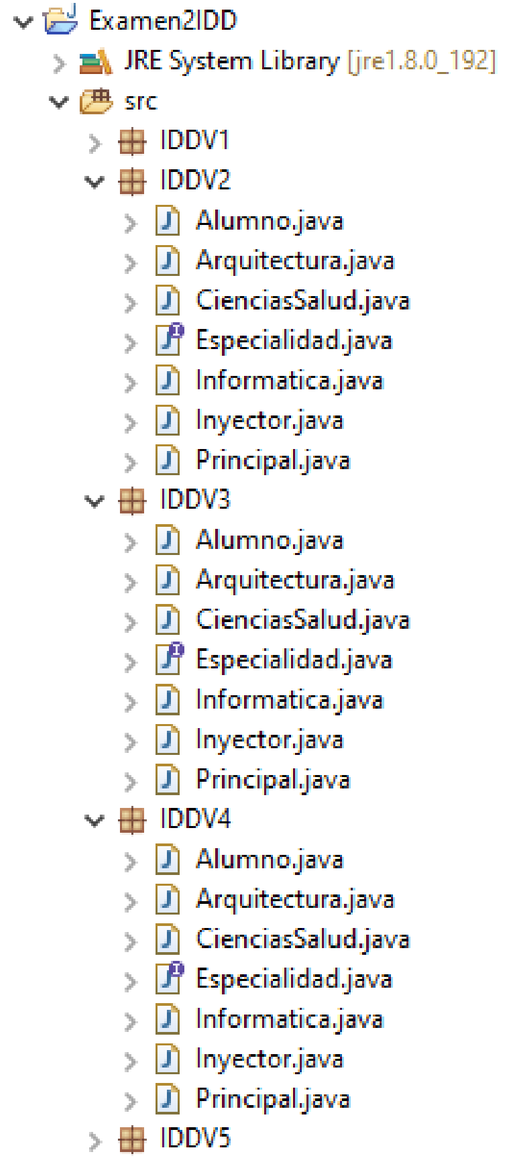
1.2- Diagrama y codifica que es inyección de dependencias

Paauete IDDV3, en este paquete ocupamos la inyección de dependencias mediante el método Setter, por lo que en la clase alumno generamos los getters y setters

```
27      //Generamos Getters y Setters
28      public Especialidad getEsp() {
29          return esp;
30      }
31
32      public void setEsp(Especialidad esp) {
33          this.esp = esp;
34      }
35
```

Por lo tanto cuando el inyector quiera inicializar sera mediante el método Setter.

```
1  package IDDV3;
2
3  public class Inyector {
4
5      //De esta forma el inyector es quien asigna la especialidad
6      static void InyEsp(Alumno alu, Alumno alu1, Alumno alu2) {
7
8          /*Como Especialidad ahora es private solo se puede
9            inicializar mediante el método setter*/
10         alu.setEsp(new CienciasSalud(" salud"));
11         alu1.setEsp(new Informatica(" informatica"));
12         alu2.setEsp(new Arquitectura(" arquitectura"));
13
14     }
15 }
```



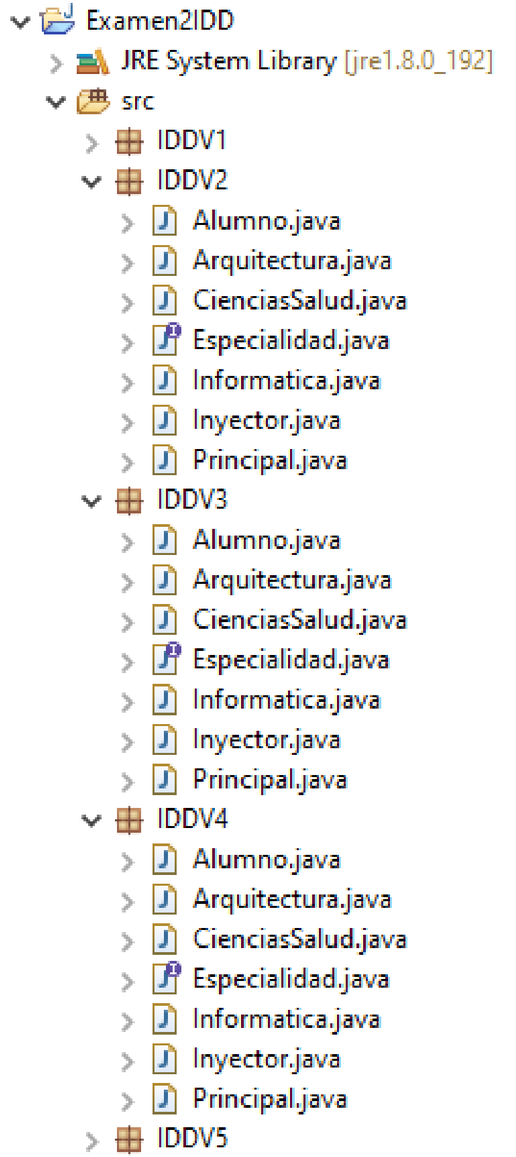
1.3- Diagrama y codifica que es inyección de dependencias

Paauete IDDV4, en este paquete ocupamos la inyección de dependencias mediante el constructor, por lo tanto cuando estamos en la clase alumno además de añadir el nombre y grupo añadiremos esp que es de tipo Especialidad.

