

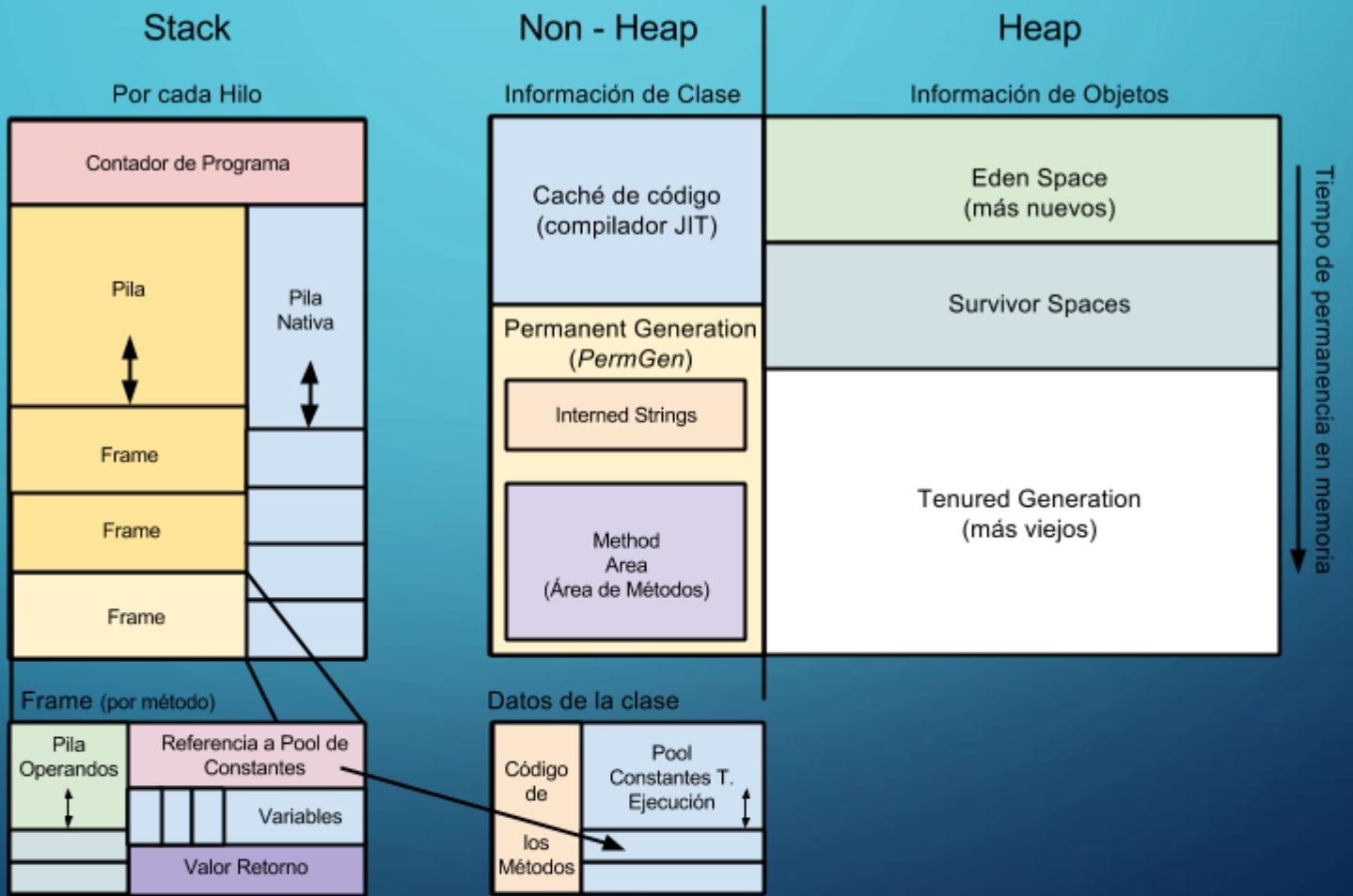
# CURSO JAVA

INSTRUCTOR

I.S.C. E.T.W. M.T.W. CARLOS URIEL DE JESÚS SÁNCHEZ GONZÁLEZ

[HTTPS://WWW.LINKEDIN.COM/IN/CARLOS-URIEL-DE-JESUS-SANCHEZ-GONZALEZ-8920A1372/](https://www.linkedin.com/in/carlosuriel-de-jesus-sanchez-gonzalez-8920a1372/)

