Resumen Teórico de Programación Orientada a Objetos (POO)

Taller de Programación
Informática - UNLP

Generado por Vertex AI y ChatGPT 4

Abril de 2024

Tabla de contenidos

Video 1

Clase 1 - Parte 1: Netbeans, Crear e Importar Proyecto, Importar Paquete de Lectura

1.1 Introducción

En este video, el profesor explica cómo usar NetBeans, un entorno de desarrollo integrado (IDE) para programar en Java. Se busca familiarizar a los estudiantes con las funciones y características de NetBeans, mostrando la creación e importación de proyectos y la importación de un paquete de lectura.

1.2 Familiarización con NetBeans

El profesor comienza mostrando el entorno de NetBeans. NetBeans es un IDE que permite escribir, compilar y depurar aplicaciones Java. Una de sus características clave es la zona de proyectos, donde se muestran todos los proyectos activos y su estructura. Esta zona facilita la organización del código y permite navegar entre archivos y paquetes.

A diferencia de entornos como Pascal, donde el código generalmente reside en un solo archivo, NetBeans permite trabajar con múltiples archivos y paquetes, permitiendo una estructura modular. Esta modularidad ayuda a los desarrolladores a organizar mejor su trabajo y facilita el mantenimiento a largo plazo.

El profesor muestra cómo abrir NetBeans desde el escritorio y aborda posibles errores al inicio, como mensajes de error relacionados con la instalación. Para resolver estos problemas, el profesor muestra cómo revisar la instalación y hacer ajustes para asegurar el correcto funcionamiento de NetBeans.

1.3 Creación e Importación de Proyectos

El profesor muestra cómo crear un nuevo proyecto en NetBeans. Crear un proyecto es el primer paso para comenzar a trabajar en el IDE. Para crear un

nuevo proyecto, se selecciona "File" y luego "New Project". Aparece un asistente que guía al desarrollador a través de los pasos necesarios para configurar el proyecto, como elegir el tipo de proyecto, asignar un nombre y directorio.

Una vez completado el proceso, el nuevo proyecto aparece en la zona de proyectos, permitiendo al desarrollador interactuar con él. El profesor recomienda elegir nombres descriptivos para los proyectos y almacenarlos en ubicaciones fáciles de recordar.

Además de crear un nuevo proyecto, el profesor muestra cómo importar un proyecto existente en NetBeans. Para hacer esto, se usa la opción "Import Project" desde el menú principal y se selecciona el directorio donde se encuentra el proyecto que se quiere importar. Una vez importado, el proyecto aparece en la zona de proyectos y se puede interactuar con él como con cualquier otro proyecto.

1.4 Importación del Paquete de Lectura

El profesor muestra cómo importar un paquete de lectura en NetBeans. Este paquete puede contener bibliotecas o recursos adicionales necesarios para que el proyecto funcione correctamente. El profesor explica que estos recursos son esenciales para asegurar el correcto funcionamiento del proyecto y muestra cómo importarlos.

Para importar el paquete de lectura, el profesor selecciona el proyecto y elige "Properties". Luego, se selecciona la sección "Libraries" y se usa "Add JAR/Folder" para agregar el paquete. El profesor muestra cómo seleccionar el archivo JAR o la carpeta que contiene los recursos necesarios y cómo verificar que el proceso fue exitoso.

El profesor aborda problemas comunes que pueden surgir al importar paquetes de lectura, como errores de ruta o falta de recursos. Recomienda revisar la ruta del archivo y asegurarse de que esté en el lugar correcto antes de intentar compilar el proyecto.

1.5 Conclusiones

El profesor concluye el video enfatizando la importancia de mantener los proyectos organizados y cómo la estructura modular de NetBeans ayuda a lograrlo. Recomienda separar el código personal del código del curso para evitar confusiones y sugiere cómo crear nuevos archivos de código fuente según las necesidades del proyecto.

Para crear nuevos archivos de código fuente, el profesor muestra cómo hacer clic con el botón derecho en el proyecto y seleccionar "New", eligiendo el tipo de archivo a crear, como "Java Main Class" o "Java Class". Recomienda mantener una estructura organizada y separar el código personal del código del curso para evitar confusiones.

Además, el profesor ofrece consejos y trucos para trabajar con NetBeans de manera eficiente. Recomienda mantener el IDE actualizado para aprovechar las últimas características y correcciones de errores. También sugiere explorar diferentes atajos de teclado para acelerar el desarrollo y trabajar de manera más fluida.

Finalmente, el profesor agradece a los estudiantes por su atención y menciona que en videos futuros se abordarán temas más avanzados, como la creación de clases Java, el uso de métodos y variables, y otros conceptos importantes en la programación con Java.

Video 2

Clase 1 - Parte 2: Java, JDK y JRE. Estructura del programa, Tipos primitivos, Operadores e Imprimir en pantalla

2.1 Introducción

El video presenta una introducción a Java como lenguaje de programación, cubriendo conceptos clave como JDK, JRE, la estructura básica de un programa Java, tipos primitivos, operadores y la capacidad de imprimir en pantalla para la depuración y ejecución del programa. Se destaca la naturaleza de Java como un lenguaje compilado y multiplataforma.

2.2 Java: compilado y multiplataforma

Java es un lenguaje compilado que transforma el código fuente en bytecode, que puede ser interpretado por la Java Virtual Machine (JVM). Esto permite que Java sea multiplataforma, lo cual es clave para la portabilidad de aplicaciones y una ventaja significativa en comparación con otros lenguajes. A diferencia de otros lenguajes, como Pascal, donde el código debe ser compilado para cada plataforma específica, Java es portátil y puede ejecutarse en cualquier entorno donde esté instalada la JVM.

El profesor menciona que esta propiedad de Java, conocida como "Write Once, Run Anywhere" (WORA), facilita el desarrollo de aplicaciones portátiles que pueden ejecutarse en múltiples sistemas operativos y arquitecturas. Esto hace que Java sea una opción atractiva para desarrolladores que desean que sus aplicaciones sean ampliamente accesibles.

2.3 JDK y JRE

El Java Development Kit (JDK) es el conjunto de herramientas necesario para desarrollar aplicaciones Java. Contiene el compilador 'javac', que convierte el

código fuente en bytecode para la JVM, y otras herramientas como depuradores, generadores de documentación y empaquetadores de aplicaciones. El Java Runtime Environment (JRE) es el entorno que permite ejecutar aplicaciones Java y contiene la JVM y las bibliotecas necesarias.

2.4 Estructura de un programa Java

Un programa Java comienza con una clase principal que tiene un método 'main', el punto de entrada para la ejecución del programa. Aquí está el ejemplo básico:

```
public class MiPrograma {
    public static void main(String[] args) {
        System.out.println("Hola, mundo!");
    }
}
```

El método 'main' debe ser público, estático y aceptar un arreglo de argumentos ('String[] args') por razones importantes:

- public: este modificador indica que el método puede ser llamado desde fuera de la clase donde está definido. Es esencial para el método main porque la JVM necesita poder acceder a este método para iniciar la ejecución del programa. Si el método main no es público, la JVM no podría encontrarlo, y el programa no se ejecutaría correctamente.
- static: este modificador significa que el método pertenece a la clase y no a una instancia específica de la clase. En otras palabras, se puede llamar al método sin crear un objeto de esa clase. Esto es clave para el método main, ya que la JVM necesita llamarlo para iniciar el programa sin tener que crear una instancia de la clase primero.

El profesor menciona errores críticos como olvidar las palabras clave 'public' y 'static', o no cerrar correctamente las llaves y paréntesis. También señala que Java es un lenguaje sensible a mayúsculas y minúsculas (case-sensitive), lo que es clave para evitar errores por capitalización incorrecta.

2.5 Tipos primitivos

Los tipos primitivos en Java son esenciales para comprender cómo se manejan los datos en el lenguaje. El uso correcto de estos tipos es clave para el rendimiento del programa y para evitar errores. Aquí están algunos ejemplos de su uso:

```java
int entero = 42;
double decimal = 3.14;
char caracter = 'A';
boolean esVerdadero = true;
```

El profesor sugiere practicar con estos tipos primitivos para comprender su funcionamiento y evitar errores comunes. El conocimiento de estos tipos es importante para desarrollar programas robustos y eficientes.

2.6 Operadores

Los operadores permiten realizar operaciones matemáticas y lógicas en Java. El profesor menciona operadores aritméticos, como ‘+‘, ‘-‘, ‘*‘, ‘/‘, ‘

Aquí tienes ejemplos del uso de operadores:

```java
int suma = 10 + 5; // suma
int resta = 15 - 7; // resta
int multiplicacion = 3 * 4; // multiplicacion
int division = 20 / 5; // division entera
int modulo = 17 % 3; // modulo
boolean esIgual = (10 == 10); // igual
boolean esDiferente = (5 != 7); // distinto
boolean esMayor = (8 > 3); // mayor
boolean yLogico = (true && false); // AND
boolean oLogico = (true || false); // OR
```

El profesor destaca errores comunes que pueden ser críticos, como usar un solo signo ‘=‘ para comparación en lugar de ‘==‘. Recomienda a los estudiantes practicar con estos operadores para entender su impacto en el flujo del programa y evitar errores lógicos.