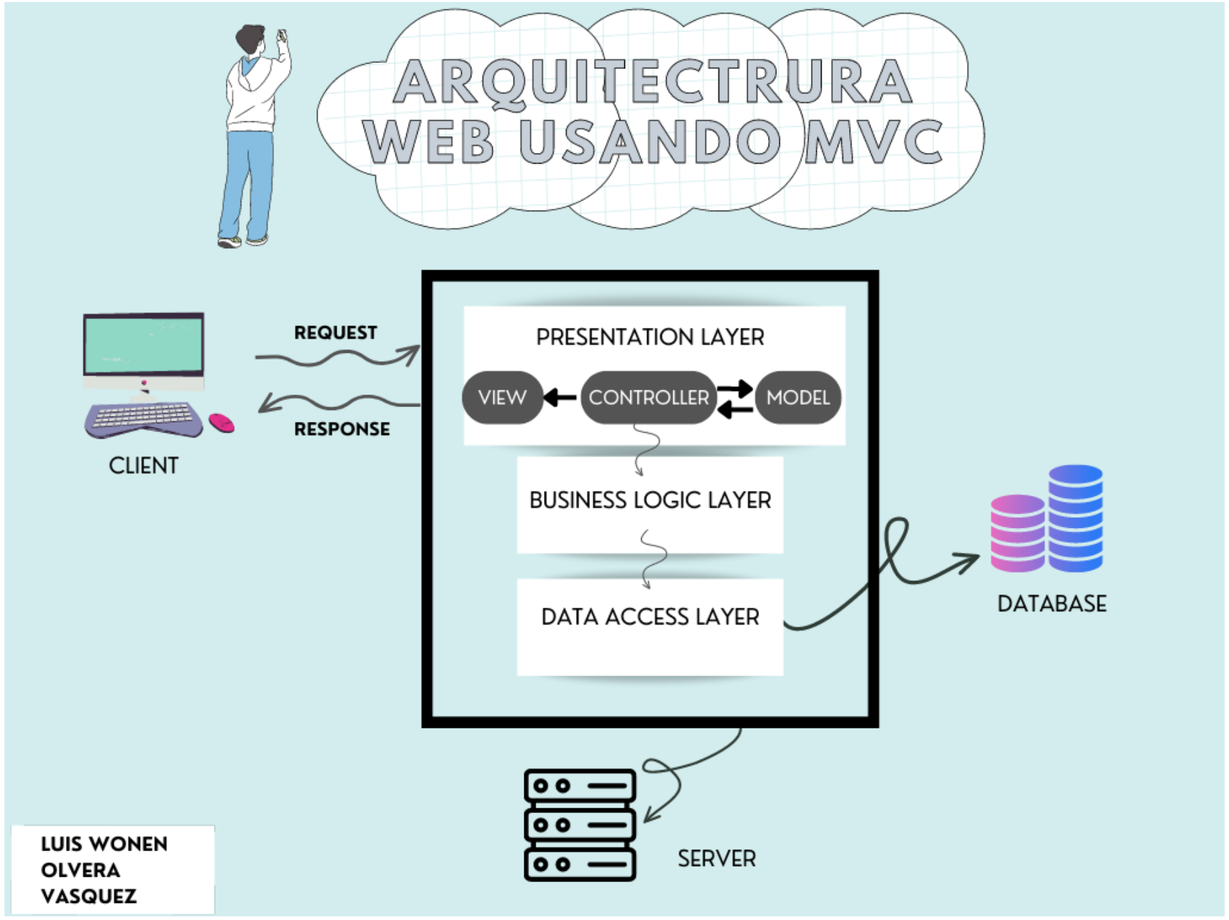
```java
package IDDV4;

public class Alumno {

    //Creamos las variables nombre y grupo
    String nombre;
    String grupo;

    private Especialidad esp;

    //Creamos el constructor de la clase
    public Alumno(String nombre, String grupo, Especialidad esp) {
        this.nombre = nombre;
        this.grupo = grupo;
        this.esp = esp;
    }
```

Por lo tanto cuando el inyector quiera inicializar sera mediante el constructor.

```java
package IDDV4;

public class Inyector {

    //De esta forma el inyector es quien asigna la especialidad
    static Alumno getAlumno() {

        //Especialidad Arquitectura = new Arquitectura(" arquitectura");
        Especialidad Informatica = new Informatica(" informática");

        return new Alumno("Eduardo Orlando","308", Informatica);

    }
}

```



2. Diagrama y explica una arquitectura web usando MVC.



MVC (Model View Controller, en español Modelo Vista Controlador) es un patrón de diseño, que tiene como principal objetivo separar los procesos de lógica de negocio, acceso a datos y la vista que le presentaremos al usuario. El patrón MVC puede ser usado para desarrollo de aplicaciones web, escritorio, móviles, entre otros.

