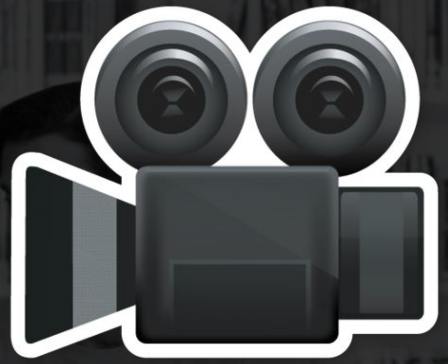
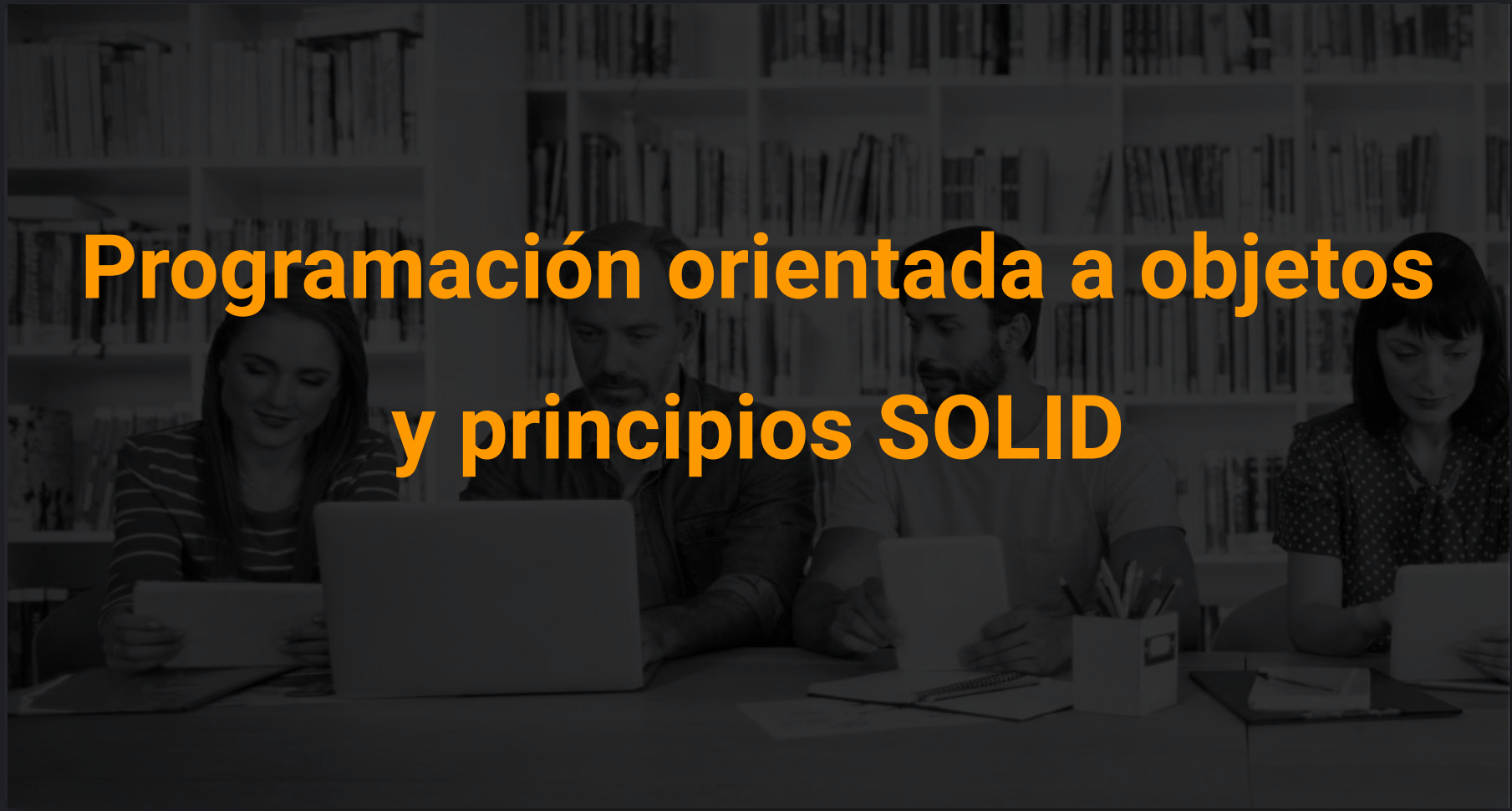
Bootcamp Java

Clase 4

Recuerda poner a grabar la clase.




