Orientación a Objetos 1

Guia teoria

Anexos: Maven, Java, Streams, Diseño, UML

Maven

Es una herramienta de gestión y construcción de proyectos, principalmente para proyectos Java. Es parte del ecosistema Apache y facilita la gestión de dependencias, la compasión, la ejecución de pruebas y la generación de informes, entre otras tareas. Utiliza un archivo XML, POM que permite especificar las dependencias con otros módulos, el orden de ejecución de estos módulos, entre otras cosas.

Dependencias: son las bibliotecas utilizadas en un proyecto. Maven facilita su incorporación y actualización, evitando problemas de versiones y asegurando que se tengan todas las dependencias necesarias para la compilación y ejecución.

Proporciona una estructura de proyecto estándar (**arquetipos**) lo que hace más fácil para los desarrolladores unirse a nuevos proyectos, ya que todos siguen la misma organización básica.

Automatiza tareas comunes como la compilación, empaquetado, despliegue y ejecución de pruebas, lo que ahorra tiempo y reduce errores.

Es compatible con otros entornos, se integra bien con herramientas de desarrollo como IDEs y sistemas de integración continua.

Java

Colecciones

Java provee un framework de colecciones muy rico conformado por un conjunto de interfaces y clases que las implementan. En este apunte describiremos solamente **ArrayList.**

Una instancia de la clase **java.util.ArrayList** constituye una colección lineal ordenada que crece dinámicamente y puede contener elementos duplicados. Ofrece acceso posicional en tiempo constante. Está indexada comenzando desde cero; es decir, el primer elemento de la lista se encuentra en la posición cero.

Permite agregar y eliminar elementos. Cada vez que se agrega un elemento, la lista lo ubica en la última posición, pero pueden agregarse en otras posiciones, lo cual genera desplazamiento de una posición en sentido creciente de todos los elementos que se encuentren ubicados a partir del lugar en el cual se va a agregar el nuevo elemento. Pueden eliminarse elementos de cualquier posición de la lista, cuando ocurre, la lista genera un desplazamiento en sentido contrario al anterior con el fin de no dejar espacios vacíos, excepto al eliminar el último elemento de la lista.

Definición y creación de colecciones

Java permite incluir, en el tipo de las colecciones, el tipo de elemento que contienen. Este tipo podría ser una clase o una interfaz, pero no un tipo primitivo, para este caso deberían utilizarse wrappers.

List<T> items = new ArrayList<T>(); Este tipo permitirá al compilador chequear que no se agreguen elementos a la lista que no sean del tipo T o alguna de sus subclases.

Protocolo de uso

Algunos métodos comunes para un ArrayList son:

- **add(<T> object)**
- **add(int index, <T> object)**
- **get(int index)**
- **size()**
- **contains(<T> object)**
- **remove(int index)**
- **clear()**
- **isEmpty()**

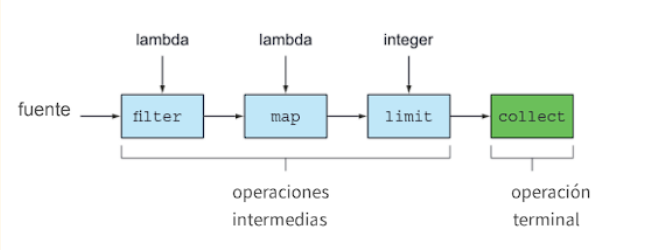
Expresiones Lambda

Son funciones anónimas que no pertenecen a ninguna clase y son utilizadas porque necesitamos utilizar una funcionalidad una única vez. Normalmente, se crean con el mero propósito de enviarlas como parámetro a una función de alto orden (función que recibe otra función como parámetro).

Su sintaxis es **(parámetros, separados, por, coma) -> {cuerpo lambda}** "->" separa los parámetros de la declaración del cuerpo de la función. Cuando se tiene un solo parámetro no es necesario el uso de paréntesis, para cualquier otro caso, es necesario incluso si no hay parámetros. Para el cuerpo no son necesarias las llaves si la expresión tiene una única línea y no necesita especificarse la cláusula return en el caso de que retorne algún valor. Si tiene más de una línea, es necesario el uso de llaves y especificar la cláusula return.

API Streams

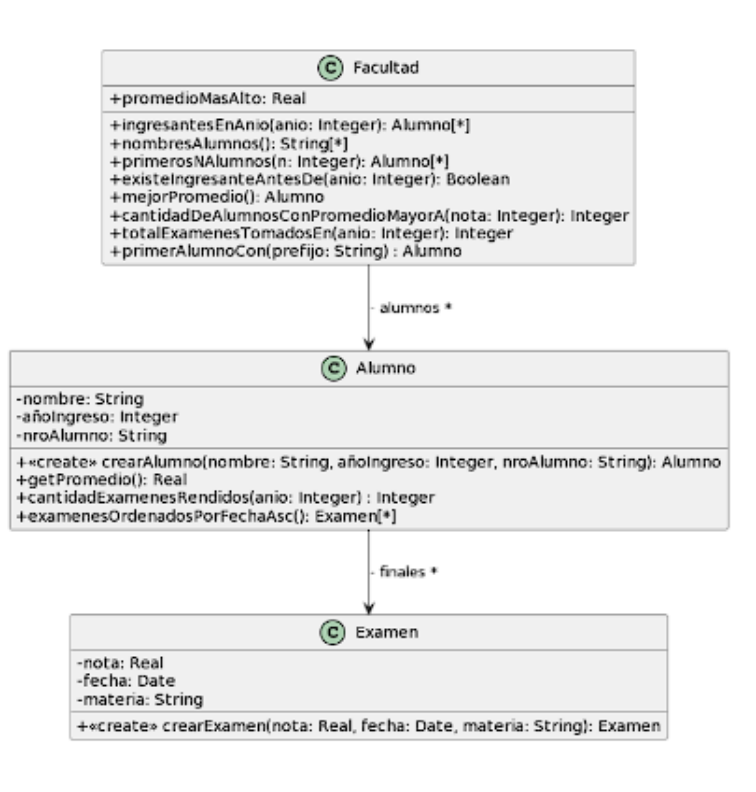
Es una API de Java que facilita el manejo de colecciones. Esto permite hacer operaciones como buscar, filtrar, sumar elementos de una colección de forma sencilla. La API Stream define una amplia variedad de operaciones con la intención de permitir su encadenamiento.



Las operaciones de la API se pueden clasificar en operaciones intermedias, que retornan un stream para permitir el encadenamiento con otras operaciones y operaciones terminales, que permiten retornar un tipo diferente a Stream, como una Lista, un Integer o void. Si no se quiere retornar un stream, siempre debe utilizarse una operación terminal.

Operaciones intermedias
filter
map
limit
sorted

Operaciones terminales
count | sum
average
findAny | findFirst
collect
anyMatch | allMatch | noneMatch
min | max



API STREAM

FILTER

Filtra los elementos de un stream según el predicado que recibe como parámetro. Ej: obtener los alumnos que ingresaron en un año dado

```
public List<Alumno> ingresantesEnAnio(int anio){
  return alumnos.stream()
  .filter(alumno->alumno.getAnioIngreso() == anio)
  .collect(Collectors.toList());
```

MAP

Genera un stream, de igual longitud que el original, a partir de aplicar la función que recibe como parámetro sobre cada elemento del stream original. Ej: obtener una lista de los nombres de todos los alumnos.

```
public List<String> nombresAlumnos(){
   return alumnos.stream()
          .map(alumno -> alumno.getNombre())
          .collect(Collectors.toList());
}
```

Op. relacionadas: MAPTODOUBLE | MAPTOINT

Genera un subtipo de stream (DoubleStream, IntStream...), de igual longitud que el original, a partir de aplicar la función que recibe como parámetro sobre cada elemento del stream original. Se utiliza cuando se trabaja con tipos primitivos como double e int respectivamente.

LIMIT

Trunca el stream dejando los primeros N elementos. Ej: obtener una lista de los primeros N alumnos.

```
public List<Alumno> primerosNAlumnos(int n){
   return alumnos
       .stream().limit(n)
       .collect(Collectors.toList()); }
```

ANYMATCH

Evalúa si existe al menos un elemento del stream que satisface el predicado que se recibe como parámetro, y en ese caso retorna verdadero. Caso contrario, retorna falso. Ej: consultar si algún alumno ingresó antes de un año dado o no.

```
public boolean existeIngresanteAntesDe(int anio){
  return alumnos.stream()
  .anyMatch(alumno->alumno.getAnioIngreso()<anio);
}
```

Op. relacionadas: ALLMATCH | NONEMATCH

Evalúa si todos los elementos (o ninguno de los elementos) del stream satisfacen el predicado que se recibe como parámetro, y en ese caso retorna verdadero. Caso contrario, retorna falso.