Tomando de referencia el diagrama a nuestro lado nos enfocaremos en el patrón MVC en una arquitectura web. El cliente hace un request a través de su navegador web puede ser Chrome, Firefox, Opera, Safari, entre otros. Este request puede ser un GET, PUT, POST, DELETE, etc. Es importante entender que el la arquitectura el MVC se encuentra dentro de la capa de presentación.

Dentro de la Vista se encuentra lo que el usuario ve en su pantalla, la lógica de la presentación, se encarga de generar la interfaz de usuario. El controlador es el cerebro, el director de orquesta, es quien se encarga de interactuar tanto con la vista como con el modelo. El controlador se encarga de recibir los request del usuario de esta forma le dice a la vista que es lo que debe mostrar y al modelo que datos debe transportar.

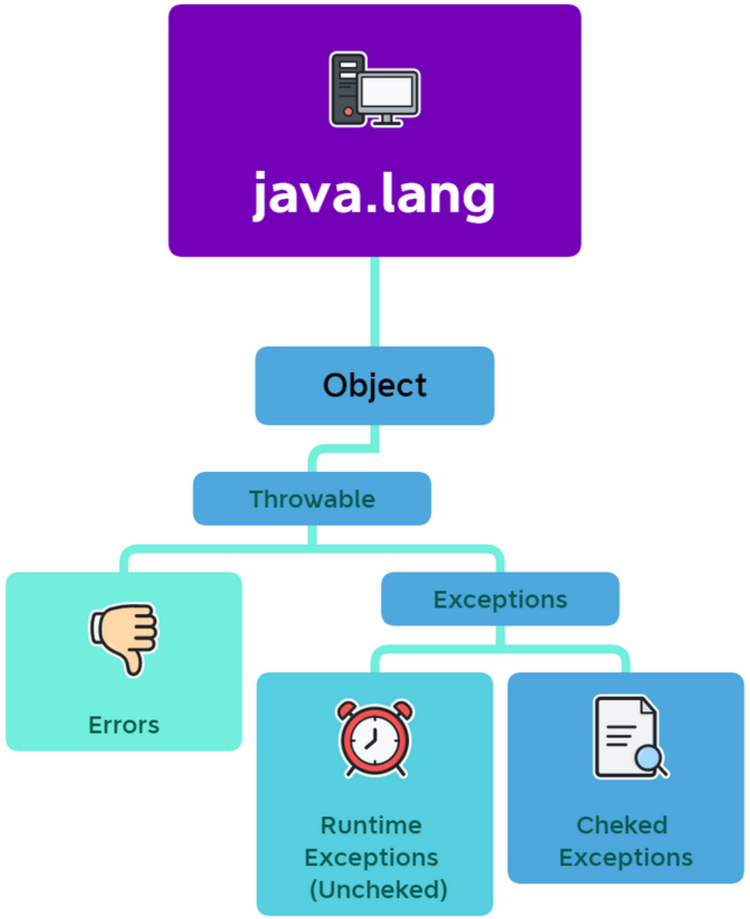
Cuando hablamos del modelo, es común pensar que este es el encargado de la lógica del negocio o de interactuar con la base de datos, pero esto es erróneo ya que el modelo solo se encarga de transportar los datos. El encargado de la lógica de negocios es la capa de lógica de negocio y quien interactúa con la base de datos es la capa de acceso a datos (ORM). En cuanto a la base de datos esta puede ser SQL o NoSQL.

3. Explica los diferentes tipos de excepciones.

Throwable: es una clase que representa todo lo que se pueda “lanzar” en Java. Contiene una instantánea del estado de la pila de memoria en el momento que se creó el objeto, lo que también llamamos stack trace o call chain.

A continuación, se mostrarán algunos métodos de la clase Throwable junto con una descripción.
https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html