2.7 Imprimir en pantalla

El método ‘System.out.println()‘ se utiliza para imprimir texto en la consola. Aquí está un ejemplo para imprimir resultados de operaciones matemáticas:

```java
public class EjemploImpresion {
    public static void main(String[] args) {
        System.out.println("El valor de c es: "+ (a+b));
    }
}
```

El profesor sugiere usar ‘System.out.println()‘ para depurar el código y comprender el flujo del programa. Recomienda experimentar con diferentes tipos de datos y combinaciones de texto y variables para obtener resultados variados.

2.8 Conclusiones

El video proporciona una introducción detallada a Java, cubriendo conceptos clave como el JDK, el JRE, la estructura básica de un programa, tipos primitivos, operadores y la capacidad de imprimir en pantalla. El profesor recomienda practicar con ejemplos simples y seguir explorando Java para obtener una comprensión sólida. La capacidad de Java para ser un lenguaje compilado y multiplataforma es importante porque permite desarrollar aplicaciones portátiles y versátiles.

Video 3

Clase 1 - Parte 3: Uso del Paquete de Lectura y Generador Aleatorio

3.1 Introducción

En esta clase, se explora el uso del paquete de lectura y la generación de valores aleatorios en Java, componentes clave para interactuar con el usuario y realizar pruebas de software. El profesor guía a los estudiantes a través de la importación del paquete, la utilización de la clase 'Lector' para leer datos desde el teclado y el uso del paquete 'GeneradorAleatorio' para generar valores aleatorios. Los ejemplos y consejos del profesor ilustran la relevancia de estas herramientas en la programación.

3.2 Uso del Paquete de Lectura

El paquete de lectura es fundamental para interactuar con el usuario a través del teclado. Para usarlo, es necesario importarlo al inicio del código:

```
import paquetelectura.Lector;
```

El profesor muestra cómo utilizar la clase 'Lector' para leer distintos tipos de datos, como 'String', 'boolean', 'int' y 'double'. El siguiente ejemplo demuestra cómo leer un 'String' y un 'double':

```
System.out.println("Ingrese su nombre");
String nombre = Lector.leerString();
System.out.println("Ingrese su sueldo");
double sueldo = Lector.leerDouble();
```

Para imprimir los resultados concatenados con un mensaje informativo, el profesor sugiere lo siguiente:

```
System.out.println(nombre + " cobra " + sueldo + " pesos");
```

El profesor aconseja usar esta estructura para interactuar con el usuario y enfatiza la importancia de la legibilidad del código. Se recomienda siempre imprimir un mensaje antes de solicitar la entrada del usuario para facilitar la comprensión.

3.3 Uso del Generador Aleatorio

El generador aleatorio permite crear datos rápidamente para pruebas y otros propósitos. Para utilizarlo, se debe importar el paquete correspondiente:

```
import paquetelectura.GeneradorAleatorio;
```

Antes de generar valores aleatorios, el profesor señala que es necesario iniciar el generador para establecer la semilla:

```
GeneradorAleatorio.iniciar();
```

El método ‘generarInt(int max)‘ genera un número entero entre 0 y ‘max - 1‘, mientras que ‘generarDouble(int max)‘ produce valores decimales dentro del mismo rango. Aquí algunos ejemplos prácticos proporcionados por el profesor:

```
int entero = GeneradorAleatorio.generarInt(20);
double doble = GeneradorAleatorio.generarDouble(500);
```

El profesor destaca la importancia de inicializar el generador aleatorio solo una vez para evitar resultados repetitivos. Para explicar el concepto, el profesor utiliza una analogía con un bolillero, indicando que si se inicializa repetidamente, se corre el riesgo de obtener resultados idénticos debido a la reutilización de la misma semilla.

3.4 Ejemplo de Uso

El profesor muestra un ejemplo práctico para aplicar los conceptos aprendidos. Este ejemplo utiliza el paquete de lectura y el generador aleatorio para simular un programa que calcula salarios aleatorios para un empleado. El objetivo es demostrar cómo se pueden usar estas herramientas para generar resultados diferentes cada vez. El código es el siguiente:

```
import paquetelectura.Lector;
import paquetelectura.GeneradorAleatorio;
GeneradorAleatorio.iniciar();
System.out.println("Ingrese su nombre");
String nombre = Lector.leerString();
System.out.println("Ingrese su sueldo inicial");
double sueldo = Lector.leerDouble();
System.out.println("Sueldo original: " + sueldo);
sueldo = GeneradorAleatorio.generarDouble(500);
System.out.println("Nuevo sueldo para " + nombre
                   + ": " + sueldo);
```

En este ejemplo, el profesor muestra cómo el generador aleatorio puede ser usado para crear variaciones en el sueldo de un empleado. El código lee el nombre del empleado y su sueldo original desde el teclado y luego genera un nuevo sueldo de manera aleatoria. El profesor destaca la utilidad de este método para simular situaciones reales y realizar pruebas con valores aleatorios.

3.5 Conclusiones

El uso del paquete de lectura y del generador aleatorio son herramientas poderosas para interactuar con el usuario y para realizar pruebas de manera eficiente. El profesor enfatiza la importancia de diseñar cuidadosamente las pruebas para asegurar que el programa sea robusto. Los valores aleatorios pueden ser útiles para pruebas rápidas, pero no deben ser el único método para validar un programa.

El profesor recomienda tener cuidado al usar el generador aleatorio, recordando que las pruebas aleatorias pueden ser útiles para ciertos casos, pero el diseño de pruebas bien planificadas es fundamental para asegurar que el programa funcione correctamente en diferentes escenarios. El consejo final del profesor es usar el generador aleatorio con responsabilidad y considerar tanto casos límites como casos promedio para asegurar la robustez del software.

Video 4

Clase 1 - Parte 4: Estructuras de Control

4.1 Introducción

En este segmento de la clase, el profesor profundiza en las estructuras de control de Java, comparándolas con las de Pascal. Se discuten ejemplos de código y se brindan consejos y recomendaciones para una mejor comprensión de estas estructuras. También se incluyen discusiones sobre cómo estas estructuras pueden ser utilizadas para diferentes propósitos y cómo optimizar su uso.

4.2 Secuencia

La secuencia es la estructura de control más básica, donde las acciones se ejecutan en el orden en que se presentan. El profesor enfatiza la importancia del orden de ejecución y el uso adecuado de los puntos y comas para evitar errores.

4.2.1 Código Java y Pascal

Comencemos con un ejemplo simple para iniciar la comparacion:

- Código Java:

```java
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

- Código Pascal:

```pascal
for i := 1 to 10 do writeln(i);
```

El profesor señala la importancia de probar cada parte del código para asegurar el funcionamiento correcto. También destaca cómo las estructuras secuenciales pueden ser simples, pero fundamentales para la comprensión de lenguajes de programación.

4.3 Selección

La selección implica tomar decisiones basadas en condiciones. El profesor introduce el if-else, señalando diferencias con Pascal, y menciona el uso de llaves para agrupar sentencias.

4.3.1 Código Java y Pascal

Comparemos ahora Java y Pascal en Estructura de Control de Seleccion:

- Código Java:

```java
int x = 10;
if (x > 5) {
    System.out.println("x es mayor que 5");
} else {
    System.out.println("x no es mayor que 5");
}
```

- Código Pascal:

```pascal
var
  x: Integer;
begin
  x := 10;
  if x > 5 then
    writeln('x es mayor que 5')
  else
    writeln('x no es mayor que 5');
end.
```

El profesor destaca la importancia de verificar las condiciones cuidadosamente y agrupar las sentencias cuando sea necesario.

4.4 Iteración

La iteración se refiere a la ejecución de un loop de acciones mientras se cumpla una condición. El profesor introduce dos tipos principales de bucles:

- while: evalúa la condición antes de ejecutar el bloque.
- do-while (repeat-until en Pascal): evalúa la condición después de ejecutar el bloque.

Comenzaremos ahora con las comparaciones de codigo para estructura de control iterativas:

4.4.1 Código While para Java y Pascal

- Código Java:

```java
int i = 10;
while (i > 0) {
    System.out.println(i);
    i--;
}
```

- Código Pascal:

```pascal
var
  i: Integer;
begin
  i := 10;
  while i > 0 do
  begin
    writeln(i);
    i := i - 1;
  end.
```

4.4.2 Código Do-While para Java y Pascal

- Código Java:

```java
int i = 10;
do {
    System.out.println(i);
    i--;
} while (i > 0);
```

- Código Pascal (repeat-until):

```
var
  i: Integer;
begin
  i := 10;
  repeat
    writeln(i);
    i := i - 1;
  until i = 0;
end.
```

El profesor recomienda prestar atención a las condiciones para evitar bucles infinitos y probar el código en incrementos pequeños para verificar su comportamiento.

4.5 Repetición

La repetición implica ejecutar cierta acción una cantidad (conocida al momento de la compilación) de veces, usando un bucle de tipo `for`. El profesor destaca la flexibilidad del `for` en Java, que permite controlar la inicialización, la condición y la expresión de incremento.

4.5.1 Código Java y Pascal

- Código Java:

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

- Código Pascal:

```
for i := 1 to 10 do
begin
  writeln(i);
end.
```

El profesor sugiere experimentar con el código para entender mejor el funcionamiento del bucle `for`.

4.6 Conclusiones

En resumen, esta parte de la clase brinda una visión general de las estructuras de control en Java y Pascal. Se destacan similitudes y diferencias, y se brindan recomendaciones útiles para evitar errores comunes y fomentar una programación clara y ordenada.

Video 5

Clase 1 - Parte 5: Vectores y Matrices

5.1 Introducción

La clase está centrada en el uso de vectores y matrices en Java, explorando sus diferencias respecto a Pascal en CADP y sus propiedades. Se muestran ejemplos prácticos y se brindan recomendaciones para el uso adecuado de estas estructuras de datos.

5.2 Vectores en Java

Los vectores o arreglos, son estructuras de datos estáticas y homogéneas donde la dimensión se determina en tiempo de ejecución. A diferencia de Pascal en CADP, donde la dimensión se establece en tiempo de compilación, en Java se puede cambiar la dimensión física al crear una nueva instancia.

5.2.1 Declaración y Creación de Vectores

Para declarar un vector en Java, se indica el tipo de dato seguido de corchetes. Por ejemplo:

```
int[] numeros;  // Declaracion de un vector de enteros
```

Para crear un vector y definir su tamaño, se usa la sentencia 'new':

```
numeros = new int[10];  // Crea un vector de 10 posiciones
```

5.2.2 Acceso y Modificación de Vectores

Para acceder a un elemento de un vector, se utiliza su índice:

```
for (int i = 0; i < numeros.length; i++) {
    System.out.println("Pos " + i + ": " + numeros[i]);
}
```

Para modificar un vector, se pueden asignar valores a través de un bucle:

```
for (int i = 0; i < numeros.length; i++) {
    numeros[i] = i * 2;  // Asigna valores al vector
}
```

El profesor recomienta:

- Usar bucles para iterar sobre los elementos del vector.
- No acceder a posiciones no inicializadas para evitar excepciones de tiempo de ejecución.
- Inicializar los vectores antes de usarlos para evitar errores.
- Definir el tamaño del vector según las necesidades del programa.

5.3 Matrices en Java

Las matrices son vectores bidimensionales. Son estructuras de datos homogéneas y estáticas donde se utilizan dos índices para acceder a los elementos. La matriz se puede crear de varias maneras, dependiendo del número de filas y columnas deseadas.

5.3.1 Declaración y Creación de Matrices

Para declarar una matriz, se usan dos corchetes y para crearla, se emplea la sentencia 'new'. Por ejemplo:

```
int[][] matriz = new int[3][4];  // Crea una matriz 3x4
```

5.3.2 Acceso y Modificación de Matrices

Para acceder a un elemento en una matriz, se necesitan dos índices: fila y columna. Aquí un ejemplo para asignar valores:

```
for (int i = 0; i < matriz.length; i++) {
    for (int j = 0; j < matriz[i].length; j++) {
        matriz[i][j] = i + j;
}
```

Para imprimir el contenido de la matriz:

```
for (int i = 0; i < matriz.length; i++) {
    for (int j = 0; j < matriz[i].length; j++) {
        System.out.println("Pos [" + i + "][" + j + "]: "
        + matriz[i][j]);
    }
}
```

Si no deseo utilizar el método length, puedo hacerlo de la siguiente manera (asumiendo que conozco las dimensiones logicas o fisicas de la matriz):

```
// Por ejemplo con dimF1 y dimF2 defindas como constantes
for (int i = 0; i < dimF1; i++) {
    for (int j = 0; j < dimF2; j++) {
        System.out.println("Pos [" + i + "][" + j + "]: "
        + matriz[i][j]);
    }
}
```

El profesor recomienta:

- Para acceder a un elemento de una matriz, se usan dos índices: fila y columna.
- No acceder a posiciones no inicializadas para evitar traer basura.
- Probar ejemplos y ejercicios prácticos para mejorar el manejo de matrices.

5.4 Conclusiones

La clase muestra las diferencias entre vectores y matrices en Java respecto a Pascal en CADP. Se brindan ejemplos prácticos y se enfatiza la importancia

de inicializar correctamente las estructuras. Se recomienda a los estudiantes practicar con ejercicios para familiarizarse con estos conceptos y evitar errores comunes en tiempo de ejecución.

Video 6

Clase 2: Objetos, Clases, Instancias y Mensajes

6.1 Introducción

En esta clase, se profundiza en el paradigma de Programación Orientada a Objetos (POO), destacando sus características principales, sus diferencias con el paradigma imperativo y sus aplicaciones prácticas en el contexto del lenguaje Java. La sesión aborda conceptos clave como objetos, clases, instanciación, encapsulamiento y mensajes, proporcionando ejemplos claros y detallados para ilustrar cómo se implementan en el desarrollo de software.

6.2 Paradigmas de Programación

El profesor comienza explicando los paradigmas de programación más comunes, destacando el paradigma imperativo y el orientado a objetos. El paradigma imperativo es el enfoque más tradicional, donde el control del flujo del programa se realiza mediante estructuras de control como bucles y condiciones. Este paradigma sigue un enfoque secuencial y jerárquico para resolver problemas.

En contraste, el paradigma orientado a objetos se basa en la abstracción de elementos del mundo real como objetos, cada uno con su propio estado y comportamiento. Este paradigma permite una mayor modularidad y encapsulamiento, permitiendo que los objetos interactúen entre sí mediante mensajes.

El profesor utiliza una analogía con la carpintería para explicar la evolución de los paradigmas. Así como un carpintero usa herramientas específicas para diferentes tareas, un programador elige el paradigma adecuado para cada problema. No se trata de reemplazar un paradigma por otro, sino de sumar nuevas herramientas a medida que se adquieren habilidades y experiencia.

6.3 Introducción a la Programación Orientada a Objetos

El profesor presenta la Programación Orientada a Objetos como un enfoque diferente para estructurar programas. A través del ejemplo de un triángulo,

muestra cómo el paradigma imperativo descompone un problema en subproblemas, mientras que el paradigma orientado a objetos trata el triángulo como un objeto con atributos y comportamientos.

En el paradigma imperativo, se aborda el problema desde una perspectiva global, descomponiéndolo en partes más pequeñas y resolviendo cada parte por separado. Por ejemplo, para calcular el área y el perímetro de un triángulo, se dividiría el problema en subproblemas como leer los lados del triángulo, calcular el área y calcular el perímetro.

En la Programación Orientada a Objetos, el triángulo se trata como un objeto con atributos como los lados y colores, y comportamientos como calcular el área y el perímetro. Esto permite una mayor modularidad y reutilización de código, ya que el objeto puede tener múltiples comportamientos y puede ser reutilizado en diferentes contextos.

6.4 Objetos y Clases

Un objeto es una abstracción de un elemento del mundo real, que combina estado interno y comportamiento. El estado interno está compuesto por atributos que caracterizan al objeto y las relaciones con otros objetos. El comportamiento se refiere a las acciones que puede realizar el objeto, implementadas a través de métodos.

Una clase es el molde a partir del cual se crean los objetos. Describe un conjunto de objetos comunes y consta de la declaración de variables de instancia y la codificación de métodos que implementan el comportamiento. Al instanciar un objeto a partir de una clase, este hereda las variables de instancia y métodos definidos por la clase.

El profesor da varios ejemplos de objetos para ilustrar el concepto. Por ejemplo, un objeto puede ser un triángulo, un perro o un auto. Cada uno tiene atributos que describen su estado interno y métodos que definen su comportamiento.

```java
public class Triangulo {
    private int lado1;
    private int lado2;
    private int lado3;
    private String colorRelleno;
    private String colorLinea;

    public double calcularArea() {
        // Implementacion para
        // calcular el area
    public double calcularPerimetro() {
        return lado1 + lado2 + lado3;
    }

    public String obtenerColorRelleno() {
        return colorRelleno;
    }
}
```

Otro ejemplo es el de un perro, cuyo estado interno puede incluir la raza, edad y color de pelaje. El comportamiento puede ser ladrar, gruñir o aullar, entre otros. Este ejemplo muestra cómo diferentes objetos pueden tener distintos comportamientos según su contexto y uso.

```java
public class Perro {
    private String raza;
    private int edad;
    private String colorPelaje;

    public void ladrar() {
        // Comportamiento para ladrar
    }

    public void aullar() {
        // Comportamiento para aullar
    }
}
```

6.5 Encapsulamiento y Ocultamiento de datos

El encapsulamiento es un principio fundamental en la Programación Orientada a Objetos. Consiste en ocultar la implementación interna del objeto, exponiendo solo una interfaz pública para interactuar con él. Esto facilita el mantenimiento y evolución del sistema, ya que permite cambiar la implementación sin afectar a otros objetos que interactúan con él.

El ocultamiento de información significa que las variables de instancia y métodos privados no son accesibles desde fuera del objeto. Solo se pueden acceder a través de métodos públicos, que forman parte de la interfaz del objeto. Esto permite que los objetos controlen su propio estado interno y comportamiento sin exponer detalles internos a otros objetos.

En el siguiente ejemplo, el método privado 'metodoPrivado' no es accesible desde fuera del objeto, lo que garantiza que solo el objeto pueda modificar su estado interno. Sin embargo, el método público 'calcularPerimetro' es parte de la interfaz del objeto y puede ser utilizado por otros objetos para obtener información.

```
public class Triangulo {
    private int lado1;
    private int lado2;
    private int lado3;

    public double calcularPerimetro() {
        return lado1 + lado2 + lado3; // Metodo publico
    }

    private double metodoPrivado() {
        // Metodo privado no accesible desde fuera
    }
}
```

6.6 Instanciación y Creación de Objetos

La instanciación es el proceso de crear un objeto a partir de una clase. Se realiza mediante el uso de la palabra clave 'new', que reserva espacio en memoria para el objeto, ejecuta el constructor y devuelve una referencia al objeto recién creado. El constructor es un método especial que inicializa el estado interno del objeto al momento de la creación.