MAX | MIN

Retorna el elemento máximo del stream de acuerdo a la expresión indicada como parámetro.
Ej: obtener el alumno con el mayor promedio

```
public Alumno mejorPromedio(){
  return alumnos.stream()
  .max((a1, a2)-> Double.compare(
        a1.getPromedio(), a2.getPromedio()))
  .orElse(null);
}
```

Si trabajamos con un stream de números (DoubleStream, IntStream...) no requiere parámetros.
Ej: se quiere obtener el promedio más alto.

```
public double promedioMasAlto(){
  return  alumnos.stream()
         .mapToDouble(alumno->alumno.getPromedio())
         .max().orElse(0);
}
```

COUNT

Retorna la cantidad de elementos en el stream. Ej: obtener la cantidad de alumnos con promedio mayor a una nota determinada

```
public int cantidadDeAlumnosConPromedioMayorA(int nota){
  return (int) alumnos.stream()
  .filter(alumno-> alumno.getPromedio() >= nota)
  .count();
```

SUM

Retorna la suma de los elementos de un stream de números (DoubleStream, IntStream...). Ej: calcular cuántos exámenes se tomaron en total a todos los alumnos, en un año determinado.

```
public int totalExamenesTomadosEn(int anio){
 return alumnos.stream()
 .mapToInt(alumno->alumno.cantidadExamenesRendidos(anio))
 .sum();
}
```

AVERAGE

Retorna el promedio de los elementos de un stream de números (DoubleStream, IntStream...).
Ej: En la clase Alumno, obtener el promedio de un alumno

```
public double getPromedio(){
  return  examenes.stream()
         .mapToDouble(examen->examen.getNota())
         .average().orElse(0);
}
```

SORTED

Ordena los elementos de un stream de acuerdo a la expresión que recibe como parámetro.
Ej: En la clase Alumno, obtener los exámenes ordenados por fecha de forma ascendente

```
public List<Examen> examenesOrdenadosPorFechaAsc(){
  return finales.stream()
     .sorted((ex1, ex2) ->
        ex1.getFecha().compareTo(ex2.getFecha()))
  .collect(Collectors.toList());
}
```

FINDFIRST | FINDANY

Ej: obtener el primer alumno cuyo nombre comience con una cadena dada. Si no existe ninguno, obtiene null.

```
public Alumno primerAlumnoCon(String x){
  return alumnos.stream()
.filter(alumno->alumno.getNombre().startsWith(x))
  .findFirst().orElse(null);
}
```

Criterios y heurísticas de diseño

Esta guía sirve como una checklist para resolver ejercicios de la materia, no es exhaustiva ni la única opción para las buenas prácticas y conceptos de programación orientada a objetos.

Malos olores de diseño

Los siguientes son malos olores en el diseño OO que no deben estar presentes en nuestros programas.

- **Envidia de atributos:** un objeto le pide a otros objetos cosas para hacer operaciones. Se evita haciendo que la tarea la debe hacer el objeto que tiene los elementos que se necesitan para la operación.

- **Clase Dios:** una clase que resuelve todo y además las demás son clases anémicas. No cumple el principio de una sola responsabilidad. Se evita viendo que otros objetos podría crear para encargarse de alguna de las responsabilidades de esta clase Dios.
- **Código duplicado:** se evita generalizando el funcionamiento en una clase y heredando, llevándolo a otro objeto y reutilizar por composición, extraerlo en un método de la misma clase y reusarlo.

- **Clase larga:** una clase es muy grande en comparación al resto. Se evita evaluando si esa clase puede delegar algo a otros objetos y evaluando que no sea una clase de Dios.

- **Método largo:** si un método tiene más de 10 renglones es una mala señal. Al igual que si es necesario incluir comentarios en medio del método. Para evitarlo debe evaluarse si el método puede dividir comportamientos en métodos individuales.

- **Objetos que conocen el id de otro:** nunca relacionar objetos por medio de claves o IDs. Para evitarlo, cuando un objeto se relaciona con otro, lo hace con una referencia. Nunca por su id.

- **Eso debería ser un objeto (obsesión por los primitivos):** a veces se modelan como strings o numerosos datos que deberían ser objetos. Cuando hacemos eso, el comportamiento que debería tener ese objeto termina estando en un lugar que no corresponde. Para evitarlo debemos pensar si eso que estoy modelando como un dato primitivo no debería ser modelado como una clase más específica.

- **Switch statements:** debería sentir mal olor cuando veo que se usa un **if** (o algo que parece un **case**, un **switch** o **if anidados**) para determinar la forma en la que se resuelve algo. Para evitarlo debemos aplicar adecuadamente el polimorfismo.

- **Variables de instancia que en realidad deberían ser temporales:** si una variable de instancia deja de tener sentido en algún momento de la vida del objeto, entonces es probable que sea temporal o que sea responsabilidad de otro. Para evitarlo debemos evaluar si esa variable es realmente un atributo del objeto o si es algo que necesito temporalmente dentro de un método.

- **Romper encapsulamiento:** nos hace perder la gran mayoría de ventajas de la OO. No solo es preocupante acceder a variables de instancia de otros objetos, además podemos romper el encapsulamiento de manera más sutil. Agregando automáticamente setters y getters por ejemplo. Si modificamos una colección que no es de otro objeto, atentamos contra el encapsulamiento. Para evitarlo debemos solo agregar setters y getters cuando sea necesario, nunca modificar una colección que no es nuestra (de otro objeto), delegar las tareas a los que tienen la información que se necesita.

- **Clase de datos o clase anémica:** una clase que parece un registro de datos debería dar mala espina. A veces los enunciados son simplificaciones que hacen que algunas clases terminen siendo así, pero por lo general sospecho cuando la clase solo tiene datos y no tiene comportamientos. Para evitarlo debemos asegurarnos que no hay comportamiento en el sistema que debería estar haciendo esa clase y lo hace otro.

- **No es-un:** una relación de herencia siempre respeta el principio es-un, sí me pregunto si B es un A la respuesta debería ser un sí, caso contrario, tiene mal olor. Para evitarlo siempre debemos evaluar esto al definir una subclase, a veces es una señal de que tanto A como B son subclases de otra que aún no definimos.

- **No quiero mi herencia:** cuando un método redefine a uno heredado, pero lo hace algo totalmente diferente, debemos desconfiar. Para evitar debemos pensar si no puedo reorganizar la jerarquía de clases para que ninguna clase herede comportamientos que no quiere.

- **Reinventando la rueda:** un principio fundamental de la POO es que las cosas se escriben una sola vez y donde corresponde. De esta manera, los módulos son más fáciles de mantener y utilizar. Tiene mal olor cuando defino un comportamiento que sospecho ya fue definido en otro lado como colecciones, iteradores o streams. Para evitarlo debemos investigar y aprender las clases y protocolos que ofrecen las librerías de objetos a mi disposición. Intento siempre utilizar un comportamiento que ya fue definido.

Estilo de programación

Los siguientes son patrones de estilo y buenas prácticas que deberían respetar nuestros programas.

- **Ofrecer constructores:** simplifica la tarea de quien crea el objeto. Garantiza una buena inicialización.

- **Nombre de mensaje que revela la intención:** que el nombre del mensaje especifique que quiere hacer, no cómo lo va a hacer.

- **Delegación a esto:** permite descomponer un método en partes que el mismo objeto resuelve. Cada método hace una cosa. Su nombre indica lo que hace. Quedan todos cortos. Permite que la subclase redefina o extienda solo un paso.

- **Métodos cortos:** para lograrlo utiliza esta delegación. Para que sean fáciles de leer, los nombres de mensajes revelan la intención.

- **Cada cosa se hace una sola vez:** para ello es importante aprender el protocolo de colecciones y otros objetos frecuentemente utilizados. Es recomendable explorar el protocolo de los objetos que voy a utilizar antes de comenzar a programar.

- **Los nombres de las variables deben indicar su rol:** los nombres siempre comienzan en minúsculas. No temas a los nombres largos de variables, con varias palabras y sintaxis de camello.

- **Piensa bien los nombres de las clases:** estos siempre inician en mayúsculas y singular. No temas a los nombres largos. Si es posible, que el nombre de la subclase, ayude a reconocer que es un caso de una superclase, por ejemplo, agregando una palabra al nombre de la superclase para definir el nombre de la subclase.

UML (Lenguaje Unificado de Modelado)

Es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema de software. Permite capturar decisiones y conocimientos.

Diagramas de Comportamiento

Diagrama de Casos de Uso

Un caso de uso es una representación del comportamiento de un sistema tal como se percibe por un usuario externo. Es una representación de una funcionalidad específica.

Describe una interacción específica entre los actores y el sistema, proporcionando un proceso completo de cómo se utiliza el sistema en situaciones reales.

El término “actor” engloba a personas, así como a otros sistemas que interactúan con el sistema.

Los elementos de un modelo de casos de usos son: actores, relaciones y casos de uso.

Actor: representa a un usuario externo, un sistema o una entidad que interactúa con el sistema que se está modelando.

Relaciones: un caso de uso incluye a otro si el primero incorpora el comportamiento especificado en el segundo. Un caso de uso se extiende a otro si puede ser mejorado o ampliado con funcionalidad adicional en ciertos escenarios, pero no siempre se aplica.