Programación orientada a objetos
y principios SOLID

Introducción - Definición de paradigma

- Podemos concebir este concepto como a un conjunto de ideas, una forma de pensar y abordar ciertas problemáticas. En nuestro caso, problemáticas informáticas que se resuelven operando de una manera específica, empleando mecanismos específicos.

Objetos en Java

01

Beneficios de la Programación orientada a objetos en Java

Otorga un marco de desarrollo sólido, generando prácticas y patrones de diseño que facilitan la automatización de tareas.

La **OOP** implementa una serie de mecanismos para lograr lo anterior; estos son: **herencia, polimorfismo, encapsulamiento y abstracción.**

01

Java: Un lenguaje de clases

En lenguajes de programación previos a Java, como por ejemplo C, la unidad de desarrollo eran las funciones.

Con la aparición de C++ y la OOP pasó un nivel más avanzado, donde la unidad de desarrollo pasaron a ser clases.

01

Java: Un lenguaje de clases

Las clases se encargan de reunir funciones (métodos) y datos que las definen (atributos). Los objetos no son otra cosa que instancias de una clase (o de varias clases, como ya veremos más adelante). Dicho de otra forma, las clases nos permiten crear objetos.

01

Constructores

La creación o instancia de un objeto, a partir de una clase, se logra a partir de un método especial que las clases incorporan, llamado **constructor**, y funciona dando instancia a los atributos de una clase. Se suele implementar un constructor vacío y otro con una cantidad específica de atributos, pueden ser todos.

```
Constructor vacio:
public Estudiante() {
}
Constructor completo:
public Estudiante(Integer id, String nombreCompleto, Float promedio,
Boolean presentismo) {
   this.id = id;
   this.nombreCompleto = nombreCompleto;
   this.promedio = promedio;
   this.presentismo = presentismo;
}
```

01

... Estructura de una clase

```
Modificador de acceso (opcional)] Class [Nombre de la clase] {
    (Atributos de la clase)
    [modificador] tipo campo1;
    ...
    [modificador] tipo campoN;
    Constructor(es) de la clase() {}
    (Métodos de la clase)
    tipo método1( parámetros); ...
    tipo métodoN( parámetros);
}
```

01

Consideraciones sobre clases

- En Java no existen variables ni métodos globales. Todas las variables y métodos deben pertenecer a una clase.
- Cuando una clase se extiende a otra hereda todas sus atributos y métodos.
- En Java no existe la herencia múltiple.
- Object es la base de toda la jerarquía de clases de Java. Si al definir una clase no se especifica la clase que extiende, por default deriva de Object.
- Por convención, los métodos se declaran como *public*, mientras que los atributos se declaran como *private*. A su vez, siempre se crea un constructor vacío.

01

Creación de una clase

A modo de ejemplo crearemos de manera completa la clase “Estudiante”, citada anteriormente.

En este bloque de código, se observa el listado de atributos y el constructor vacío.

```
public class Estudiante {
    private Integer id;
    private String nombreCompleto;
    private Float promedio;
    private Boolean presentismo;

    public Estudiante() {
    }
```

01

Creación de una clase

A continuación vemos el constructor completo y el método get & set de uno de los atributos (el resto no se listan, dado que son idénticos, pero con el nombre cambiado) y por último el método toString(), que transforma los atributos del objeto en caracteres.

