**PROGRAMACIÓN ORIENTA A OBJETOS – POO**

La Programación orientada a Objetos (POO) es considerada como un paradigma de programación. ¿Qué es un paradigma? Es todo aquel modelo, patrón o ejemplo que debe o puede seguirse en determinadas situaciones. En otras palabras, puede ser considerado como una determinada forma de resolver problemas en ciertas situaciones específicas.

La POO se basa en la idea de un mundo lleno de objetos, en donde la resolución de los problemas que puedan existir puede llevarse a cabo mediante el modelado/abstracción de objetos de la vida real.

Al mismo tiempo la POO tiene una serie de propiedades:

⭐ Abstracción

⭐ Encapsulamiento

⭐ Polimorfismo

⭐ Herencia

**¿Qué son las clases? ¿Cómo crear una?**

Las clases son plantillas o moldes que nos permiten construir objetos. Es una de las principales formas de abstraer objetos de la vida real para utilizarlos en el mundo lógico en las aplicaciones Java que desarrollemos.

Por ejemplo: Si tuviéramos la clase auto, esta contendría las principales partes/elementos que tiene un auto para poder construirlo o representarlo a nivel lógico.

Todas las clases tienen las siguientes características o particularidades:

✅ Representan «entidades» del mundo real.

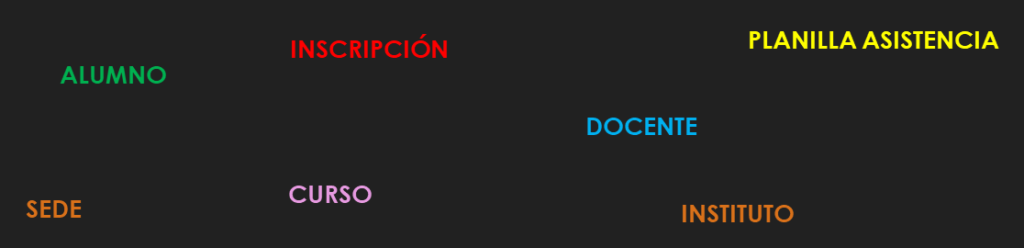
✅ Poseen atributos y métodos.

✅ Para poder hacer uso de ellas, debemos crear «instancias» u «objetos».

**Ejercicio: Identificación de clases**

Supongamos que una academia nos solicita implementar un sistema de asistencia para un curso en particular. Para ello, tenemos que identificar cuáles podrían ser las clases del sistema.

A continuación, te dejamos una lista de «posibles clases». ¿Cuáles de ellas crees que podrían ser modeladas para el sistema de asistencias? Ten en cuenta que las clases generalmente están asociadas a SUSTANTIVOS en SINGULAR.



¿Pudiste identificar cuáles podrían ser las clases? ¿Si? ¿No? De todas maneras te dejamos a continuación la resolución del ejercicio para que puedas validarlo:



**Atributos de una clase**

Los atributos son características que posee una clase. Son variables contenidas y establecidas por los objetos y normalmente cuentan con un tipo de dato asociado.

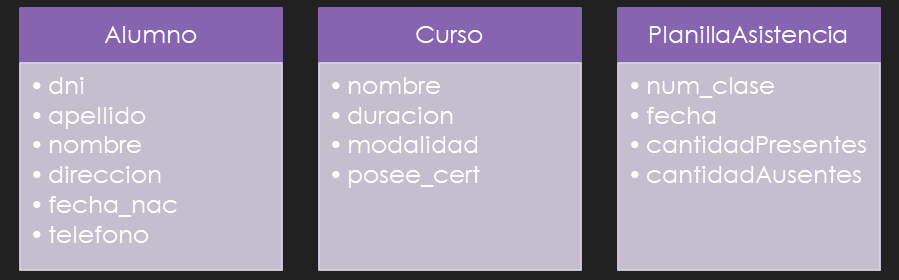
Por ejemplo, supongamos que queremos armar una clase para un PetShop que represente a las Mascotas que podrían arribar al mismo. ¿Cuáles serían los atributos de esta clase?



**Ejercicio: Identificación de atributos**

Te proponemos que pienses en las 3 clases que detectamos en el ejercicio anterior para un sistema de asistencia de un curso. A partir de eso… ¿Cuáles creerías que podrían llegar a ser los atributos de cada una de las clases?