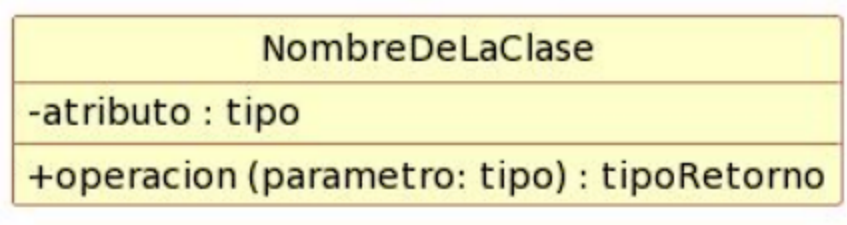
Diagramas de Casos de Uso - conversación

Comprar producto			
1-	Se indica el producto que se quiere comprar		
		2-	Se agrega el producto al carrito de compras
3.	Se procede a terminar la compra		
		4-	Se calcula el total
		5-	Se muestran las opciones de pago
6	Se elige una opción de pago		
		7-	Se crea una orden de compra, registrando el pago
	acciones del actor		respuesta del sistema

Diagramas de Clases

Una clase es una descripción de conjunto de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica.

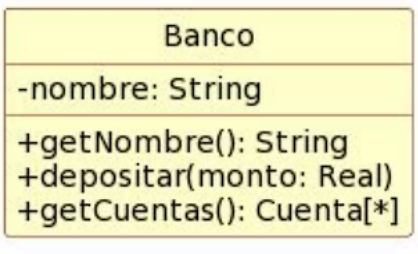
Una clase es representada gráficamente por cajas con tres comportamientos.



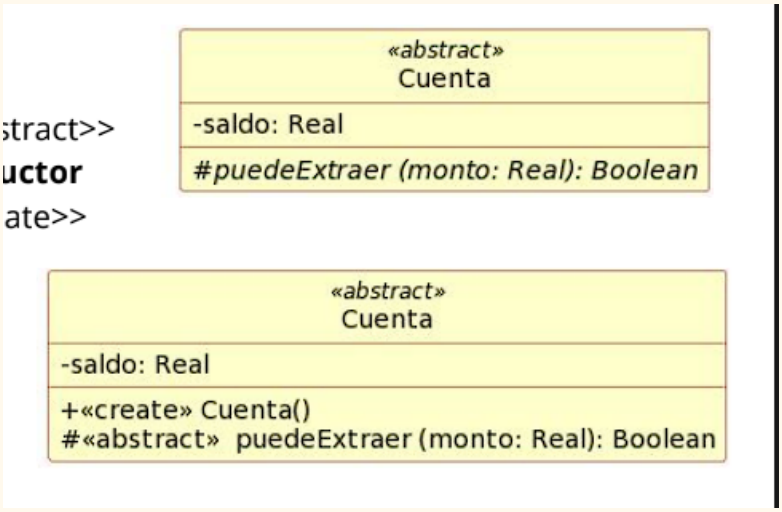
Nombre de la clase: debe ser singular, empezar con mayúsculas y estilo CamelCase. Si es clase abstracta el estereotipo <<abstract>> estará en cursiva.

Atributos de la clase: tienen visibilidad (privada, protegida, pública o paquete), nombre en estilo camelCase comenzando en minúscula, un tipo (tipos UML: integer, real, boolean, string) y tienen un valor por defecto.

Operaciones de la clase (métodos): tienen visibilidad y nombre igual al atributo de la clase, usan parámetros con nombre en estilo camelCase, poseen un tipo de retorno (nada o no específico, un objeto previamente indicando su clase, una colección se indica con el nombre de la clase [*]).

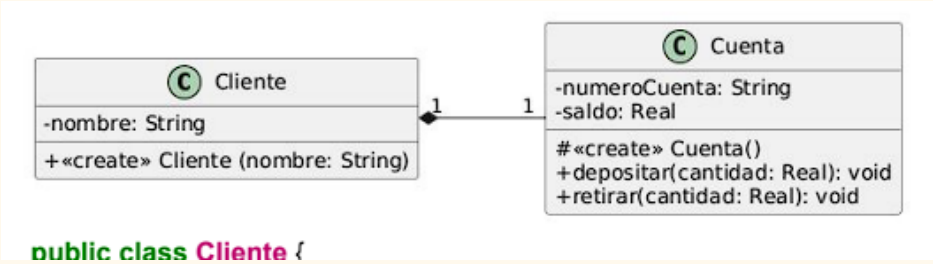


Abstracto: si el método es abstracto, usa cursiva y el estereotipo <<abstract>>.
Constructor: si el método es un constructor usa el estereotipo <<create>>,

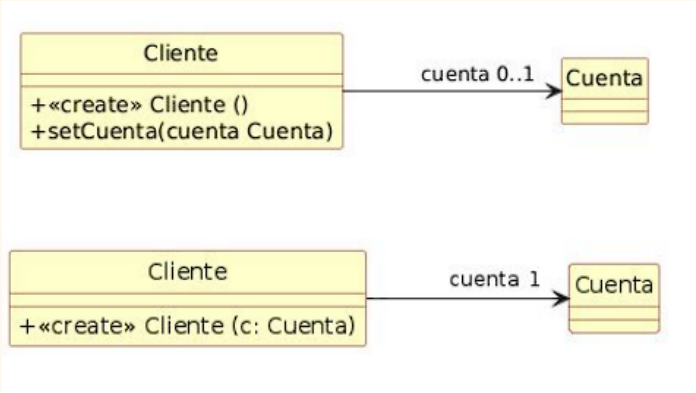


Asociación: representa la relación entre las clases.

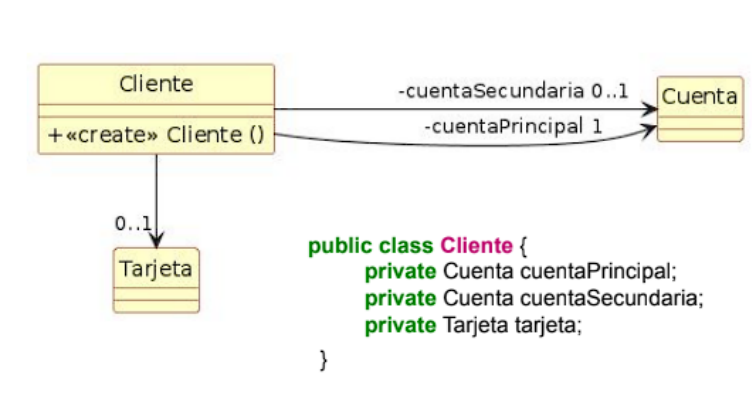
- **Navegabilidad:** indica la dirección en la que se puede acceder a una clase desde otra. Una flecha en el extremo de una línea indica que una clase puede acceder a otra; sin flecha, la navegación es bidireccional.



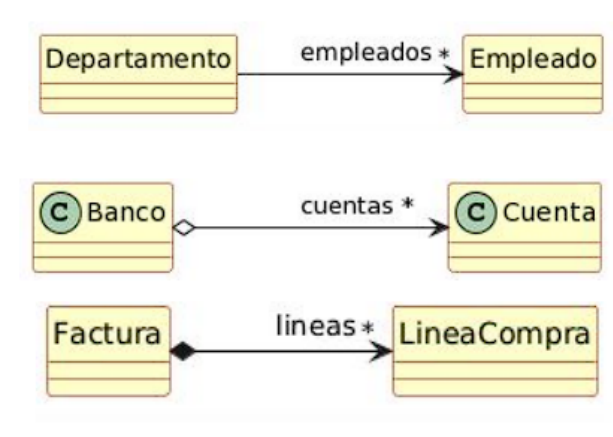
- **Multiplicidad:** indica cuántas instancias de una clase pueden estar asociadas a una instancia de otra. Se representa con valores como 1, 0..* (muchos), 1..* (uno o más).



- **Nombre de rol:** es un indicador opcional que indica el rol de la clase en la asociación. Se escribe junto al extremo de la línea de asociación.



- **Tipos: simple** (conexión básica entre dos clases, no hay una dependencia fuerte entre ellas), **agregación** (relación "tiene un" donde una clase es contenedor de otras clases contenidas que pueden existir independientemente, se representa con un rombo vacío), **composición** (relación "es parte de" donde las clases contenidas dependen totalmente de la clase contenedor. Si el contenedor se elimina sus partes también, se representa con un rombo sólido.).



Interfaces: una interfaz define un conjunto de operaciones que una clase o componente debe implementar. Actúa como un contrato que establece qué métodos deben ser implementados.

En UML las interfaces se representan mediante un rectángulo con el nombre interfaz precedido del estereotipo <<interfaz>>.

La implementación de los métodos definidos en la interfaz es responsabilidad de las clases que la implementan.

Implementación y herencia: una clase se extiende a una clase abstracta, significa que debe proveer las implementaciones de los métodos abstractos. En cambio, con una interfaz, la clase proporciona la implementación de todos los métodos definidos en esa interfaz.

Dependencias: indica que un cambio en la especificación de una clase puede afectar a otra clase que depende de ella. Esta relación se utiliza para mostrar que una clase o componente utiliza o necesita a otro para funcionar correctamente, pero sin mantener una referencia duradera a él.