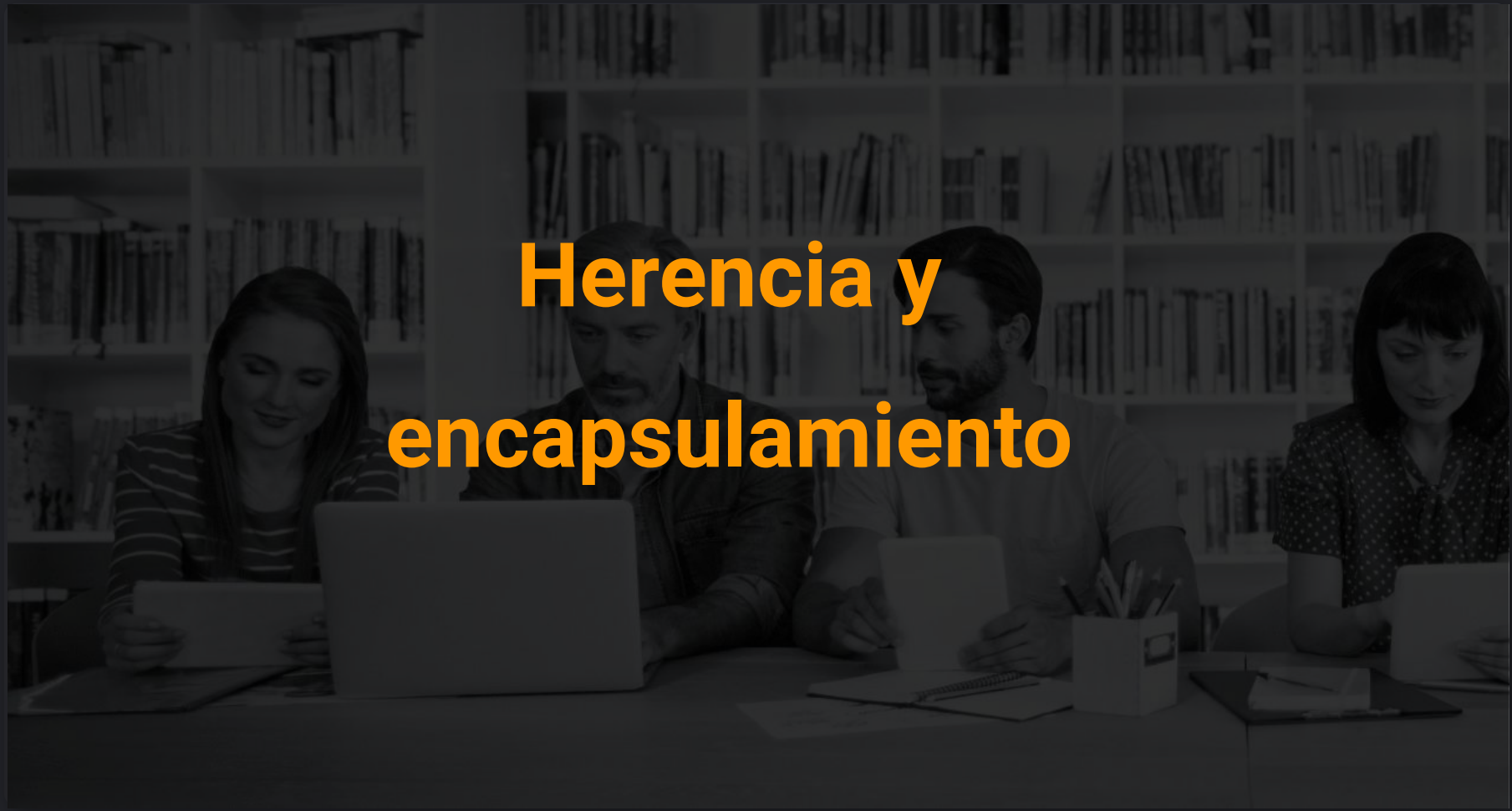
```
public Estudiante(Integer id, String nombreCompleto, Float promedio, Boolean presentismo) {
   this.id = id;
   this.nombreCompleto = nombreCompleto;
   this.promedio = promedio;
   this.presentismo = presentismo;
}
public Integer getId() {
   return this.id;
}
```

01

Creación de una clase

Con la anotación @Override forzamos al compilador a que emplee este método toString() que acabamos de crear, en vez de utilizar el que viene por defecto en la clase Object.

```
public void setId(Integer id) {
        this.id = id;
    }
    @Override
    public String toString() {
        return "{" + " id='" + getId() + "'" + ", nombreCompleto='" + getNombreCompleto() + "'" + ",
        promedio='"  + getPromedio() + "'" + ", presentismo='" + isPresentismo() + "'" + "}";
    }
```


Herencia y
encapsulamiento

02

Herencia y encapsulamiento de objetos

Idéntico a como se emplea el concepto coloquialmente, la **herencia** dentro de este paradigma implica el traspaso de atributos y métodos entre clases hijas y padres **(subclass y superclass)**. Mientras que el **encapsulamiento**, es la capacidad de una clase de ocultar ciertos elementos al exterior

La especificidad en este proceso se logra a través de los ya mencionados **modificadores de acceso.**

02

Modificadores de acceso

Cada vez que instanciamos un método o un atributo en una clase (tal como se hizo en el ejemplo anterior), le asignamos un modificador de acceso. Con esto estamos especificando para quiénes va a estar disponible cada componente. Por convención se suelen instanciar atributos (datos) de manera privada y métodos públicos.

