

INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS

1. CLASES I

1.1. DEFINICIÓN

- ❖ Una clase en POO es como un "molde" para crear objetos.
- ❖ Define un *tipo de dato que encapsula datos y comportamientos (métodos)*.
- ❖ Es una plantilla desde la cual se crean objetos.
 - Cada nuevo objeto creado a partir de la clase se denomina **instancia**.
 - A veces los términos objeto e instancia se usan indistintamente ya que **cuando instanciamos una clase estamos creando un objeto**.

1. CLASES II

1.2. COMPONENTES I

1.2.1. ATRIBUTOS O PROPIEDADES

❖ Los **atributos** o **propiedades** son variables que definen las propiedades o características del tipo de objeto que la clase representa.

▫ **EJEMPLO:** En una clase **Coche**, atributos podrían ser **marca, modelo, color**.

1.2.2. MÉTODOS

❖ Los **métodos** son funciones definidas dentro de una clase que describen las acciones o comportamientos que los objetos de esa clase pueden realizar.

▫ **EJEMPLO:** En la clase **Coche**, un método podría ser **arrancar()**.

1. CLASES III

1.2. COMPONENTES II

EJEMPLO

❖ Esto es solamente el “molde” para crear objetos Coche, pero aún no existe ningún objeto coche. Para crear un objeto Coche tendremos que usar el **operador new** de forma similar a como hacíamos con los arrays, pero eso lo veremos más adelante.

```
public class Coche {  
    // Atributos  
    String marca;  
    String modelo;  
    String color;  
  
    // Método  
    public void arrancar() {  
        System.out.println("El coche está arrancando");  
    }  
}
```

1. CLASES IV

1.3. ATRIBUTOS Y MÉTODOS DE UNA CLASE FREnte A LOS DE UN OBJETO I

❖ Recordemos que los atributos y/o los métodos de una clase pueden ser estáticos, de tal forma que pertenecerán a la clase y no a cada objeto que se cree a partir de ella, es decir, los atributos y métodos estáticos pertenecen a la clase en sí y no a instancias individuales. Por tanto, se pueden acceder sin necesidad de crear un objeto de la clase.

❖ **EJEMPLO:** (Ver siguiente diapositiva) En este ejemplo, **cantidadCoches** es un atributo estático que cuenta el número de instancias de **Coche** creadas. El constructor **Coche(String marca)** inicializa el atributo **marca** y actualiza **cantidadCoches**. Además, se muestra un método estático **getCantidadCoches** que permite acceder al número de coches sin necesidad de una instancia de **Coche**. Tranquilidad: Más adelante hablaremos de qué son y para qué se usan los constructores.

1. CLASES V

1.3. ATRIBUTOS Y MÉTODOS DE UNA CLASE FREnte A LOS DE UN OBJETO II

```
public class Coche {  
    // Atributo estático  
    static int cantidadCoches;  
  
    // Atributo de instancia  
    String marca;  
  
    // Constructor  
    public Coche(String marca) {  
        this.marca = marca;  
        cantidadCoches++;  
    }  
  
    // Método de instancia  
    public void arrancar() {  
        System.out.println("El coche " + marca + " está arrancando");  
    }  
  
    // Método estático  
    public static int getCantidadCoches() {  
        return cantidadCoches;  
    }  
}
```

2. OBJETOS I

2.1. DEFINICIÓN

- ◆ Un **objeto** es una instancia de una clase. Al crear un objeto, se reserva espacio en la memoria para almacenar un ente que tiene los atributos y puede realizar los métodos definidos en su clase.

2.2. CREACIÓN (OPERADOR new) I

- ◆ Se usa el operador **new** para crear un objeto a partir de una clase. Esto inicializa una nueva instancia de esa clase.
- ◆ La sintaxis general es **NombreClase nombreObjeto = new NombreClase();**

2. OBJETOS II

2.2. CREACIÓN (OPERADOR new) II

- ◆ **EJEMPLO:** En este ejemplo, **miCoche** es un objeto de la clase **Coche**. Se crea con **new Coche()** y luego se llama al método **arrancar()** que está definido en la clase **Coche**.

```
public class Main {  
    public static void main(String[] args) {  
        // Creación de un objeto de la clase Coche  
        Coche miCoche = new Coche();  
  
        // Uso de un método del objeto  
        miCoche.arrancar();  
    }  
}
```

3. CONSTRUCTORES I

3.1. DEFINICIÓN

- ❖ Los **constructores** en la Programación Orientada a Objetos (POO) son métodos especiales que se utilizan para inicializar los objetos de una clase. Son fundamentales en cualquier lenguaje de programación que soporte POO, como Java.
- ❖ Los constructores permiten controlar cómo se inicializan los objetos y garantizan que los objetos se creen en un estado consistente y válido. Quedará claro con el ejemplo.

