Contenido

1. Clases

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.**

1.2. Componentes

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()**.

```
public class Coche {
    // Atributos
    String marca;
    String modelo;
    String color;

    // Método
    public void arrancar() {
        System.out.println("El coche está arrancando");
    }
}
```

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.

1.3. Atributos y métodos de una clase frente a los de un objeto

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: 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.

```
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

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)

- 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();**

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

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.

3.2. Características

1. **Nombre del Constructor**: Tiene el mismo nombre que la clase a la que pertenece.
2. **Inicialización de Objetos**: Se ejecutan automáticamente en el momento de la creación de un objeto (cuando se usa **new**).
3. **No tienen Tipo de Retorno**: A diferencia de los métodos ordinarios, los constructores no especifican un tipo de retorno, ni siquiera **void**.

4. **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.

5. **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.

3.3. Ejemplos

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;
    }
}
```

3.3.2. 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");
```

4. Bibliotecas de clases y objetos de Java

4.1. Biblioteca de objetos en Java

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.

4.1.1. Componentes Clave de la JCL

1. **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.

2. **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.

3. **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.

4.1.2. 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.

4.1.3. 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.

4.1.4. 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
public final Class<?> getClass()	Retorna la instancia de `Class` que representa la clase del objeto.
public int hashCode()	Retorna un código hash para el objeto, utilizado en estructuras de datos hash.
public boolean equals(Object obj)	Compara si el objeto actual es "igual" al objeto proporcionado según la implementación.
protected Object clone()	Crea y retorna una copia del objeto si la clase implementa `Cloneable`.
public String toString()	Retorna una representación en cadena del objeto.
public final void notify()	Despierta a un solo hilo que esté esperando en el monitor del objeto.
public final void notifyAll()	Despierta a todos los hilos que están esperando en el monitor del objeto.
public final void wait(long timeout)	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.
public final void wait(long timeout, int nanos)	Hace que el hilo actual espere con más precisión en tiempo.
public final void wait()	Hace que el hilo actual espere indefinidamente hasta que otro hilo invoque `notify()` o `notifyAll()` para este objeto.
protected void finalize()	Método llamado por el recolector de basura antes de que el objeto sea destruido, ahora obsoleto desde Java 9.

4.1.5. Clases envoltorio (Wrapper Classes)

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.

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**

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
```

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
```

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**.

4.2. Java versus Javax

La diferencia principal entre las bibliotecas **java** y **javax** en Java radica en sus propósitos y en la evolución histórica de sus usos:

4.2.1. Propósito y Contenido

1. **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.

2. **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.

4.2.2. 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.

4.2.3. Uso y Compatibilidad

1. 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.

2. 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.

4.2.4. 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	swing	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.

5. Conceptos fundamentales de la POO

5.1. Principios básicos

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

1. **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.

2. **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.

3. **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.

4. **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.

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

5.2. Encapsulamiento y visibilidad

En la POO, "encapsulamiento" y "visibilidad" son dos conceptos clave:

1. **Encapsulamiento**: Se refiere a la práctica de ocultar los detalles internos del funcionamiento de una clase y solo exponer las operaciones que son seguras y necesarias para el usuario de esa clase. Esto significa que el estado interno de un objeto (sus atributos) se protege del acceso directo desde fuera de la clase. Las interacciones con el objeto se realizan a través de métodos (funciones) que controlan cómo se accede y modifica ese estado interno.

2. **Visibilidad**: Está relacionada con cómo se controla el acceso a los miembros (atributos y métodos) de una clase. En Java, existen varios modificadores de acceso como **public**, **private**, **protected**, y el acceso por defecto (sin modificador). Estos determinan si otros objetos o clases pueden acceder a los miembros de una clase, ayudando así a implementar el encapsulamiento. Por ejemplo, un atributo marcado como **private** solo puede ser accedido por métodos dentro de la misma clase.

El encapsulado y la visibilidad son fundamentales para crear un código robusto, modular y fácil de mantener, elementos esenciales en la programación orientada a objetos.

5.2.1. Visibilidad de los atributos de una clase

Te recuerdo que la **visibilidad de los métodos** fue tratada con detalle en el punto 1.1.6 del tema 2 (Estructura y bloques fundamentales de un programa informático).

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 mantenible.

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.

1. 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.
- **EJEMPLO**:

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

    public String getMatricula() {
        return matricula;
    }

    public void setMatricula(String m) {
        matricula = m;
    }
}
```