```
Triangulo tri = new Triangulo(10, 10, 10,
                "amarillo", "violeta");
```

6.7 Envío de Mensajes y Polimorfismo

El envío de mensajes es el mecanismo para interactuar con los objetos. Un mensaje provoca la ejecución de un método y puede llevar datos (parámetros) o devolver un resultado. El polimorfismo se refiere a la capacidad de objetos de diferentes clases de responder al mismo mensaje de diferentes maneras, dependiendo de su estado interno.

El profesor explica que el envío de mensajes se realiza utilizando el operador punto (‘.‘), seguido del nombre del método y, opcionalmente, los parámetros necesarios. Esto permite invocar métodos específicos para interactuar con objetos y obtener resultados.

Para enviar un mensaje a un objeto, se utiliza el operador punto (‘.‘) seguido del nombre del método y los parámetros necesarios. El siguiente ejemplo muestra cómo se envía un mensaje a un objeto para obtener su longitud y otro para obtener un carácter en una posición específica.

```
public class Demo01EnvioMensaje {
    public static void main(String[] args) {
      String saludo1 = new String("hola");
      System.out.println(saludo1.length());
      // Imprime 4
      System.out.println(saludo1.charAt(0));
      // Imprime 'h'
      System.out.println(saludo1.toUpperCase()
                        .equals("HOLA"));
      // Compara con mayusculas
    }
}
```

El polimorfismo permite que objetos de diferentes clases respondan al mismo mensaje de diferentes maneras. Esto se logra mediante la herencia y la sobrescritura de métodos. El profesor da el ejemplo de objetos gráficos como círculos y triángulos, que pueden tener el mismo método "dibujar", pero implementado de manera diferente según su estado interno. A continuación, se muestra un ejemplo simple de polimorfismo en Java (que sera complementado en otros ejemplos mas complejos en los próximos videos).

```java
abstract class Forma {
    abstract void dibujar();
}
class Circulo extends Forma {
    void dibujar() {
        System.out.println("Dibujando un circulo.");
    }
}
class Cuadrado extends Forma {
    void dibujar() {
        System.out.println("Dibujando un cuadrado.");
    }
}
public class DemoPolimorfismo {
    public static void main(String[] args) {
        Forma miForma;
        Circulo miCirculo = new Circulo();
        Cuadrado miCuadrado = new Cuadrado();

        miForma = miCirculo;
        miForma.dibujar();  // Dibujando un circulo.

        miForma = miCuadrado;
        miForma.dibujar();  // Dibujando un cuadrado.
    }
}
```

6.8 Recolector de Basura y Manejo de Memoria

El recolector de basura es un componente importante en la programación en Java. Se encarga de liberar memoria de objetos no referenciados, evitando problemas de fuga de memoria y optimizando el uso de recursos. El profesor explica que Java incluye un componente especial llamado "Garbage Collector", que detecta objetos no referenciados y los elimina de la memoria.

El recolector de basura tiene un costo asociado, ya que implica un proceso adicional que se ejecuta en segundo plano. Sin embargo, su uso es fundamental para garantizar que el programa no consuma memoria innecesariamente y que los objetos no referenciados se eliminen correctamente.

6.9 Conclusiones

En resumen, la Programación Orientada a Objetos es un paradigma poderoso para el desarrollo de software, permitiendo una mayor modularidad, encapsulamiento y polimorfismo. Al utilizar objetos, clases, instanciación y mensajes, se puede estructurar el software de manera más flexible y escalable.

El encapsulamiento permite ocultar la implementación interna del objeto, facilitando el mantenimiento y la evolución del sistema. El polimorfismo permite

reutilizar código y crear sistemas más versátiles. El uso de mensajes para interactuar con objetos proporciona flexibilidad y versatilidad en la implementación de soluciones.

El recolector de basura es un componente esencial en Java, que ayuda a mantener un uso eficiente de la memoria. Aunque tiene un costo asociado, su uso es fundamental para evitar problemas de fuga de memoria y mantener el sistema funcionando de manera óptima.

Con estos conceptos clave, la Programación Orientada a Objetos se presenta como un paradigma poderoso y versátil para el desarrollo de software, permitiendo crear aplicaciones más robustas y mantenibles.

Video 7

Clase 3 - Parte 1: Crear Clase

7.1 Introducción

Se ven conceptos fundamentales de la Programación Orientada a Objetos (POO) en Java y se centra en la creación de clases personalizadas. El ejemplo principal utilizado es la clase "Libro", que representa un libro con varios atributos y métodos asociados. A través de este ejemplo, el profesor ilustra conceptos como encapsulamiento, getters, setters y el método toString. Se incluyen fragmentos de código, así como las recomendaciones mencionadas a lo largo de la clase.

7.2 Creación de Clases y Atributos

El profesor comienza explicando qué es una clase y cómo actúa como un molde para crear objetos. En este contexto, una clase puede reutilizarse para crear múltiples instancias. El ejemplo utilizado es la creación de una clase que represente libros, donde cada libro tiene atributos como título, primer autor, editorial, año de edición, ISBN y precio.

```
public class Libro {
    private String titulo;
    private String primerAutor;
    private String editorial;
    private int anioEdicion;
    private String ISBN;
    private double precio;
}
```

Los atributos representan el estado del objeto, y cada atributo tiene un tipo de dato asociado. El profesor resalta que las variables de instancia deben ser privadas para lograr encapsulamiento, evitando el acceso directo desde fuera de la clase. La encapsulación es un concepto clave en la POO, ya que permite proteger datos sensibles y controlar el acceso a ellos. El profesor también menciona

que es posible inicializar las variables de instancia con valores por defecto al momento de la declaración. Por ejemplo, para la clase Libro, se pueden inicializar valores por defecto para el año de edición y el precio:

```
public class Libro {
    private String titulo = "Desconocido";
    private int anioEdicion = 2015;
    private double precio = 100.0;
}
```

Estas inicializaciones proporcionan valores predeterminados en caso de que no se proporcionen durante la creación de la instancia.

7.3 Encapsulamiento y Métodos Getter y Setter

El encapsulamiento es fundamental en la POO porque ayuda a mantener el control sobre los datos y protege el estado interno de los objetos. Para permitir el acceso controlado a las variables de instancia, se utilizan métodos "getter", que devuelven el valor de un atributo, y métodos "setter", que permiten modificar el valor. Los getters y setters permiten a los usuarios de la clase interactuar con sus atributos de manera controlada, sin acceder directamente a las variables privadas. Aquí hay un ejemplo de getters y setters para la clase Libro:

```java
public class Libro {
    private String titulo;
    private double precio;

    // Getter para el titulo
    public String getTitulo() {
        return titulo;
    }

    // Setter para el titulo
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    // Getter para el precio
    public double getPrecio() {
        return precio;
    }

    // Setter para el precio
    public void setPrecio(double precio) {
        this.precio = precio;
    }
}
```

El profesor recomienda siempre utilizar getters y setters para acceder y modificar las variables de instancia privadas. Este enfoque permite controlar y validar los valores antes de asignarlos a las variables, evitando errores y asegurando la consistencia del estado interno de la clase.

7.4 Constructores y Método toString

Un constructor es un método especial que se ejecuta cuando se crea una nueva instancia de la clase. Se utiliza para inicializar las variables de instancia y establecer valores por defecto si es necesario. El profesor muestra cómo crear un constructor para la clase Libro que recibe varios parámetros para inicializar los atributos:

```java
public class Libro {
    private String titulo;
    private String primerAutor;
    private String editorial;
    private int aniooEdicion;
    private String ISBN;
    private double precio;

    // Constructor con parametros
    public Libro(String titulo, String primerAutor,
                 String editorial, int aniooEdicion,
                 String ISBN, doble precio) {
        this.titulo = titulo;
        this.primerAutor = primerAutor;
        this.editorial = editorial;
        este.anioEdicion = anioEdicion;
        este.ISBN = ISBN;
        este.precio = precio;
    }
}
```

Los constructores permiten crear instancias con valores iniciales específicos, lo cual es útil para personalizar los objetos al momento de su creación. El profesor también menciona que Java proporciona un constructor por defecto si no se define uno, pero es recomendable definir un constructor propio para tener más control sobre la inicialización.

El método toString es un método especial en Java que devuelve una representación en forma de cadena del objeto. Es útil para depurar y para obtener una descripción del objeto. El profesor muestra cómo sobrescribir el método toString para personalizar la salida y proporcionar información significativa:

```java
public class Libro {
    // Otros atributos y metodos

    // Metodo toString personalizado
    @Override
    public String toString() {
        return titulo + " por " + primerAutor + " - "
        + editorial + " (" + anioEdicion + "), ISBN: "
        + ISBN + ", Precio: " + precio;
    }
}
```

El profesor recomienda sobrescribir el método toString para proporcionar una representación clara y significativa del objeto. Esto permite imprimir detalles importantes cuando se llama a System.out.println(objeto), lo que facilita

la depuración y el seguimiento del estado de los objetos.

7.5 Parámetros y Pasaje por Valor

En Java, los parámetros se pasan por valor, lo que significa que el método recibe una copia del valor del parámetro actual. Este principio aplica a datos primitivos y a referencias de objetos. Sin embargo, el efecto de este pasaje por valor varía dependiendo del tipo de parámetro.

7.5.1 Parámetros de Datos Primitivos

Para datos primitivos como enteros, decimales o booleanos, el pasaje por valor significa que cualquier cambio realizado dentro del método solo afecta a la copia, sin modificar el valor original. El profesor muestra un ejemplo para ilustrar este concepto:

```
public class EjemploPasajePorValor {
    public static void hacerUno(int y) {
        y++; // Modifica la copia del parametro
    }

    public static void main(String[] args) {
        int x = 1;
        hacerUno(x); // Pasaje por valor
        System.out.println("Valor de x: " + x);
        // Imprime: Valor de x: 1
    }
}
```

En este ejemplo, el método hacerUno recibe un parámetro entero y lo incrementa. Sin embargo, al imprimir el valor de x después de llamar a hacerUno, se observa que el valor no ha cambiado, ya que solo se modificó la copia del parámetro dentro del método.

7.5.2 Parámetros de Objetos

Cuando se trata de objetos, el pasaje por valor significa que el parámetro formal recibe una copia de la referencia al objeto. Esto implica que cualquier cambio realizado en el estado interno del objeto afectará al objeto original. El profesor proporciona un ejemplo para demostrar esta idea:

```java
public class EjemploPasajePorValor {
    public static hacerDos(Libro libro) {
        libro.setTitulo("Nuevo Titulo");
        // Modifica el objeto original
    }

    public static void main(String[] args) {
        Libro libro1 =  new Libro("Java Programming",
                        "Autor A", "Editorial X", 2022,
                        "1234567890", 50.0);
        hacerDos(libro1); // Modifica el objeto original
        System.out.println("Titulo del libro1: " +
                           libro1.getTitulo());
        // Imprime: Titulo del libro1: Nuevo Titulo
    }
}
```

En este caso, el método hacerDos recibe un objeto de tipo Libro y modifica su título. Después de llamar a hacerDos, el cambio se refleja en el objeto original, demostrando que, aunque el parámetro se pasa por valor, la referencia al objeto es la misma, por lo que los cambios afectan al objeto original.

Sin embargo, si el método asigna un nuevo objeto al parámetro, esto no afecta al objeto original porque se está trabajando con una copia de la referencia. Aquí hay un ejemplo para ilustrar esta diferencia:

```java
public class EjemploPasajePorValor {
    public static hacerTres(Libro libro) {
        libro = new Libro("Otro Libro", "Autor B",
                          "Editorial Y",2023,
                          "0987654321", 60.0);
                          // Nuevo objeto
    }

    public static void main(String[] args) {
        Libro libro1 = new Libro("Java Programming",
                                 "Autor A", "Editorial X",
                                  2022, "1234567890", 50.0);
        hacerTres(libro1);
        // No modifica el objeto original
        System.out.println("Titulo del libro1: " + 0
                           libro1.getTitulo());
        // Imprime: Titulo del libro1: Java Programming
    }
}
```

```java
public class EjemploPasajePorValor {
    public static hacerTres(Libro libro) {
        libro = new Libro("Otro Libro", "Autor B",
                          "Editorial Y", 2023,
                          "0987654321", 60.0);
                          // Nuevo objeto
    }

    public static void main(String[] args) {
        Libro libro1 = un Libro("Java Programming",
                                "Autor A", "Editorial X",
                                2022, "1234567890", 50.0);
        hacerTres(libro1);
        // No modifica el objeto original
        System.out.println("Titulo del libro1: " +
                           libro1.getTitulo());
        // Imprime: Titulo del libro1: Java Programming
    }
}
```

En este caso, el método hacerTres asigna un nuevo objeto al parámetro, pero como el parámetro es una copia de la referencia, este cambio no afecta al objeto original. Esto demuestra la importancia de entender cómo funciona el pasaje por valor en Java.

7.6 Métodos de Clase y Constructores Adicionales

Los métodos de clase son funciones que definen el comportamiento de la clase y permiten interactuar con sus objetos. Estos métodos pueden ser públicos o privados, dependiendo de si se desea que sean accesibles desde fuera de la clase o solo internamente.

El profesor muestra cómo crear diferentes métodos para la clase Libro, como setters y getters adicionales, así como métodos personalizados para realizar acciones específicas. A continuación se muestra un ejemplo de métodos para la clase Libro:

```java
public class Libro {
    // Atributos privados
    private String titulo;
    private String primerAutor;
    private String editorial;
    private int anioEdicion;
    private String ISBN;
    private double precio;

    // Constructor por defecto
    public Libro() {
        este.titulo = "Desconocido";
        este.primerAutor = "Autor Desconocido";
        esta.editorial = "Editorial Desconocida";
        este.anioEdicion = 2022;
        este.ISBN = "1234567890";
        este.precio = 50.0;
}

    // Metodo para obtener informacion del libro
    public String obtenerInformacion() {
        return "Titulo: " + titulo + ", Autor: " +
               primerAutor + ", Editorial: " +
               editorial + ", Anio: " + anioEdicion +
               ", ISBN: " + ISBN + ", Precio: " + precio;
    }

    // Otros metodos como getters y setters
}
```

En este ejemplo, se incluye un constructor por defecto que inicializa las variables de instancia con valores predeterminados. También se muestra un método personalizado obtenerInformacion, que devuelve información detallada del libro como una cadena. Estos métodos permiten interactuar con los objetos de manera controlada y proporcionan funcionalidad adicional para la clase.

7.7 Conclusiones

El profesor concluye la clase enfatizando la importancia del encapsulamiento, los métodos getter y setter, el método toString, y la diferencia entre pasaje por valor para datos primitivos y para objetos. También resalta la necesidad de definir constructores para personalizar la creación de objetos y el uso de métodos para implementar el comportamiento de la clase.

Video 8

Clase 3 - Parte 2: Interacciones entre Objetos

8.1 Introducción

La clase 3, Parte 2, explora conceptos de programación orientada a objetos (POO) en Java, centrándose en la interacción entre objetos. El profesor aborda temas clave como la instanciación de objetos, la declaración de constructores, la sobrecarga, la interacción entre objetos y el uso de la referencia "this". Estos conceptos son fundamentales para comprender y aplicar la POO en proyectos de software.

8.2 Instanciación de Objetos

La instanciación es el proceso de crear nuevas instancias de una clase. En Java, se utiliza la palabra clave "new" para crear objetos, lo que desencadena una serie de pasos que incluyen la asignación de valores por defecto a las variables de instancia y la llamada a un constructor para personalizar la creación del objeto.

El profesor describe cómo la referencia al objeto se almacena en una variable para que se pueda interactuar con el objeto más tarde. Los objetos pueden tener métodos que definen su comportamiento y variables de instancia que representan su estado.

8.2.1 Ejemplo de Instanciación de Objetos

```
public class Producto {
    private String nombre;
    private double precio;

    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    public String getNombre() {
        return this.nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public double getPrecio() {
       return this.precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }
}

public class ProgramaPrincipal {
    public static void main(String[] args) {
        Producto producto1 = new Producto("P 1", 100);
        Producto producto2 = new Producto("P 2", 150);

        System.out.println("Nombre del Producto 1: " +
                           producto1.getNombre());
        System.out.println("Precio del Producto 2: "+
                          producto2.getPrecio());
    }
}
```

Este ejemplo muestra cómo se instancian objetos de la clase "Producto" utilizando un constructor personalizado. El uso de "new" es fundamental para la instanciación, y los métodos getter y setter permiten interactuar con el objeto para obtener y establecer valores.

8.3 Declaración de Constructores

Un constructor es un método especial que se utiliza para inicializar objetos. El profesor menciona que cada clase en Java debe tener al menos un constructor.

Si no se define uno explícitamente, Java proporciona un constructor predeterminado sin parámetros. Sin embargo, al agregar un constructor personalizado, Java ya no incluye automáticamente el constructor predeterminado.

El constructor puede recibir diferentes parámetros para personalizar la creación del objeto. El profesor destaca que un constructor debe tener el mismo nombre que la clase y puede realizar tareas adicionales como inicializar variables de instancia y asignar valores personalizados.

8.3.1 Ejemplo de Declaración de Constructores

```
public class Libro {
    private String titulo;
    private String editorial;
    private int anioEdicion;
    private String ISBN;

    public Libro(String titulo, String editorial,
                 int anioEdicion, String ISBN) {
        this.titulo = titulo;
        this.editorial = editorial;
        this.anioEdicion = anioEdicion;
        this.ISBN = ISBN;
    }

    public String getTitulo() {
        return this.titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public String getEditorial() {
        return this.editorial;
    }

    public void setEditorial(String editorial) {
        this.editorial = editorial;
    }
```

Este ejemplo muestra cómo se declara un constructor personalizado para la clase "Libro". El constructor recibe parámetros como "título", "editorial", "añoEdición", y "ISBN", y asigna estos valores a las variables de instancia correspondientes. El profesor destaca que esta capacidad de personalización es clave para la programación orientada a objetos.

8.4 Sobrecarga de Constructores

La sobrecarga de constructores permite tener varios constructores con el mismo nombre pero con diferentes parámetros. El profesor señala que esto brinda flexibilidad para crear objetos de diferentes maneras, permitiendo adaptaciones para satisfacer diversas necesidades. La sobrecarga facilita la creación de objetos con diferentes configuraciones y evita la necesidad de crear múltiples clases para diferentes propósitos.

8.4.1 Ejemplo de Sobrecarga de Constructores

```
public class Libro {
    private String titulo;
    private String editorial;
    private int anioEdicion;
    private String ISBN.