3. CONSTRUCTORES II

3.2. CARACTERÍSTICAS

- ❖ **Nombre del Constructor:** Tiene el mismo nombre que la clase a la que pertenece.
- ❖ **Inicialización de Objetos:** Se ejecutan automáticamente en el momento de la creación de un objeto (cuando se usa **new**).
- ❖ **No tienen Tipo de Retorno:** A diferencia de los métodos ordinarios, los constructores no especifican un tipo de retorno, ni siquiera void.
- ❖ **Sobrecarga de Constructores:** Una clase puede tener varios constructores, cada uno con diferentes parámetros. Esto se llama "sobrecarga de constructores" y permite inicializar objetos de diferentes maneras.
- ❖ **Constructor por Defecto:** Si no se define explícitamente un constructor, la mayoría de los lenguajes de programación (incluido Java) proporcionan un constructor por defecto sin parámetros que inicializa el objeto con valores predeterminados.

3. CONSTRUCTORES III

3.3. EJEMPLOS I

3.3.1. CLASE A USAR EN EL EJEMPLO

- ❖ Los ejemplos de constructores que vamos a ver se basan en esta definición de la clase Coche:

```
public class Coche {  
    String marca;  
    String modelo;  
  
    // Constructor sin parámetros  
    public Coche() {  
        marca = "Desconocido";  
        modelo = "Desconocido";  
    }  
  
    // Constructor con parámetros  
    public Coche(String marca, String modelo) {  
        this.marca = marca;  
        this.modelo = modelo;  
    }  
}
```

3. CONSTRUCTORES IV

3.3. EJEMPLOS II

3.3.2. EJEMPLO DE USO DE CONSTRUCTOR SIN PARÁMETROS

- ❖ Crea un objeto **Coche** con marca y modelo por defecto.

```
Coche miCoche = new Coche();
```

3.3.3. EJEMPLO DE USO DE CONSTRUCTOR CON PARÁMETROS

- ❖ Crea un objeto **Coche** con marca y modelo específicos.

```
Coche miOtroCoche = new Coche("Toyota", "Corolla");
```

SE PROPONE LA REALIZACIÓN DE LA TAREA 1

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA I

4.1. BIBLIOTECA DE OBJETOS EN JAVA I

- ❖ La biblioteca de objetos en Java, conocida como **Java Class Library (JCL)**, es una extensa colección de clases e interfaces predefinidas que forman parte del entorno de ejecución de Java. La JCL proporciona una amplia gama de funcionalidades, desde operaciones básicas de entrada/salida hasta interfaces gráficas de usuario y acceso a bases de datos.

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA II

4.1. BIBLIOTECA DE OBJETOS EN JAVA II

4.1.1. COMPONENTES CLAVE DE LA JCL

- ❖ **java.lang:** Este paquete es fundamental, ya que contiene clases que son esenciales para el diseño del lenguaje Java. Incluye clases como **Object**, que es la superclase de todas las clases en Java, y **String**, además de clases para tipos de datos primitivos (envoltorios como **Integer**, **Double**, etc.), y el sistema de manejo de excepciones.
- ❖ **java.util:** Ofrece clases de utilidades que son comúnmente usadas. Incluye colecciones (como listas, mapas y conjuntos), frameworks, y clases para fecha y hora, eventos, manipulación de tokens, y aleatorización.
- ❖ **java.io:** Proporciona clases e interfaces para el sistema de entrada/salida en Java. Esto incluye la lectura y escritura de datos a través de flujos (streams), archivos y directorios. Es fundamental para cualquier operación de E/S, como trabajar con archivos o comunicarse a través de la red.

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA III

4.1. BIBLIOTECA DE OBJETOS EN JAVA III

4.1.2. ESTRUCTURA Y RELACIONES

- ❖ La estructura de la JCL es jerárquica y orientada a objetos. Todas las clases en Java heredan de la clase **Object** directa o indirectamente, lo que significa que comparten ciertos métodos comunes.
- ❖ Las clases e interfaces en estos paquetes están interrelacionadas. Por ejemplo, muchas clases en **java.io** utilizan tipos de **java.lang**, y las colecciones en **java.util** a menudo se utilizan junto con **java.io** para el procesamiento de datos.
- ❖ Además de estos paquetes, la JCL contiene numerosos otros paquetes para tareas específicas como **java.net** para la red, **java.sql** para el acceso a bases de datos, y **javax.swing** para interfaces gráficas de usuario, entre otros.

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA IV

4.1. BIBLIOTECA DE OBJETOS EN JAVA IV

4.1.3. IMPORTANCIA EN LA PROGRAMACIÓN JAVA

- ❖ La JCL proporciona un conjunto estándar de clases y métodos que se pueden utilizar para desarrollar aplicaciones Java. Esto permite a los desarrolladores centrarse en la lógica específica de su aplicación sin tener que reinventar soluciones comunes.
- ❖ Facilita la portabilidad y la interoperabilidad, ya que las aplicaciones desarrolladas con estas bibliotecas pueden ejecutarse en cualquier máquina virtual de Java (JVM) sin necesidad de modificar el código.
- ❖ En resumen, la Java Class Library es esencial para el desarrollo en Java, proporcionando una base sólida y un conjunto extenso de herramientas para los desarrolladores.