02

Modificadores de acceso

Estos se encargan de dar distintos niveles de acceso, según lo siguiente:

	private	protected	public	package
clase	√	√	√	√
subclase		√	√	
paquete		√	√	√
exterior			√	

De izquierda a derecha, la gráfica anterior representa lo siguiente:

- **private:** lo declarado como privado, solo estará disponible dentro de la misma clase.
- **protected:** lo heredado de manera protegida, solo estará disponible para la clase, las clases que se generen a partir de esta (subclases) y todas las demás clases disponibles en el paquete (conjunto de clases).
- **public**: público, estará disponible para todo el entorno.
- **package:** disponible para clases que integren el mismo paquete.

Definiciones en POO

- **Package:** se trata de un conjunto de clases. Es común encontrar a todas las clases vinculadas a un determinado proceso o sección de la aplicación en un mismo paquete.

- **Estado:** se define de esta forma al conjunto de valores que adoptan los atributos de un objeto, en un momento dado.

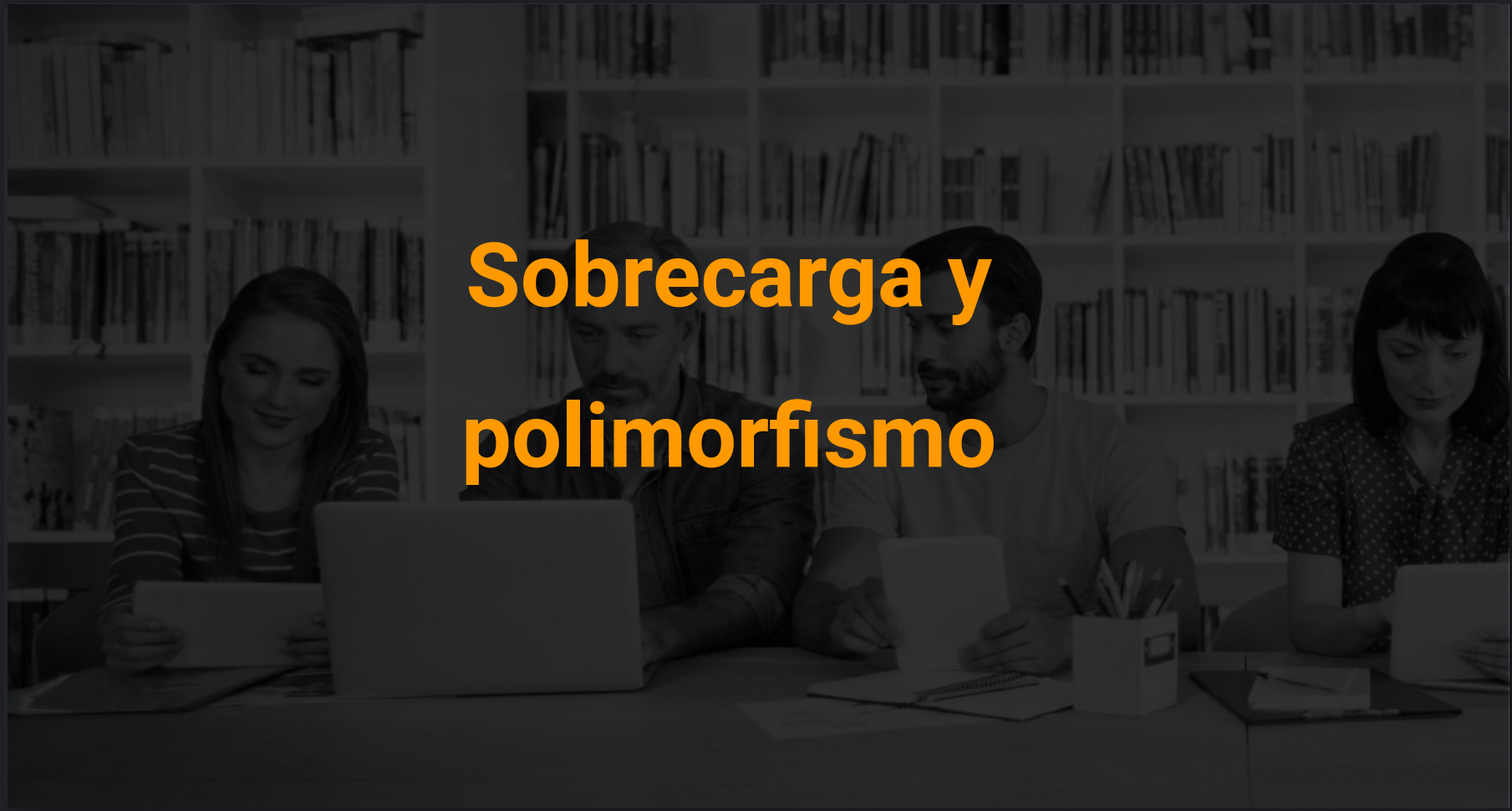
- **Static:** el campo static será el mismo para todas las instancias de la clase.