    public Libro(String titulo, String editorial,
                 int anioEdicion, String ISBN) {
        this.titulo = titulo;
        this.editorial = editorial;
        this.anioEdicion = anioEdicion;
        this.ISBN = ISBN;
    }

    public Libro(String titulo) {
        this.titulo = titulo;
        this.editorial = "Editorial Desconocida";
        this.anioEdicion = 0;
        this.ISBN = "ISBN Desconocido";
    }

    public Libro() {
        this.titulo = "titulo Desconocido";
        this.editorial = "Editorial Desconocida";
        this.anioEdicion = 0;
        this.ISBN = "ISBN Desconocido";
    }
}
```

La sobrecarga permite tener varios constructores con diferentes combinaciones de parámetros, proporcionando flexibilidad para instanciar objetos de diferentes maneras. El profesor destaca que al utilizar la sobrecarga, se puede personalizar la creación de objetos sin necesidad de cambiar el diseño de la clase base.

8.5 Interacción entre Objetos

En la programación orientada a objetos, la interacción entre objetos es fundamental para coordinar acciones y realizar tareas comunes. El profesor explica que los objetos se comunican entre sí mediante referencias y mensajes, lo que permite que colaboren para resolver problemas y realizar funciones complejas.

8.5.1 Ejemplo de Interacción entre Libros y Autores

```
public class Autor {
    private String nombre;
    private String biografia;
    public Autor(String nombre, String biografia) {
        this.nombre = nombre;
        this.biografia = biografia;
    }
    public String getNombre() {
        return this.nombre.
    }
    public void setNombre(String nombre) {
        this.nombre = nombre.
    }
    public String getBiografia() {
        return this.biografia;
    }
    public void setBiografia(String biografia) {
        this.biografia = biografia;
    }
}
public class Libro {
    private String titulo;
    private Autor primerAutor;
    private String editorial;
    private int anioEdicion;
    private String ISBN.
    public Autor getPrimerAutor() {
        return this.primerAutor;
    }
    public void setPrimerAutor(Autor primerAutor) {
        this.primerAutor = primerAutor;
    }
    public String getTitulo() {
        return this.titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public double getPrecio() {
        return this.precio;
    }
    public void setPrecio(double precio) {
        this.precio = precio;
    }
}
```

El uso de referencias entre objetos permite coordinar acciones y distribuir responsabilidades. El profesor señala que esta interacción entre objetos fomenta

la modularidad y facilita el mantenimiento del software, ya que cada objeto tiene una responsabilidad específica.

8.6 La Referencia "this"

La referencia "this" es un elemento clave en Java para referenciar al objeto actual que está ejecutando un método. El profesor explica que "this" es útil para evitar ambigüedades cuando los parámetros tienen el mismo nombre que las variables de instancia. También se puede usar para llamar a métodos del mismo objeto o para acceder a variables de instancia.

8.6.1 Ejemplo de Uso de "this"

```
public class Libro {
    private String titulo;
    private double precio;

    public void setTitulo(String titulo) {
        this.titulo = titulo.
    }

    public double getPrecioFinalConIva() {
        return this.precio * 1.21.
    }
}
```

El uso de "this" garantiza que el método opere sobre el objeto correcto, evitando errores cuando hay ambigüedades en los nombres de variables y parámetros. El profesor destaca que "this" es esencial para mantener la consistencia y el buen funcionamiento del código.

8.7 Conclusiones

El profesor concluye la clase alentando a los estudiantes a practicar los conceptos aprendidos y aplicar los consejos y trucos dados durante la sesión. Se recomienda realizar ejercicios que involucren la creación de clases y la modificación de programas principales para poner en práctica los conceptos de programación orientada a objetos.

El profesor deja abierta la posibilidad de explorar otros temas relacionados con la programación orientada a objetos en futuras sesiones, alentando a los estudiantes a profundizar en la interacción entre objetos y en el uso de constructores y sobrecarga para crear programas más flexibles y robustos.

El profesor proporciona varios consejos a lo largo de la clase para mejorar la comprensión y la eficiencia en la programación orientada a objetos. Algunos de los consejos y recomendaciones destacados incluyen:

- **Uso de Constructores Personalizados:** El profesor enfatiza la importancia de definir constructores personalizados para inicializar objetos con valores específicos. Esto proporciona flexibilidad en la creación de objetos y evita errores relacionados con valores por defecto.
- **Sobrecarga de Métodos y Constructores:** El profesor muestra cómo la sobrecarga permite definir múltiples métodos y constructores con el mismo nombre pero con diferentes parámetros. Esto brinda flexibilidad para adaptar la creación de objetos a diferentes escenarios.
- **Interacción entre Objetos:** El profesor destaca la importancia de la interacción entre objetos y cómo se puede utilizar para resolver problemas complejos de manera colaborativa. La referencia a otros objetos y el envío de mensajes son elementos clave en la programación orientada a objetos.
- **Uso de "this":** El profesor explica que "this" es esencial para referenciar el objeto actual y evitar ambigüedades cuando los parámetros tienen el mismo nombre que las variables de instancia. Esto garantiza que los métodos operen sobre el objeto correcto y evita errores en el código.

Video 9

Clase 4 - Parte 1: Jerarquía y Herencia

9.1 Introducción

Este documento presenta un resumen extenso sobre la herencia y jerarquía en la programación orientada a objetos, basándose en la transcripción de una clase y documentos complementarios. El objetivo es profundizar en cómo estos conceptos fundamentales permiten la reutilización de código y estructuran el desarrollo de software, utilizando ejemplos en Java para ilustrar su aplicación práctica.

9.2 Fundamentos de la Programación Orientada a Objetos (POO)

La programación orientada a objetos es un paradigma de desarrollo que se centra en la creación de objetos que encapsulan datos y funciones relacionados. Durante la clase, el profesor comenzó con una revisión de cómo las clases se utilizan para modelar objetos con estados internos y comportamientos, utilizando atributos para almacenar datos y métodos para definir acciones. Se discutió la importancia de la encapsulación para mantener ocultos los detalles internos del objeto y exponer solo aquellas interfaces necesarias para la interacción con otros objetos.

```java
public class Libro {
    private String titulo;
    private Autor autor;
    private String isbn;
    public Libro(String titulo, Autor autor, String isbn){
        this.titulo = titulo;
        this.autor = autor;
        this.isbn = isbn;
    }
    public String getTitulo() {
        return titulo;
    }
    public Autor getAutor() {
        return autor;
    }
    public String getIsbn() {
        return isbn;
    }
}
public class Autor {
    private String nombre;
    private String biografia;
    public Autor(String nombre, String biografia) {
        this.nombre = nombre;
        this.biografia = biografia;
    }
    public String getNombre() {
        return nombre;
    }
    public String getBiografia() {
        return biografia;
    }
}
```

También se vio como interactúan los objetos entre sí, como en el caso de la clase 'Libro' que contiene un objeto de la clase 'Autor'. Esta relación de composición permite modelar estructuras más complejas y reflejar la realidad de manera más precisa. En esta nueva clase se discutirán los conceptos de herencia, polimorfismo y jerarquía como practicas de programación orientada a objetos que permiten mejorar la eficiencia y escalabilidad del código.

9.3 Motivación: Duplicación de código

La duplicación de código es un problema común en el desarrollo de software que puede complicar el mantenimiento y la escalabilidad de los proyectos. El profesor destacó este problema utilizando dos clases que comparten funcionalidades pero están implementadas de forma independiente. Este ejemplo prepara el contexto para introducir cómo la herencia puede optimizar la estructura del código y

reducir la redundancia.

Se utilizó el ejemplo de dos clases, ‘Automovil‘ y ‘Bicicleta‘, que comparten atributos y métodos comunes pero también tienen diferencias que justifican su separación inicial antes de introducir la herencia.

```java
public class Automovil {
    private double velocidad;
    private String color;
    private int numeroDePuertas;
    // Atributo especifico de Automovil

    public Automovil(String color, int numeroDePuertas) {
        this.color = color;
        this.velocidad = 0.0;
        this.numeroDePuertas = numeroDePuertas;
    }

    public void acelerar(double incremento) {
        velocidad += incremento;
    }

    public String getColor() {
        return color;
    }

    public int getNumeroDePuertas() {
        return numeroDePuertas;
    }
}

public class Bicicleta {
    private double velocidad;
    private String color;
    private boolean tieneCanasta;
    // Atributo especifico de Bicicleta

    public Bicicleta(String color, boolean tieneCanasta) {
        this.color = color;
        this.velocidad = 0.0;
        this.tieneCanasta = tieneCanasta;
    }

    public void acelerar(double incremento) {
        velocidad += incremento;
    }

    public String getColor() {
        return color;
    }

    public boolean tieneCanasta() {
        return tieneCanasta;
    }
}
```

Este código destaca cómo ‘Automovil‘ y ‘Bicicleta‘ gestionan propiedades y métodos similares para ‘velocidad‘ y ‘color‘, y cómo cada uno tiene un atributo adicional que refleja sus características únicas. El profesor (utilizando otro ejemplo) señaló que, a pesar de estas diferencias, la duplicación de código en métodos y atributos comunes podría eliminarse mediante la implementación de una superclase común. Esta discusión anticipa el potencial de la herencia para consolidar el código común y al mismo tiempo permitir diferencias específicas entre las subclases.

9.4 Introducción a la Herencia

La herencia permite que una clase derive de otra, tomando sus atributos y métodos como base, lo cual promueve una mayor organización y reutilización de código. El profesor ilustró este concepto mediante la extensión de clases, donde la clase derivada (subclase) hereda las propiedades de la clase base (superclase), permitiendo modificar o añadir nuevas funcionalidades.