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA V

4.1. BIBLIOTECA DE OBJETOS EN JAVA V

4.1.4. LA CLASE OBJECT

❖ La clase **Object** en Java es la superclase de todas las clases en el lenguaje. Cualquier clase que crees en Java hereda de **Object** de manera implícita si no se especifica otra superclase. Esto significa que cualquier objeto en Java tiene al menos los métodos que **Object** define.

❖ La clase **Object** proporciona los siguientes métodos, cada uno con su propia funcionalidad específica:

Método	Descripción
<code>public final Class<?> getClass()</code>	Retorna la instancia de "Class" que representa la clase del objeto.
<code>public int hashCode()</code>	Retorna un código hash para el objeto, utilizado en estructuras de datos hash.
<code>public boolean equals(Object obj)</code>	Compara si el objeto actual es "igual" al objeto proporcionado según la implementación.
<code>protected Object clone()</code>	Crea y retorna una copia del objeto si la clase implementa "Cloneable".
<code>public String toString()</code>	Retorna una representación en cadena del objeto.
<code>public final void notify()</code>	Despierta a un solo hilo que esté esperando en el monitor del objeto.
<code>public final void notifyAll()</code>	Despierta a todos los hilos que están esperando en el monitor del objeto.
<code>public final void wait(long timeout)</code>	Hace que el hilo actual espere hasta que otro hilo invoque "notify()" o "notifyAll()" para este objeto o que haya transcurrido un cierto tiempo de espera.
<code>public final void wait(long timeout, int nanos)</code>	Hace que el hilo actual espere con más precisión en tiempo.
<code>public final void wait()</code>	Hace que el hilo actual espere indefinidamente hasta que otro hilo invoque "notify()" o "notifyAll()" para este objeto.
<code>protected void finalize()</code>	Método llamado por el recolector de basura antes de que el objeto sea destruido, ahora obsoleto desde Java 9.

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA VI

4.1. BIBLIOTECA DE OBJETOS EN JAVA VI

4.1.5. CLASES ENVOLTORIO (WRAPPER CLASSES) I

❖ Las clases envoltorio o clases de envoltura (Wrapper Classes) en Java son un grupo de clases utilizadas para convertir tipos de datos primitivos (como **int**, **char**, **double**, etc.) en objetos. La necesidad de estas clases surge porque las estructuras de datos en la API de colecciones de Java solo trabajan con objetos y no con tipos primitivos. Además, las clases envoltorio proporcionan una serie de métodos útiles para operaciones como la conversión, comparación, y otras operaciones con valores primitivos.

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA VII

4.1. BIBLIOTECA DE OBJETOS EN JAVA VII

4.1.5. CLASES ENVOLTORIO (WRAPPER CLASSES) II

❖ Cada tipo primitivo en Java tiene una clase envoltorio correspondiente:

- **byte** - **Byte**
- **short** - **Short**
- **int** - **Integer**
- **long** - **Long**
- **float** - **Float**
- **double** - **Double**
- **char** - **Character**
- **boolean** - **Boolean**

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA VIII

4.1. BIBLIOTECA DE OBJETOS EN JAVA VIII

4.1.5. CLASES ENVOLTORIO (WRAPPER CLASSES) III

❖ **EJEMPLO 1:** **Integer**

❖ La clase **Integer** se utiliza para envolver un valor del tipo primitivo **int** en un objeto. Un ejemplo de su uso es en las colecciones, como **ArrayList<Integer>**, donde no se pueden usar tipos primitivos.

```
int i = 10;  
Integer integerObject = Integer.valueOf(i); // envolviendo  
int intValue = integerObject.intValue(); // extrayendo
```

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA IX

4.1. BIBLIOTECA DE OBJETOS EN JAVA IX

4.1.5. CLASES ENVOLTORIO (WRAPPER CLASSES) IV

❖ EJEMPLO 2: Double

- ❖ De manera similar, la clase **Double** envuelve un valor del tipo primitivo **double** en un objeto. Esto permite que los valores **double** se puedan usar donde se requieren objetos, como en las colecciones genéricas.

```
double d = 10.5;  
Double doubleObject = Double.valueOf(d); // envolviendo double  
double doubleValue = doubleObject.doubleValue(); // extrayendo
```

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA X

4.1. BIBLIOTECA DE OBJETOS EN JAVA X

4.1.5. CLASES ENVOLTORIO (WRAPPER CLASSES) V

- ❖ Estas clases también proporcionan constantes útiles como **MIN_VALUE**, **MAX_VALUE**, y métodos para convertir a y desde cadenas (**String**), como **parseInt**, **parseDouble**, y **toString**.