A continuación te dejamos una posible solución. Tené en cuenta que aunque no hayas llegado a la misma, la tuya también puede ser válida, dado que el modelado de clases depende mucho del escenario sobre el que se trabaje y la interpretación de cada programador.



**¿Qué son los métodos? ¿Cómo crear uno?**

Las operaciones o **métodos** son acciones contenidas en una clase y ayudar a definir el comportamiento de la misma, diciendo cuáles son las acciones que ésta puede hacer.

Dentro de un sistema, los métodos suelen estar representados como **verbos en infinitivo (ar, er, ir)**y puede tener opcionalmente valores de entrada (Parámetros) y valores de salida (Retorno).

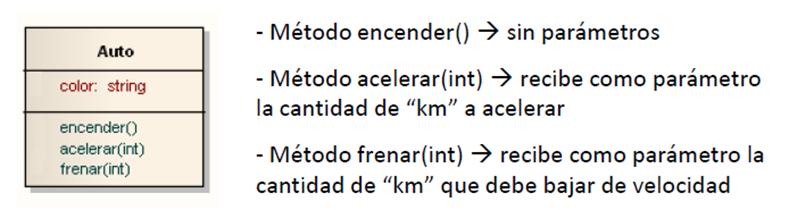
Existen métodos que pueden ser **Procedimientos**(no retornan un valor) o **Funciones**(retornan un valor de un tipo de dato en particular).

**Valores de entrada o parámetros**

Los parámetros son valores que pueden ser enviados en un método. Los métodos toma los parámetros como **valores de entrada**, y así puede realizar las acciones necesarias a partir de los mismos.

Todos los parámetros deben tener un tipo de dato asociado (como así también pueden haber parámetros vacíos).

**Por ejemplo:**



**Valor de Salida o Retorno**

La **salida**de un método es un valor en particular que el mismo retorna luego de haber realizado una serie de acciones o procesos. Los valores de entrada son **datos**, y los valores de salida son considerados  generalmente como **información.**

Todos los valores de salida deben tener un tipo de dato asociado. En los métodos, es posible retornar un **único valor de salida**y ésta acción se lleva a cabo mediante la palabra reservada **return.**



Public = Modificador de acceso.

**Métodos Constructores y Objetos**

En **Java**, los **objetos** son **instancias de clases** y se crean mediante la invocación de un método llamado **constructor**. Un objeto en programación es, generalmente, la representación lógica de un objeto en la vida real. Posee un **estado**(de acuerdo a sus atributos) y posee un **comportamiento**(realizan operaciones de acuerdo a sus métodos).

Un **constructor** es un método especial que se llama automáticamente cuando se crea un objeto de una clase.

Los **constructores** se utilizan para inicializar los atributos de un objeto cuando se crea. Cada clase puede tener uno o más constructores y cada constructor tiene un nombre idéntico al de la clase propiamente dicha.

**Getters y Setters**

En **Java**, los **getters** y **setters** son métodos utilizados para acceder y modificar los valores de los **atributos de una clase** de manera controlada y segura. Los **getters** se utilizan para obtener el valor de un atributo, mientras que los **setters** se utilizan para establecer o modificar el valor de un atributo.

En términos simples, los getters y setters permiten que los atributos de una clase se mantengan privados y no se puedan acceder directamente desde fuera de la clase.

**Propiedades de la POO**

En **Java**, la **Programación Orientada a Objetos (POO)** se basa en cuatro conceptos fundamentales:

* Abstracción
* Herencia
* Polimorfismo
* Encapsulamiento

Veamos un poquito algunas características de cada uno de ellos:

* **Abstracción:** La abstracción en la POO es el proceso de identificar y separar las características esenciales de un objeto de la vida real para representarlas de forma abstracta o lógica en un programa. En Java, esto se logra mediante la creación de clases y la definición de sus atributos y métodos.
* **Herencia**: La herencia es un mecanismo en Java que permite que una clase herede propiedades y métodos de una clase padre/madre.
* **Polimorfismo**: Etimológicamente podemos decir que significa Poli: Muchas, Morfismo: Formas. El polimorfismo en Java es la capacidad de un objeto para tomar muchas formas diferentes.
* **Encapsulamiento:** El encapsulamiento es un mecanismo que permite ocultar los detalles de implementación de un objeto y proporciona una interfaz pública para que otros objetos interactúen con él.

