Forma

Guía de aprendizaje

Descubriendo la lógica de clases en Java

Experiencia 1

Desarrollo Orientado a Objetos I

Facilitador disciplinar: Marcelo Crisóstomo Carrasco

Asesor par: Alberto Campos Vidal

**Índice**

[Introducción a la experiencia 4](#_Toc202450165)

[Resultados de aprendizaje 5](#_Toc202450166)

[Conceptos relevantes 6](#_Toc202450167)

[Preguntas activadoras 6](#_Toc202450168)

[Actividades de la experiencia 7](#_Toc202450169)

[Semana 1: Introducción a la Programación Orientada a Objetos 8](#_Toc202450170)

[¿Qué es la Programación Orientada a Objetos (POO)? 8](#_Toc202450171)

[¿Qué es el Desarrollo Orientado a Objetos (DOO)? 10](#_Toc202450172)

[Modificadores de acceso 15](#_Toc202450173)

[Métodos get y set (accesores y mutadores) 17](#_Toc202450174)

[Constructores 19](#_Toc202450175)

[Método toString 21](#_Toc202450176)

[Revisemos lo aprendido 23](#_Toc202450177)

[Semana 2: Construyendo soluciones con composición de clases y buenas prácticas 24](#_Toc202450178)

[Organización del proyecto en paquetes (estructura modular) 24](#_Toc202450179)

[Más allá del código: buenas prácticas de programación 32](#_Toc202450180)

[¿Qué es Javadoc y para qué sirve? 32](#_Toc202450181)

[Revisemos lo aprendido 34](#_Toc202450182)

[Semana 3: Realizando validación, excepciones y relaciones entre clases 35](#_Toc202450183)

[Validación de atributos (ejemplo: RUT chileno) 35](#_Toc202450184)

[Manejo de excepciones personalizadas 37](#_Toc202450185)

[Integración y reutilización de clases 39](#_Toc202450186)

[¿Cómo saber cuándo relacionar clases? 44](#_Toc202450187)

[Revisemos lo aprendido 46](#_Toc202450188)

[Cierre de la experiencia 47](#_Toc202450189)

[Referencias 48](#_Toc202450190)

[Lecturas de la experiencia 48](#_Toc202450191)

[Apuntes 49](#_Toc202450192)

[Solucionario Revisemos lo aprendido 50](#_Toc202450193)

# Introducción a la experiencia

En esta experiencia de aprendizaje abordaremos los fundamentos esenciales de la Programación Orientada a Objetos (POO) aplicada con el lenguaje Java, uno de los lenguajes más utilizados y demandados en la industria del software. A lo largo de estas primeras semanas, construiremos desde cero el conocimiento necesario para representar entidades del mundo real a través de clases, objetos y estructuras modulares.

Comenzaremos comprendiendo qué es una clase, cómo se definen sus atributos y métodos, y cuál es la lógica detrás de conceptos como encapsulamiento, visibilidad, constructores y el método toString(). Luego avanzaremos hacia el diseño de clases compuestas, integrando elementos como direcciones, validaciones y estructuras reutilizables mediante composición. También abordaremos la organización del proyecto en paquetes lógicos y el uso de buenas prácticas de programación, incluyendo la documentación profesional del código fuente con Javadoc.

Esta etapa es sumamente importante, ya que establece las bases sobre las que se construirá todo el desarrollo posterior de la asignatura y del proyecto final. Aprenderás no solo a programar con sintaxis correcta, sino a pensar como un desarrollador orientado a objetos, aplicando criterios de diseño que te permitirán crear sistemas mantenibles, escalables y robustos.

# Resultados de aprendizaje

### El estudiante al finalizar esta experiencia de aprendizaje será capaz de:

RA1.Desarrolla soluciones aplicando los principios fundamentales de la programación orientada a objetos, integrando la composición de clases y buenas prácticas de programación.

### Indicadores de logro:

**IL1.** Desarrolla soluciones orientadas a objetos mediante clases que representen entidades relevantes del dominio del problema, ocultando detalles de implementación no esenciales.

**IL2.** Aplica buenas prácticas de programación, como nombrado significativo, modularidad, documentación y composición de clases, en el desarrollo de soluciones orientadas a objetos.

**IL3.** Genera soluciones algorítmicas aplicando el concepto de composición de clases para la reutilización de código.

# Conceptos relevantes

| Clases | Objetos | Atributos |
| --- | --- | --- |
| Métodos | Encapsulamiento | Visibilidad (Private o Public) |
| Constructor | Javadoc | Organización en paquetes |
| toString | Reutilización de Clases | Getters y Setters |
| POO | DOO | Has - a |

# Preguntas activadoras

* ¿Por qué crees que el Desarrollo Orientado a Objetos (DOO) es el enfoque más utilizado en la industria para construir software escalable y mantenible?
* Te has preguntado ¿cómo representarías a una persona, un producto o una dirección dentro de un programa?
* ¿Qué ventajas crees que tiene organizar bien el código desde el principio?

# Actividades de la experiencia

En la siguiente experiencia de aprendizaje se realizarán tres actividades, dos formativas en la semana 1 y 2, y otra actividad sumativa en la semana 3, donde deberás:

**Semana 1:**

Diseñar clases aplicando los principios fundamentales de la programación orientada a objetos, como la definición de atributos y métodos, uso de constructores, encapsulamiento y representación de objetos de la vida real en código.

**Semana 2:**

Integrar una relación de composición entre clases y organizar el proyecto en paquetes ordenados, aplicando buenas prácticas de desarrollo y documentación del código fuente mediante comentarios y herramientas profesionales.

**Semana 3:**

Construir un sistema orientado a objetos que integre múltiples clases relacionadas, validación de datos y manejo de errores mediante excepciones personalizadas. Esta actividad será evaluada como evidencia de tu dominio sobre composición, reutilización y estructura modular del código.

# Semana 1: Introducción a la Programación Orientada a Objetos

## ¿Qué es la Programación Orientada a Objetos (POO)?

La Programación Orientada a Objetos (POO) es un paradigma ampliamente adoptado para desarrollar sistemas escalables, reutilizables y mantenibles. Este enfoque permite representar entidades del mundo real mediante objetos, los cuales se crean a partir de clases que definen atributos (características) y métodos (acciones).