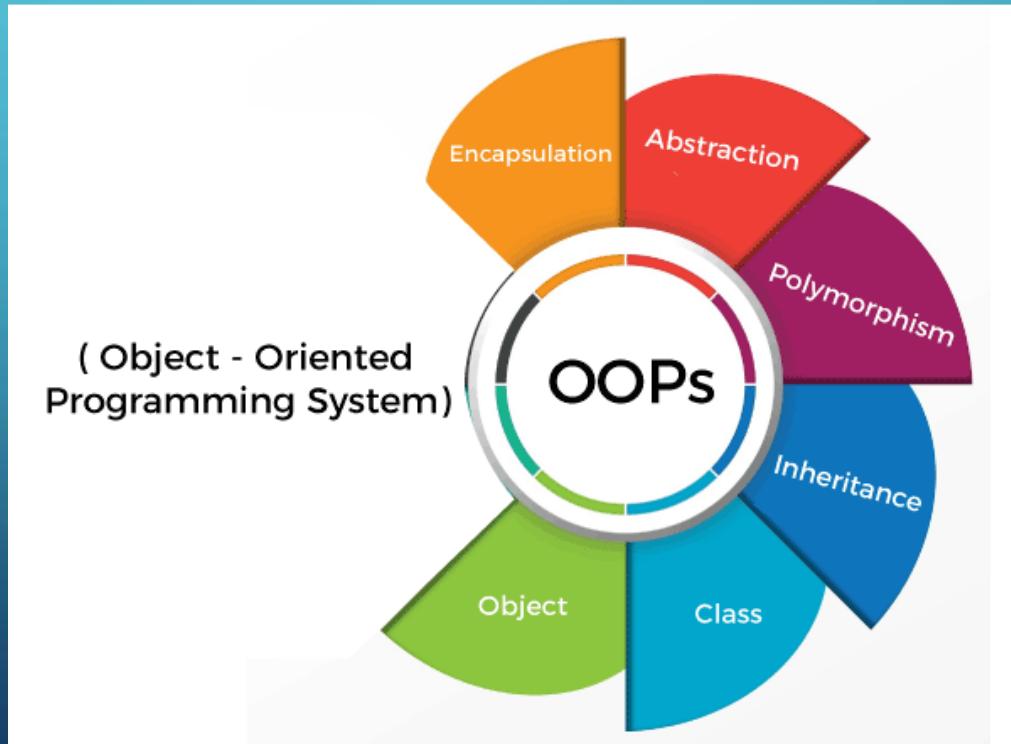
# MEMORIA DE JVM



# POO

## PROGRAMACIÓN ORIENTADA A OBJETOS

- Es un **paradigma de programación** (una forma de organizar y pensar el código) que se basa en el uso de **objetos** para modelar entidades del mundo real o conceptos abstractos.



# OBJETOS Y CLASES

Clase → Es como una plantilla o molde que define las características (atributos) y comportamientos (métodos) de un objeto.

Ejemplo: Persona puede ser una clase.

Objeto → Es una instancia de una clase, es decir, un elemento concreto creado a partir de una clase.