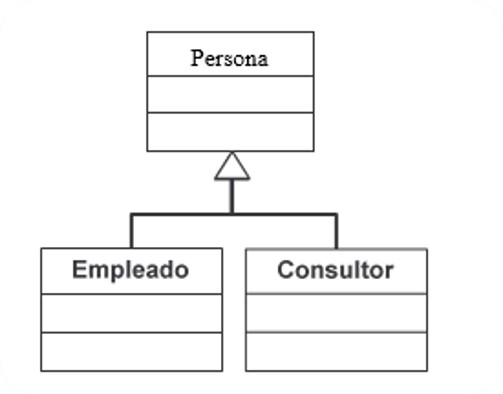
Habiéndolas conocido un poco, veamos un poco más en detalle una de las más importantes y utilizadas a la hora de desarrollar con Java. ¡La herencia!

**Herencia**

En la programación orientada a objetos, la herencia es un concepto clave que permite a una clase adquirir los atributos y comportamientos de otra.

La herencia nos permite reutilizar clases. Generalmente al utilizar herencia se crea una nueva clase que «hereda» o «extiende» métodos y atributos de una clase ya existente sin tener que reescribir el código asociado a esta última. Esta nueva clase es denominada generalmente como **subclase**o también **clase hija**y puede poseer atributos y métodos que no existan en la clase original. Por otro lado, la clase original es conocida como **super clase**o también **clase padre/madre**.

Al igual que la «herencia genética» en la vida real, donde los hijos heredamos ciertas características y comportamientos de nuestros padres, en la «herencia de programación» se cumple exactamente el mismo concepto. ¡Veamos un ejemplo!



**Persona** es una clase genérica que sirve para almacenar datos en común de todas las personas, como el nombre, la dirección, el número de teléfono, etc.

**Empleado**hereda estos mismos atributos de la clase Persona, pero puede incluir atributos propios, como por ejemplo, num de legajo, puesto, cargo, etc.

**Consultor** también hereda los mismos atributos en común de la clase persona, mientras que puede incluir otros atributos propios, como empresa Consultora, id de consultor, etc.

Al mismo tiempo, Empleado y Consultor son entre si clases hermanas. Al igual que en la vida real, ambas heredan las mismas características de s clase padre, sin embargo, ambas pueden poseer de igual manera atributos y métodos que sean propios y que las diferencia entre si.

Public class nombre\_clase extends clase\_padre = si una clase es hija se la declara con la palabra extends después del nombre.

**Polimorfismo**

El **Polimorfismo en Java** se refiere a la capacidad de un objeto de tomar muchas formas diferentes (poli = muchas, morfismo = formas). En términos más simples, significa que un objeto puede ser tratado como si fuera de varios tipos.

En **Java**, esto se logra a través de la **herencia** y los **métodos de sobreescritura**. Supongamos que tienes una clase base llamada «**Animal**» y clases derivadas como «**Perro**» y «**Gato**«. Tanto los perros como los gatos son animales, por lo que pueden heredar de la clase base «**Animal**«.

El **polimorfismo** permite tratar un perro o un gato como un animal genérico. Por ejemplo, si tienes un método que toma un parámetro de tipo «Animal», puedes pasarle tanto un objeto perro como un objeto gato. Esto se debe a que los objetos perro y gato son subtipos de Animal.

Otro ejemplo clásico es con **estructuras** como por ejemplo un **vector/array**. Si declaramos un vector de tipo Animal, dentro de el podremos poner objetos de tipo Perro y objetos de tipo Gato por más que sean distintos por el simple hecho de heredar de la clase Animal y tener eso en común.

**Encapsulamiento**

El E**ncapsulamiento en Java** es una propiedad que nos permite ocultar los detalles internos de una clase y proporcionar acceso controlado a sus atributos y métodos, mediante palabras reservadas llamadas «**modificadores de acceso**«.