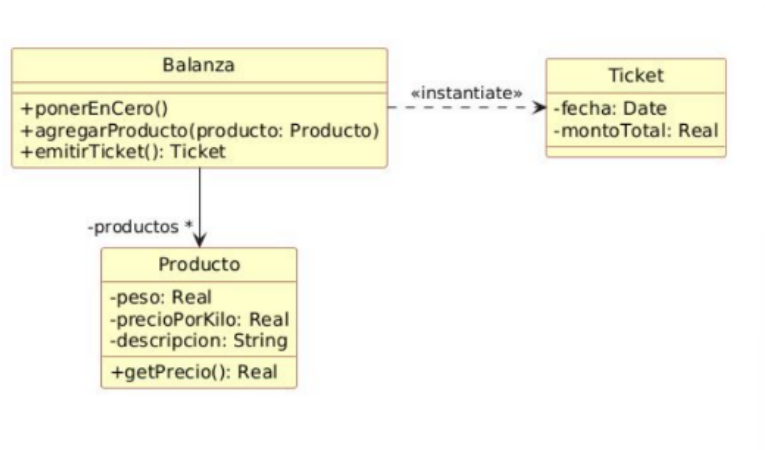


Diagrama de Objetos

Permiten visualizar una instancia específica de un sistema en un momento dado. Se pueden mostrar los valores de los atributos y los links que son las referencias a otros objetos.

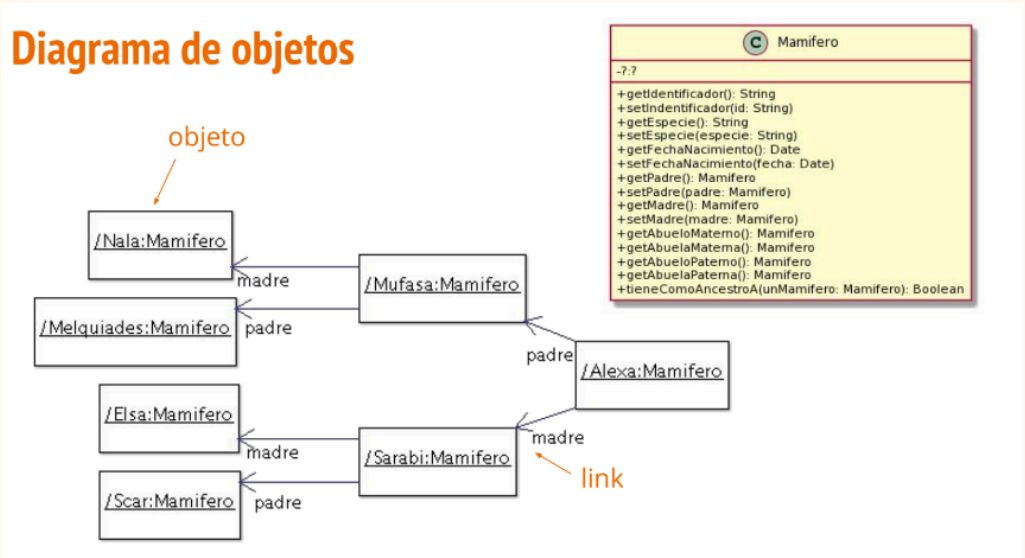


Diagrama de Paquetes

Permiten la agrupación de clases. Son útiles para mostrar la organización de un sistema y como los elementos se agrupan y relacionan entre sí.

Se quiere:

- Una alta cohesión dentro de un paquete. Los elementos dentro de un paquete están relacionados.
- Poco acoplamiento entre ellos, exportando solo aquellos elementos necesarios e importando solo los necesarios.

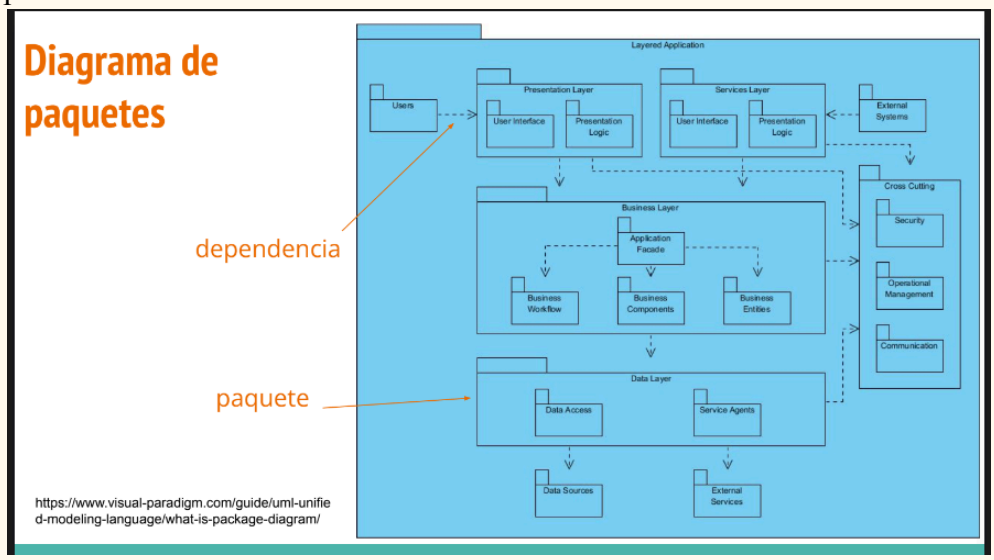
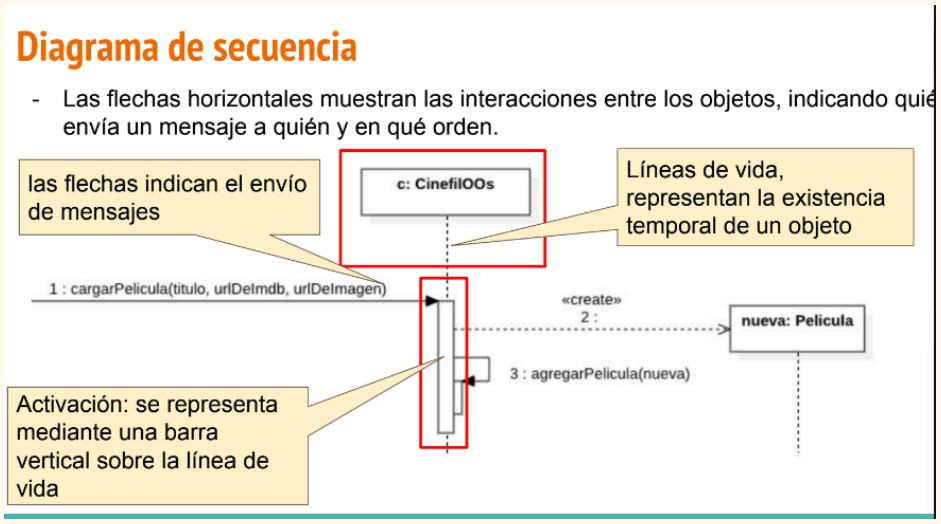


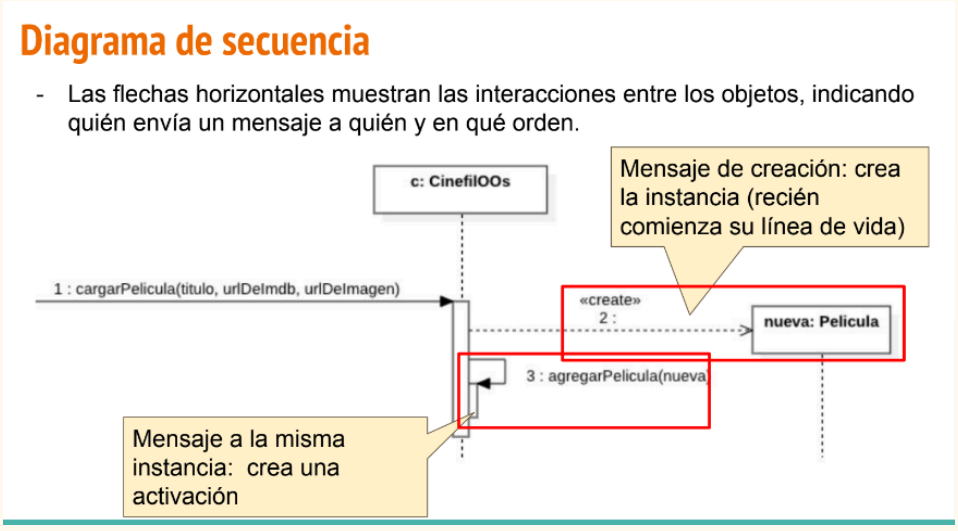
Diagrama de Secuencia

Es un tipo de diagrama de interacción porque describe cómo y en qué orden colabora un grupo de objetos.

Muestra claramente cómo interactúan distintos objetos en un sistema a lo largo del tiempo.

En un diagrama de secuencia, los objetos se representan en la parte superior del diagrama y el tiempo avanza de arriba hacia abajo.