Java es uno de los lenguajes más utilizados para enseñar y aplicar la POO, gracias a su sintaxis clara, su fuerte tipado y su orientación completamente basada en objetos. Comprender este paradigma es fundamental para cualquier desarrollador, ya que establece las bases para construir soluciones robustas y estructuradas.

A continuación, te presentamos una infografía que explica de manera visual los principales conceptos de la Programación Orientada a Objetos.

Figura 1

*Programación Orientada a Objetos*

*Nota.* Es un paradigma de programación que organiza las funciones en entidades llamadas objetos.EDteam (2020). *¿Qué es la Programación orientada a objetos?* [ed.team](https://ed.team/comunidad/que-es-la-programacion-orientada-a-objetos)

Este enfoque promueve la creación de sistemas claros, reutilizables y fáciles de mantener, gracias a principios fundamentales como:

* **Encapsulamiento:** protege los datos internos de un objeto, exponiendo solo lo necesario.
* **Composición:** permite que un objeto esté formado por otros objetos.
* **Herencia:** posibilita que una clase herede propiedades y comportamientos de otra.
* **Polimorfismo:** permite utilizar métodos de manera flexible según el tipo de objeto.



Importante

La POO facilita el desarrollo de sistemas complejos al dividirlos en partes más pequeñas, comprensibles y reutilizables.

Comprender la Programación Orientada a Objetos (POO) es el primer paso para adoptar un enfoque más estructurado y natural al desarrollo de software. Sin embargo, conocer sus principios no basta por sí solo: es necesario trasladar esa comprensión teórica al proceso completo de creación de sistemas: el Desarrollo Orientado a Objetos (DOO).

## ¿Qué es el Desarrollo Orientado a Objetos (DOO)?

El Desarrollo Orientado a Objetos (DOO) consiste en aplicar los principios de la POO en el diseño, construcción y evolución de un sistema. Implica pensar orientado a objetos desde el análisis del problema: identificar las entidades del dominio, definir sus relaciones y responsabilidades, y estructurar el software de forma coherente y escalable.

Mientras la POO representa el paradigma teórico, el DOO es su aplicación práctica para resolver problemas reales mediante software.

La POO ofrece ventajas claras frente a otros enfoques, como la programación estructurada o procedimental. Sus principales beneficios son:

* Organización estructurada del código, facilitando la lectura, mantenimiento y colaboración.
* Reutilización de componentes, reduciendo duplicación y errores.
* Código limpio y mantenible, siguiendo principios como Don't Repeat Yourself (DRY).
* Desarrollo más rápido y eficiente, gracias a estructuras reutilizables y bien definidas.
* Escalabilidad, permitiendo ampliar el sistema sin comprometer funcionamiento base.

Para aplicar el Desarrollo Orientado a Objetos de manera efectiva, es esencial comprender los componentes básicos que lo hacen posible. Entre ellos, las clases, los objetos y los atributos desempeñan un papel central: las clases actúan como moldes que definen las características y comportamientos comunes; los objetos son instancias concretas creadas a partir de esas clases; y los atributos representan las propiedades que distinguen a cada objeto. Estos elementos permiten traducir las entidades del mundo real en estructuras de software claras, reutilizables y coherentes con el enfoque orientado a objetos.

### ¿Qué es una Clase?

Una clase en Java es una plantilla o modelo que define cómo será un tipo de objeto. Dentro de una clase se describen las propiedades (llamadas atributos) y las acciones que puede realizar (llamadas métodos). Cuando usamos una clase, estamos creando objetos que tienen esas propiedades y comportamientos.

Figura 2

*Relación entre clase y objetos: analogía con un plano de casa*

*Nota.* La imagen utiliza una metáfora visual para explicar la diferencia entre clase y objetos en programación orientada a objetos. Fiorentino, K. (2024). *Clases y Objetos: Fundamentos para Optimizar Juegos.* [*platzi.com*](https://platzi.com/cursos/programacion-basica/clases-y-objetos/)

En otras palabras, una clase es como el plano de una casa, y un objeto es como los edificios que se construye a partir de ese plano.

**Clase como molde**

Una clase en Java es un molde o plantilla que define cómo serán los objetos que creemos a partir de ella. En ella se describen las características (atributos) y comportamientos (métodos) que tendrá cada objeto.

Figura 3

*Definición de una clase en Java: ejemplo básico de Fruta*

*Nota.* Imagen que muestra un ejemplo de clase que define el molde con dos atributos: nombre y color. JetBrains (2021). IntelliJ IDEA. (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

### ¿Qué es un objeto?

Figura 4

*Relación entre clase y objetos*

*Nota.* La imagen representa una clase que actúa como un molde o plantilla, a partir del cual se pueden crear múltiples instancias denominadas objetos. MGPanel. (2022). ¿Qué es la Programación Orientada a Objetos? [MGPanel.org](https://www.mgpanel.org/post/-que-es-la-programacion-orientada-a-objetos-)

En Java, un objeto es una instancia de una clase. Es decir, cuando creamos un objeto, estamos construyendo algo real a partir de un molde que definimos antes (la clase).

**Cada objeto tiene:**

* Sus **propios valores** en los atributos (por ejemplo, el nombre "Manzana").
* La capacidad de realizar **acciones** definidas por los métodos (que aprenderás más adelante).

**Objeto como instancia**

Un objeto es una instancia de una clase. Cuando usamos la palabra clave new, estamos creando una versión real y única de esa clase en la memoria, es como usar un molde (la clase) para fabricar una pieza específica (el objeto). Por ejemplo, en el siguiente código:

Figura 5

*Instanciación de un objeto desde una clase en Java*

*Nota*. Imagen de ejemplo que muestra el uso de un molde (la clase) para fabricar una pieza específica JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

En este caso:

* Fruta es la clase (el molde).
* miFruta es un objeto (una instancia concreta).
* Usamos new Fruta() para **crear** ese objeto en la memoria.

**Relación entre clase y objeto**

La clase es el plano, y el objeto es la construcción. Usamos clases para definir cómo será cada objeto, y luego creamos tantos objetos como necesitemos.