- ❖ A partir de Java 5, la característica de autoboxing y unboxing maneja automáticamente la conversión entre los tipos primitivos y sus clases envoltorio correspondientes, haciendo que sea más fácil trabajar con ellos en situaciones donde se necesitan objetos. Por ejemplo, puedes asignar directamente un int a un Integer y viceversa sin necesidad de llamar explícitamente a métodos como valueOf o intValue.

SE PROPONE LA REALIZACIÓN DE LA TAREA 2

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA XI

4.2. JAVA versus JAVAX I

4.2.1. PROPÓSITO Y CONTENIDO

- ❖ **java:** El prefijo **java** se utiliza para las clases y paquetes fundamentales en el lenguaje Java. Estos incluyen funcionalidades básicas del lenguaje, como manipulación de cadenas, matemáticas, entrada/salida, utilidades de colección, y la API básica de red. Estos paquetes son considerados esenciales para la programación en Java.
- ❖ **javax:** Inicialmente, el prefijo **javax** se utilizaba para paquetes de extensión de la API de Java. Históricamente, incluía APIs que no eran parte del núcleo del lenguaje Java, pero que eran proporcionadas por Sun Microsystems como extensiones estándar. Con el tiempo, algunos de estos paquetes se han vuelto esenciales y ampliamente utilizados, aunque mantienen el prefijo javax. Ejemplos incluyen APIs para interfaces gráficas de usuario (como Swing en **javax.swing**), APIs para correo electrónico, y servicios web.

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA XII

4.2. JAVA versus JAVAX II

4.2.2. HISTORIA Y EVOLUCIÓN

- ❖ Originalmente, **javax** se creó para contener componentes que no formaban parte del JDK estándar pero que podían ser agregados por los desarrolladores si se necesitaban. Con el tiempo, algunos de estos paquetes se han integrado más estrechamente en el JDK, aunque conservan el prefijo **javax** por razones de compatibilidad y legado.
- ❖ Algunas funcionalidades que comenzaron en **javax** han sido tan exitosas y fundamentales que eventualmente se trasladaron al paquete **java**. Sin embargo, esto no es lo común y la mayoría de los paquetes **javax** permanecen bajo ese nombre.

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA XIII

4.2. JAVA versus JAVAX III

4.2.3. USO Y COMPATIBILIDAD

- ❖ Los paquetes **java** son considerados como parte integral de cualquier implementación del entorno de ejecución de Java (JRE) y están disponibles por defecto. Son esenciales para la programación básica en Java.
- ❖ Los paquetes **javax**, aunque ampliamente disponibles y utilizados, pueden ser más específicos para ciertos tipos de aplicaciones y no necesariamente se consideran fundamentales para todas las aplicaciones Java.
- ❖ En resumen, **java** y **javax** representan dos conjuntos de bibliotecas dentro del ecosistema Java, con **java** centrado en funcionalidades fundamentales y **javax** incluyendo extensiones y funcionalidades especializadas que originalmente no eran parte del núcleo de Java.

4. BIBLIOTECAS DE CLASES Y OBJETOS EN JAVA XIV

4.2. JAVA versus JAVAX IV

4.2.4. TABLA RESUMEN

- ❖ En esta tabla solamente aparecen las clases más representativas de **java** y **javax**.

Paquete Principal	Subpaquete	Descripción
java	lang	Contiene clases fundamentales como Object, String, clases de envoltura para tipos primitivos y el sistema de manejo de excepciones.
	util	Proporciona colecciones, frameworks, y clases para fecha y hora, eventos, y aleatorización.
	io	Incluye clases e interfaces para el sistema de entrada/salida en Java.
	math	Ofrece clases para operaciones matemáticas avanzadas y precisión arbitraria.
	awt	Contiene todas las clases para crear interfaces de usuario e imágenes.
	sql	Proporciona el acceso a bases de datos mediante JDBC.
	net	Incluye clases para desarrollar aplicaciones de red.
	javax	Ofrece un conjunto de componentes para interfaces gráficas de usuario más sofisticadas que AWT.
	imageio	Proporciona clases para leer y escribir imágenes en formatos comunes.
	crypto	Contiene clases e interfaces para criptografía.
	sql	Amplía las funcionalidades de JDBC para el acceso a bases de datos.