```
public class Figura {
    private String color;

    public Figura(String color) {
        this.color = color;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}


public class Circulo extends Figura {
    private double radio;
    public Circulo(double radio, String color) {
        super(color);
        this.radio = radio;
    }

    public double calcularArea() {
        return Math.PI * Math.pow(radio, 2);
    }
}
```

9.5 Profundización en Polimorfismo

El polimorfismo es un principio de POO que permite que objetos de diferentes clases respondan a las mismas acciones de diferentes maneras. Se abordó cómo el polimorfismo y la herencia trabajan juntos para permitir que objetos de una subclase puedan ser tratados como objetos de una superclase, pero comportándose de manera única según su tipo específico.

```
public class Triangulo extends Figura {
    private double base;
    private double altura;

    public Triangulo(double base, double altura, String col
        super(color);
        this.base = base;
        this.altura = altura;
    }

    @Override
    public double calcularArea() {
        return (base * altura) / 2;
    }
}
```

9.6 Herencia en Práctica: Ejemplos Avanzados

El concepto de herencia fue explorado más a fondo mediante ejemplos que ilustran la creación de jerarquías de clases. La discusión se centró en cómo estas jerarquías facilitan la gestión del código en proyectos más complejos y cómo pueden ayudar a evitar la repetición de código.

```
public class Cuadrado extends Figura {
    private double lado;

    public Cuadrado(double lado, String color) {
        super(color);
        this.lado = lado;
    }

    @Override
    public double calcularArea() {
        return lado * lado;
    }
}
```

9.7 Búsqueda de Método en la Jerarquía de Clases

La búsqueda de métodos en una jerarquía de clases es un aspecto crucial de la herencia. Se explicó cómo Java busca el método apropiado para ejecutar cuando se invoca un método en un objeto, siguiendo la cadena de herencia desde la subclase hasta la superclase hasta encontrar el método o lanzar un error si no se encuentra.

```
// En la clase Cuadrado, se invoca el metodo calcularArea
// especifico de Cuadrado.
Cuadrado cuadrado = new Cuadrado(10, "rojo");
System.out.println("Area cuadrado: " +
                   cuadrado.calcularArea());

// Si se invoca un metodo que no esta en Cuadrado
// pero si en Figura,
// Java busca hacia arriba en la jerarquia.
System.out.println("Color cuadrado: "
                   + cuadrado.getColor());
```

9.8 Consejos para Usar Herencia eficientemente

El profesor compartió varios consejos y mejores prácticas para implementar herencia de manera efectiva en proyectos de programación. Estos incluyen: - Evitar la Herencia Excesiva: Usar herencia solo cuando es lógico y no complicar la jerarquía de clases innecesariamente. - Principio de Sustitución de Liskov: Asegurarse de que las subclases puedan reemplazar a las superclases sin afectar el comportamiento del programa. - Uso de Polimorfismo: Aprovechar el polimorfismo para mejorar la flexibilidad y la extensibilidad del código.

9.9 Conclusiones

La herencia y el polimorfismo son piedras angulares de la programación orientada a objetos que proporcionan un marco robusto para estructurar programas de manera lógica y eficiente. A través de los ejemplos y discusiones en clase, se demostró cómo estos conceptos no solo facilitan la reutilización de código, sino que también permiten la construcción de sistemas más flexibles y mantenibles. Al adherirse a las buenas prácticas y comprender profundamente cómo Java maneja la herencia y el polimorfismo, los programadores pueden maximizar los beneficios de estos poderosos principios de diseño.

Video 10

Clase 4 - Parte 2: Clases y Métodos Abstractos

10.1 Introducción

En esta segunda parte de la serie sobre programación orientada a objetos, nos enfocamos en la profundización de la herencia a través de la introducción de clases y métodos abstractos. Estos conceptos no solo refuerzan la jerarquía y la estructura en el diseño de software orientado a objetos, sino que también ofrecen una manera elegante de manejar situaciones donde la creación directa de instancias de ciertas clases no es deseada o necesaria. Exploraremos cómo estos mecanismos permiten una definición más clara y formal de las responsabilidades que deben cumplir las subclases en un entorno de herencia.

10.2 Fundamentos de Clases Abstractas y Métodos Abstractos

El concepto de clases abstractas se introduce como una evolución natural en la gestión de la herencia dentro de la programación orientada a objetos. Una clase abstracta se define como una clase que no puede ser instanciada por sí misma, sirviendo únicamente como una superclase para otras subclases. Esta característica es crucial para evitar la duplicación de código y asegurar que ciertos métodos esenciales sean implementados por todas las subclases derivadas.

```java
public abstract class Figura {
    private String colorRelleno;
    private String colorLinea;

    public String getColorRelleno() {
        return this.colorRelleno;
    }

    public void setColorRelleno(String color) {
        this.colorRelleno = color;
    }

    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

10.3 La Necesidad de Métodos Abstractos

Los métodos abstractos son aquellos que no tienen implementación en la clase abstracta donde se declaran, lo que obliga a cada subclase concreta a proporcionar una implementación específica. Esto asegura que todas las subclases mantengan una interfaz uniforme para ciertas operaciones, fortaleciendo el principio de polimorfismo y aumentando la coherencia del diseño.

```java
    public class Cuadrado extends Figura {
        private double lado;

        public Cuadrado(double lado, String colorRelleno,
                        String colorLinea) {
            setColorRelleno(colorRelleno);
            setColorLinea(colorLinea);
            this.lado = lado;
        }

        @Override
        public double calcularArea() {
            return lado * lado;
        }

        @Override
        public double calcularPerimetro() {
            return 4 * lado;
        }
    }
```

10.4 Aplicaciones Prácticas de Clases Abstractas

La aplicación de clases abstractas se extiende más allá de la simple prevención de instancias. Estas clases actúan como cimientos para arquitecturas de software donde diferentes componentes comparten una serie de características comunes pero también requieren implementar comportamientos específicos que dependen del contexto de la subclase.

10.5 Beneficios de Utilizar Clases Abstractas

Las clases abstractas ofrecen varios beneficios en el desarrollo de software orientado a objetos. Al centralizar y compartir código común, reducen la duplicación y facilitan el mantenimiento. Además, al definir una estructura clara que las subclases deben seguir, las clases abstractas mejoran la integridad del diseño y aseguran que todas las extensiones de una clase base manejen adecuadamente las funcionalidades esenciales.

10.6 Forzando la Implementación de Comportamientos Esenciales

Uno de los mayores beneficios de las clases abstractas es la capacidad de obligar a las subclases a implementar métodos que son cruciales para el funcionamiento del sistema. Esto es especialmente útil en entornos donde el comportamiento de las subclases debe ser garantizado y uniforme, independientemente de las diferencias en los detalles de implementación de cada subclase.

```
public class Automovil extends Vehiculo {
    public Automovil() {
        super("Automovil");
    }

    @Override
    public void mover() {
        System.out.println("El automovil se
                            mueve sobre ruedas.");
    }
}

public class Avion extends Vehiculo {
    public Avion() {
        super("Avion");
    }

    @Override
    public void mover() {
        System.out.println("El avion vuela
                            en el aire.");
    }
}
```

10.7 Mejoras en el Polimorfismo

El uso de clases abstractas refuerza el polimorfismo al permitir que objetos de diferentes clases derivadas sean tratados como objetos de una sola clase base. Esto simplifica la gestión de diferentes tipos de objetos que comparten el mismo interfaz pero exhiben comportamientos diferentes bajo las mismas operaciones.

```
public static void iniciarMovimiento(Vehiculo v) {
    v.mover();
    // Llamada polimorfica: el metodo especifico
    // se determina en tiempo de ejecucion.
}

public static void main(String[] args) {
    Vehiculo miAuto = new Automovil();
    Vehiculo miAvion = new Avion();
    iniciarMovimiento(miAuto);
    iniciarMovimiento(miAvion);
}
```

10.8 Conclusiones

Las clases y métodos abstractos son herramientas poderosas en la programación orientada a objetos que ofrecen robustez y flexibilidad en el diseño de software. Permiten una clara separación y definición de lo que debe ser implementado, asegurando una base sólida para futuras extensiones y mejoras. Al requerir que las subclases concretas implementen métodos abstractos específicos, las clases abstractas garantizan que todas las implementaciones derivadas se adhieran a un contrato común, lo cual es esencial para mantener la coherencia y la predictibilidad del comportamiento en sistemas complejos.

Video 11

Clase 4 - Parte 3: Super, Polimorfismo y Binding Dinámico

11.1 Introducción

En esta última parte de la serie sobre programación orientada a objetos, nos centramos en conceptos avanzados que son cruciales para una gestión eficaz de la herencia y el comportamiento dinámico en sistemas orientados a objetos. Los conceptos de 'super', polimorfismo y binding dinámico no solo proporcionan herramientas para escribir código robusto y mantenible, sino que también permiten una mayor flexibilidad y reusabilidad del código en diversas aplicaciones de software.

11.2 Uso Avanzado de 'super'

El operador 'super' en Java es esencial para interactuar con la clase padre desde una subclase. Este operador permite a las subclases acceder a métodos o constructores de la clase base, facilitando la extensión y modificación de comportamientos heredados sin la necesidad de reescribir métodos completamente.