### Atributos (variables de instancia)

Los atributos son variables o datos que representan las características de un objeto. Se definen dentro de la clase, pero fuera de cualquier método. Por ejemplo, una fruta puede tener nombre, peso, si está madura, etc.

Figura 6

*Relación entre atributos y métodos de un objeto*

*Nota.* Esta ilustración representa cómo un objeto en la programación orientada a objetos encapsula tanto datos (atributos) como funcionalidad (métodos). EDteam (2019). ¿Qué es la programación orientada a objetos? [ed.team/blog/](https://ed.team/blog/que-es-la-programacion-orientada-objetos)

**Tipos de datos comunes:**

* int: números enteros → int peso;
* double: números decimales → double precio;
* String: texto → String nombre;
* boolean: lógico → boolean estaMadura;

**Ejemplo de código:**

Figura 7

*Atributo variable*

*Nota.* Imagen de ejemplo de atributo variables que representan las características de Fruta. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

## Modificadores de acceso

En la Programación Orientada a Objetos (POO), controlar el acceso a los distintos elementos de una clase es esencial para proteger la integridad de los datos y mantener una estructura segura y ordenada en el código. Esta capacidad se logra mediante los modificadores de acceso –también conocidos como de visibilidad– que permiten definir el alcance de acceso a atributos, métodos, constructores y clases desde otras partes del programa, y son:

* public (público)
* protected (protegido)
* *(default)* (predeterminado)
* private (privado)

Estos modificadores son clave para aplicar uno de los principios fundamentales de la POO: **el encapsulamiento,** el cual establece límites claros entre las partes internas de una clase y su interacción con el exterior.

Para comprender mejor cómo estos modificadores afectan el acceso en diferentes contextos, observa la siguiente imagen:

Figura 8

*Modificadores de acceso en Java y su visibilidad*

*Nota.* Esta imagen explica los niveles de acceso de miembros (atributos y métodos) en Java: privado, protegido, público y el acceso por defecto (predeterminado). MfgRobots. (s.f.). Modificadores de acceso Java. [mfgrobots.com](https://es.mfgrobots.com/ooip/java/1003003435.html)

**Acceso desde:**

* public: cualquier clase, sin importar el paquete.
* protected: la misma clase, clases del mismo paquete y subclases, incluso en otros paquetes.
* *(default)*: también llamado package-private (sin especificar ningún modificador): accesible solo desde el mismo paquete.
* private: solo accesible dentro de la misma clase.

Una vez comprendida la estructura básica de una clase y la relación con sus objetos, es momento de explorar elementos fundamentales que permiten interactuar con ellos de manera más segura, flexible y controlada. A continuación, abordaremos herramientas clave como los métodos get y set, que facilitan el acceso a atributos privados; los constructores, que permiten inicializar objetos de distintas formas; el método especial toString(), que define cómo se representa un objeto como texto; y, finalmente, veremos cómo se crean objetos desde el método main, punto de entrada de cualquier aplicación Java. Estos conceptos son esenciales para desarrollar programas orientados a objetos que sean robustos, organizados y fáciles de mantener.

## Métodos get y set (accesores y mutadores)

La Programación Orientada a Objetos utiliza métodos especiales que permiten consultar o modificar estos valores sin comprometer la integridad del objeto. Es en este contexto donde entran en juego los métodos *get* y *set*, también conocidos como accesores y mutadores.

Los métodos *get* y *set* permiten acceder o modificar atributos privados de manera controlada, este enfoque forma parte del principio de encapsulamiento, que busca proteger los datos internos del objeto y dar acceso solo a través de métodos definidos.

Cuando un atributo está marcado como private, no puede ser utilizado directamente desde fuera de la clase. Para manejar este atributo, se crean métodos públicos:

* **El método *get* (accesor)** devuelve el valor actual del atributo.
* **El método *set* (mutador)** permite cambiar el valor del atributo, pudiendo además incluir validaciones antes de asignar.

**Por ejemplo:**

Clase Fruta, podríamos tener un atributo nombre privado y los métodos getNombre() y setNombre() para trabajar con él. Así evitamos accesos no deseados y mantenemos el control sobre cómo se usan los datos.

Figura 9

*Clase Fruta*

*Nota.* Imagen de ejemplo de uso los métodos getNombre() y setNombre(). JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

## Constructores

Un constructor es un método especial dentro de una clase que se ejecuta automáticamente cuando se crea un objeto con la palabra clave new.

Su principal función es inicializar los atributos del objeto en el momento de su creación, es decir, asignarles un valor inicial para que el objeto esté listo para usarse desde el comienzo.

* **Constructor por defecto** (implícito): si no escribes ningún constructor, Java crea uno por ti automáticamente (no visible en el código):

**Figura 10**

*Constructor por defecto ej.* Fruta

*Nota.* Imagen ejemplo de constructor. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

**Este constructor no hace nada más que permitir instanciar**: Fruta f = new Fruta();

* **Constructor sin parámetros** (explícito): tú lo defines sin recibir datos, pero puedes asignar valores por defecto:

Figura 11

*Constructor sin parámetros*

*Nota.* Imagen de ejemplo con valor asignado manualmente. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

**Se invoca así:** Fruta f = new Fruta();

* **Constructor con parámetros:** permite crear el objeto con datos personalizados desde el inicio:

Figura 12

*Constructor con parámetro*

*Nota.* Imagen de ejemplo donde se ejecuta al crear un objeto y recibe un nombre. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

**Se invoca así:** Fruta f = new Fruta("Kiwi");

## Método toString

El método toString() en Java es un método especial heredado de la clase base Object, que sirve para mostrar una representación textual de un objeto cuando se imprime, por ejemplo, con System.out.println(). Al sobrescribir este método, podemos personalizar la forma en que se muestra el objeto, como en el siguiente ejemplo con la clase Videojuego:

Figura 13

*Clase Videojuego*

*Nota.* Ejemplo de personalización de la forma que se muestra el objeto. Retorna texto descriptivo. JetBrains (2021). IntelliJ IDEA. (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

**Creación de un objeto desde el método main:** para crear un objeto, usamos la palabra clave new:

Figura 14

*Método Main: punto de inicio*

*Nota.* Ejemplo de Metodo main, punto de inicio. JetBrains. (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

* **new Videojuego("Zelda"):** crea una nueva instancia con el título proporcionado.
* **miJuego.toString():** accede al método del objeto que devuelve su descripción.