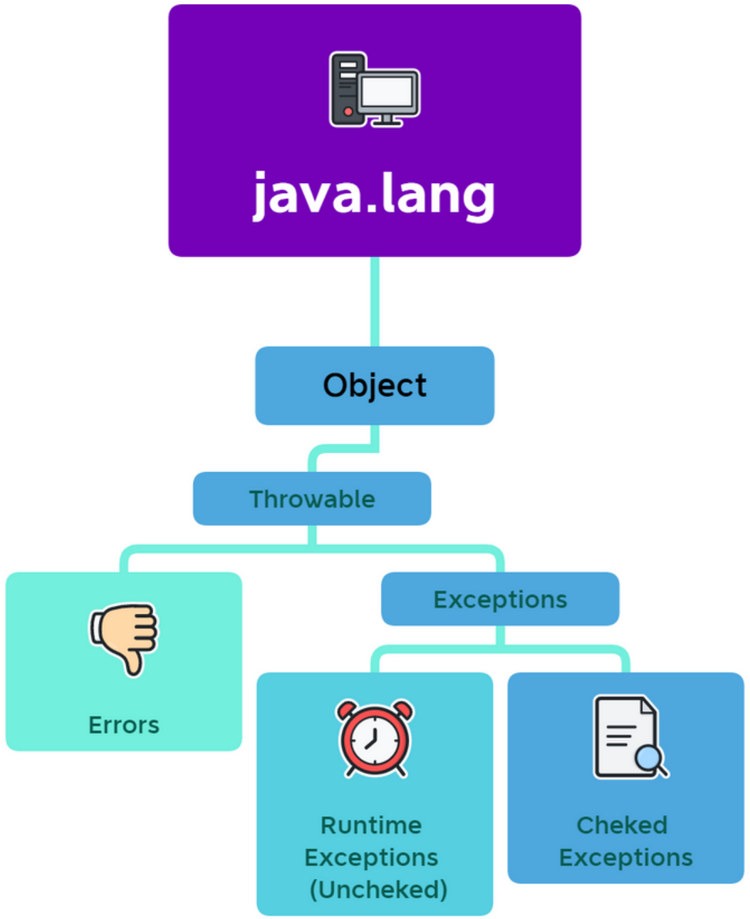
- **getMessage**(): Se usa para obtener un mensaje de error asociado con una excepción.
- **printStackTrace**(): Se utiliza para imprimir el registro del stack donde se ha iniciado la exepción.
- **toString**(): Se utiliza para mostrar el nombre de una excepción junto con el mensaje que devuelve getMessage().
- **void printStrackTrace**: Visualiza un objeto y la traza de pila de llamadas lanzada.



3.1 Explica los diferentes tipos de excepciones.

Error: nos indica un problema grave, ya que este problema no puede ser resuelto de ninguna manera, por lo que un programa normalmente se detiene. Algunos ejemplos de error pueden ser: si se quiere acceder al disco duro, pero este se encuentra dañado, no tener conexión a internet, un fallo en la electricidad, no tener permiso para modificar un archivo, entre otros. Podemos concluir que estos errores son ajenos al programa.

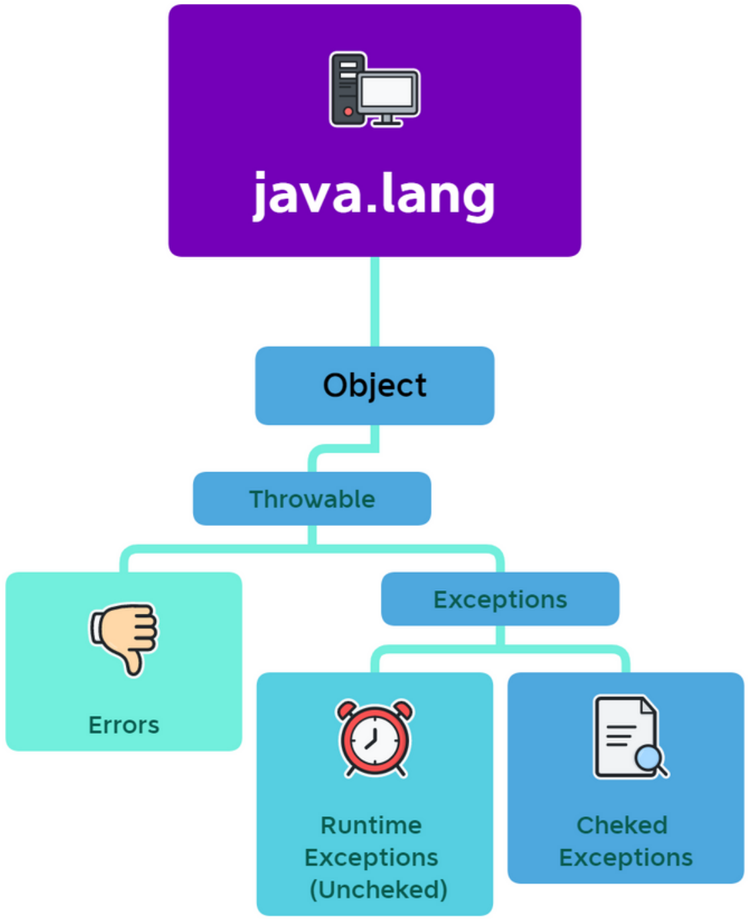
- **AnnotationFormatError:** Se genera cuando el analizador de anotaciones intenta leer una anotación de un archivo de clase y determina que la anotación tiene un formato incorrecto. https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/AnnotationFormatError.html
- **AssertionError:** Lanzado para indicar que una aserción ha fallado. https://docs.oracle.com/javase/8/docs/api/java/lang/AssertionError.html
- **CoderMalfunctionError:** Se genera un error cuando el decodeLoopmétodo de a CharsetDecoder, o el encodeLoopmétodo de a CharsetEncoder, genera una excepción inesperada.https://docs.oracle.com/javase/8/docs/api/java/nio/charset/CoderMalfunctionError.html
- **FactoryConfigurationError:** Se lanza cuando existe un problema con la configuración con Parser Factories. Este error generalmente se generará cuando no se pueda encontrar o instanciar la clase de una fábrica de analizadores especificada en las propiedades del sistema.https://docs.oracle.com/javase/8/docs/api/javax/xml/parsers/FactoryConfigurationError.html



3.2 Explica los diferentes tipos de excepciones.

Runtime Exception (Unchecked): En este tipo de excepciones el compilador no obliga que se tenga que colocar las cláusulas try catch. Por lo anterior lo único que podemos hacer es que nuestro programa termine fallando y muestre el mensaje de error al usuario, de esta manera después de conocer el error podemos implementar un try-catch para que el compilador nos permita ejecutar el programa.