**Modificadores de Acceso en Java**

En **Java**, hay cuatro modificadores de acceso que controlan la visibilidad y accesibilidad de las clases, atributos, métodos y constructores. A continuación te contamos de forma sencilla qué permite llevar a cabo cada uno de ellos:

1. **Public**: Es el modificador de acceso más abierto y utilizado. Los miembros (atributos, métodos, constructores) declarados como públicos son accesibles desde cualquier clase y paquete. Pueden ser utilizados por cualquier objeto que tenga acceso a la clase.
2. **Protected:** Los miembros declarados como protected son accesibles SOLO dentro de la misma clase, en clases hijas (subclases) y dentro del mismo paquete.
3. **Private:** Es el modificador de acceso más restrictivo. Los miembros declarados como privados son accesibles solo dentro de la misma clase. No pueden ser accedidos ni siquiera por clases derivadas de la misma.
4. **Default (sin modificador):** Si no se especifica ningún modificador de acceso, se considera el acceso por defecto. Los miembros con acceso por defecto son accesibles solo dentro del mismo paquete. No pueden ser accedidos desde clases en paquetes diferentes.

**Ejemplo**

Veamos un ejemplo práctico. Supongamos que tenemos una clase llamada «Persona» con atributos como nombre, apellido, edad y dirección. En lugar de permitir que otros objetos accedan directamente a estos atributos, podemos declararlos como «privados» utilizando el modificador de acceso «private». Esto significa que solo la propia clase va a poder acceder a ellos.

Supongamos que creamos también en esta clase los getters y setters y los establecemos como «Public». Los getters son métodos que devuelven el valor de un atributo y los setters son métodos que establecen un nuevo valor para el atributo, al ser públicos van a poder ser accesibles desde cualquier otra clase a través de la instancia o creación de un objeto de la clase en cuestión. Estos métodos públicos actúan como intermediarios para acceder a los atributos privados de la clase.

**Clases Abstractas**

Sin dudas, las **clases abstractas** son uno de los temas MÁS TEMIDOS en lo que es **POO** con **Java**… Pero… ¡No te preocupes! En este clase vas a aprender con todo **¿Qué son?** y **¿Para qué se utilizan?**

⚠️ Existe un concepto muy parecido que son las INTERFACES, no te preocupes que en las próximas clases hablaremos de ellas y de su diferencia con las clases abstractas, pero por ahora, arranquemos a hablar de éstas últimas.

Una **clase abstracta en Java** es una clase que no puede ser instanciada directamente, es decir, no se pueden (o deben) crear objetos de esa clase específica. Se utiliza como una **clase base o plantilla** para otras clases relacionadas.

La principal característica de una **clase abstracta** es que puede tener **métodos abstractos**. Un método abstracto es abstracto cuando está declarado pero sin su implementación, es decir, no contiene entre sus «llaves» nada de código.  En un **método abstracto**, solo se declara su firma, incluyendo el nombre del método y los parámetros que acepta cerrando la línea con «;», nada más que eso.

Cuando una clase declara al menos un **método abstracto**, se debe marcar como abstracta utilizando la palabra clave **«abstract»**. Una **clase abstracta** puede tener tanto métodos abstractos como métodos concretos (implementados), pero si o si debe implementar al menos un método abstracto para considerarse como clase abstracta como tal.

Las clases que heredan de una **clase abstracta**, llamadas **subclases o clases hijas**, deben proporcionar una implementación para todos los métodos abstractos heredados. Esto significa que las subclases deben completar/implementar los métodos abstractos declarados en la clase abstracta.

Una **clase abstracta** se utiliza como una abstracción general de la cual se derivan clases más específicas. Proporciona una estructura común y define los métodos abstractos que deben implementarse en las subclases. Esto promueve la **reutilización del código** y la creación de **jerarquías de clases**.

**Interfaces**

Las interfaces en Java son una colección de métodos abstractos y atributos que pueden ser únicamente constantes. Básicamente, se puede ver a una interfaz como un conjunto de comportamientos/métodos que una clase puede luego implementar.

En lugar de proporcionar una implementación concreta de los métodos, una interfaz solo declara los nombres de los métodos y los parámetros que aceptan. Las clases que implementan una interfaz deben ser las encargadas de proporcionar la implementación de todos los métodos declarados en la misma.

⚠️ En pocas palabras, así como una clase abstracta podemos verla como una «plantilla» de atributos y métodos abstractos para otras clases; a las interfaces podemos verlas como «plantillas» de métodos o comportamientos, ya que no es considerada una buena práctica que las mismas posean atributos y en caso de que los posean, deben ser constantes.