SE PROPONE LA REALIZACIÓN DE LA TAREA 3

5. CONCEPTOS FUNDAMENTALES DE LA POO I

5.1. PRINCIPIOS BÁSICOS I

❖ Los principios básicos de la programación orientada a objetos son **cuatro** y cada uno aporta una dimensión importante al diseño y desarrollo de software:

- **Encapsulamiento:** Se refiere a la agrupación de datos y los métodos que los manipulan dentro de una clase, ocultando los detalles internos del funcionamiento de la clase al mundo exterior. Esto ayuda a proteger el estado interno del objeto y promueve la modularidad.
- **Abstracción:** Implica enfocarse en las características esenciales de un objeto, ignorando las menos importantes o accidentales. En la práctica, se traduce en la creación de clases que representan conceptos abstractos y definiciones de interfaces. La abstracción permite trabajar a un nivel más genérico, facilitando así la gestión de la complejidad.

5. CONCEPTOS FUNDAMENTALES DE LA POO II

5.1. PRINCIPIOS BÁSICOS II

- **Herencia:** Permite que una clase (subclase) herede propiedades y métodos de otra clase (superclase). La herencia promueve la reutilización del código y puede ayudar a establecer una jerarquía de clases dentro de la aplicación.
- **Polimorfismo:** Significa 'muchas formas' y en la programación orientada a objetos, se refiere a la capacidad de una variable de tipo base para referenciar objetos de diferentes tipos derivados y así, ejecutar sus métodos específicos, a pesar de la referencia común. Esto permite que se escriban programas más flexibles y extensibles.

SE PROPONE LA REALIZACIÓN DE LA TAREA 4

❖ Estos principios son la base para crear programas estructurados, eficientes y fáciles de mantener, facilitando el desarrollo de software complejo y de alta calidad.

5. CONCEPTOS FUNDAMENTALES DE LA POO III

5.2. ENCAPSULAMIENTO Y VISIBILIDAD I

5.2.1. VISIBILIDAD DE LOS ATRIBUTOS DE UNA CLASE I

❖ La **visibilidad de los métodos** fue tratada con detalle en el [punto 1.1.6 del tema 2.](#)

- ❖ La elección del modificador de acceso para los atributos de una clase es una decisión importante en el diseño de software ya que afecta directamente a la encapsulación, la seguridad y el acoplamiento de las clases. Es fundamental utilizar estos modificadores de manera efectiva para crear un código robusto y mantenable.
- ❖ La **visibilidad** de los atributos en una clase Java determina cómo y desde dónde se puede acceder a estos atributos. Java proporciona varios niveles de visibilidad a través de modificadores de acceso. Estos son: **private**, **default** (sin modificador), **protected**, y **public**. A continuación, vamos a ver la utilidad de cada uno de ellos acompañado de un ejemplo.

5. CONCEPTOS FUNDAMENTALES DE LA POO IV

5.2. ENCAPSULAMIENTO Y VISIBILIDAD II

5.2.1. VISIBILIDAD DE LOS ATRIBUTOS DE UNA CLASE II

❖ Privado (**private**)

- **Descripción:** Cuando un atributo es declarado como **private**, solo puede ser accedido dentro de la clase donde se declara.
- **Uso típico:** Se utiliza para ocultar los datos internos de la clase y forzar el acceso a estos datos a través de métodos públicos (**getters** y **setters**). Los métodos de tipo **getter** y **setter** usan la sintaxis **getNombreAtributo** y **setNombreAtributo**, respectivamente.

```
public class Coche {  
    private String matricula; // Solo accesible dentro de la clase Coche  
  
    public String getMatricula() {
```

5. CONCEPTOS FUNDAMENTALES DE LA POO V

5.2. ENCAPSULAMIENTO Y VISIBILIDAD III

5.2.1. VISIBILIDAD DE LOS ATRIBUTOS DE UNA CLASE III

❖ Sin modificador (**Default**)

- **Descripción:** Si un atributo no tiene modificador, se le asigna el acceso por defecto. Estos atributos son accesibles solo dentro de las clases del mismo paquete.
- **Uso típico:** Utilizado cuando se quiere permitir el acceso a nivel de paquete, manteniendo la encapsulación fuera de este.

```
class Motor {  
    int potencia; // Accesible dentro de cualquier clase en el mismo paquete  
  
    int getPotencia() {
```

5. CONCEPTOS FUNDAMENTALES DE LA POO VI

5.2. ENCAPSULAMIENTO Y VISIBILIDAD IV

5.2.1. VISIBILIDAD DE LOS ATRIBUTOS DE UNA CLASE IV

❖ Protegido (**protected**)

- **Descripción:** Los atributos **protected** son accesibles dentro de la misma clase, en clases del mismo paquete, y en subclases incluso si estas están en diferentes paquetes.
- **Uso típico:** Es útil cuando se espera que una propiedad sea accesible en las clases heredadas.

```
public class Vehiculo {  
    protected int ruedas; // Accesible en subclases y en el mismo paquete  
  
    protected int getRuedas() {
```

5. CONCEPTOS FUNDAMENTALES DE LA POO VII

5.2. ENCAPSULAMIENTO Y VISIBILIDAD V

5.2.1. VISIBILIDAD DE LOS ATRIBUTOS DE UNA CLASE V

❖ Público (**public**)