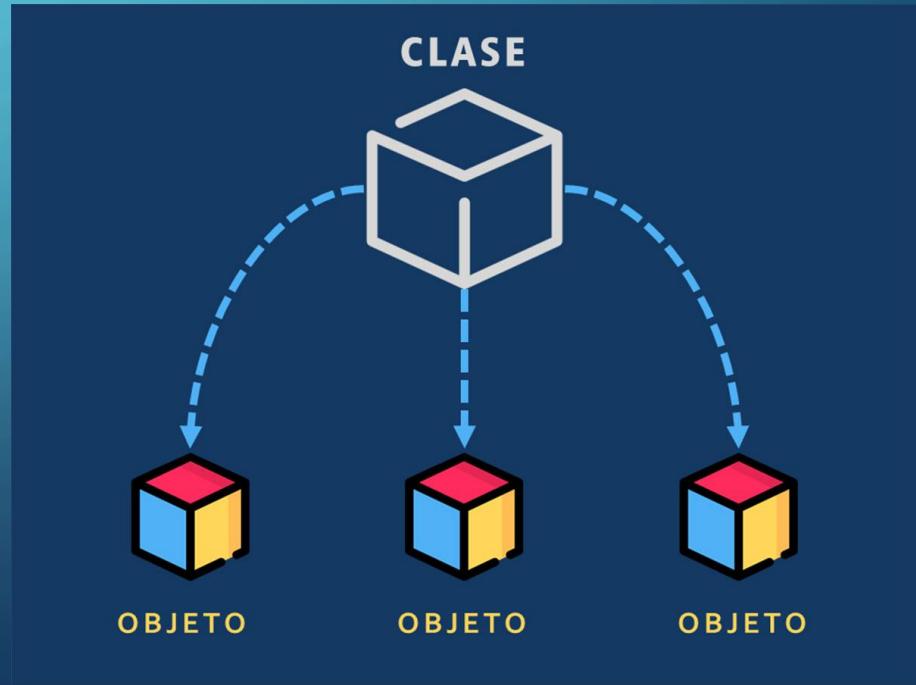
Ejemplo: persona1 = new Persona("Carlos", 30).

Atributos → Son las propiedades o datos que tiene un objeto.

Ejemplo: nombre, edad.

Métodos → Son las funciones o acciones que un objeto puede realizar.

Ejemplo: caminar(), hablar().



# ABSTRACCIÓN

- En Programación Orientada a Objetos (POO), la abstracción es el principio que consiste en enfocarse en los aspectos esenciales de un objeto y ocultar los detalles innecesarios.
- Es decir:  Nos quedamos con lo que un objeto hace y no necesariamente con cómo lo hace.
- Ejemplo: Imagina la clase Auto : A ti como usuario te interesa poder encenderlo, acelerar o frenar.
- No necesitas saber cómo funcionan internamente los pistones, la inyección de gasolina o la transmisión

# ENCAPSULACIÓN, HERENCIA, POLIMORFISMO

- Encapsulación: Proteger los datos internos de un objeto y exponer solo lo necesario.
- Herencia: Permitir que una clase (hija) herede atributos y métodos de otra (padre).
- Polimorfismo: Permitir que un mismo método se comporte de distintas maneras según el contexto

# MODIFICADORES DE ACCESO

- En Java los modificadores de acceso controlan la visibilidad de clases, atributos y métodos. Los principales son:
- **public** → accesible desde cualquier clase.
- **private** → accesible solo dentro de la misma clase.
- **protected** → accesible dentro del mismo paquete y por clases hijas (herencia).
- **default (sin palabra clave)** → accesible solo dentro del mismo paquete

# CONSTRUCTORES

- Un constructor es un método especial de una clase que se ejecuta automáticamente cuando se crea un objeto (new). Sirve para inicializar los atributos de la clase (dar valores iniciales).
- Se llama igual que la clase.
- No tiene tipo de retorno (ni siquiera void).
- Se ejecuta solo una vez, en el momento de crear el objeto con new.