Esto permite trabajar con múltiples videojuegos, cada uno con sus propios valores y comportamientos.

# Revisemos lo aprendido

Antes de continuar explorando este recurso, pon en práctica tus conocimientos contestando las siguientes preguntas:

1. Ana está desarrollando una aplicación para gestionar frutas. Define una clase Fruta con atributos como nombre y color. Sin embargo, su compañero le dice que debe proteger estos atributos del acceso directo desde otras clases. **¿Qué principio de POO debería aplicar Ana para cumplir esta recomendación?**
2. Composición.
3. Polimorfismo.
4. Encapsulamiento.
5. Herencia.
6. Luis crea la clase Libro y desea que, al ejecutar System.out.println(miLibro); aparezca una descripción legible con el título y autor del libro. Actualmente, al imprimir el objeto no se muestra información clara. **¿Qué debe hacer Luis para personalizar esa salida al imprimir el objeto?**
7. Crear un método llamado imprimirLibro().
8. Sobrescribir el método equals() de la clase Object.
9. Llamar al constructor de la clase nuevamente.
10. Sobrescribir el método toString()
11. Martina está creando una aplicación de registro de mascotas. Define una clase Mascota con atributos nombre, edad y raza. Para crear una instancia específica desde el inicio necesita definir un mecanismo adecuado. **¿Qué tipo de constructor le recomiendas?**
12. Constructor con parámetros.
13. Constructor vacío.
14. Constructor por defecto.
15. Constructor estático.

# Semana 2: Construyendo soluciones con composición de clases y buenas prácticas

## Organización del proyecto en paquetes (estructura modular)

En esta segunda semana, aprenderás a modelar relaciones entre clases usando composición, a organizar tu proyecto en paquetes dentro de IntelliJ IDEA, y a documentar profesionalmente tus clases con Javadoc. También aplicarás buenas prácticas de programación que te ayudarán a construir sistemas más limpios y mantenibles. Lo harás trabajando en un ejemplo contextualizado en una empresa ficticia del sur de Chile llamada PuertoGames, dedicada a la venta de videojuegos.

Una buena práctica en el desarrollo profesional es organizar las clases en paquetes según su funcionalidad. En este caso, usaremos la siguiente estructura:

* **model:** para clases que representan entidades (como Cliente, Direccion).
* **app:** para la clase principal con el método main().
* **util:** para herramientas o validadores auxiliares (se usará más adelante).

### ¿Cómo crear un paquete en IntelliJ IDEA?

Para construir un sistema orientado a objetos claro y mantenible, es fundamental **organizar correctamente las clases en paquetes**, según su funcionalidad. A continuación, te presentamos una **guía paso a paso** para estructurar tu proyecto en **IntelliJ IDEA**, de forma profesional y modular.

**Paso 1: Crear paquetes en IntelliJ IDEA**

1. Haz clic derecho sobre la carpeta src.

Figura 15

*Carpeta src*

*Nota.* Imagen de paso a paso para crear un paquete en IntelliJ IDEA. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

2. Selecciona New > Package

Figura 16

*Selecciona Package*

*Nota.* Imagen de paso a paso para crear un paquete en IntelliJ IDEA. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

3. Escribe el nombre del paquete, por ejemplo: model, y presiona Enter.

Figura 17

*New Package*

*Nota.* Imagen de paso a paso para crear un paquete en IntelliJ IDEA. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

4. Repite el proceso para crear los paquetes app y util.

**Figura 18**

*Carpeta src*

*Nota.* Imagen de paso a paso para crear un paquete en IntelliJ IDEA. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

### Paso 2: Mover la clase Main al paquete app

1. Debes tener tu paquete app creado

Figura 19

*Paquete app creado*

*Nota.* Imagen de paso a paso para crear un paquete en IntelliJ IDEA. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

2. Arrástrala con el mouse hacia la carpeta app.

Figura 20

*Arrastrar Main a app*

*Nota.* Imagen de paso a paso para crear un paquete en IntelliJ IDEA. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

3. Aparecerá una ventana de confirmación. Haz clic en el botón **Refactor** para aplicar el cambio correctamente.

Figura 21

*Ventana de confirmación*

*Nota.* Imagen de paso a paso para crear un paquete en IntelliJ IDEA. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

4. IntelliJ actualizará automáticamente el package en la parte superior del archivo.

Figura 22

*Actualización package*

*Nota.* Imagen de paso a paso para crear un paquete en IntelliJ IDEA. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

### Paso 3: Crear y organizar las clases del sistema

Para comprender cómo se estructura un sistema orientado a objetos en Java, se analiza un conjunto de clases que permite observar cómo se organiza y ejecuta un sistema simple aplicando los principios de la Programación Orientada a Objetos. Ahora vamos a crear las clases, una por una, dentro de sus respectivos paquetes.

1. Clase Direccion (en paquete model)

Figura 23

*Creación de clase Direccion*

*Nota*. Imagen ejemplo que representa una dirección asociada a un cliente en PuertoGames. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

**Explicación:**

* La clase Direccion se ubica dentro del paquete model.
* Cada atributo representa una parte de la dirección.
* Se usa Javadoc para describir la clase y cada propiedad.