2. 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.
- **EJEMPLO**:

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

    int getPotencia() {
        return potencia;
    }
}
```

3. 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.
- **EJEMPLO**:

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

    protected int getRuedas() {
        return ruedas;
    }
}

public class Bicicleta extends Vehiculo {
    public void setRuedas(int r) {
        ruedas = r; // Acceso permitido
    }
}
```

4. 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.
- **EJEMPLO**:

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

5.3. Relaciones entre clases

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:

1. **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.
2. **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**.
3. **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**.
4. **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.
5. **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.

Estas relaciones son esenciales para comprender cómo los objetos interactúan en un programa y cómo se puede estructurar el código de manera eficiente y lógica.

6. Trabajando con algunas clases de la JCL

6.1. Manejando fechas y horas en Java

6.1.1. Clase Date

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

Método	Descripción	Ejemplo
Date()	Constructor que inicializa el objeto con la fecha y hora actuales.	`` `Date fechaActual = new Date();` ``
Date(long date)	Constructor que inicializa el objeto con la fecha y hora especificadas en milisegundos desde el 1 de enero de 1970.	`` `Date fechaEspecifica = new Date(1625097600000L);` ``
boolean after(Date when)	Comprueba si la fecha del objeto es posterior a la fecha especificada.	`` `fechaActual.after(fechaEspecifica)` ``
boolean before(Date when)	Comprueba si la fecha del objeto es anterior a la fecha especificada.	`` `fechaActual.before(fechaEspecifica)` ``
Object clone()	Clona el objeto `` `Date` ``.	`` `Date fechaClonada = (Date) fechaActual.clone();` ``
int compareTo(Date anotherDate)	Compara dos fechas.	`` `fechaActual.compareTo(fechaEspecifica)` ``
boolean equals(Object obj)	Comprueba si dos fechas son iguales.	`` `fechaActual.equals(fechaEspecifica)` ``
long getTime()	Devuelve la fecha y hora en milisegundos desde el 1 de enero de 1970.	`` `long tiempo = fechaActual.getTime();` ``
int hashCode()	Devuelve un hash code para el objeto `` `Date` ``.	`` `int hash = fechaActual.hashCode();` ``
void setTime(long time)	Establece la fecha y hora con un valor en milisegundos desde el 1 de enero de 1970.	`` `fechaActual.setTime(1625097600000L);` ``
String toString()	Convierte la fecha a `` `String` ``.	`` `String fechaComoString = fechaActual.toString();` ``

EJEMPLO:

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

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.

Método	Descripción	Ejemplo
Calendar getInstance()	Obtiene una instancia de `Calendar` con la fecha y hora actuales.	`Calendar calendario = Calendar.getInstance();`
int get(int field)	Devuelve el valor del campo especificado del calendario.	`int año = calendario.get(Calendar.YEAR);`
void set(int field, int value)	Establece el valor del campo especificado del calendario.	`calendario.set(Calendar.YEAR, 2023);`
void add(int field, int amount)	Añade o resta la cantidad especificada de tiempo al campo de calendario dado.	`calendario.add(Calendar.DAY_OF_MONTH, 5);`
void roll(int field, boolean up)	Añade o resta una unidad del campo de calendario especificado sin cambiar campos mayores.	`calendario.roll(Calendar.MONTH, false);`

Método	Descripción	Ejemplo
Date getTime()	Devuelve un objeto `Date` que representa el tiempo de este `Calendar`.	`Date fecha = calendario.getTime();`
void setTime(Date date)	Establece el tiempo del calendario con el valor del objeto `Date` proporcionado.	`calendario.setTime(new Date());`
long getTimeInMillis()	Devuelve el tiempo del calendario en milisegundos.	`long tiempoMilis = calendario.getTimeInMillis();`
void setTimeInMillis(long millis)	Establece el tiempo en milisegundos del calendario.	`calendario.setTimeInMillis(1625097600000L);`
int getActualMaximum(int field)	Devuelve el valor máximo que puede tener el campo especificado.	`int ultimoDia = calendario.getActualMaximum(Calendar.DAY_OF_MONTH);`
int getActualMinimum(int field)	Devuelve el valor mínimo que puede tener el campo especificado.	`int primerDia = calendario.getActualMinimum(Calendar.DAY_OF_MONTH);`

EJEMPLO:

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

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.