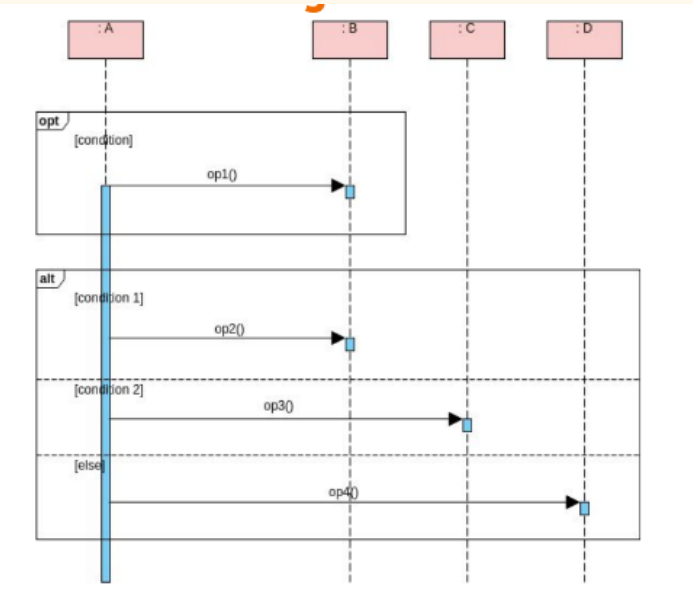




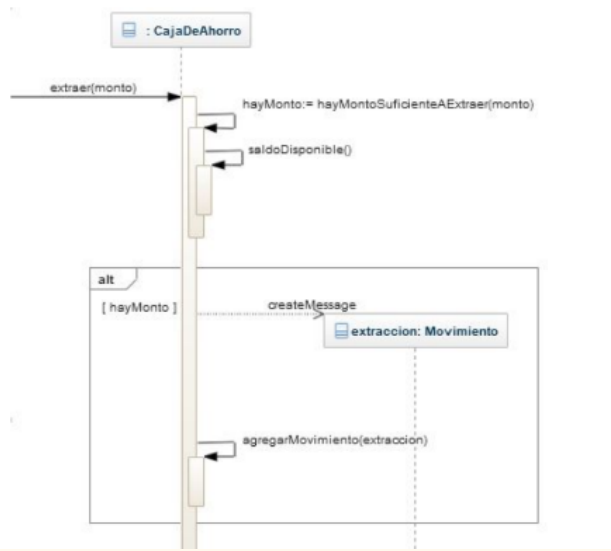
Combined Fragment: un fragmento combinado es un elemento que se utiliza para representar la lógica de control y las estructuras condicionales en una secuencia de interacción entre objetos. A través de ellos se pueden especificar bloques repetitivos, opcionales y alternativos, entre otros.

Fragmentos más utilizados:

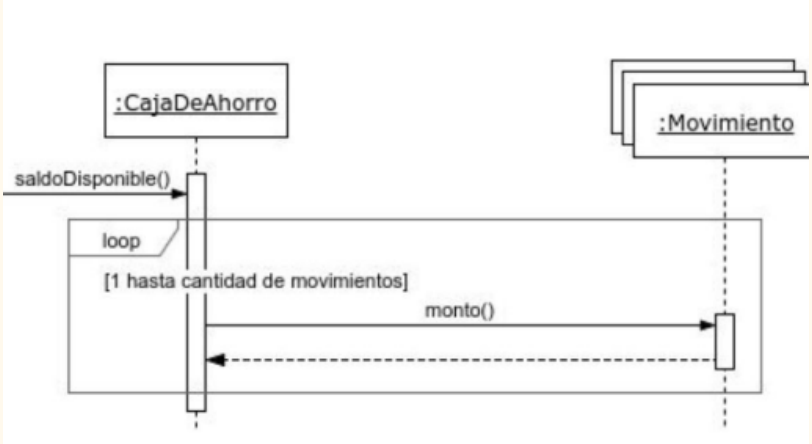
- **opt (opcional):** representa una parte de la secuencia de interacción que puede o no ejecutarse, dependiendo de una condición booleana. Si la condición es verdadera, se ejecuta la parte opcional; de lo contrario, se omite.



- **alt (alternativa):** se utiliza para modelar una elección entre diferentes opciones de interacción. En cada opción se evalúa una condición booleana para determinar cuál de las opciones se ejecutará.



- **loop (bucle):** se utiliza para modelar repeticiones de una secuencia de interacción. Puede especificar el número de repeticiones o utilizar una condición para controlar la terminación del bucle.

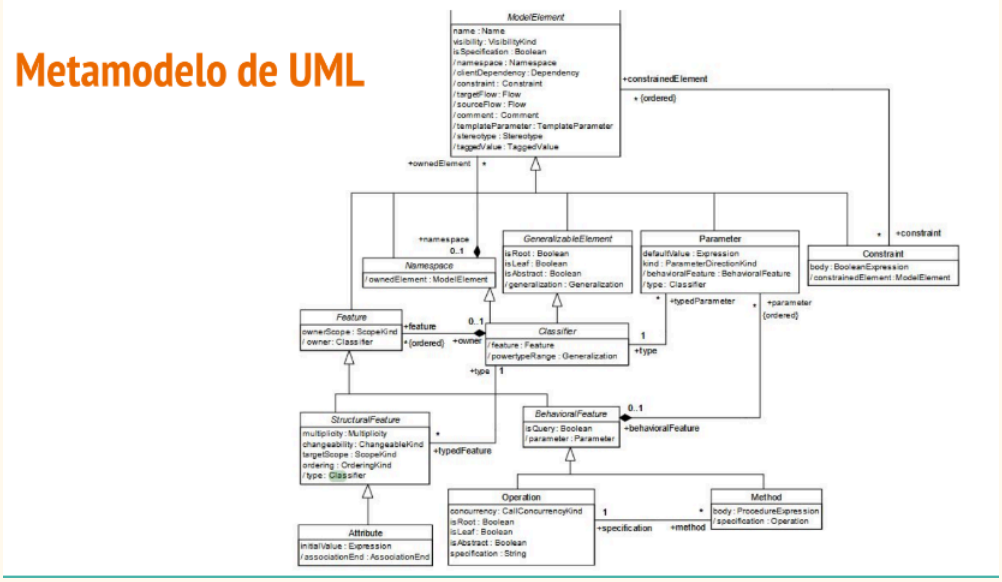


- **break.**

Herramientas UML

UML se define como un documento de especificación de UML.

El metamodelo de UML es una especificación que define las construcciones y elementos básicos que pueden utilizarse para crear diagramas en UML. Es esencialmente una descripción formal de cómo se estructuran y relacionan los elementos.



Herramientas de modelado

Pueden ser tanto gráficas como basadas en metamodelo UML.

- **Herramienta gráfica:** permite dibujar elementos con la misma imagen que UML.
- **Herramienta basada en el metamodelo UML:** estas herramientas se centran más en la manipulación directa de los elementos del metamodelo UML.

Algunas herramientas

- **PlantUML:** herramienta que pasa de texto a imagen, no tiene chequeo de metamodelo. Permite dibujar cosas ajenas a UML, o con errores semánticos. Tiene versión online. En UML no es posible una jerarquía en ciclo.
- **starUML:** realiza el chequeo de algunas reglas definidas en el metamodelo. Es inestable.
- **GenMyModel:** realiza el chequeo de algunas reglas definidas en el metamodelo. Tiene versión online. No es posible una jerarquía en ciclo ni es posible dibujarlo.

Clase 1: Introducción a objetos

Un software construido con objeto es un conjunto de objetos que colaboran enviando mensajes. Todo el cómputo ocurre dentro del código de los objetos. La clave de éxito es poder agregar una nueva funcionalidad, reemplazar objetos o modificarlos y que el sistema no se rompa.

Los sistemas están compuestos únicamente por objetos que colaboran para llevar a cabo sus responsabilidades. Los objetos son responsables de conocer sus propiedades, conocer otros objetos y llevar a cabo determinadas acciones.

Aspectos de interés en esta definición.

- No existe un objeto “main”.
- Cuando codificamos, describimos clases.
- Una jerarquía de clases no indica lo mismo que una jerarquía top-down.
- Cuando se ejecuta el programa, lo que tenemos son objetos que cooperan y que se crean dinámicamente durante la ejecución del programa.
- Podemos pensar la interacción usuario / software de la misma manera.
- Este mismo modelo nos permite entender otros modelos de computación: viendo a los objetos como proveedores de servicios, por ejemplo.
- Este mismo modelo no asume objetos localizados en el mismo espacio de memoria, pueden estar distribuidos.

Cambio en cómo “pensamos” el software

- La estructura general cambia, en vez de una jerarquía: main, procedures, sub-procedures, tenemos una red de objetos que se comunican.
- Pensamos en qué cosas hay dentro del software y cómo se comunican entre sí.
- Hay un shift mental crítico en la forma en la cual pensamos el software como objetos.

Objeto

Es una abstracción de una entidad del dominio del problema (persona, producto, cuenta bancaria, auto, etc.). Puede representar también conceptos del espacio de la solución (estructuras de datos, archivos, iconos, ventanas, etc.).

Un objeto tiene:

- **Identidad:** para distinguir un objeto de otro.
- **Conocimiento:** con base en sus relaciones con otros objetos y estado interno.
- **Comportamiento:** conjunto de mensajes que un objeto sabe responder.