2. Clase Cliente (en paquete model)

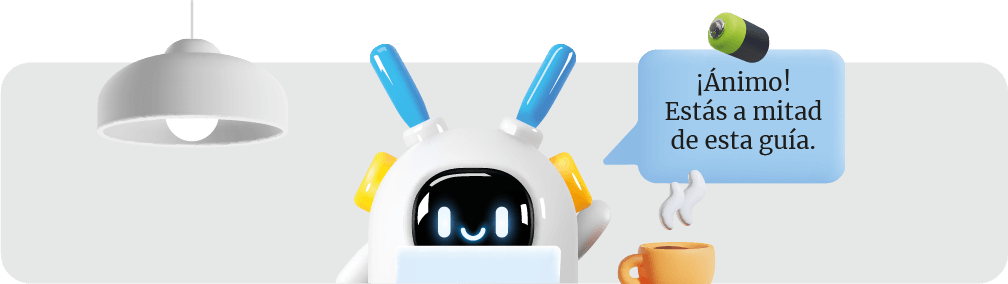
Figura 24

*Creación de clase Cliente*

*Nota*. Representa un cliente de la empresa PuertoGames con nombre y dirección. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

**Explicación:**

* La clase Cliente tiene un atributo Direccion, lo que implementa composición.
* Cliente está contenida también en el paquete model.
* Es un modelo simple para representar información básica de un cliente de videojuegos.



3. Clase Main (en paquete app)

Figura 25

*Clase principal*

*Nota.* Ejemplo que ejecuta el sistema de prueba para PuertoGames. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

Al finalizar estos pasos, habrás estructurado correctamente un sistema Java con:

* Separación clara entre entidades (model), lógica principal (app) y herramientas (util).
* Clases bien documentadas.
* Aplicación de principios como encapsulamiento y composición.
* Un entorno de desarrollo ordenado, ideal para seguir escalando tu aplicación.

## Más allá del código: buenas prácticas de programación

Crear clases y paquetes no es solo una tarea técnica: también es una oportunidad para aplicar buenas prácticas desde el inicio. Estas prácticas hacen que tu código no solo funcione, sino que además sea más fácil de entender, mantener y mejorar. Al realizar esta primera organización de tu sistema, ya estás incorporando varias de ellas:

* **Responsabilidad única:** cada clase representa una sola entidad (Cliente, Direccion).
* **Código organizado:** el uso de paquetes mantiene el proyecto limpio y estructurado.
* **Nombres claros y descriptivos:** evita usar nombres como x, a1 o obj.
* **Separación entre lógica de datos y presentación:** los datos se almacenan en las clases del modelo; el programa principal (Main) se encarga de mostrarlos.
* **Documentación con Javadoc:** describe para qué sirve cada clase y cada método, facilitando la comprensión del código por parte de otras personas (o del mismo programador en el futuro).

## Logotipo Javadoc¿Qué es Javadoc y para qué sirve?

Uno de los pilares de un proyecto profesional es su **documentación clara y actualizada**. Java incorpora una herramienta oficial para esto: **Javadoc**, que permite generar documentación automática del código fuente a partir de comentarios especiales. Estos comentarios deben comenzar con /\*\* y describen clases, métodos y atributos.

**¿Por qué usarlo?**

* Facilita la comprensión del código por parte de otros desarrolladores.
* Permite generar documentación técnica en formato HTML.
* Ayuda a mantener un estándar de calidad profesional.

**¿Cómo se implementa?**

Ejemplo aplicado a una clase:

Figura 26

*Representa una dirección física de un cliente*

*Nota.* En esta imagen se visualiza la clase Direccion.java, parte del paquete model, en un proyecto Java. JetBrains (2021). IntelliJ IDEA. (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

Puedes usar estos comentarios tanto para clases como para métodos y atributos. En entornos profesionales, su uso es altamente valorado.

Link de interés

Revisa la documentación oficial de Oracle para conocer más sobre Javadoc: <https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>

# Revisemos lo aprendido

Antes de continuar explorando la guía de aprendizaje, pon en práctica tus conocimientos contestando las siguientes preguntas:

1. Estás desarrollando una aplicación en IntelliJ IDEA para gestionar clientes y sus direcciones. Decides seguir buenas prácticas de organización del código. **¿En qué paquete deberías ubicar la clase** Cliente**, que representa una entidad del dominio?**

1. En el paquete util.
2. En el paquete app.
3. En la carpeta raíz del proyecto.
4. En el paquete model.

2. En PuertoGames, se está desarrollando una clase Cliente que contiene una dirección. Esto se modela agregando un atributo Direccion dentro de Cliente. **¿Qué concepto de la POO estás aplicando al incluir una clase dentro de otra?**

1. Herencia
2. Encapsulamiento
3. Composición
4. Polimorfismo

3. Tu equipo de trabajo quiere asegurarse de que el código Java sea fácilmente comprensible por futuros programadores. Deciden usar Javadoc para documentar clases y métodos. **¿Cuál de los siguientes beneficios justifica el uso de Javadoc en proyectos Java?**

1. Permite ejecutar código desde la documentación.
2. Facilita la creación de paquetes automáticamente.
3. Ayuda a generar documentación técnica en formato HTML.
4. Reduce el tiempo de compilación.

# Semana 3: Realizando validación, excepciones y relaciones entre clases

## Validación de atributos (ejemplo: RUT chileno)

En esta semana trabajarás en la construcción de un sistema modular más completo. Aprenderás a validar datos sensibles como el RUT, a manejar errores con excepciones personalizadas, y a relacionar varias clases entre sí de manera coherente. También comprenderás cuándo y cómo establecer relaciones de asignación entre clases.

El ejemplo seguirá basándose en la empresa ficticia PuertoGames, ahora incorporando validación de clientes con RUT.

### ¿Por qué son necesarias las validaciones?

Las validaciones son fundamentales porque garantizan que los datos ingresados por los usuarios cumplan con un formato correcto y coherente antes de ser utilizados por el sistema. En el caso del RUT chileno, validar su estructura evita errores como entradas mal escritas, números inválidos o fraudes, asegurando así la integridad de la información. Sin estas validaciones, el sistema podría almacenar datos defectuosos que causarían fallos en procesos posteriores, como búsquedas, autenticación o generación de documentos, afectando la fiabilidad y seguridad de la aplicación.

Cuando trabajamos con datos importantes como un RUT, no basta con almacenarlos: debemos validarlos para evitar errores o fraudes.