- **Descripción:** Los atributos **public** son accesibles desde cualquier parte del programa, siempre que se tenga una instancia de la clase.
- **Uso típico:** Aunque no es una práctica común exponer directamente atributos como públicos (por cuestiones de encapsulamiento), se pueden usar para constantes o en casos donde el control directo es necesario.

```
public class Configuracion {  
    public static final String VERSION = "1.0"; // Accesible globalmente  
}
```

5. CONCEPTOS FUNDAMENTALES DE LA POO VIII

5.3. RELACIONES ENTRE CLASES I

❖ En este apartado se incluyen varios conceptos clave que describen cómo las clases interactúan y se asocian entre sí. Estas relaciones son fundamentales para el diseño y la arquitectura de sistemas de software. Los principales tipos de relaciones son:

- **Herencia:** Una clase (subclase o clase derivada) hereda atributos y métodos de otra clase (superclase o clase base). La herencia establece una relación "es un/a" (is-a), donde la subclase es un tipo específico de la superclase.
- **Asociación:** Indica que una clase tiene una relación con otra clase. Esta relación es generalmente de tipo "usa un/a" (uses-a) o "tiene un/a" (has-a). Por ejemplo, una clase **Coche** puede tener una asociación con la clase **Motor**.

5. CONCEPTOS FUNDAMENTALES DE LA POO VIII

5.3. RELACIONES ENTRE CLASES I

- **Agregación:** Es un tipo especial de asociación que representa una relación "todo-parte" donde las partes pueden existir independientemente del todo. Por ejemplo, un **Coche** (todo) puede tener varios **Rueda** (partes), pero las **Rueda** pueden existir sin el **Coche**.
- **Composición:** También es una relación "todo-parte", pero más fuerte que la agregación. En la composición, las partes no pueden existir independientemente del todo. Por ejemplo, un **Motor** es parte de un **Coche** y no tiene sentido sin él.
- **Dependencia:** Indica que una clase depende de otra clase. Si se cambia la clase de la cual depende, es probable que la clase dependiente también deba cambiar.

SE PROPONE LA REALIZACIÓN DE LA TAREA 5

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL I

6.1. MANEJANDO FECHAS Y HORAS EN JAVA I

6.1.1. CLASE Date

❖ La clase **Date** representa un momento específico en el tiempo.

❖ **EJEMPLO:**

```
Date ahora = new Date();  
System.out.println("Fecha actual: " + ahora);
```

❖ **NOTA:** Puedes ver una tabla resumen de sus métodos más relevantes en los apuntes del tema.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL II

6.1. MANEJANDO FECHAS Y HORAS EN JAVA II

6.1.2. CLASE Calendar

- ❖ La clase **Calendar** proporciona métodos para convertir fechas y horas entre un momento específico y un conjunto de campos de calendario.

❖ **EJEMPLO:**

```
Calendar calendario = Calendar.getInstance();
int año = calendario.get(Calendar.YEAR);
System.out.println("Año actual: " + año);
```

- ❖ **NOTA:** Puedes ver una tabla resumen de sus métodos más relevantes en los apuntes del tema.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL III

6.1. MANEJANDO FECHAS Y HORAS EN JAVA III

6.1.3. CLASE GregorianCalendar

- ❖ La clase **GregorianCalendar** es una concreción de **Calendar** que utiliza el calendario gregoriano. Hereda los métodos de **Calendar** y añade algunos específicos para el calendario gregoriano, como **isLeapYear(int year)** para ver si el año es o no bisiesto.

❖ **EJEMPLO:**

```
GregorianCalendar gregCal = new GregorianCalendar();
boolean esBisiesto = gregCal.isLeapYear(2024);
System.out.println("2024 es bisiesto: " + esBisiesto);
```

- ❖ **NOTA:** Puedes ver una tabla resumen de sus métodos más relevantes en los apuntes del tema.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL IV

6.1. MANEJANDO FECHAS Y HORAS EN JAVA IV

6.1.4. CLASE TimeZone

- ❖ La clase **TimeZone** representa una zona horaria.

❖ EJEMPLO:

```
// Creando una instancia de SimpleTimeZone para GMT
SimpleTimeZone zonaGMT = new SimpleTimeZone(0, "GMT");

// Imprimiendo detalles de la zona horaria
System.out.println("ID de la zona horaria: " + zonaGMT.getID());
System.out.println("Desplazamiento respecto a UTC: " + zonaGMT.getRawOffset() + " milisegundos");
```

- ❖ **NOTA:** Puedes ver una tabla resumen de sus métodos más relevantes en los apuntes del tema.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL V

6.1. MANEJANDO FECHAS Y HORAS EN JAVA V

6.1.5. CLASE SimpleTimeZone

- ❖ La clase **SimpleTimeZone** es una subclase de **TimeZone** que representa zonas horarias con transiciones simples, como el horario de verano.