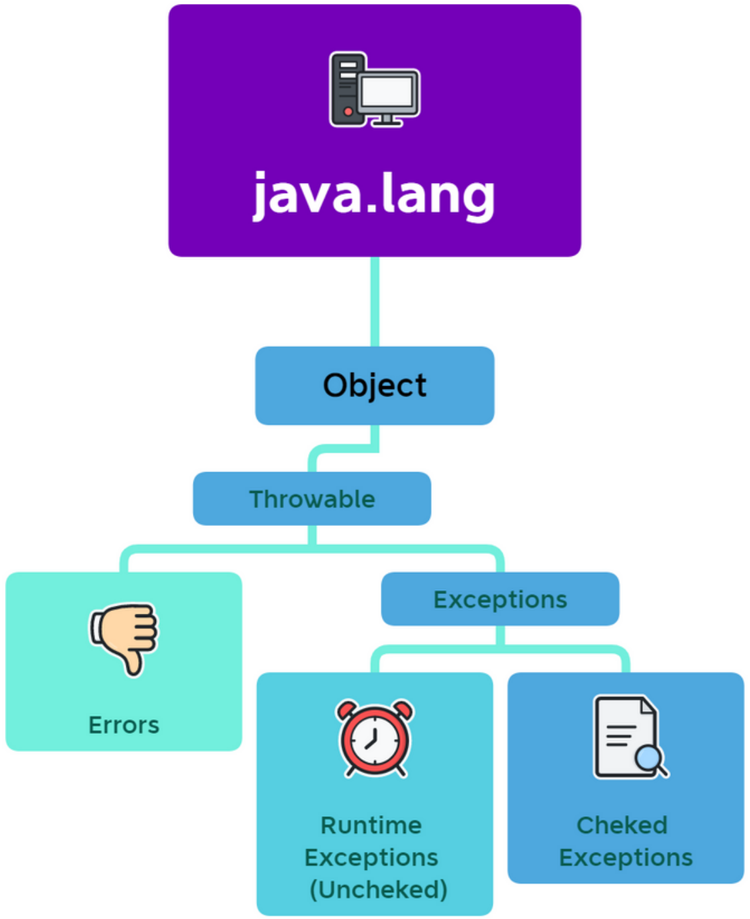
- **ArithmeticException:** Lanzado cuando se ha producido una condición aritmética excepcional. Por ejemplo, un entero "dividir por cero" arroja una instancia de esta clase. https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html
- **IllegalArgumentException:** Lanzado para indicar que a un método se le ha pasado un argumento ilegal o inapropiado. https://docs.oracle.com/javase/8/docs/api/java/lang/IllegalArgumentException.html
- **ArrayStoreException:** Se lanza para indicar que se ha intentado almacenar el tipo de objeto incorrecto en una matriz de objetos. https://docs.oracle.com/javase/8/docs/api/java/lang/ArrayStoreException.html
- **IndexOutOfBoundsException**: Se lanza para indicar que un índice de algún tipo (como una matriz, una cadena o un vector) está fuera de rango. https://docs.oracle.com/javase/8/docs/api/java/lang/IndexOutOfBoundsException.html



3.3 Explica los diferentes tipos de excepciones.

Checked Exception: En este tipo de excepciones el compilador obliga a que se utilicen las cláusulas try-catch u obliga a lanzar las excepciones. Por lo anterior es menos probable que este tipo de excepciones ocurra ya que el compilador nos indica cuando tenemos un error de este tipo.

- **PrintException:** La clase PrintException encapsula una condición de error relacionada con la impresión que ocurrió mientras se usaba una instancia del servicio de impresión. https://docs.oracle.com/javase/8/docs/api/javax/print/PrintException.htm
- **PrinterException**: La PrinterExceptionclase y sus subclases se utilizan para indicar que se ha producido una condición excepcional en el sistema de impresión. https://docs.oracle.com/javase/8/docs/api/java/awt/print/PrinterException.html
- **IOException**: Señala que se ha producido una excepción de E/S de algún tipo. Esta clase es la clase general de excepciones producidas por operaciones de E/S fallidas o interrumpidas. https://docs.oracle.com/javase/8/docs/api/java/io/IOException.html



4. Explica los cuatro propósitos del final con código.

Variables locales: Cuando se pone el final al declarar las variables locales, la estaremos convirtiendo en una constante local, por lo cual su valor ya no se podrá modificar. Por lo tanto, al declarar una variable local con final es necesario inicializarla sino mostrará problemas al compilar.

Atributos: Cuando ponemos final al declarar un atributo, como en las variables locales lo volvemos constante por lo que es necesario también darle un valor inicial o bien que se inicialice mediante un parámetro en el constructor. Por lo anterior, una vez que tenga valor ya no será mutable.