### Clase Rut (paquete model)

Figura 27

*Validación de atributos*

*Nota.* Esta figura representa la clase Rut implementada en Java, la cual representa un Rol Único Tributario (RUT) chileno. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

**Explicación:**

* El constructor valida que el RUT tenga el formato XXXXXXXX-Y, usando una **expresión regular** con matches("[0-9]+-[0-9kK]").
* Si el formato es incorrecto, se lanza una excepción personalizada mediante el uso de throw new IllegalArgumentException("...").
* throw indica que se está lanzando manualmente una excepción.
* new crea una nueva instancia del objeto de excepción.
* IllegalArgumentException es una clase de excepción de Java que se utiliza cuando se pasa un argumento inválido a un método o constructor.
* getNumero() devuelve el RUT almacenado, pero solo si ha pasado la validación.
* toString() permite imprimir una representación legible del objeto en consola, por ejemplo: RUT: 12345678-K.

## Manejo de excepciones personalizadas

Las excepciones son errores que pueden ocurrir durante la ejecución del programa. Podemos lanzar nuestras propias excepciones para controlar mejor los errores y entregar mensajes más claros.

### Clase RutInvalidoException (paquete util)

Figura 28

*Manejo de excepciones personalizadas*

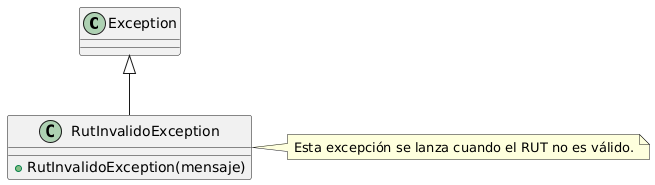
*Nota.* En esta figura se presenta la clase RutInvalidoException, una excepción personalizada en Java creada para manejar errores relacionados con formatos inválidos de RUT. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

**Explicación:**

* Esta clase extiende (extends) la clase base Exception, lo que significa que hereda su comportamiento y se convierte en una subclase de excepción personalizada. En otras palabras, RutInvalidoException es un tipo de excepción que funciona como cualquier otra excepción de Java, pero está diseñada específicamente para el contexto del RUT.
* El uso de super(mensaje) llama al constructor de la clase padre (Exception) y le entrega el mensaje de error que será mostrado si ocurre esta excepción. Esta clase permite lanzar errores propios del negocio (por ejemplo, cuando un RUT no tiene el formato esperado) y mejora la claridad y legibilidad del código, ya que diferencia los errores de validación de RUT de otros errores más genéricos.
* Se puede usar en vez de **IllegalArgumentException** cuando queremos clasificar de forma más precisa los tipos de error en una aplicación.

Figura 29

*Diagrama de excepción*



*Nota*. El diagrama muestra la jerarquía de herencia en la que la clase RutInvalidoException extiende a la clase Exception de Java. PlantUML. (2024). PlantText (1.2024.7). [Software] [PlantText.com](https://www.planttext.com/)

## Integración y reutilización de clases

Integramos ahora la clase Rut como atributo dentro de Cliente para aplicar uno de los principios más importantes de la programación orientada a objetos: la reutilización y modularidad del código. En lugar de volver a escribir la lógica de validación del RUT dentro de cada clase que lo necesite, encapsulamos esa lógica en una clase independiente (Rut) y luego la integramos en otras clases como Cliente. Esto no solo evita la duplicación de código, sino que mejora la organización, facilita el mantenimiento y asegura que cualquier cambio o mejora en la validación se refleje automáticamente en todas las clases que usen RUT. Así, la clase Cliente puede centrarse en representar al cliente, mientras que la clase Rut se encarga exclusivamente de validar e interpretar el número de identificación.

### Clase Direccion (paquete model)

Comenzaremos agregando la clase Direccion, que nos permitirá representar de forma clara y estructurada la información geográfica básica de un cliente, como su calle, ciudad y región. Esta clase será utilizada por otras clases como Cliente, promoviendo la reutilización del código y una separación de responsabilidades que mejora la organización del sistema.

Figura 30

*Clase Dirección, paquete model*

*Nota.* Ejemplo que representa la dirección de un cliente. JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

**Explicación:**

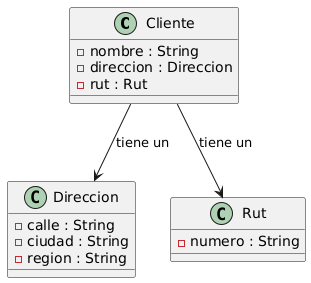
* Clase sencilla que almacena los datos de dirección del cliente.
* Se utiliza el método toString() para mostrar la dirección completa en un solo texto.

### Modificación de Cliente (paquete model)

Una vez definidas las clases Rut y Direccion, ahora estamos en condiciones de integrarlas dentro de una clase más compleja: Cliente. Esta clase representa a una persona o entidad que contiene tanto un nombre como su información de dirección y su RUT. Al reutilizar las clases que ya creamos, no solo simplificamos el diseño del código, sino que también aplicamos principios fundamentales como la modularidad, la reutilización y la separación de responsabilidades.

Figura 31

*Diagrama de clases*



*Nota*. Este diagrama de clases muestra la relación de composición entre la clase Cliente y dos clases auxiliares: Direccion y Rut. PlantUML. (2024). PlantText (1.2024.7). [Software] [PlantText.com](https://www.planttext.com/)

Este diagrama muestra cómo la clase Cliente se compone de otras clases (Rut y Direccion) mediante relaciones de tipo “has-a”. Esto permite reutilizar el código y mantener una estructura clara y modular.

Figura 32

*Clase Dirección, Cliente y Rut*

*Nota.* La clase Direccion, ubicada en el paquete model, representa la dirección física de un cliente. JetBrains. (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

**Explicación:**

* Ahora Cliente contiene un Rut, lo que implica una **relación de asignación**.
* También contiene una instancia de Direccion, integrando así múltiples objetos en una sola entidad.
* Las clases Rut y Direccion se **reutilizan** en distintos contextos donde se necesiten esas estructuras, sin tener que duplicar código o lógica de validación.

### Clase principal “Main” de prueba (paquete app)

Ahora que ya contamos con las clases Cliente, Direccion y Rut correctamente estructuradas e integradas, es momento de ponerlas en práctica. En esta clase Main simularemos la creación de un cliente con todos sus datos, incluyendo validación del RUT. Esta clase actúa como punto de entrada para ejecutar y comprobar el funcionamiento del sistema, integrando lo aprendido sobre modularidad, validaciones y manejo de excepciones.

