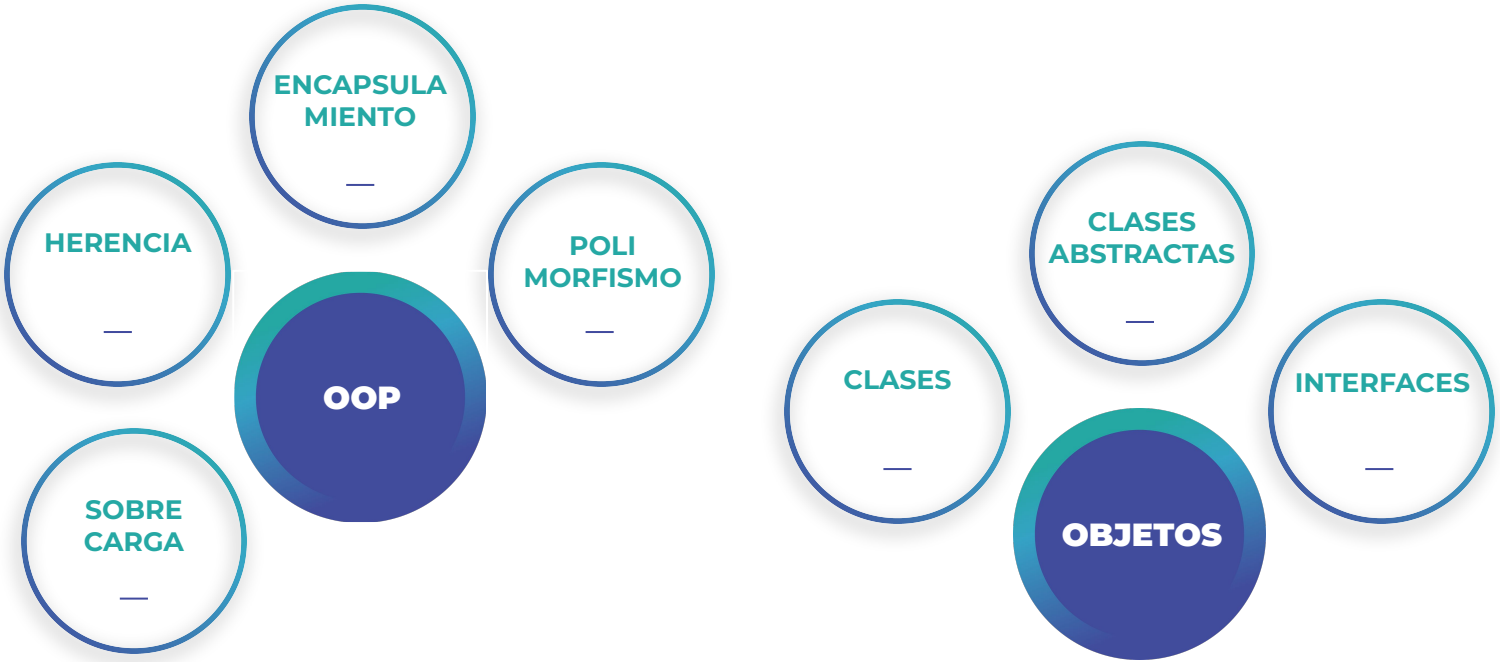
Argentina Programa

Programación orientada a objetos, interfaces y clases abstractas

Mapa conceptual



PREGUNTA GENERAL

¿Qué definición le darían a “paradigma”? ¿Llevado a la programación?

PARADIGMA

Podemos concebir esto 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

Programación orientada a objetos (POO)

01

BENEFICIO DE LA POO 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.**

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.

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.

CLASE  MÉTODOS  ATRIBUTOS

CONSTRUCTORES DE OBJETOS

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;
}
```

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);
}
```

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.

Creación completa de una clase

Caso práctico

CREACIÓN COMPLETA DE UNA CLASE

A modo de ejemplo crearemos de manera completa la clase “Estudiante”, citado 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() {
    }
```

CREACIÓN COMPLETA 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;
}
```

CREACIÓN COMPLETA 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.**

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 quienes va a estar disponible cada componente. Por convención se suelen instanciar atributos (datos) de manera privada y métodos públicos.

Modificadores de acceso

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

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

Modificadores de acceso

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 OOP

- **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 OOP

- **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.

Comunicación entre objetos

El modelado de objetos no sólo tiene en consideración los objetos de un sistema, sino también sus interrelaciones.
Los objetos interactúan enviando mensajes unos a otros. Tras la recepción de un mensaje el objeto actuará. La acción puede ser el envío de otros mensajes, el cambio de su estado, o la ejecución de un método.

Sobrecarga y polimorfismo

03

Sobrecarga

Sobrecargar, 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.

¿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() {
}
```

Usando los constructores sobrecargados

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() {
}
```

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.

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.

Ejemplo polimorfismo

Redactemos el código de cómo se aplicaría 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...
```

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.

Abstracción e interfaces

04

¿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”.

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.**

Función de las clases abstractas

Básicamente, se emplean como medio para desarrollar **subclases.** En otras palabras, las clases hijas (subclases) heredan de la clase padre (superclase) ciertos atributos y métodos, siendo las clases herederas las encargadas de implementar y definir estos métodos.

👉En el ejemplo anterior, “Animales”, es una clase abstract, mientras que “Perros” no.

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

Generamos una clase abstracta

Generamos la clase abstracta Animal y luego generamos la clase Perro, qué hereda a Animal y explícita como implementar los distintos métodos

```
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
```

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

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

¿PUEDE UNA CLASE TENER TODOS SUS MÉTODOS ABSTRACTOS?

La respuesta es **SÍ** y 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.**

¿Puede una clase tener todos sus métodos abstractos?

La respuesta es **SÍ** y 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.**

¿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.

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 implemente. 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.

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

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

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

Elementos finales de una clase

05

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.

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.

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;
}
```

¡Muchas gracias!