FVL.java | cartas.java | CartasInglesas.java

```
1
2  public class FVL {
3
4      final int numero = 13; //Atributo
5
6      public static void main(String[] args) {
7          final int Min_valor = 1;//Variable local
8          final int Max_valor = 10;
9
10         int num;
11         num = Max_valor - Min_valor;
12
13         System.out.print(num);
14     }
15
16 }
17
```

4.1 Explica los cuatro propósitos del final con código.

Clase: Cuando agregamos el final a una clase, todo estará bien en la misma, pero nos limitará en la cuestión de la herencia ya que una clase final no puede ser padre, si creamos otra clase e intentamos extender la clase final nos marcará un error. Otro aspecto del final es que no se puede aplicar en una clase abstracta.

FVL.java | cartas.java | CartasInglesas.java

```
1
2  public final class cartas {
3
4      int NMC;
5
6      public cartas(int nMC) {
7          super();
8          NMC = nMC;
9      }
10
11     void asignar(int a) {
12     }
13 }
```

FVL.java | cartas.java | CartasInglesas.java

```
1
2  public class CartasInglesas extends cartas {
3
4  }
5
```

4.2 Explica los cuatro propósitos del final con código.

Método: Cuando agregamos el final a un método, este método no puede ser sobrescrito por las clases hijas. De la misma forma cuando se agrega el final a un método abstracto nos enviará un error.

```
1
2  public class cartas {
3
4      int NMC;
5
6      public cartas(int nMC) {
7          super();
8          NMC = nMC;
9      }
10
11     final void asignar(int a) {
12     }
13 }
14
```

```
1
2  public class CartasInglesas extends cartas {
3
4      public CartasInglesas(int nMC) {
5          super(nMC);
6          // TODO Auto-generated constructor stub
7      }
8
9      @Override
10     final asignar(int a) {
11     }
12
13
14 }
```

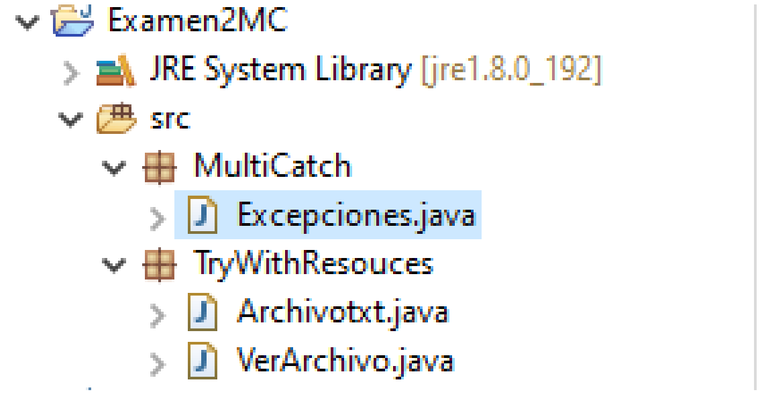
5. Exponer un código con multicacth, try with resources

Para responder esta pregunta se creo el proyecto Examen2MC, dentro del cual se encuentra el paquete multicatch y la clase Excepciones.

El objetivo de este programa es exponer el multicatch para esto hicimos uso de los enteros y operaciones aritméticas básicas.

Primero declamos num1 y num2 estos enteros son los que seran asignados por el usuario desde consola. Se declararon los enteros suma. resta, mult y div para almacenar el resultado de las operaciones.

Lo anterior se realizo dentro del un bloque try y el catch se utilizara para las excepciones.




```java
package MultiCatch;

import java.util.InputMismatchException;

public class Excepciones {


    public static void main(String [] args) {
        //Declaramos las variables locales que se ocuparan en las siguientes operaciones
        //int num = 13;
        int num1,num2;
        int suma, resta, mult, div;


        try {
            //Pediremos dos números desde consola para realizar las operaciones
            System.out.println("Porfavor, ingrese un número: ");
            num1 = extracted().nextInt();

            System.out.println("Porfavor, ingrese otro número: ");
            num2 = extracted().nextInt();
```

```
}catch(ArithmeticException e) {
        //Esta excepcion nos dice que aritméticamente no se puede dividir un entero en 0
    System.out.println("Aritméticamente no se puede divir entre 0");
}catch(InputMismatchException e) {
        //Esta excepcion nos dice que el valor debe ser un entero
        //Si escribimos una cadena, un decimal, etc nos marcara esta excepción
        System.out.println("Se debe ingresar un entero");
}catch(Exception e) {
    e.printStackTrace(System.out);
}finally {
    //El código que se encuentre en el finally siempre se ejecutará
    System.out.println("El objetivo del programa es hacer operaciones con enteros");
}
```

5.1 Exponer un código con multicacth, try with resources

Al ser num1 y num2 asignados desde consola, hice pruebas para saber que excepciones podría obtener, al finalizar encontre: ArithmeticException y InputMismatchException.

La primera funciona cuando a pesar de asignar números enteros la opreación no es aritméticamente posible, la segunda excepcion es cuando lo ingresado por consola no pertenece a un entero.