Figura 33

*Clase principal “Main”*

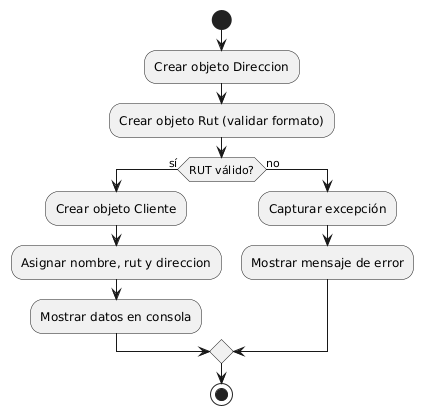
*Nota.* Ejemplo que contiene clase Cliente, Direccion y Rut en el paquete model JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. [Jetbrains](https://www.jetbrains.com/idea/)

**Explicación:**

* Se crea una **dirección** y un **RUT válidos** usando objetos previamente definidos.
* Se integran todos los datos en un único objeto Cliente, demostrando la reutilización de clases.
* Se utiliza un bloque try-catch para **manejar errores de validación**, específicamente cuando el RUT no cumple con el formato esperado.

Figura 34

*Flujo de ejecución de la clase Main*



*Nota.* Diagrama que ilustra la lógica implementada en el programa de la clase Main de Semana 3. Representa gráficamente cómo se construye un objeto Cliente, validando previamente su RUT. PlantUML. (2024). PlantText (1.2024.7). [Software] [PlantText.com](https://www.planttext.com/)

Este diagrama muestra el flujo lógico del programa principal. Comienza con la creación de la dirección y el RUT, valida su formato y, según el resultado, crea el cliente o lanza una excepción. Finalmente, si todo es correcto, imprime los datos del cliente en consola.

## ¿Cómo saber cuándo relacionar clases?

Relacionamos clases cuando una entidad necesita o depende de otra para existir o funcionar correctamente dentro del sistema. Esta decisión se basa en el análisis del modelo de dominio, es decir, cómo se comportan las entidades en el mundo real que estamos modelando.

**En este caso específico:**

* Un cliente tiene una dirección → Esto representa una relación de composición. La clase Cliente contiene una instancia de Direccion porque sin dirección, el cliente no estaría completo. La dirección es parte del cliente, y si el cliente se elimina, su dirección también deja de tener sentido.
* Un cliente tiene un RUT → Esto es una relación de asignación o asociación. El RUT es un dato que identifica al cliente, pero no necesariamente forma parte de su estructura interna. Podría existir como una entidad reutilizable para validar o autenticar otras clases (por ejemplo: Proveedor, Usuario, Trabajador, etc.).

**Regla práctica:**

Si una clase “tiene un(a)” de otra, probablemente debas usar una relación de asignación o composición. A este tipo de relación en programación orientada a objetos se le llama “has-a” (del inglés: tiene un(a)).

**¿Qué significa "has-a"?**

"Has-a" es una expresión común en programación orientada a objetos que describe una relación en la que una clase contiene una instancia de otra como atributo. Es decir, una clase está compuesta por o usa otra clase como parte de su definición.

Figura 35

*Comparación entre relaciones*

La figura contiene dos conjuntos de clases. En la parte izquierda se muestra una relación de herencia. La clase Estudiante hereda de la clase Persona, lo que se indica mediante una flecha con triángulo blanco apuntando desde Estudiante hacia Persona con la etiqueta “is-a” (es un tipo de). La clase Persona contiene un atributo nombre de tipo String, mientras que Estudiante agrega el atributo carrera, también de tipo String.

En la parte derecha se presenta una relación de composición. La clase Cliente contiene un atributo rut del tipo Rut, lo que indica una relación “has-a” (tiene un). Esta relación está representada por una flecha con punta negra apuntando desde Cliente hacia Rut. Cliente también tiene un atributo nombre de tipo String, y Rut contiene un atributo numero de tipo String.

*Nota*. Esta figura muestra dos tipos de relaciones entre clases en la programación orientada a objetos: herencia (is-a) e inclusión o composición (has-a). PlantUML. (2024). PlantText (1.2024.7). [Software] [PlantText.com](https://www.planttext.com/)

**Descripción:**

Muestra la diferencia entre herencia (is-a) y composición/asignación (has-a), conceptos clave para estructurar correctamente sistemas orientados a objetos.

**Por ejemplo:**

* Cliente has-a Direccion → El cliente tiene una dirección.
* Cliente has-a Rut → El cliente tiene un RUT.

Esta relación es clave para lograr un diseño modular, reutilizable y fácil de mantener. Al encapsular información y responsabilidades en clases separadas (como Rut para validación y Direccion para ubicación), puedes construir sistemas más claros y robustos.

# Revisemos lo aprendido

Las siguientes preguntas aplicadas te ayudarán a reforzar lo aprendido en esta semana, especialmente en temas como validación del RUT, uso de excepciones personalizadas y relaciones entre clases en sistemas orientados a objetos.

1. Estás desarrollando una clase Cliente para la empresa PuertoGames. Un compañero implementa la validación del RUT directamente dentro de la clase Cliente, usando una estructura if simple. Tú propones usar una clase separada llamada Rut. **¿Por qué es mejor encapsular la validación del RUT en una clase independiente?**

1. Porque el código es más corto al evitar estructuras de control complejas.
2. Porque separa responsabilidades y permite reutilizar la lógica en otras clases.
3. Porque las clases independientes no pueden lanzar excepciones.
4. Porque Java obliga a que el RUT esté en una clase propia.

2. Durante las pruebas, al ingresar un RUT en formato incorrecto, el sistema lanza una IllegalArgumentException. El equipo quiere mejorar la claridad del error para distinguir entre tipos de fallos. **¿Qué debes hacer para mejorar el manejo de este error?**

1. Usar un try-catch dentro de la clase Rut que oculte el error.
2. Dejar que el programa se detenga para forzar al usuario a corregir el RUT.
3. Cambiar el tipo de dato de String a int para evitar letras en el RUT.
4. Crear una excepción personalizada como RutInvalidoException.