# CONSTRUCTORES

- **public constructor**
  - Puedes crear objetos desde cualquier clase y cualquier paquete.
  - `public Persona(String nombre, int edad) { ... }`
  - El más común para clases que quieras instanciar libremente.
- **private constructor**
  - Solo se puede usar dentro de la misma clase.
  - Muy usado en patrones de diseño como Singleton (para que no cualquiera cree objetos).
  - `private Persona(String nombre, int edad) { ... }`
- **protected constructor**
  - Solo se puede usar dentro del mismo paquete o en clases hijas (subclases).
  - `protected Persona(String nombre, int edad) { ... }`
- **default (sin modificador)**
  - Solo accesible desde el mismo paquete.
  - `Persona(String nombre, int edad) { ... } // paquete-privado`

# @Override

- En Java, `@Override` es una anotación que se usa cuando una subclase sobrescribe (`override`) un método de su clase padre (o de una interfaz).
- ¿Qué significa sobrescribir?
  - Una clase hija hereda métodos de la clase padre. Si necesita que ese método tenga un comportamiento distinto, puede reescribirlo en la subclase. Ahí entra `@Override`, para indicar explícitamente que estás sobrescribiendo un método heredado.
- Cosas importantes de `@Override`:
  - No es obligatorio, pero es buena práctica porque:
    - Si escribes mal el nombre del método, el compilador te marca error.
    - Garantiza que realmente estás sobrescribiendo algo existente.
    - Se usa con métodos heredados, nunca con atributos ni constructores.
    - También aplica cuando implementas un método de una interfaz:

# MIEMBROS ESTÁTICOS

- Un miembro estático (atributo o método) pertenece a la clase en lugar de a los objetos.
- No necesitas crear una instancia (new) para usarlos.
- Todos los objetos de esa clase comparten el mismo valor.

# CONSTANTES

- Una constante en Java es una variable cuyo valor no puede cambiar después de asignarse.
- Se declara con final.
- Si además es static, será compartida por toda la clase y se suele escribir en MAYÚSCULAS.

# TIPOS DE DATO PRIMITIVO

- Los tipos primitivos son los tipos de datos básicos que Java maneja directamente.
- No son objetos.
- Se almacenan de manera eficiente en memoria.
- Son inmutables (no tienen métodos asociados).

# TIPOS DE DATOS PRIMITIVO

Tipo	Tamaño	Valores ejemplo
byte	8 bits	-128 a 127
short	16 bits	-32,768 a 32,767
int	32 bits	-2,147,483,648 a 2,147,483,647
long	64 bits	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
float	32 bits	3.14, 0.5, -2.7
double	64 bits	3.14159, -0.0001
char	16 bits	'a', 'Z', '9'
boolean	1 bit	true / false

# CLASES CONTENEDORAS WRAPPER CLASS

- Cada tipo primitivo tiene su clase contenedora (wrapper class) en Java.
- Permite tratar los primitivos como objetos.
- Se usan para colecciones (ArrayList, HashMap) que solo aceptan objetos.
- Proporcionan métodos útiles para conversión, comparación, etc.

# CLASES CONTENEDORAS CORRESPONDIENTES

Primitivo

byte

short

int

long

float

double

char

boolean

Wrapper Class

Byte

Short

Integer

Long

Float

Double

Character

Boolean

# DIFERENCIAS CLAVE

## Característica

Es objeto

Tamaño

Métodos

Uso en colecciones

## Primitivo

No

Fijo

Ninguno

No

## Wrapper Class

Sí

Igual al primitivo

Muchos métodos

Sí (ArrayList, HashMap, etc.)

Primitivos → datos básicos y eficientes.

Wrapper Classes → objetos que “envuelven” los primitivos para poder usar métodos y colecciones.

# STRING

- Representa **cadenas de texto**.
- Es **inmutable**: una vez creada, no se puede cambiar.
- No es **primitiva**, es **objeto**.
- Puedes usar muchos métodos útiles: `.length()`, `.substring()`, `.toUpperCase()`, etc.
  - `String nombre = "Carlos";`
  - `System.out.println(nombre.length()); // 6`
  - `System.out.println(nombre.toUpperCase()); // "CARLOS"`