11.2.1 Funcionalidad de 'super'

El uso de 'super' se manifiesta en dos formas principales: invocación de constructores de la clase padre y acceso a métodos que han sido sobrescritos por la subclase. Este mecanismo es especialmente útil en el contexto de constructor chaining, donde cada constructor en la jerarquía de herencia debe ser inicializado correctamente.

11.2.2 Ejemplo Detallado de 'super'

Consideremos una jerarquía de clases donde 'super' juega un papel crucial en la inicialización y comportamiento de las subclases:

```java
public class Figura {
    protected String color;

    public Figura(String color) {
        this.color = color;
    }

    public void dibujar() {
        System.out.println("Dibuja una figura de color "
                           + color);
    }
}

public class Circulo extends Figura {
    private double radio;

    public Circulo(double radio, String color) {
        super(color);  // Llama al constructor de Figura
        this.radio = radio;
    }

    @Override
    public void dibujar() {
        super.dibujar();
        // Invoca el metodo dibujar de Figura
        System.out.println("Dibuja un circulo de radio "
                           + radio);
    }
}
```

Este código ilustra cómo ‘super‘ es utilizado para asegurar que la funcionalidad de la clase base se preserve mientras se añaden características adicionales en la subclase.

11.3 Refactorización y Extensión de Métodos Comunes

Refactorizar y extender métodos comunes a través de la herencia permite un diseño de software más limpio y mantenible. Al centralizar el código común en la clase base, las subclases pueden extender o modificar este comportamiento básico sin duplicar código, lo que facilita las actualizaciones y el mantenimiento del software.

11.3.1 Implementación de Métodos Comunes

Mover código común, como métodos de utilidad o implementaciones predeterminadas de operaciones, a una clase base reduce la redundancia y mejora la

coherencia del diseño. Por ejemplo, en un sistema de UI, una clase base ‘Componente‘ podría implementar métodos para estilos comunes y manejo de eventos, mientras que subclases como ‘Boton‘ y ‘Slider‘ pueden especificar comportamientos y estilos particulares.

```
public abstract class Componente {
    protected String estilo;
    public Componente(String estilo) {
        this.estilo = estilo;
    }
    public void aplicarEstilo() {
        System.out.println("Aplicando estilo " + estilo);
    }
}
public class Boton extends Componente {
    public Boton(String estilo) {
        super(estilo);
    }
    @Override
    public void aplicarEstilo() {
        super.aplicarEstilo();
        System.out.println("Estilo especifico
                            del boton aplicado.");
    }
}
```

Este ejemplo muestra cómo la funcionalidad común de aplicación de estilos se centraliza en la clase ‘Componente‘, mientras que la clase ‘Boton‘ extiende esta funcionalidad para incluir aspectos específicos de los botones.

Continuaré con la segunda parte del documento en el próximo mensaje, detallando aún más sobre polimorfismo y binding dinámico, y proporcionando ejemplos adicionales para ilustrar estos conceptos.

11.4 Polimorfismo en Acción

El polimorfismo es una característica fundamental de la programación orientada a objetos que permite a objetos de diferentes clases responder de manera diferente al mismo método o mensaje. Esto no solo aumenta la flexibilidad del código, sino que también permite la creación de interfaces más genéricas y reutilizables.

11.4.1 Detalles del Polimorfismo

Polimorfismo se manifiesta principalmente en dos formas: polimorfismo de inclusión, donde objetos de diferentes subclases se tratan como objetos de una superclase, y polimorfismo paramétrico, que se refiere al uso de tipos genéricos en programación.

11.4.2 Ejemplos de Polimorfismo

Consideremos un sistema de formas geométricas donde cada forma tiene un método para calcular su área, pero la implementación varía según la forma específica:

```
public abstract class Forma {
    public abstract double calcularArea();
}
public class Rectangulo extends Forma {
    private double ancho;
    private double alto;
    public Rectangulo(double ancho, double alto) {
        this.ancho = ancho;
        this.alto = alto;
    }
    @Override
    public double calcularArea() {
        return ancho * alto;
    }
}
public class Circulo extends Forma {
    private double radio;
    public Circulo(double radio) {
        this.radio = radio;
    }
    @Override
    public double calcularArea() {
        return Math.PI * radio * radio;
    }
}
public static void imprimirArea(Forma forma) {
    System.out.println("El area de la forma es: "
                       + forma.calcularArea());
}
```

Este código ilustra cómo diferentes objetos (‘Rectangulo‘, ‘Circulo‘) pueden ser tratados como instancias de ‘Forma‘ y cómo cada uno responde de manera diferente al mismo mensaje (‘calcularArea‘).

11.5 Binding Dinámico y su Importancia

El binding dinámico es el proceso por el cual la llamada a un método se resuelve en tiempo de ejecución en lugar de en tiempo de compilación. Esto es esencial para implementar polimorfismo y para que el código pueda adaptarse a diferentes tipos de objetos en tiempo de ejecución.

11.5.1 Cómo Funciona el Binding Dinámico

En Java, el binding dinámico ocurre automáticamente para todos los métodos no estáticos. Java utiliza el tipo real del objeto para determinar qué método se debe invocar, incluso si el método es invocado usando una referencia de tipo de superclase.

11.5.2 Ejemplo de Binding Dinámico

Usando el ejemplo anterior de 'Forma', el método 'calcularArea' es seleccionado basado en el tipo real del objeto, ya sea 'Rectangulo' o 'Circulo', en el momento de la llamada:

```
Forma miForma = new Circulo(3);
System.out.println(miForma.calcularArea());
// Usa el metodo calcularArea de Circulo

miForma = new Rectangulo(4, 5);
System.out.println(miForma.calcularArea());
// Usa el metodo calcularArea de Rectangulo
```

Este comportamiento demuestra cómo Java maneja el binding dinámico, permitiendo que el mismo código funcione con diferentes tipos de objetos y realice operaciones específicas según el tipo real del objeto en tiempo de ejecución.

11.6 Interrelación entre Polimorfismo y Binding Dinámico

Polimorfismo y binding dinámico son conceptos estrechamente vinculados que juntos forman la base para muchas de las características más poderosas de la programación orientada a objetos. El polimorfismo permite que diferentes clases proporcionen diferentes implementaciones de un mismo método, mientras que el binding dinámico determina cuál de estas implementaciones se debe ejecutar en tiempo de ejecución.

11.6.1 Polimorfismo: Flexibilidad en el Comportamiento

El polimorfismo se manifiesta cuando una referencia de superclase puede apuntar a objetos de cualquiera de sus subclases, permitiendo que ese objeto ejecute su propia versión de un método. Esto proporciona una increíble flexibilidad en el diseño del software, permitiendo que nuevos tipos de objetos sean introducidos sin alterar el código que los utiliza.

11.6.2 Binding Dinámico: Implementación del Polimorfismo

El binding dinámico es el mecanismo por el cual se implementa el polimorfismo en tiempo de ejecución. Cuando se invoca un método a través de una referencia de superclase, el binding dinámico asegura que se ejecute la versión correcta del método, dependiendo del tipo real del objeto referenciado. Esto es esencial para que el polimorfismo funcione correctamente en la práctica.

11.6.3 Ejemplo Ilustrativo de Polimorfismo y Binding Dinámico

Consideremos un sistema donde diferentes tipos de trabajadores necesitan calcular su pago de manera diferente:

```
public abstract class Trabajador {
    public abstract double calcularPago();
}
public class Asalariado extends Trabajador {
    private double salario;
    public Asalariado(double salario) {
        this.salario = salario;
    }
    @Override
    public double calcularPago() {
        return salario / 52;
    }
}
public class PorHora extends Trabajador {
    private double horas;
    private double tarifa;
    public PorHora(double horas, double tarifa) {
        this.horas = horas;
        this.tarifa = tarifa;
    }
    @Override
    public double calcularPago() {
        return horas * tarifa;
    }
}
public static void imprimirPago(Trabajador trabajador) {
    System.out.println("Pago: "
                       + trabajador.calcularPago());
}
Trabajador alice = new Asalariado(52000);
Trabajador bob = new PorHora(40, 15);
imprimirPago(alice);
imprimirPago(bob);
```

Este ejemplo muestra cómo 'Trabajador', una clase abstracta, puede ser extendida por 'Asalariado' y 'PorHora', cada una implementando el método 'calcularPago' de manera diferente. La función 'imprimirPago' puede aceptar cualquier objeto 'Trabajador' y, gracias al binding dinámico, el método 'calcularPago' apropiado se invoca dependiendo del tipo real del objeto pasado.

11.7 Conclusiones

La comprensión profunda y la aplicación efectiva de 'super', polimorfismo y binding dinámico son fundamentales para cualquier programador que trabaje con Java y otros lenguajes orientados a objetos. Estos conceptos permiten desarrollar software que es robusto, mantenible, y extensible. A través del uso adecuado de estas características, los desarrolladores pueden crear sistemas más modulares y reutilizables, que son capaces de manejar una amplia variedad de situaciones de manera eficiente.

La relación entre polimorfismo y binding dinámico es fundamental para el diseño de sistemas robustos y flexibles en la programación orientada a objetos. La capacidad de tratar diferentes objetos a través de una interfaz común, mientras se permite que sus comportamientos varíen, es lo que hace que el polimorfismo sea tan poderoso. El binding dinámico, al resolver llamadas a métodos en tiempo de ejecución, asegura que el comportamiento correcto se ejecute en función del tipo específico de objeto, maximizando así la flexibilidad y reusabilidad del código.