Definiciones en POO

- **Final:** el campo debe ser inicializado y no se puede modificar.
- **Superclase:** es la clase de la cual se heredan atributos o métodos. Por ejemplo, si hablamos de la jerarquía Animales perros, Animales es la superclase de la clase Perros.
- **Subclase:** clase que hereda atributos o métodos de otra clase. Continuando con el ejemplo anterior, la clase Beagle será una subclase de Perros.

Definiciones en Clase Object

- **toString():** devuelve una cadena que describe el objeto.
- **hashCode():** devuelve el código hash asociado con el objeto invocado.
- **equals():** determina si un objeto es igual a otro.
- **getClass():** obtiene la clase de un objeto en tiempo de ejecución.


Sobrecarga y
polimorfismo

03

Sobrecarga

A nivel programación, implica usar un mismo nombre, dentro de un contexto, más de una vez. En lenguajes previos a **Java**, esto no estaba permitido.

Si recordamos la estructura de una clase analizada previamente, observaremos que podían alojar más de un método constructor. Generalmente estos se instancian con el mismo nombre de la clase a la que pertenecen, y ponen en evidencia que estos métodos están siendo sobrecargados.

03

¿Cómo resuelve Java la sobrecarga?

El sistema define qué función usar en cada caso, a partir de los atributos pasados a las mismas. Continuando con el ejemplo de los constructores, siempre se tendrá uno vacío (esto se usa por convención y practicidad) y otros con las distintas cantidades de atributos que podemos usar.

```
public Estudiante(Integer id, String nombreCompleto, Float promedio, Boolean presentismo) {
   this.id = id;
   this.nombreCompleto = nombreCompleto;
   this.promedio = promedio;
   this.presentismo = presentismo;
}
public Estudiante() {}
```

03

Mismo nombre, diferentes resultados

Esta es la posibilidad que brinda la **sobrecarga de funciones.**
Sintetizando la idea alrededor de los **constructores**, se deberá tener uno por cada conjunto de atributos que se quiere instanciar. Esto mismo puede ser llevado adelante con otras funciones.

03

Polimorfismo

Habíamos hecho mención de que sobrecarga y polimorfismo son conceptos muy cercanos, siendo este último la propiedad que permite que distintos objetos desempeñen métodos idénticos, pero de distinta forma. Por poner un ejemplo sencillo, distintos animales, se comunican, se trasladan, se alimentan de distintas formas, pero todos ejercen la misma acción.

03

Ejemplo de polimorfismo

Redactemos el código de cémo se aplicaria lo enunciado anteriormente.

```
public class Perro {
 // Atributos y métodos de la clase
   public String comunicarse(){
        return "Ladrando...";
    }
}
Llamado:
System.out.println(perro1.comunicarse());

Salida:
Ladrando...
```