Método	Descripción	Ejemplo
GregorianCalendar()	Constructor que crea un calendario gregoriano con la fecha y hora actuales.	`` `GregorianCalendar gCal = new GregorianCalendar();` ``
GregorianCalendar(int year, int month, int dayOfMonth)	Constructor que crea un calendario gregoriano para la fecha especificada.	`` `GregorianCalendar gCal = new GregorianCalendar(2023, Calendar.MARCH, 10);` ``
GregorianCalendar(int year, int month, int dayOfMonth, int hourOfDay, int minute)	Constructor que crea un calendario gregoriano para la fecha y hora especificadas.	`` `GregorianCalendar gCal = new GregorianCalendar(2023, Calendar.MARCH, 10, 15, 30);` ``
GregorianCalendar(int year, int month, int dayOfMonth, int hourOfDay, int minute, int second)	Constructor que crea un calendario gregoriano para la fecha, hora y segundo especificados.	`` `GregorianCalendar gCal = new GregorianCalendar(2023, Calendar.MARCH, 10, 15, 30, 45);` ``
void setGregorianChange(Date date)	Establece la fecha en la que el calendario gregoriano fue adoptado.	`` `gCal.setGregorianChange(new Date());` ``
Date getGregorianChange()	Devuelve la fecha en la que el calendario gregoriano fue adoptado.	`` `Date cambioGregoriano = gCal.getGregorianChange();` ``
boolean isLeapYear(int year)	Determina si el año especificado es bisiesto en el calendario gregoriano.	`` `boolean esBisiesto = gCal.isLeapYear(2024);` ``

EJEMPLO:

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

6.1.4. Clase TimeZone

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

Método	Descripción	Ejemplo
static TimeZone getTimeZone(String ID)	Devuelve la zona horaria para el ID especificado.	`` `TimeZone zona = TimeZone.getTimeZone("GMT");` ``
static String[] getAvailableIDs()	Devuelve todos los IDs de zonas horarias disponibles.	`` `String[] zonas = TimeZone.getAvailableIDs();` ``
String getID()	Devuelve el ID de la zona horaria.	`` `String idZona = zona.getID();` ``
static TimeZone getDefault()	Obtiene la zona horaria predeterminada.	`` `TimeZone zonaDefault = TimeZone.getDefault();` ``
boolean inDaylightTime(Date date)	Determina si la fecha especificada está en horario de verano en esta zona horaria.	`` `boolean esHorarioVerano = zona.inDaylightTime(new Date());` ``
int getOffset(long date)	Devuelve la cantidad de tiempo que se debe agregar a la hora UTC para obtener la hora local para la fecha especificada.	`` `int offset = zona.getOffset(System.currentTimeMillis());` ``
void setID(String ID)	Establece el ID de la zona horaria.	`` `zona.setID("America/Los_Angeles");` ``
int getRawOffset()	Devuelve la cantidad de tiempo en milisegundos que se debe agregar a la hora UTC para obtener la hora estándar en esta zona horaria.	`` `int rawOffset = zona.getRawOffset();` ``
void setRawOffset(int offsetMillis)	Establece el desplazamiento en milisegundos de la hora UTC para esta zona horaria.	`` `zona.setRawOffset(3600000);` ``
static void setDefault(TimeZone zone)	Establece la zona horaria predeterminada.	`` `TimeZone.setDefault(zona);` ``