❖ EJEMPLO:

```
SimpleTimeZone stz = new SimpleTimeZone(3600000, "Europe/Madrid");
System.out.println("Zona horaria: " + stz.getID());
```

- ❖ **NOTA:** Puedes ver una tabla resumen de sus métodos más relevantes en los apuntes del tema.

SE PROPONE LA REALIZACIÓN DE LA TAREA 6

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL VI

6.2. MANEJANDO NÚMEROS ALEATORIOS EN JAVA I

- ❖ Los números aleatorios en Java son útiles para una variedad de aplicaciones como juegos, simulaciones, pruebas de algoritmos, y en situaciones donde se requiere una selección o comportamiento impredecible.

- ❖ El uso de números aleatorios en Java es fundamental en diversas áreas de programación. Elegir la clase adecuada depende de la necesidad específica de seguridad, tipo de datos y control sobre el rango de los números generados.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL VII

6.2. MANEJANDO NÚMEROS ALEATORIOS EN JAVA II

6.2.1. CLASES PARA GENERAR NÚMEROS ALEATORIOS I

- ❖ **Math.random():** Genera un número decimal (double) entre 0.0 y 1.0.

- ❖ **java.util.Random:** Una clase más versátil que permite generar números aleatorios de varios tipos (int, long, float, double, boolean).

- ❖ **java.security.SecureRandom:** Extiende **java.util.Random**, más segura y adecuada para criptografía.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL VIII

6.2. MANEJANDO NÚMEROS ALEATORIOS EN JAVA III

6.2.1. CLASES PARA GENERAR NÚMEROS ALEATORIOS II

❖ **EJEMPLO:** `Math.random()` es útil para casos simples.

```
double numeroAleatorio = Math.random();
System.out.println("Número aleatorio entre 0.0 y 1.0: " + numeroAleatorio);
```

❖ **EJEMPLO:** `Random` ofrece más flexibilidad y control (p.ej., especificar límites).

```
Random generador = new Random();
int numeroEnteroAleatorio = generador.nextInt(100); // Entre 0 y 99
System.out.println("Número entero aleatorio entre 0 y 99: " + numeroEnteroAleatorio);
```

❖ **EJEMPLO:** `SecureRandom` es esencial para aplicaciones que requieren alta seguridad, como en criptografía.

```
SecureRandom generadorSeguro = new SecureRandom();
int numeroSeguro = generadorSeguro.nextInt();
System.out.println("Número entero aleatorio seguro: " + numeroSeguro);
```

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL IX

6.2. MANEJANDO NÚMEROS ALEATORIOS EN JAVA IV

6.2.2. LA CLASE RandomGenerator I

❖ La clase `RandomGenerator` en Java es una parte de la API de generadores de números aleatorios introducida en Java 17. Proporciona una forma más moderna y flexible de generar números aleatorios y es parte del paquete `java.util.random`.

❖ `RandomGenerator` en Java es una adición valiosa para los desarrolladores que buscan una solución más robusta, flexible y de alto rendimiento para la generación de números aleatorios, especialmente en aplicaciones modernas y entornos concurrentes.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL X

6.2. MANEJANDO NÚMEROS ALEATORIOS EN JAVA V

6.2.2. LA CLASE RandomGenerator II

❖ Propósitos de RandomGenerator:

- ❑ **Flexibilidad Mejorada:** Ofrece una gama más amplia de algoritmos para la generación de números aleatorios, permitiendo a los desarrolladores elegir el que mejor se adapte a sus necesidades.
- ❑ **Interfaz Unificada:** Proporciona una interfaz común para varios algoritmos de generación de números aleatorios, lo que facilita cambiar entre diferentes algoritmos sin necesidad de cambiar el código.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL XI

6.2. MANEJANDO NÚMEROS ALEATORIOS EN JAVA VI

6.2.2. LA CLASE RandomGenerator III

❖ Propósitos de RandomGenerator: (continuación)

- ❑ **Mejoras de Rendimiento:** Algunos de los algoritmos disponibles en **RandomGenerator** están optimizados para ofrecer un mejor rendimiento en ciertas situaciones.
- ❑ **Uso en Aplicaciones Paralelas:** Algunas implementaciones de **RandomGenerator** están diseñadas para usarse en entornos de programación paralela y concurrente, minimizando la contención y mejorando el rendimiento en tales escenarios.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL XII

6.2. MANEJANDO NÚMEROS ALEATORIOS EN JAVA VII

6.2.2. LA CLASE RandomGenerator IV

❖ EJEMPLO básico de uso:

```
import java.util.RandomGenerator;

public class EjemploRandomGenerator {
    public static void main(String[] args) {
        RandomGenerator randomGenerator = RandomGenerator.getDefault();

        int numeroAleatorio = randomGenerator.nextInt(100); // Genera un número aleatorio entre 0 y 99
        System.out.println("Número aleatorio: " + numeroAleatorio);
    }
}
```