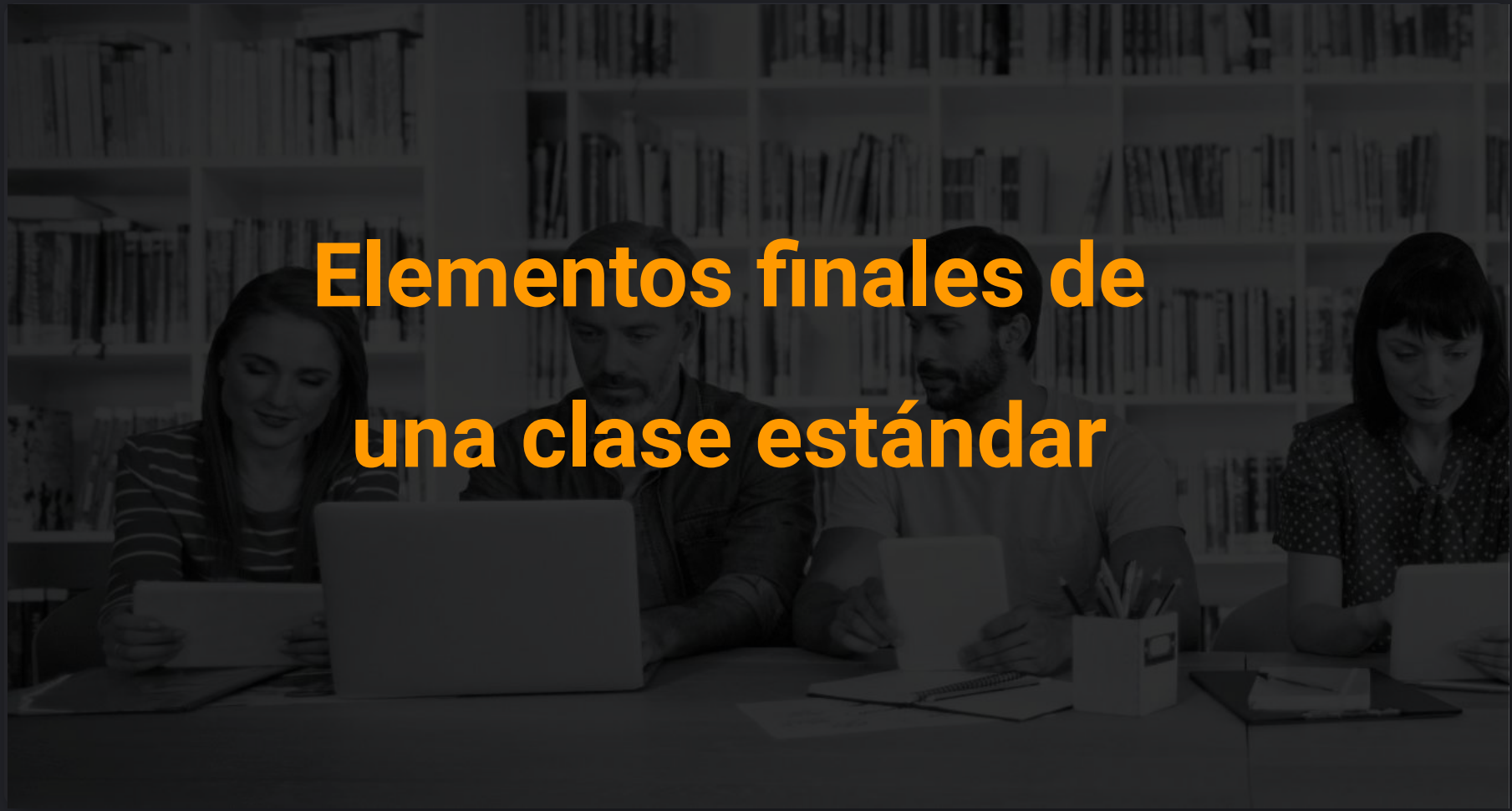
```
public class Gato {
 // Atributos y métodos de la clase
   public String comunicarse(){
        return "Maullando...";
    }
}
Llamado:
System.out.println(gato1.comunicarse());

Salida:
Maullando...
```

03

Conclusiones sobre polimorfismo y sobrecarga

- **Sobrecargar** un método implica que, dentro de una clase, una función pueda aparecer más de una vez, pero con distintos argumentos, y acorde a los parámetros recibidos al momento de su llamado; es que el sistema define qué función específicamente usar.

- **Polimorfismo** son funciones con mismo nombre (pueden o no tener los mismo atributos), pero ejercidas por distintos objetos.


Elementos finales de
una clase estándar

04

Setear y obtener atributos

Si bien los constructores son métodos necesarios en un clase, aparecen otros métodos que por convención se suelen incluir dentro de la creación de una clase: métodos setter & getter (setear, establecer y obtener).

- El primero, como su nombre lo indica, permite asignar (setear) valores a un atributo.
- Mientras que los métodos getter permiten obtener, leer, el valor de un determinado atributo.

04

ToString()

Como cierre, al estándar de clases en Java se les suele agregar el método toString() sobreescrito (por eso lleva la anotación @Override).

Se tiene conocimiento de que este método se hereda automáticamente de la clase Object. Sin embargo es buena práctica sobreescribirlo, redefiniendo de manera más adecuada para la clase en cuestión.

04

... Ejemplos, toString(), get & set

A partir de la clase Perro, citada a lo largo de toda la clase:

```
@Override
public String toString() {
    return "{" +
        " nombre='" + getNombre() + "'" +
        ", peso='" + getPeso() + "'" +
        ", habitat='" + getHabitat() + "'" +
        ", peligro='" + isPeligro() + "'" +
        "}";
}
```

```
public float getPeso() {
    return this.peso;
}

public void setPeso(float peso) {
    this.peso = peso;
}
```

Abstracción e interfaces

05

¿Se pueden crear objetos de cualquier clase?