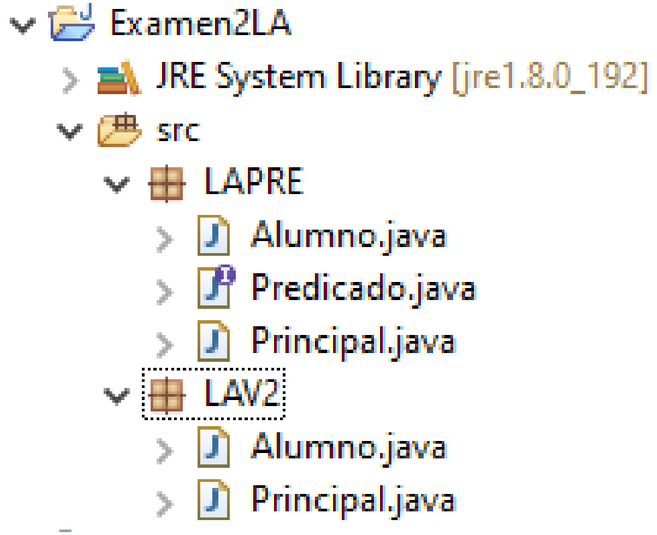
Decidí añadir el finally con el prósosito de explicar que aunque se ejecute o no una excepción siempre se ejecutará el bloque de código dentro de él.

```
<terminated> Excepciones [Java Application] C:\Program Files\Java\jre1.8.0_192\bi
Porfavor, ingrese un número:
1000
Porfavor, ingrese otro número:
20
El resultado de la suma es: 1020
El resultado de la resta es: 980
El resultado de la mult es: 20000
El resultado de la div es: 50
El objetivo del programa es hacer operaciones con enteros
```

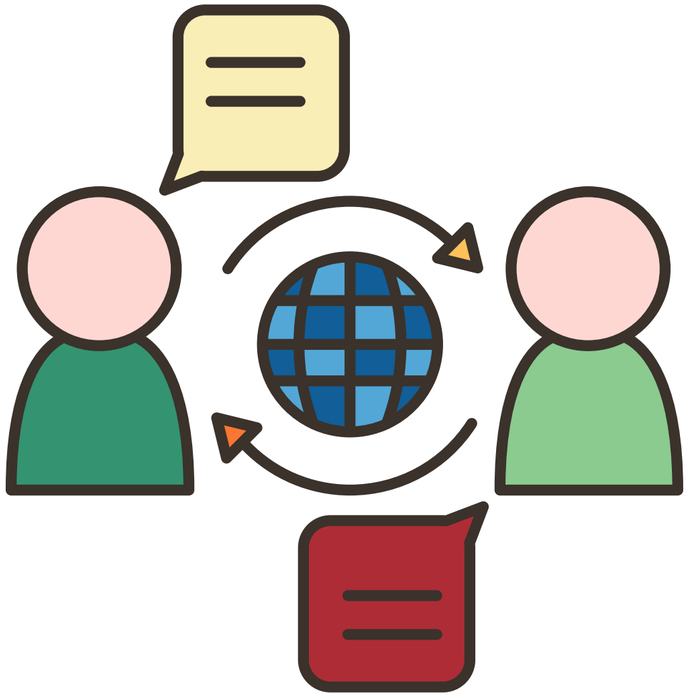
```
<terminated> Excepciones [Java Application] C:\Program Files\Java\jre1.8.0_192\
Porfavor, ingrese un número:
10
Porfavor, ingrese otro número:
0
Aritméticamente no se puede divir entre 0
El objetivo del programa es hacer operaciones con enteros
```

5.2 Exponer un código con multicacth, try with resources

En la carpeta Examenes/Semana 2, encontraremos el programa Examen2LA, en el cual tendremos 2 paquetes LAPRE y LAV2. En ambos paquetes tenemos la clase principal, en esta clase principal usaremos el scanner y ocuparemos el try with resources para cerrar el Scanner ya que al pedir datos desde consola podriamos tener excepciones por ocupamos el try with resources para cerrar el scanner.




```
19          try (Scanner sc = new Scanner(System.in)) {
20              List<Alumno> listaAlumnos = new ArrayList<>();
21
```



6. Explica la diferencia entre los procesos síncronos y asíncronos.

Proceso síncrono: Síncrono es un adjetivo que describe objetos o eventos que están coordinados en el tiempo. Un proceso síncrono En la comunicación de programa a programa, la comunicación síncrona requiere que cada extremo en un intercambio de comunicación responda a su vez sin iniciar una nueva comunicación.

Una actividad típica que podría usar un protocolo sincrónico sería una transmisión de archivos de un punto al otro. A medida se recibe cada trasmisión, se devuelve una respuesta que indica éxito o la necesidad de reenviar. Cada transmisión de datos sucesiva requiere una respuesta a la transmisión previa, antes de que pueda iniciarse una nueva.