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");
```

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.

Método	Descripción	Ejemplo
SimpleTimeZone(int timeOffset, String ID)	Constructor que crea una zona horaria con un desplazamiento de tiempo y un ID específico.	`` `SimpleTimeZone zona = new SimpleTimeZone(3600000, "Europe/Madrid");` ``
void setStartRule(int month, int dayOfWeekInMonth, int dayOfWeek, int time)	Establece la regla para iniciar el horario de verano.	`` `zona.setStartRule(Calendar.MARCH, -1, Calendar.SUNDAY, 2 * 3600000);` ``
void setEndRule(int month, int dayOfWeekInMonth, int dayOfWeek, int time)	Establece la regla para finalizar el horario de verano.	`` `zona.setEndRule(Calendar.OCTOBER, -1, Calendar.SUNDAY, 2 * 3600000);` ``
void setStartYear(int year)	Establece el primer año en el que se aplica la zona horaria.	`` `zona.setStartYear(2023);` ``
int getRawOffset()	Devuelve el desplazamiento de tiempo estándar de la zona horaria respecto a UTC.	`` `int offset = zona.getRawOffset();` ``
void setRawOffset(int offsetMillis)	Establece el desplazamiento de tiempo estándar de la zona horaria respecto a UTC.	`` `zona.setRawOffset(3600000);` ``
boolean inDaylightTime(Date date)	Determina si la fecha especificada está en horario de verano en esta zona horaria.	`` `boolean esHorarioVerano = zona.inDaylightTime(new Date());` ``
boolean useDaylightTime()	Comprueba si esta zona horaria utiliza el horario de verano.	`` `boolean utilizaHorarioVerano = zona.useDaylightTime();` ``
int getOffset(long date)	Devuelve el desplazamiento total (estándar más horario de verano) para la fecha especificada.	`` `int totalOffset = zona.getOffset(System.currentTimeMillis());` ``
void setDSTSavings(int millisSavedDuringDST)	Establece la cantidad de milisegundos que se ahorran durante el horario de verano.	`` `zona.setDSTSavings(3600000);` ``

EJEMPLO:

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

6.2. Manejando números aleatorios en Java

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.

6.2.1. Clases para generar números aleatorios

1. **Math.random()**: Genera un número decimal (double) entre 0.0 y 1.0.
2. **java.util.Random**: Una clase más versátil que permite generar números aleatorios de varios tipos (int, long, float, double, boolean).
3. **java.security.SecureRandom**: Extiende **java.util.Random**, más segura y adecuada para criptografía.

EJEMPLOS:

- **Math.random()** es útil para casos simples.

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

- **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);
```

- **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);
```

6.2.2. La clase RandomGenerator

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.

Propósitos de RandomGenerator:

1. **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.

2. **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.

3. **Mejoras de Rendimiento**: Algunos de los algoritmos disponibles en **RandomGenerator** están optimizados para ofrecer un mejor rendimiento en ciertas situaciones.

4. **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.

EJEMPLO básico de uso:

```java
import java.util.random.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.

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.

EJEMPLO: Supongamos que quieres 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);
    }
}
```

En este código:

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

Este método es una forma efectiva de controlar el rango máximo de los números **double** generados con **RandomGenerator**.

Tabla resumen con los métodos más útiles de RamdomGenerator:

Aquí tienes una tabla con información sobre la clase RandomGenerator en Java, incluyendo los nombres, descripciones y ejemplos:

Nombre	Descripción	Ejemplo
getDefault()	Obtiene una instancia del generador de números aleatorios usando el algoritmo predeterminado.	`` `RandomGenerator random = RandomGenerator.getDefault();` ``
nextInt()	Genera un número entero aleatorio.	`` `int num = random.nextInt(100); // Entre 0 y 99` ``
nextLong()	Genera un número largo (long) aleatorio.	`` `long numLargo = random.nextLong();` ``
nextDouble()	Genera un número decimal (double) aleatorio.	`` `double numDecimal = random.nextDouble();` ``
nextFloat()	Genera un número de coma flotante (float) aleatorio.	`` `float numFloat = random.nextFloat();` ``
nextBoolean()	Genera un valor booleano (true o false) aleatorio.	`` `boolean valorBool = random.nextBoolean();` ``
nextBytes(byte[] bytes)	Rellena el array de bytes proporcionado con datos aleatorios.	`` `byte[] bytes = new byte[10]; random.nextBytes(bytes);` ``

6.2.3. ¡Cuidado! No son números aleatorios

En realidad, lo que se generan son números pseudoaleatorios, no números aleatorios en el sentido estricto. Esto es importante desde una perspectiva técnica y matemática.

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.

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.

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.

7. Despedida

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