3. En tu sistema, un objeto Cliente contiene un objeto Direccion. Tu compañera pregunta si esto es herencia, ya que Cliente “tiene” una dirección. **¿Qué tipo de relación representa esta conexión?**

1. Herencia (is-a)
2. Agregación (uses-a)
3. Composición o asignación (has-a)
4. Asociación directa (equals-a)

# Cierre de la experiencia

Durante esta experiencia diste tus primeros pasos en la construcción de sistemas orientados a objetos utilizando Java. Aprendiste a modelar entidades reales mediante clases, a organizar tu código en paquetes, y a aplicar principios fundamentales como la composición, la reutilización de clases, la validación de atributos y el manejo de errores con excepciones personalizadas.

Estas habilidades no solo te permiten escribir código funcional, sino diseñar soluciones limpias, mantenibles y pensadas para crecer. Ya estás desarrollando una mentalidad profesional: piensas en modularidad, te preocupas por la estructura del sistema, y reconoces el valor de validar y proteger los datos desde el inicio.

Con estas herramientas, estás listo para abordar desafíos más complejos en las siguientes semanas, como el desarrollo de un proyecto integrador que consolide todo lo aprendido. Te invitamos a seguir explorando, practicando y cuestionando: así se construyen los verdaderos desarrolladores.

# Referencias

* EDteam (2020). ¿Qué es la Programación orientada a objetos? <https://ed.team/comunidad/que-es-la-programacion-orientada-a-objetos>
* JetBrains (2021). IntelliJ IDEA (2022.3.2). [Software]. <https://www.jetbrains.com/idea/>
* Modificadores de acceso. (2012). Programación Orientada a Objetos. <https://picarcodigo.blogspot.com/2012/05/modificadores-de-acceso.html>
* Schools (s.f.). Java OOP. <https://www.w3schools.com/java/java_oop.asp>

# Lecturas de la experiencia

* Capítulo 1: Fundamentos de la Programación Orientada a Objetos (POO)

M. Hernández, B. (2023). *Programación orientada a objetos en Java. Buenas prácticas.* Ediciones de la U. <https://webezproxy.duoc.cl/login?url=http://biblioteca.duoc.cl/bdigital/elibros/a51968-Programacion/17/> Páginas 21 a 38.

* Capítulo 4: Métodos: métodos constructores, *set* y *get*

M. Hernández, B. (2023). *Programación orientada a objetos en Java. Buenas prácticas.* Ediciones de la U. <https://webezproxy.duoc.cl/login?url=http://biblioteca.duoc.cl/bdigital/elibros/a51968-Programacion/61/> Páginas 67-107.

* JAVA OOP

W3schools. (s.f.). *Java OOP.* <https://www.w3schools.com/java/java_oop.asp>

Te recomendamos explorar los recursos de W3Schools Java donde encontrarás explicaciones claras, ejemplos ejecutables y ejercicios interactivos:

<https://webezproxy.duoc.cl/login?url=http://biblioteca.duoc.cl/bdigital/elibros/a51968-Programacion/17/>

# Apuntes

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Solucionario Revisemos lo aprendido

## Semana 1

* **Pregunta 1 -** Respuesta correcta: Encapsulamiento.

**Retroalimentación:** El encapsulamiento permite ocultar los atributos internos de una clase, exponiéndolos sólo mediante métodos públicos como get y set. Esto garantiza mayor seguridad y control del acceso a los datos.

* **Pregunta 2 -** Respuesta correcta: Sobrescribir el método toString().

**Retroalimentación:** El método toString() se sobrescribe para definir cómo se mostrará el contenido del objeto de forma personalizada al imprimirlo.

* **Pregunta 3 -** Respuesta correcta: Constructor con parámetros.

**Retroalimentación:** El constructor con parámetros permite inicializar un objeto con valores específicos en el momento de su creación, lo cual es muy útil para asignar atributos desde el comienzo.

## Semana 2

* **Pregunta 1 -** Respuesta correcta: En el paquete *model*.

**Retroalimentación:** La clase Cliente representa una entidad del dominio y, por tanto, debe ir en el paquete model, siguiendo el principio de separación por responsabilidades.

* **Pregunta 2 -** Respuesta correcta: Composición.

**Retroalimentación:** Se está aplicando **composición**, ya que la clase Cliente contiene un objeto Direccion, representando una relación "tiene una".

* **Pregunta 3 -** Respuesta correcta: Ayuda a generar documentación técnica en formato HTML.

**Retroalimentación:** Javadoc permite generar documentación profesional en HTML, lo que facilita la comprensión y mantenimiento del código por parte del equipo.

## Semana 3

* **Pregunta 1 -** Respuesta correcta: Porque separa responsabilidades y permite reutilizar la lógica en otras clases.

**Retroalimentación:** Separar la validación del RUT en su propia clase promueve la modularidad y la reutilización, principios fundamentales del diseño orientado a objetos. Así, otras clases como Proveedor o Empleado también podrían usar la clase Rut sin duplicar código.

* **Pregunta 2 -** Respuesta correcta: Crear una excepción personalizada como RutInvalidoException.

**Retroalimentación:** Una excepción personalizada permite identificar errores específicos del dominio (como un RUT inválido) de forma más clara y controlada, mejorando la comprensión del código y la experiencia del usuario.

* **Pregunta 3 -** Respuesta correcta: Composición o asignación (has-a).

**Retroalimentación**: Cuando una clase “tiene un(a)” objeto de otra clase como atributo, estamos ante una relación de composición o asignación (has-a), no de herencia. Esto mejora el diseño modular y reutilizable del sistema.



Reservados todos los derechos Fundación Instituto Profesional Duoc UC. No se permite copiar, reproducir, reeditar, descargar, publicar, emitir, difundir, de forma total o parcial la presente obra, ni su incorporación a un sistema informático, ni su transmisión en cualquier forma o por cualquier medio (electrónico, mecánico, fotocopia, grabación u otros) sin autorización previa y por escrito de Fundación Instituto Profesional Duoc UC La infracción de dichos derechos puede constituir un delito contra la propiedad intelectual.