La respuesta es NO. Para establecer una correspondencia con el ejemplo visto hace unos instantes. Sería de utilidad emplear una clase “Animales” de la cual se derivan, todos los animales previamente creados. Pero nunca crearíamos un Animal en concreto (Por lo Animales se denomina “Clase Abstracta”)

- Si existe la clase Animales y la clase Perros no podremos crear animales, sin antes pasar por clases más específicas, como en este caso clase “perros”.

05

Clases abstractas

Técnicamente las clases abstractas son aquellas que no especifican el funcionamiento de la totalidad de sus métodos.

👉 Dicho en otras palabras, declaran la existencia de los mismos pero no su implementación, dejando este detalle a cargo de las subclases.

05

Empleo de una clase abstracta

Retomemos el ejemplo anterior, donde contábamos con un conjunto de animales (perro, gato, vaca, mono, canario, y pez). Además de los atributos generados durante el desafío, agregaremos una serie de métodos, pero a partir de la clase abstracta “Animales”, los métodos serán: comunicarse y trasladarse.

05

Generamos una clase abstracta

Generamos la clase abstracta Animal y luego generamos la clase Perro, que hereda a Animal y explicita cómo implementar los distintos métodos.

```
public class Perro extends Animal{
    //Métodos y atributos de clase Perro

    public String comunicarse(){
        return "Ladrando...";
    }
    public String trasladarse(){
        return "Cuadrupedo";
    }
}
```

05

... Generamos una clase abstracta

```
public abstract class Animal {
    public abstract String comunicarse();
    public abstract String trasladarse();
}
```

```
Llamado:
System.out.println(perro1.comunicarse());
System.out.println(perro1.trasladarse());
Salida:
Ladrando...
Cuadrupedo
```

05

Interfaces

Desde aquí nace otro elemento clave dentro del desarrollo en Java y la OOP: las **interfaces.**

Las clases con todos sus métodos abstractos y sus atributos establecidos se ubican en una categoría particular y reciben el nombre de interfaces.

05

¿Qué tiene de particular una interfaz?

- A diferencia de las clases abstractas, las interfaces se implementan, no heredan. Por lo que podemos implementar más de una interfaz, adquiriendo mayor funcionalidad.

- Como se vio en el esquema de herencias y modificadores de acceso, hay posibilidades de mostrar datos a partir de la herencia. Mientras que a partir de la implementación, los datos no son visibles, quedando encapsulados.

- Todo lo implementado desde una interfaz puede ser sobreescrito en las nuevas clases, dando nueva funcionalidad.

05

Estructura de una interfaz e implementación

Continuando con el ejemplo previo, podemos definir Animales como una interfaz, y que alguno de los animales creados previamente, la implementa. No se desarrollará mucho más en esta ocasión, dado que este es el mecanismo más empleado a lo largo de Java para el desarrollo Back End y tendremos ejemplos más detallados en próximas oportunidades.

05

Estructura de una interfaz e implementación

- Nótese la diferencia de cómo las interfaces se implementan con implements, mientras que las clases se heredan con extends.

```
public interface Animal {
   //Atributos y métodos de la interface
}
```

```
public class Perro implements Animal {
   //Atributos y métodos de Perro
}
```


Principios SOLID

06

Definición

El término es un acrónimo creado por Robert C. Martin; representa **5 principios básicos de programación orientada a objetos** que de seguirlos adecuadamente nos permitirán tener un buen diseño de programación legible y mantenible.

06

Definición

Entre ellos encontramos:

- Principio Responsabilidad única (Single responsibility)
- Principio Abierto/Cerrado (Open/Close)
- Principio de sustitución de Liskov (Liskov substitution)
- Principio de Segregación de Interfaces (Interface segregation)
- Principio de inversión de dependencias (Dependency inversion)

06

Principio de responsabilidad única

indica que cada clase debe tener una tarea concreta.
Es relativamente frecuente que algún desafío que enfrentamos al programar se resuelva con un método que no está directamente relacionado con la responsabilidad de la clase.

Efectivamente aparece un problema cuando el método necesario es insertado en la clase con la que venimos trabajando, en lugar de generar una nueva.

06

Principio de responsabilidad única

```
public class Triangle {
	private Double base;
	private Double height;

	public Double getArea() {
		return (base*height)/2;
	}
	public String printHtmlOuput() {
		return "<h1>Result:</h1><p>The areas is "+getArea()+"</p>";
	}
}
```

En la clase triangulo la responsabilidad única debe ser la de comportarse como un triángulo, pero imprimir un resultado no es un comportamiento directo de un triángulo, por lo cual debemos crear una clase para dicha responsabilidad.