El estado interno

Determina su conocimiento. Está dado por las propiedades básicas del objeto, otros objetos con los cuales colabora para llevar a cabo sus responsabilidades. El estado interno se mantiene en las variables de instancia del objeto. Es privado el objeto, ningún otro objeto puede accederlo.

Variables de instancia

En general, las variables son referencias (punteros) a otros objetos con los cuales el objeto colabora. Algunas pueden ser atributos básicos.

Comportamiento

Un objeto se define en términos de su comportamiento. El comportamiento indica qué sabe hacer el objeto, sus responsabilidades. Se especifica a través del conjunto de mensajes que el objeto sabe responder, llamado protocolo.

La realización de cada mensaje se especifica a través del método. Cuando un objeto recibe un mensaje, responde activando el método asociado. El que envía el mensaje delega en el receptor la manera de resolverlo, que es privada del objeto.

Envío de un mensaje

Para poder enviarle un mensaje a un objeto, hay que conocerlo. Al enviarle un mensaje a un objeto, este responde activando el método asociado al mensaje. Como resultado del envío de un mensaje, puede retornar un objeto.

Especificación de un mensaje

Se especifica mediante un nombre, que es correspondiente al protocolo del objeto receptor, y parámetros que son la información necesaria para resolver el mensaje.

Métodos

Es la contraparte funcional del mensaje, expresa la forma de llevar a cabo la semántica propia de un mensaje particular.

Básicamente, puede realizar tres acciones:

- Modificar el estado interno del objeto.
- Colaborar con otros objetos enviándoles mensajes.
- Retornar y terminar.

En un sistema diseñado correctamente, un objeto no debería realizar ninguna operación vinculada a la interfaz o a la interacción. En la mayoría de entornos de desarrollo es incluso imposible hacerlo.

Formas de conocimiento

Para que un objeto conozca a otro lo debe poder "nombrar". Decimos que se establece una ligadura entre un nombre y un objeto.

Podemos identificar tres formas de conocimiento o tipos de relaciones entre objetos:

- **Conocimiento interno:** variables de instancias.
- **Conocimiento externo:** parámetros.
- **Conocimiento temporal:** variables temporales.

También existe una cuarta forma de conocimiento especial, las pseudo variables (como this o self).

Encapsulamiento

Es la cualidad de los objetos de ocultar los detalles de implementación y su estado interno al mundo exterior.

Características:

- Esconde detalles de implementación.
- Protege el estado interno de los objetos.
- Un objeto solo muestra su cara visible por medio de su protocolo.
- Los métodos y su estado quedan escondidos para cualquier otro objeto. Es el objeto quien decide que se publica.
- Facilita modularidad y reutilización.

Clases e instancias

Una clase es una descripción abstracta de un conjunto de objetos.

Cumplen tres roles:

- Agrupan el comportamiento común a sus instancias
- Definen la forma de sus instancias.
- Crean objetos que son instancias de ellas.

En consecuencia, todas las instancias de una clase se comportan de la misma manera. Cada instancia mantendrá su propio estado interno.

Especificación de clases

Las clases se especifican por medio de un nombre, el estado o estructura interna que tendrán sus instancias y los métodos asociados que definen el comportamiento.

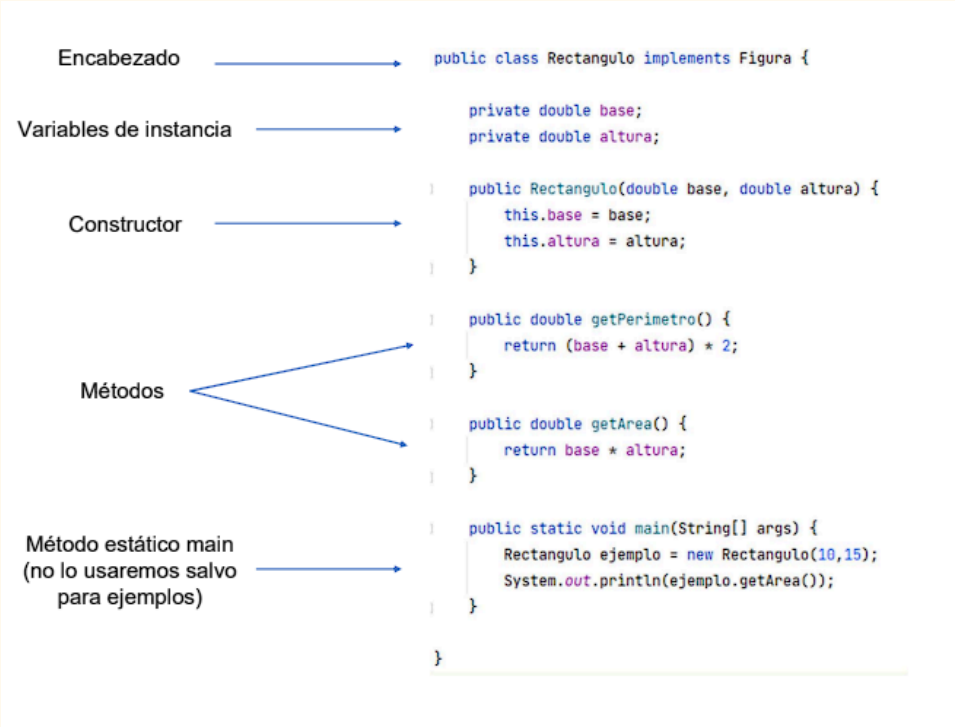
Instanciación

Es el mecanismo de creación de objetos. Los objetos se instancian a partir de un molde. La clase funciona como molde. Un nuevo objeto es una instancia de una clase. Todas las instancias de la misma clase tendrán la misma estructura interna y responderán al mismo protocolo de la misma manera.

Inicialización

Para que un objeto esté listo para llevar a cabo sus tareas hace falta inicializarlo. Esto significa darle valor a sus variables.

Clase 2: Relaciones objetosas



Un objeto conoce a otro porque es su responsabilidad mantener a ese otro objeto en el sistema (guardan relación estrecha) o necesita delegarle trabajo (enviarle mensajes).

Un objeto conoce a otro cuando tiene una referencia en una variable de instancia (**relación duradera)**. Le llega una referencia como parámetro **(relación temporal)**. Lo crea (**relación temporal / relación duradera)**. Lo obtiene enviando mensajes a otros que conoce (**relación temporal).**

this / self (otros lenguajes)

Es una referencia a sí mismo, un objeto que se comunica consigo mismo. Una pseudo variable no se le puede asignar un valor porque toma el valor automáticamente cuando un objeto comienza a ejecutar un método.

Hace referencia al ejecutor del método y se utiliza para: descomponer métodos largos, reutilizar comportamiento repetido en varios métodos, aprovechar comportamientos heredados o pasar una referencia para que otros puedan enviar mensajes.

Operador ==

Las variables son punteras a objetos y más de una variable puede apuntar al mismo objeto. Para saber si dos variables apuntan al mismo objeto se usa "==", un operador que no puede redefinirse.

Método equals()

Se utiliza para evaluar si dos objetos son iguales, la igualdad se define en función del dominio.

Relaciones entre objetos y chequeo de tipos

Java es un lenguaje estáticamente fuertemente tipado, es decir, debemos indicar el tipo de todas las variables. El compilador chequea la correctitud de nuestro programa respecto a los tipos.

Se asegura de que no enviamos mensajes a objetos que no los entienden.

Cuando declaramos el tipo de una variable, el compilador controla que solo enviamos esa variable, acorde a su tipo.

Cuando asignamos un objeto a otra variable, el compilador controla que su clase sea compatible con el tipo de la variable.

Tipos de lenguajes OO (simplificado)

Cada clase en Java define explícitamente un tipo, se pueden utilizar clases para dar tipo a las variables.

Asignar un objeto a una variable no afecta al objeto, es decir, no cambia su clase. La clase de un objeto se establece cuando se crea y no cambia más.

Interfaces

- Una clase define un tipo y también implementa los métodos correspondientes.
- Una variable tipada con una clase solo acepta instancias de esa clase.
- Una interfaz nos permite declarar tipos sin tener que ofrecer implementación.
- Puedo utilizar interfaces como tipos de variables.
- Las clases deben declarar explícitamente qué interfaces implementan.
- Una clase puede implementar varias interfaces.
- El compilador chequea que la clase implemente las interfaces que declara, salvo que sea una clase abstracta.

Un objeto que conoce a muchos

Las relaciones de un objeto con muchos se implementan mediante colecciones. Decimos que un objeto conoce a muchos, pero en realidad conoce una colección, que tiene referencias a esos muchos. Para modificar y explorar la relación se envían mensajes a la colección.

Method Lookup

Cuando un objeto recibe un mensaje, se busca en su clase un método cuya firma se corresponda con el mensaje. En un lenguaje con tipado estático como java sabemos que lo encontrará, mientras que un lenguaje dinámico podría no hacerlo lo que sería un error en tiempo de ejecución.