# BIGDECIMAL

- Se usa para números decimales con gran precisión, especialmente en dinero o cálculos científicos.
- Permite evitar errores de redondeo que ocurren con float o double.
- Es inmutable y tiene métodos para operaciones matemáticas.
  - `import java.math.BigDecimal;`
  - `BigDecimal precio = new BigDecimal("19.99");`
  - `BigDecimal cantidad = new BigDecimal("3");`
  - `BigDecimal total = precio.multiply(cantidad);`
  - `System.out.println(total); // 59.97`

# BIGINTEGER

- Igual que BigDecimal pero para enteros muy grandes, más allá del límite de long.
- También es inmutable y tiene métodos como .add(), .multiply(), .pow().
  - import java.math.BigInteger;
  - BigInteger a = new BigInteger("12345678901234567890");
  - BigInteger b = new BigInteger("98765432109876543210");
  - BigInteger suma = a.add(b);
  - System.out.println(suma); // 11111111101111111100

# LOCALDATE

- Representa solo fechas (día, mes, año) sin hora.
- Forma parte de la API de Java 8 Date & Time (`java.time`).
- Es inmutable y tiene métodos como `.plusDays()`, `.minusMonths()`
  - `import java.time.LocalDate;`  
`LocalDate hoy = LocalDate.now();`
  - `LocalDate mañana = hoy.plusDays(1);`
  - `System.out.println(hoy); // 2025-09-22 (por ejemplo)`
  - `System.out.println(mañana); // 2025-09-23`

# LOCALDATETIME

- Representa fecha y hora (año, mes, día, hora, minuto, segundo)
- También inmutable, con métodos similares a LocalDate.
  - Import `java.time.LocalDateTime;`
  - `LocalDateTime ahora = LocalDateTime.now();`
  - `LocalDateTime masTarde = ahora.plusHours(3);`
  - `System.out.println(ahora);`
  - `System.out.println(masTarde);`

# RESUMEN

Tipo	Es primitivo	Qué representa	Características importantes
String	✗ No	Texto	Inmutable, métodos útiles
BigDecimal	✗ No	Decimal de alta precisión	Inmutable, operaciones precisas
BigInteger	✗ No	Enteros muy grandes	Inmutable, operaciones grandes
LocalDate	✗ No	Fecha sin hora	Inmutable, manipulación de fechas
LocalDateTime	✗ No	Fecha y hora	Inmutable, manipulación completa de tiempo

# LISTAS

- En Java, una lista es una estructura de datos que permite almacenar múltiples elementos en un orden específico y permitir duplicados.
- Pertenece al paquete `java.util`.
- Forma parte de la colección `List` dentro del framework de colecciones.
- Puedes acceder a los elementos por índice (posición).
- Características principales
  - Ordenada → mantiene el orden de inserción.
  - Puede contener duplicados → puedes tener elementos repetidos.
  - Dinámica → el tamaño puede crecer o reducirse automáticamente.
- Métodos más importantes:
  - `add(elemento)` → agregar un elemento
  - `get(indice)` → obtener un elemento
  - `remove(indice o elemento)` → eliminar
  - `size()` → obtener tamaño
  - `contains(elemento)` → verificar existencia
  - `clear()` → vaciar la lista

# LISTAS

- Tipos de listas comunes en Java

Clase

ArrayList

LinkedList

Vector

Características principales

Implementación basada en **array dinámico**, acceso rápido por índice.

Implementación basada en **lista doblemente enlazada**, más eficiente para inserciones/eliminaciones en medio de la lista.

Similar a ArrayList, pero **sincronizado** (hilos), menos usado hoy.

- Diferencias con el array

Característica

Tamaño

Métodos

Orden

Duplicados

Array

Fijo

Limitados

Sí

Sí

Lista (List)

Dinámico

Muchos métodos útiles (add, remove, contains, etc.)