06

Principio Open/Close

El principio Open/Close se refiere a que una entidad de software debe estar abierta a la expansión y cerrada a la modificación.

Para mantener este principio deberíamos agregar nuevas clases y métodos sin modificar el comportamiento de los ya existentes.

Una manera simple y efectiva para poner el práctica este principio es usar interfaces y así delegar la funcionalidad a los objetos de que implementan.

06

Principio Open/Close

- Supongamos que necesitamos crear un módulo que calcule el área de un círculo.
- Luego, necesitamos también calcular el área de un triángulo. (En este punto es donde alguien podría caer en el error de desarrollar lógica que determine la condición de la figura y, a partir de eso, lleve adelante el cálculo).
- Lo correspondiente sería disponer de una interfaz “figuras”, establecer el cálculo de área desde ahí. Y que cada “figura” resuelva ese cálculo según corresponda.

06

Principio Open/Close

- Con esto estamos expandiendo cada figura en sí misma (ahora cada una calcula el área por su cuenta), mientras que no modificamos la Interfaz inicial.

06

Principio de Sustitución de Liskov

Si bien parte de un principio originado en ciencias más duras, por lo que su enunciado no es del todo “amigable”, podemos resumirlo de la siguiente manera:

Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

06

Principio de Sustitución de Liskov

Veamos ahora un ejemplo frecuente:

- Podemos pensar a un cuadrado como a un rectángulo, con la particularidad de que su ancho y su altura coinciden.
- Ante esto, podríamos hacer que la clase “Cuadrado” extienda de “Rectángulo”.
- Si nos ponemos a hilar fino en cuestiones de cálculo e implementación, llegaremos a la conclusión de que lo más prolijo en este caso sería nuevamente hacer uso de interfaces, y que el cuadrado no extienda directamente del rectángulo, dado que no cumple el principio de Liskov.

06

Principio de segregación de interfaces

En este principio se establece que:

- Los clientes de un programa dado solo deberían conocer de éste aquellos métodos que realmente usan, y no aquellos que no necesitan usar.

Este principio fue concebido con la finalidad de mantener un sistema desacoplado de los demás sistemas de los cuales depende.
El problema viene cuando un sistema está tan acoplado que es casi imposible realizar modificaciones de forma aislada sin que esto lleve a modificaciones colaterales.

06

Principio de segregación de interfaces

Si durante una implementación de una interfaz algún método de dicha interfaz no tiene realmente funcionalidad y dejamos vacío el método a implementar o propagamos una excepción en él, no estamos cumpliendo con el **Principio de segregación de interfaces.**

Esto ocurre cuando tenemos lo que se denomina una **“interface pesada”**. Lo que debemos hacer es crear interfaces más pequeñas denominadas “**interfaces de rol**”.

06

Principio de Inversión de dependencias

La definición original de este principio es la siguiente:

- Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.
- Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

06

Principio de Inversión de dependencias

Para esclarecer esto, nuevamente vemos un ejemplo:

- Supongamos que tenemos un DTO de personas.
- En paralelo, tenemos una clase encargada de guardar info de este DTO en una base de datos MySQL. A partir de un servicio con la correspondiente lógica de negocio.

```
public class ServicePerson {
	public void savePerson(Person person) {
		MySql mysql = new MySql();
		mysql.savePerson(person);
	}
}
```

06

Principio de Inversión de dependencias

- El problema viene cuando en lugar de guardar en la base de datos Mysql quieres guardar en Oracle o en lugar de guardarlo en base de datos quieres tratarlo de otra manera. Es decir, cuando necesitamos hacer modificaciones en nuestro proyecto.

- Instanciar la clase como se hace en el ejemplo anterior genera un fuerte acoplamiento, lo que implica tener que rehacer el código y evita la reutilizar el mismo.

06

Principio de Inversión de dependencias

La solución a esto, aplicando la inversión de dependencias, se lleva adelante de la siguiente forma:

- Creamos una interfaz que abstrae la clase encargada de efectuar las operaciones contra la base de datos, supongamos **IPersistence.**
- Modificamos la clase encargada de hacer las operaciones, para que implemente dicha interfaz.
- Por último, modificamos la clase de Servicio, para evitar que cree una instancia de la clase de MySQL y así evitar el acoplamiento entre ellas.

06

Principio de Inversión de dependencias

```
public class ServicePerson {

	private IPersistence persistence;

	public ServicePerson(IPersistence persistence) {
		this.persistence = persistence;
	}
	public void savePerson(Person person) {
		persistence.save(person);
	}
}
```

¡Muchas gracias!