⚠️ Por otro lado, una de las características más importantes de las interfaces en Java es que le brindan la posibilidad a este de implementar la conocida «Herencia múltiple».

En forma resumida podemos decir que una interfaz:

Es una colección de métodos abstractos con propiedades (atributos) CONSTANTES.

Solamente puede extender o implementar otras interfaces (la cantidad que quiera).

Da a conocer qué se debe hacer (métodos) pero sin mostrar su implementación (Solo puede tener métodos abstractos).

Solo puede tener métodos con acceso público (no pueden ser protected o private)

Solo puede tener “variables” public static final (o sea constantes)

La palabra “abstract” en la definición de métodos no es obligatoria.

Generalmente las interfaces indican el “PUEDE HACER” de un objeto.

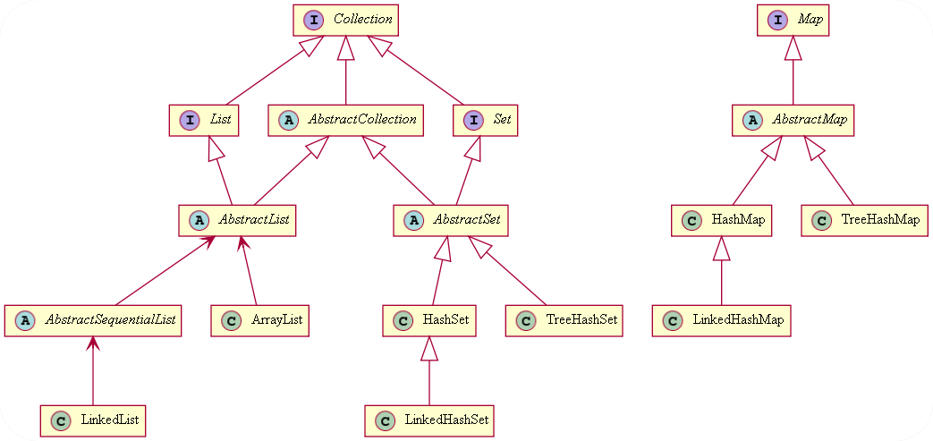
**Clases Abstractas vs Interfaces**

Llegamos al momento favorito de todos… ¡Poner en práctica todo lo aprendido hasta ahora! … Sin embargo, para poder resolver correctamente y llevar el paso a paso del siguiente ejercicio necesitás repasar y tener bien en claro las diferencias entre una clase abstracta y una interfaz, por  lo que a continuación te dejamos un resumen de las principales diferencias que tienen entre si:

* Las **clases abstractas** e **interfaces** ayudan a ordenar código (obliga a las clases a utilizar los mismos métodos y no crear repetidos).
* Una **clase abstracta** puede tener métodos que sean abstractos o que no lo sean, mientras que las **interfaces** sólo y exclusivamente pueden definir métodos abstractos.
* Una **clase abstracta** puede heredar o extender de cualquier clase (independientemente de que esta sea abstracta o no), mientras que **una interfaz** solamente puede implementar otras interfaces.
* Una **clase abstracta** puede heredar de una sola clase (abstracta o no) mientras que **una interfaz** puede extender de varias interfaces de una misma vez.
* En una **clase abstracta**, los métodos abstractos pueden ser public o protected. En **una interfaz** solamente puede haber métodos públicos.
* Las **clases abstractas** exlican mayoritariamente el “ES/SER” de un objeto mientras que  
  **las interfaces** explican el “PUEDE HACER”.

**¿Qué son las Collections en Java?**

Las colecciones son estructuras similares a los arreglos pero con la principal característica de que son dinámicos (su tamaño y cantidad de elementos puede variar en el tiempo). En Java, las colecciones se emplean mediante la interfaz “Collection”, que permite implementar una serie de métodos comunes como ser: Añadir, Eliminar, Obtener el tamaño de la colección, etc.