Polimorfismo

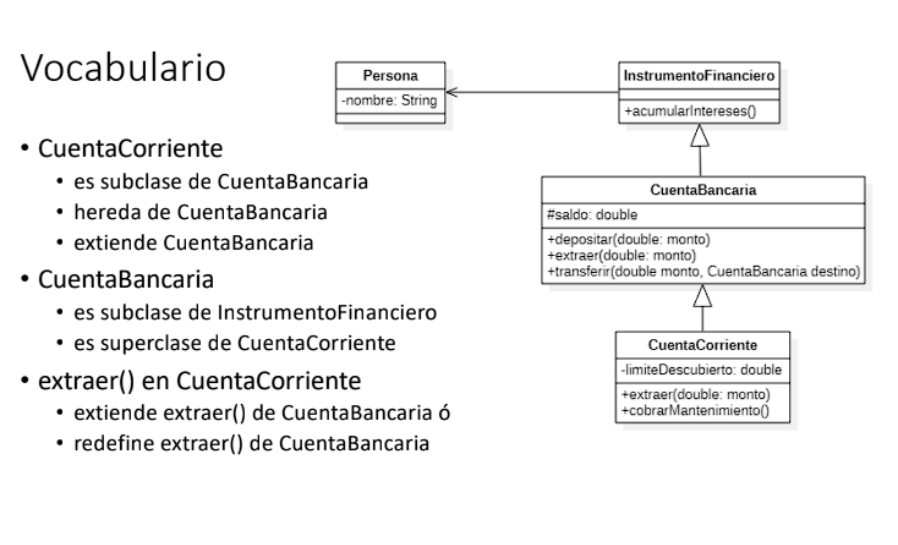
Objetos de distintas clases son polimórficos con respecto a un mensaje si todos lo entienden, aun cuando cada uno lo implemente de un modo diferente. Implica que un mismo mensaje puede enviarse a objetos de distinta clase, los cuales podrían ejecutar métodos diferentes en respuesta al mismo mensaje.

Cuando dos clases Java implementan una interfaz, se vuelven polimorfismo respecto a los métodos de la interfaz.

Bien aplicado permite repartir mejor las responsabilidades (delegar), desacopla objetos y mejora la cohesión (cada uno hace lo suyo), concentra cambios (reduce su impacto), permite extender sin modificar (agregando nuevos objetos), lleva a código más genérico y reusable, nos permite programar por protocolo, no por implementación.

Clase 3: Herencia

Es el mecanismo que permite a una clase heredar estructuras y comportamientos a otra clase. Sirve como estrategia de reuso de código, conceptos o definiciones. En Java solo existe la herencia simple. Es una característica transitiva de los objetos.



Preguntarse "es un" es la regla para identificar usos adecuados de herencia. Si suena bien en el lenguaje del dominio, es probable que sea un uso adecuado.

Method Lookup

Cuando un objeto recibe un mensaje, se busca en su clase un método cuya firma se corresponda con el mensaje. En un lenguaje dinámico podría no encontrarlo (error en tiempo de ejecución). En un lenguaje con tipado estático sabemos que lo entenderá (aunque no sepamos qué hará).

Cuando un objeto recibe un mensaje, se busca en su clase un método cuya firma corresponda con el mensaje. Si no lo encuentra, sigue buscando en la superclase de su clase, y en la superclase de esta, y así hasta que no haya más superclases.

Overriding

Es sobreescribir métodos. La búsqueda en la cadena de superclases termina tan pronto encuentra un método cuya firma coincide con la buscada. Si heredaba un método con la misma firma, el mismo queda oculto (override). No es frecuente (podría ser mala práctica).

Super

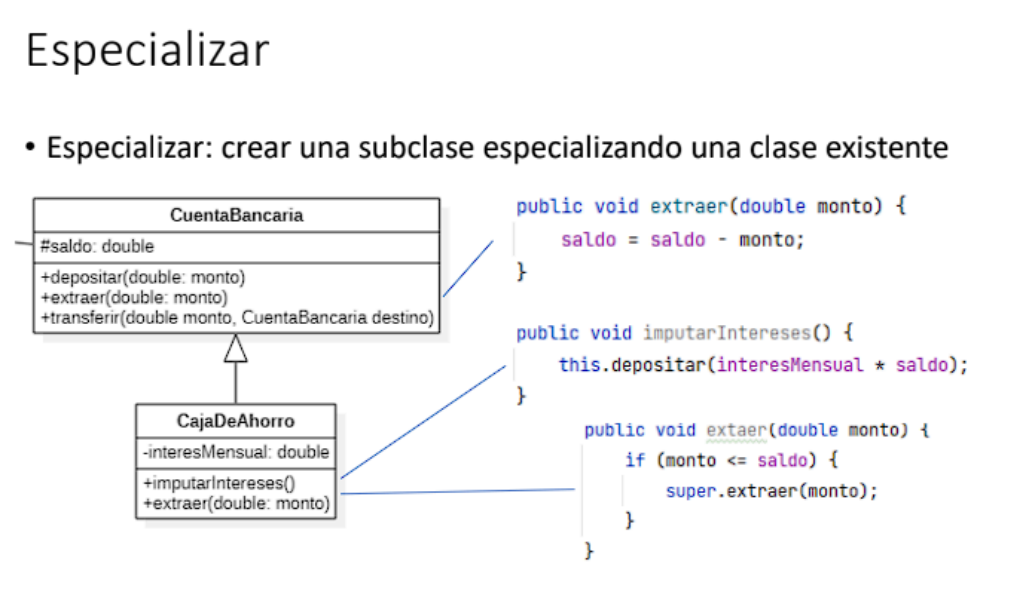
Podemos pensar a super como una “pseudo-variable”, no podemos asignarle un valor y toma un valor automáticamente cuando un objeto comienza a ejecutar un método.

En un método, super y this hacen referencia al objeto que lo ejecuta, es decir, al receptor del mensaje.

Utilizar super en lugar de esto solo cambia la forma en la que se hace el method lookup. Se utiliza para solamente extender comportamientos heredados, es decir, implementar un método e incluir el comportamiento que se heredaba para él.

Cuando se recibe un mensaje, la búsqueda de métodos comienza en la clase inmediata superior a aquella donde está definido el método que envía el mensaje, sin importar la clase del receptor.

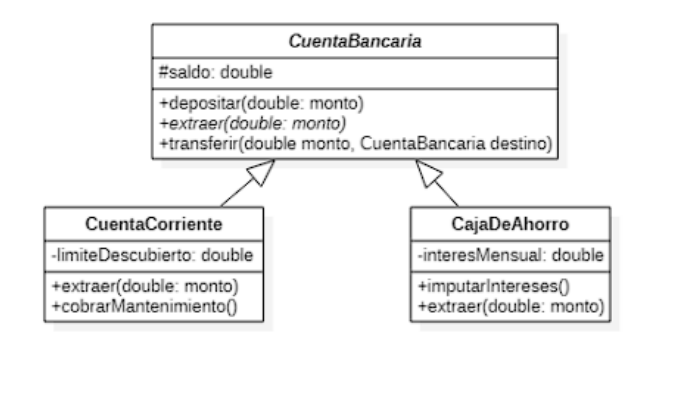
Los constructores en Java son subrutinas que se ejecutan en la creación de objetos, no se heredan. Si quiero reutilizar el comportamiento de otro constructor, debe invocarlo explícitamente usando el método super() al inicio del constructor.



Clase abstracta

Una clase abstracta captura el comportamiento y estructuras que sería común a otras clases. Una clase abstracta no tiene instancias (no modela algo completo). Seguramente será especializada. Puede declarar comportamiento abstracto y utilizarlo para implementar comportamiento concreto.

Generalizar: introducir una superclase que abstrae aspectos comunes a otras, suele resultar en una clase abstracta.



Situaciones de uso de herencia

- **Sub clasificar para especializar:** la subclase extiende métodos para especializarse. Ambas clases son concretas.
- **Herencia para especificar:** la superclase es una combinación de métodos concretos y abstractos. La subclase implementa los métodos abstractos.
- **Sub clasificar para extender:** la subclase agrega nuevos métodos.

Malas prácticas de herencia:

- **Heredar para construir:** heredó comportamientos y estructuras, pero cambió el tipo. Se debe evitar aunque nos vamos a cruzar con ejemplos en el código de otros. Sí, es posible, reemplazar por composición.
- **Sub clasificar para generalizar:** la subclase implementa métodos para hacerlos más generales. Solo tiene sentido si no puedo reordenar la jerarquía para especializarme.
- **Sub clasificar para limitar:** la subclase implementa un método para que deje de funcionar, limitarlo. Solo tiene sentido si no puedo reordenar la jerarquía para especializarme.
- **Herencia indecisa (sub clasificación por varianza):** tengo dos clases con un mismo tipo y algunos métodos compartidos. No puedo decidir cuál es la subclase y cuál la superclase. Resolverlo buscando una superclase común (generalización).

Tipos en lenguajes OO (simplificado)

Es el conjunto de firmas de operaciones o métodos.

Sabiendo esto:

- Cada clase en Java define explícitamente un tipo.
- Cada subclase define explícitamente un subtipo.
- Cada instancia de una clase A es del tipo definido por esa clase.

Donde espero un objeto de una clase A, aceptar un objeto de cualquier subclase A, pero no a la inversa.