- ❖ En este ejemplo, **RandomGenerator.getDefault()** obtiene una instancia de un generador de números aleatorios utilizando el algoritmo predeterminado. Luego se utiliza para generar un número entero aleatorio.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL XIII

6.2. MANEJANDO NÚMEROS ALEATORIOS EN JAVA VIII

6.2.2. LA CLASE RandomGenerator V

❖ Generando números aleatorios de tipo double con RandomGenerator:

- ▣ Es posible acotar el rango máximo de un número generado como **double** utilizando **RandomGenerator**. Sin embargo, a diferencia de los métodos para enteros, no hay un método directo en **RandomGenerator** que permita especificar un límite superior para **double**. En su lugar, se puede multiplicar el valor generado, que por defecto está en el rango [0.0, 1.0), por el límite superior deseado.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL XIV

6.2. MANEJANDO NÚMEROS ALEATORIOS EN JAVA IX

6.2.2. LA CLASE RandomGenerator VI

❖ **EJEMPLO:** Vamos a generar un número **double** aleatorio entre 0.0 y 10.0

```
import java.util.random.RandomGenerator;

public class EjemploRandomDouble {
    public static void main(String[] args) {
        RandomGenerator randomGenerator = RandomGenerator.getDefault();

        double limiteSuperior = 10.0;
        double numeroAleatorio = randomGenerator.nextDouble() * limiteSuperior; // Genera un número entre 0.0 y 10.0

        System.out.println("Número double aleatorio entre 0.0 y " + limiteSuperior + ": " + numeroAleatorio);
    }
}
```

Ver tabla de los apuntes para más información sobre los métodos de RandomGenerator

- ❑ **randomGenerator.nextDouble()** genera un número **double** en el rango [0.0, 1.0].
- ❑ Multiplicamos este número por **limiteSuperior** para escalar el rango a [0.0, 10.0].

❖ Así controlamos el rango máximo de los números generados con **RandomGenerator**.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL XV

6.2. MANEJANDO NÚMEROS ALEATORIOS EN JAVA X

6.2.3. ¡CUIDADO! NO SON NÚMEROS ALEATORIOS I

❖ En realidad, lo que se generan son números pseudoaleatorios.

❖ **Números Pseudoaleatorios:**

- ❑ **Definición:** Los números pseudoaleatorios son secuencias de números que parecen ser aleatorios, pero en realidad son generados por un algoritmo determinista. Este algoritmo, dado un valor inicial llamado "semilla" (seed), produce una secuencia de números que tiene propiedades estadísticas de aleatoriedad.
- ❑ **Predecibilidad:** Debido a que se basan en un algoritmo determinista, si conoces la semilla y el algoritmo, puedes predecir toda la secuencia de números.
- ❑ **Uso en Programación:** En la mayoría de los lenguajes de programación, incluido Java, los métodos para generar números aleatorios en realidad producen números pseudoaleatorios. Esto es suficiente para la mayoría de las aplicaciones, como juegos, simulaciones y pruebas.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL XVI

6.2. MANEJANDO NÚMEROS ALEATORIOS EN JAVA XI

6.2.3. ¡CUIDADO! NO SON NÚMEROS ALEATORIOS II

❖ Números Verdaderamente Aleatorios:

- ❑ **Definición:** Los números verdaderamente aleatorios son aquellos que provienen de un proceso físico aleatorio, como el ruido térmico, la desintegración radiactiva, etc.
- ❑ **Imposibilidad en Computadoras Ordinarias:** Las computadoras convencionales no pueden generar números verdaderamente aleatorios utilizando solamente software, ya que se basan en algoritmos deterministas.

6. TRABAJANDO CON ALGUNAS CLASES DE LA JCL XVII

6.2. MANEJANDO NÚMEROS ALEATORIOS EN JAVA XII

6.2.3. ¡CUIDADO! NO SON NÚMEROS ALEATORIOS III

❖ Conclusión:

- ❑ En el contexto de la programación en Java y el uso de clases como **Random**, **SecureRandom**, o **RandomGenerator**, estamos trabajando con generadores de números pseudoaleatorios. Estos son adecuados para la mayoría de las aplicaciones de software, pero es crucial entender su naturaleza determinista, especialmente en contextos donde la seguridad y la imprevisibilidad son críticas, como en criptografía. En esos casos, se puede recurrir a métodos más sofisticados que combinan algoritmos pseudoaleatorios con entradas aleatorias de hardware para aumentar la imprevisibilidad.

SE PROPONE LA REALIZACIÓN DE LA TAREA 7

7. DESPEDIDA



Ya conoces el poder de la librería de objetos y clases de Java, ¡Ahora toca hacer tus propias clases y objetos!