Tipos principales de Collections

**Listas**

Las listas son un conjunto de elementos relacionados entre si que tienen un determinado orden. Su tamaño es dinámico (puede cambiar en el tiempo)

En Java existen diferentes tipos de Listas:

* ArrayLists
* LinkedLists
* Stack

**ArrayLists**

Entre los tipos de listas, unas de las más utilizadas son las **ArrayLists.**Las mismas tienen estas características:

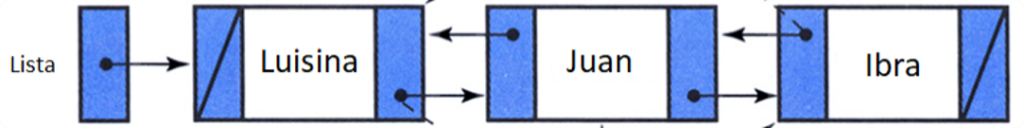
* **ArrayList** es una clase que se representa como una matriz dinámica que permite almacenar elementos.
* Hereda de la clase **AbstractList**, la cual implementa la interfaz **List**.
* Permite colecciones o elementos duplicados.
* El orden de los registros es el orden en el que fueron insertados.
* Permite acceso aleatorio (tiene índice)
* Manipulación lenta (recorrer tooodo el **Arraylist** para hacer un cambio)

**LinkedLists**

Una **LinkedList** en **Java** es una estructura de datos que representa una lista doblemente enlazada.  Se diferencia de los ArrayLists porque sus elementos no se almacenan en ubicaciones contiguas de memoria.

Es una clase.

En una **LinkedList**, cada elemento, llamado nodo, contiene un valor y una referencia al siguiente nodo en la lista. El primer nodo se llama «cabeza» y el último nodo se llama «cola». Si un nodo no tiene un siguiente nodo, su referencia es nula, lo que indica el final de la lista.



La **LinkedList** proporciona métodos para agregar, eliminar y acceder a elementos en la lista. Por ejemplo, se pueden agregar elementos al principio o al final de la lista utilizando los métodos **addFirst()** y **addLast()**. También eliminar elementos con los métodos **removeFirst()** y **removeLast()**. Además, es posible siempre acceder a elementos individuales utilizando el método **get()** o recorrer la lista utilizando un iterador o un **for each**.

La principal ventaja de una **LinkedList** es que permite una inserción y eliminación eficiente de elementos en cualquier posición de la lista, ya que solo se necesitan actualizar las referencias de los nodos adyacentes. Sin embargo, el acceso a elementos en una posición específica es menos eficiente que en un arreglo, ya que la **LinkedList** debe recorrer los nodos desde el principio o desde la posición inicial hasta llegar al elemento deseado.

**Diferencias entre ArrayLists y LinkedLists**

**ArrayList** y **LinkedList** son dos implementaciones de la **interfaz List** en **Java** que se utilizan para almacenar y manipular colecciones de elementos. Aunque ambos cumplen la misma funcionalidad básica, existen diferencias clave en cómo se almacenan y acceden a los elementos, lo que afecta su rendimiento en diferentes situaciones.

**ArrayList** utiliza un arreglo interno para almacenar los elementos de la lista de manera contigua en memoria. Esto permite un acceso rápido a elementos individuales utilizando índices, lo que significa que se puede acceder a cualquier elemento directamente. Sin embargo, la inserción o eliminación de elementos en posiciones intermedias puede ser costosa, ya que los elementos adyacentes deben desplazarse para hacer espacio o llenar el vacío.

En resumen, **ArrayList** es ideal cuando se requiere un acceso rápido a elementos específicos y las operaciones de inserción y eliminación no son frecuentes o críticas para el rendimiento.

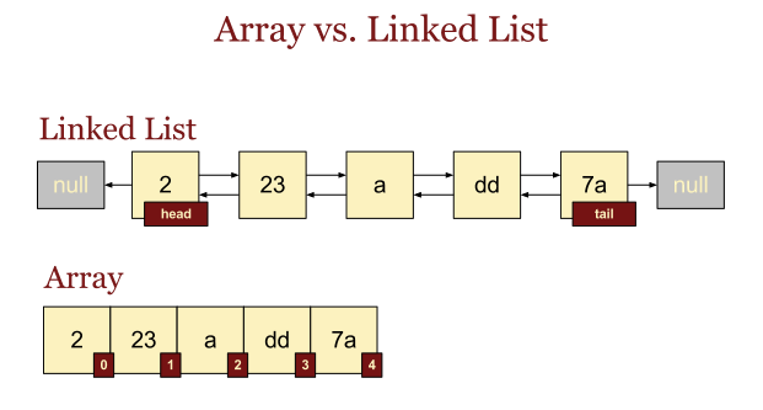
Por otro lado, **LinkedList** utiliza una estructura de **lista enlazada**, donde cada elemento (nodo) contiene una referencia al siguiente nodo en la lista. Los elementos no están almacenados de manera contigua en memoria, lo que hace que el acceso directo a elementos mediante índices sea menos eficiente que en ArrayList.