Sí

Sí

# HASHMAP

- Un HashMap es una colección que almacena pares clave-valor (key-value).
- Forma parte del framework de colecciones (java.util).
- Permite acceso rápido a los valores mediante su clave.
- No mantiene un orden específico de los elementos.
- Características principales
  - Cada clave es única; los valores pueden repetirse.
  - Permite valores null y una clave null (solo una).
  - Muy eficiente para búsquedas rápidas, gracias al hashing.
  - Clases relacionadas:
    - HashMap → no sincronizado, rápido.
    - LinkedHashMap → mantiene el orden de inserción.
    - TreeMap → mantiene los elementos ordenados por clave.

# HASHMAP

## Métodos Más comunes

Método	Descripción
<code>put(key, value)</code>	Agrega un par clave-valor
<code>get(key)</code>	Obtiene el valor asociado a la clave
<code>remove(key)</code>	Elimina un par clave-valor
<code>containsKey(key)</code>	Verifica si existe la clave
<code>containsValue(value)</code>	Verifica si existe el valor
<code>keySet()</code>	Devuelve un conjunto (Set) de claves
<code>values()</code>	Devuelve colección de valores
<code>entrySet()</code>	Devuelve un conjunto de pares clave-valor

HashMap: colección clave-valor, clave única, acceso rápido.

Ideal cuando necesitas buscar, actualizar o eliminar elementos por clave de manera eficiente.

Si quieres mantener orden, usa LinkedHashMap.

Si quieres orden natural por clave, usa TreeMap.

# CLASES ABSTRACTAS

- Una clase que no se puede instanciar directamente (`new` no funciona).
- Puede tener:
  - Métodos abstractos (sin implementación)
  - Métodos concretos (con implementación)
  - Atributos
- Sirve para compartir código común entre varias clases relacionadas.
- Puedes tener atributos y métodos con implementación.
- Puedes heredar solo una clase abstracta (Java no permite herencia múltiple).

# INTERFACES

- Una plantilla pura que define solo métodos (y constantes).
- Todas las clases que implementen la interfaz deben dar implementación a esos métodos.
- Desde Java 8, pueden tener métodos default con implementación.
- Permiten herencia múltiple, porque una clase puede implementar varias interfaces.
- Una clase puede implementar varias interfaces.
- Las interfaces definen un contrato obligatorio.
- No pueden tener atributos de instancia normales, solo constantes (public static final).

# DIFERENCIAS

## Característica

Instanciable

Métodos concretos

Métodos abstractos

Atributos

Herencia múltiple

## Uso típico

Clase abstracta → código común + abstracción, herencia simple.

Interfaz → contrato puro, herencia múltiple posible, ideal para comportamiento común en clases no relacionadas.

## Clase abstracta

No

Sí

Sí

Sí (variables normales)

Solo una clase

Compartir código común entre  
clases relacionadas

## Interfaz

No

Desde Java 8 (default)

Sí (implícitos)

Solo constantes (public static final)

Puede implementar muchas

Definir contrato que muchas  
clases diferentes pueden cumplir

# EXCEPCIONES

- Una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal del programa.
- Puede ser causada por errores como: división entre cero, acceso a un índice inválido en un array, archivos no encontrados, etc.
- Permite manejar errores sin que el programa se detenga abruptamente.
- Jerarquía básica
- Todas las excepciones en Java derivan de la clase Throwable.
  - Exception → errores que el programa puede manejar (checked exceptions y runtime exceptions).
  - Error → errores graves del sistema (generalmente no se manejan).

# MANEJO DE EXCEPCIONES

- Se usa **try-catch-finally** para capturar y manejar excepciones.
- Otras palabras clave
- **throw** → lanza una excepción manualmente.
  - `throw new IllegalArgumentException("Valor inválido");`
- **throws** → indica que un método puede lanzar una excepción.
  - `public void leerArchivo() throws IOException { ... }`
- **finally** → bloque que se ejecuta siempre, haya o no excepción.
- **try-with-resources** → para manejar recursos automáticamente (archivos, conexiones).
- Las excepciones evitan que el programa se detenga abruptamente.
- Se clasifican en checked y unchecked.
- Se manejan con **try-catch-finally**.
- Permiten informar y reaccionar a errores de manera controlada.