Significa que todos los involucrados en una actividad deben realizar su parte al mismo tiempo. Tales eventos a veces se llaman eventos en tiempo real o en vivo. Dichos eventos incluyen sesiones de chat, sesiones de pantalla compartida y pizarra, y videoconferencias.

6.1 Explica la diferencia entre los procesos síncronos y asíncronos.

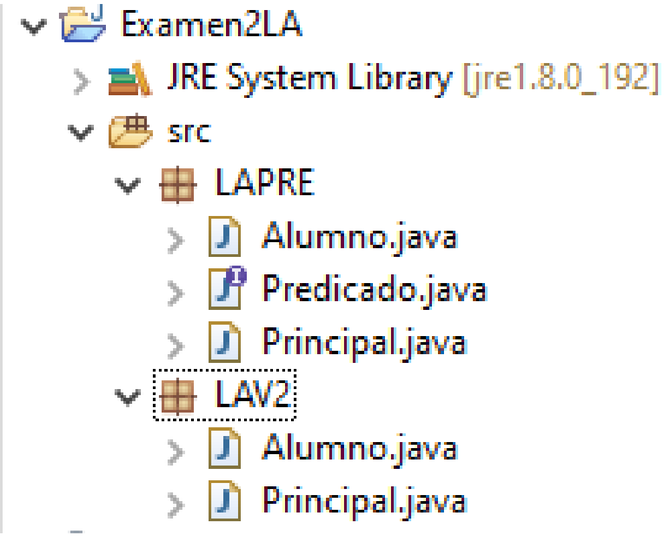
Proceso asíncrono: Un proceso asíncrono es un proceso o una función que ejecuta una tarea "en segundo plano" sin que el usuario tenga que esperar a que finalice la tarea.

Son aquellas que los participantes pueden experimentar cuando lo deseen. Los materiales de aprendizaje publicados permanentemente y las evaluaciones calificadas automáticamente son claramente asíncronas: los estudiantes pueden leerlas en cualquier momento.

Diferencia entre procesos síncronos y asíncronos: En programación comúnmente se dice que los procesos de un programa que se ejecutan de forma independiente se llaman asíncronos, mientras que si son procesos ejecutados en respuesta a otro proceso serán síncronos.

7. Realiza un ejercicio usando Lambdas

En la carpeta Examenes/Semana 2, encontraremos el programa **Examen2LA**, en el cual tendremos 2 paquetes LAPRE y LAV2. Dentro del primero paquete tenemos el ejemplo de Lambdas usando el predicado y en el segundo paquete tenemos el ejemplo de Lambdas con el uso de predicate.



En ambos ejemplos tenemos la misma base, una lista de alumnos que se define por consola, a continuación se ira llenando la lista, pidiendo el nombre, la especialidad y la matricula. Para esto los primeros 2 datos serán String y la matrícula será un entero.

La diferencia primordial entre el primer y segundo ejemplo es el uso de la interface Predicado ya que las clases Alumno en ambos ejemplos son iguales. Al usar la interface Predicado nosotros definimos el método abstracto y lo utilizamos en la clase Principal. En cambio si usamos Predicate java nos proporciona el método Test.

Predicado.java

```
1  package LAPRE;
2
3  @FunctionalInterface
4  public interface Predicado<T> {
5
6      abstract boolean probar(T t);
7  }
8
```

7.1 Realiza un ejercicio usando Lambdas

En la clase alumno declaramos las variables privadas nombre, especialidad y matricula. Generamos el contructor de la clase, los getters y setter además del método toString

```
public class Alumno {

    private String nombre;
    private String especialidad;
    private int matricula;

    //Generamos el contructor con todos los valores
    public Alumno(String nombre, String especialidad, int matricula) {
        super();
        this.nombre = nombre;
        this.especialidad = especialidad;
        this.matricula = matricula;
    }
```

```
17      //Generamos el método toString
18      @Override
19      public String toString() {
20          return "Alumno [nombre=" + nombre + ", especia
21      especialidad + ", matricula=" + matricula + "]";
22      }
```

```
24      //Generamos Getters y Setters
25      public String getNombre() {
26          return nombre;
27      }
28      public void setNombre(String nombre)
29          this.nombre = nombre;
30      }
31      public String getEspecialidad() {
32          return especialidad;
```

7.2 Realiza un ejercicio usando Lambdas

```
mostrarAlumno(listaAlumnos, w -> w.getNombre().length()>3);

mostrarAlumno(listaAlumnos, w -> w.getMatricula() > 1000);
```

En ambos ejemplos implementamos las mismas lambdas, lo que cambia es el método mostrarAlumno, como se vera debajo de este texto estará la implemantación del Predicado y después estará el Predicate.

```
51      static void mostrarAlumno(List<Alumno> lista, Predicado<Alumno> p ) {
52          for(Alumno e :lista) {
53              if (p.probar(e))
54                  System.out.println(e);
55          }
56
57      }
58
59  }
```

```
52      static void mostrarAlumno(List<Alumno> lista, Predicate<Alumno> p ) {
53          for(Alumno e :lista) {
54              if (p.test(e))
55                  System.out.println(e);
56          }
57
58      }
59
60  }
```