Sin embargo, **LinkedList** es eficiente para la inserción y eliminación de elementos en cualquier posición, ya que solo se necesitan actualizar las referencias de los nodos adyacentes.

En otras palabras, si es necesario realizar operaciones frecuentes de inserción o eliminación en diferentes posiciones de la lista, **LinkedList** puede ser más adecuado.

En conclusión, la elección entre **ArrayList** y **LinkedList** depende de las operaciones que se realizarán con mayor frecuencia en tu programa. Si necesitas un acceso rápido a elementos individuales y las operaciones de inserción y eliminación no son frecuentes, **ArrayList** es una buena opción. Por otro lado, si las operaciones de inserción y eliminación son más frecuentes y la accesibilidad directa a elementos no es una prioridad, **LinkedList** puede ser más eficiente.

Antes de tomar una decisión de cuál de las dos utilizar, hay que considerar las características y requisitos específicos del programa  a desarrollar para tomar la decisión más adecuada.



**Stacks (Pilas)**

Una **pila**, también conocida como **stack** en inglés, es una estructura de datos en la que los elementos se agregan y eliminan según un principio conocido como «LIFO» (Last-In, First-Out), es decir, el último elemento agregado es el primero en ser eliminado.

Utilizan una serie de métodos propios para realizar ciertas operaciones. Entre ellos se encuentran:

* **push:** Coloca un elemento al tope de la pila.
* **pop:** Borra el último elemento de la pila.
* **isEmpty:** Devuelve true o false dependiendo si la pila está vacía o no.
* **peek:** Mostrar el elemento tope de la lista SIN ELIMINAR
* **search:** Buscar elemento específico de la pila.

**Excepciones en Java**

En **Java**, una **excepción** es un evento o condición inusual que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de ejecución. Las excepciones se utilizan para manejar y señalar errores y situaciones excepcionales.

Muchas cosas pueden causar **excepciones**, entre ellas:

* Errores de hardware
* Operaciones matemáticas no posibles (Ej: dividir por cero)
* Errores de programa (Ej: error de desbordamiento de un arreglo)
* Apertura de un archivo inexistente
* Entre otras…

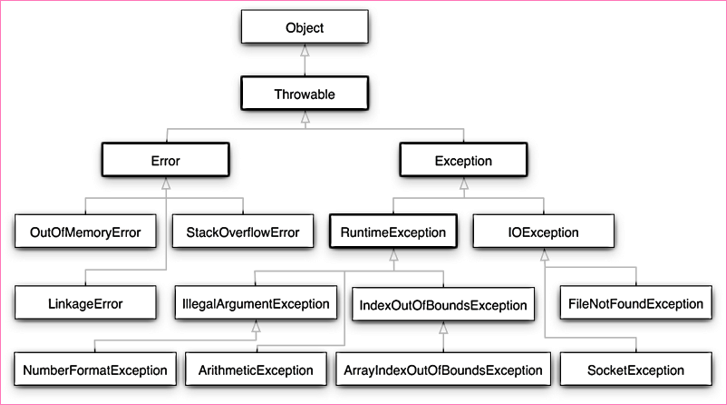
**Tipos de Excepciones**

Hay dos tipos bien diferenciados:

* Propias de **Java** como lenguaje
* Personalizadas

**Propias de Java**

Entre las propias de Java podemos encontrarnos con las siguientes:

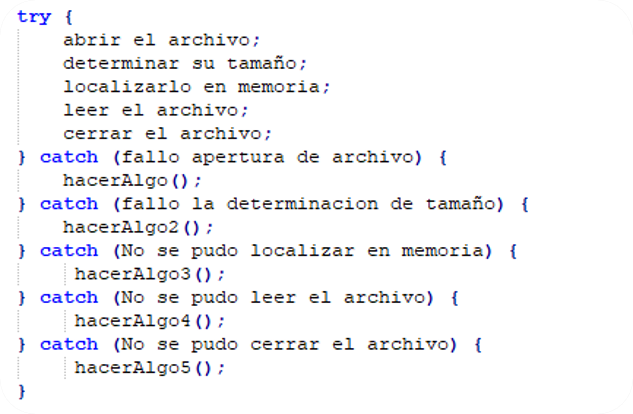


Cada una de ellas puede ser utilizada en diferentes situaciones, por ejemplo, **FileNotFoundException**cuando no se encuentra un archivo o **ArrayIndexOutOfBoundsException**cuando se intenta acceder a una posición fuera de un array.

**Personalizadas**

Para poder construir excepciones propias, hay que tener en cuenta los siguientes bloques:

* **try:** Bloque donde puede ocurrir la excepción
* **catch:**Es como el “else” del try (se ejecuta al dispararse una excepción en el bloque try)
* **finally:**Bloque de código que se ejecuta siempre (sin importar si hubieron errores o no)
* **throw:**Lanzar una excepción cualquiera
* **throws:**Determinar qué excepciones puede lanzar un método

Ejemplo de cómo podría ser la estructura de realizar una serie de acciones que puedan captar una excepción personalizada.

**Relaciones entre Clases en Java**

En Java, las **relaciones entre clases** se establecen mediante **conceptos de programación orientada a objetos**, como la **herencia**, la **composición** y la **asociación**.

Estas relaciones permiten establecer conexiones y dependencias entre diferentes clases para modelar la **estructura** y el **comportamiento** de un sistema.

Anteriormente vimos cómo funciona la herencia, ahora pasemos a ver cómo son las relaciones **1 a 1**, **1 a n** y **n a n** entre **clases**.

**Relación 1 a 1**

Este tipo de relación se lleva a cabo mediante la creación de un objeto como atributo o variable de instancia dentro de una clase. En este tipo de relación, cada objeto de la primera clase tiene una referencia única y exclusiva a un objeto de la segunda clase, y viceversa. Por ejemplo:

//ejemplo relación 1 a 1 entre dos clases

public class ClaseA {

private int id;  
private String nombre;  
private ClaseB objetoB;

// …

}

**Relación 1 a n**

En **programación orientada a objetos (POO)**, una relación uno a muchos implica que un objeto de una clase se relaciona con varios objetos de otra clase.

En este tipo de relación, un objeto de la primera clase puede tener referencias múltiples a objetos de la segunda clase, pero cada objeto de la segunda clase solo está asociado a un único objeto de la primera. Esto se logra generalmente mediante el uso de **Collections**, como ser por ejemplo **List, ArrayList, LinkedList, Hash**, etc.

Por ejemplo, supongamos que queremos relacionar la clase Compra con la clase Items, donde cada compra puede tener varios Items, lo haríamos de la siguiente manera:

//ejemplo relación 1 a n entre dos clases

public class Compra {

private int id;  
private String nombre;  
private List<Item> listaItems;

// …

}

**Relación n a n**

Una relación **muchos a muchos (n a n)** en Java con POO implica que múltiples objetos de una clase se pueden relacionar con múltiples objetos de otra y viceversa.

Esto generalmente se logra mediante el uso de Collections en «ambos lados» de las relaciones, es decir, en ambas clases que se quieren relacionar entre si, dado que recordemos que, las relaciones n a n no son ni más ni menos que dos relaciones 1 a n enfrentadas entre si.

Por otro lado, como generalmente las relaciones n a n luego también son mapeadas en bases de datos, existen casos en donde se puede aplicar la misma teoría que en base de datos, donde es posible la creación de una clase intermedia que agregue información extra a las relaciones existentes entre ambas clases. Sin embargo, en casos donde se usen ORMs (como por ejemplo JPA + Hibernate), no suele ser necesaria esta creación de clase intermedia ya que el mismo ORM (concepto que veremos en clases más adelante) se encarga de crear todo de forma automática.

Supongamos un ejemplo donde una clase Alumnos tiene una relación n a n con una clase Materia, donde muchos Alumnos pueden estar en muchas materias, y muchas materias pueden tener muchos alumnos:

//ejemplo relación n a n entre dos clases  
//Clase Alumno

public class Alumno {

private int id;  
private String nombreAlumno;  
private List<Materia> listaMaterias;

// …

}

//ejemplo relación n a n entre dos clases  
//Clase Materia

public class Materia{

private int id;  
private String nombreMateria;  
private List<Alumno> listaAlumnos;

// …

}