Modificadores de visibilidad con herencia

Variable privada en una clase A:

- Las instancias de la subclase A tendrán esa variable de instancia.
- En los métodos de la subclase de A no se puede referenciar.

Método privado en una clase A:

- Las instancias de sus subclases entenderán el mensaje.
- En los métodos de las subclases de A no puedo enviar el método a this

Los métodos y variables declarados como protegidos pueden ser vistos en las subclases.

Clases abstractas e interfaces

Una clase abstracta es una clase:

- Puede usarse como tipo.
- Puede o no tener métodos abstractos.
- Ofrece implementación de algunos métodos.
- Sus métodos concretos pueden depender de sus métodos abstractos.

Una interfaz define un tipo:

- Sirve como contrato.
- Puede extenderse a otras.

Se pueden implementar muchas interfaces pero solo se puede heredar de una clase.

Clase 4: Colecciones

Es un grupo de objetos. Permiten duplicados y los mismos no están ordenados. Los elementos de una colección no son accedidos a través de un índice. Por el contrario, su única forma de recuperación es uno a uno, secuencialmente.

Buenas prácticas

- Nunca modificó una colección que obtuve de otro objeto.
- Cada objeto es responsable de mantener los invariantes de sus colecciones.
- Solo el dueño de la colección puede modificarla.
- Recordar que una colección puede cambiar luego de que la obtenga.

Librería/framework de colecciones

Todos los lenguajes OO ofrecen librerías de colecciones. Buscan abstracciones, interoperabilidad, performance, reúso y productividad.

Las colecciones admiten, generalmente, contenido heterogéneo en términos de clase, pero homogéneo en términos de comportamiento.

La librería de colecciones de java se organiza en términos de:

- **Interfaces:** representa la esencia de distintos tipos de colecciones.
- **Clases abstractas:** captura aspectos comunes de implementación.
- **Clases concretas:** implementaciones concretas de las interfaces.
- **Algoritmos útiles:** implementados como métodos estáticos.

Algunos tipos comunes de colecciones (interfaces)

- **List (java.util.List)**
  - Admite duplicados.
  - Sus elementos están indexados por enteros de 0 en adelante (índice).

- **Set (java.util.set)**
  - No admite duplicados.
  - Sus elementos no están indexados, ideal para chequear pertenencia.

- **Map (java.util.map)**
  - Asocia objetos que actúan como claves a otros que actúan como valores.

- **Queue (java.util.Queue)**
  - Maneja el orden en que se recuperan los objetos (LIFO, FIFO, por prioridad, etc.).

Generics y polimorfismo

Las colecciones admiten cualquier objeto en su contenido. Cuanto más sepa el compilador respecto al contenido de la colección, mejor podrá chequear lo que hacemos.

Contenido homogéneo da lugar a polimorfismo. Al definir y al instanciar una colección indico el tipo de su contenido.

Operaciones sobre colecciones

Siempre que tenemos colecciones repetimos las mismas operaciones:

- Ordenar respecto a algún criterio.
- Recorrer y hacer algo con todos sus elementos.
- Encontrar un elemento.
- Filtrar para quedarme solo con algunos elementos.
- Recolectar algo de todos los elementos.
- Reducir.

Nos interesa escribir código que sea independiente, tanto como sea posible, del tipo de colección que utilizamos.

Ordenando colecciones - Comparador: en Java, para ordenar nos valemos de un comparador. Los TreeSet usan un comparador para mantenerse ordenados. Para ordenar, por ejemplo, List, le enviamos el mensaje sort, con un comparador como parámetro.

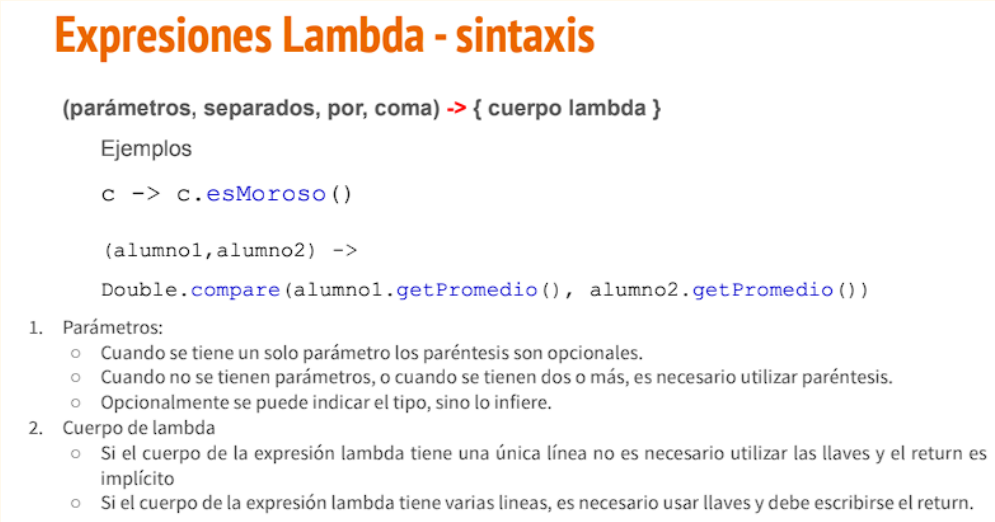
Recorrer colecciones: es una operación frecuente. El loop de control es un lugar más donde cometer errores. El código es repetitivo y queda atado a la estructura o tipo de la colección.

Iterador - iteración externa: todas las colecciones entienden el mensaje iterador(). Proporciona una manera de recorrer sobre los elementos de una colección de forma secuencial sin exponer su representación interna. Esto es útil para trabajar con diferentes tipos de colecciones de manera uniforme. El iterador es polimórfico.

Si es necesario hacer múltiples operaciones, se deben hacer una después de otra.

Streams

Expresiones Lambda: son métodos anónimos, no tienen nombre ni pertenecen a ninguna clase. Son útiles para parametrizar lo que otros objetos deben hacer, decirle a otros objetos que avisen cuando pase algo (callbacks).



Stream - iteración interna

Expresamos lo que queremos de una forma más abstracta y declarativa, es decir, un código más fácil de entender y mantener. Las operaciones se combinan para formar pipelines.

No almacena los datos, sino que provee acceso a una fuente de datos subyacente. Cada operación produce un resultado, pero no modifica la fuente. Potencialmente sin final.

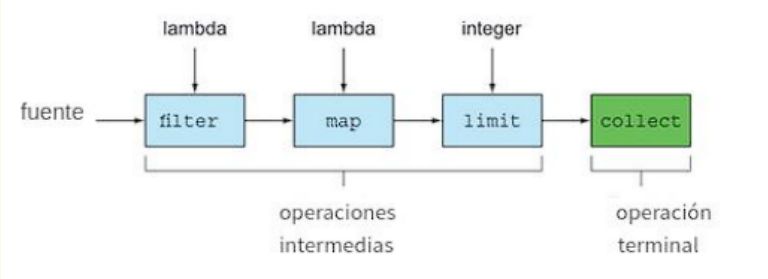
Consumibles: los elementos se procesan de forma secuencial y se descartan después de ser consumidos.

La forma más frecuente de obtenerlos es vía el mensaje stream() a una colección.

Pipelines

Para construir un pipeline se encadenan envíos de mensajes. Una fuente, de la que se obtienen los elementos. Cero o más operadores intermedios, que devuelven un nuevo stream. Operaciones terminales, que retornan un resultado.

La operación terminal guía el proceso. Las operaciones intermedias son Lazy: se calculan y procesan solo cuando es necesario, es decir, cuando se realiza una operación terminal que requiere el resultado.



Stream Pipelines - Algunos ejemplos

Operaciones intermedias	Operaciones terminales
filter de un stream obtengo otro con igual o menos elementos.	count | sum average
map de un stream obtengo otro con el mismo número de elementos pero eventualmente de distinto tipo	findAny | findFirst retorna Optional collect
limit de un stream obtengo otro de un numero especifico de elementos	anyMatch | allMatch | noneMatch retorna boolean
sorted de un stream obtengo otro ordenado	min | max retorna Optional

Optional

Se utiliza para representar un valor que podría estar presente o ausente en un resultado. Son una forma de manejar la posibilidad de valores nulos de manera más segura y explícita.

Algunos métodos en Streams, como **findFirst()** o **max()** devuelve un **Opcional** para representar un resultado.

Luego, se puede utilizar métodos de **Optional** como **ifPresent(), orElse(), orElseGet()**, entre otros, para manipular y obtener el valor de manera segura.

Operación intermedia: filter()

El mensaje filter retorna un nuevo stream que solo deja pasar los elementos que cumplen cierto predicado. El predicado es una expresión lambda que toma un elemento y resulta en true o false.

```
List<Alumno> ingresantesEnAnio2020 = alumnos.stream()
        .filter(alumno -> alumno.getAnioIngreso() == 2020)
        .collect(Collectors.toList());
```

Operación intermedia: map()

El mensaje map() nos da un stream que transforma cada elemento de entrada aplicando una función que indiquemos. La función de transformación recibe un elemento del stream y devuelve un objeto.

```
List<Factura> facturas = this.getFacturas();
Set<String> cuits = facturas.stream()
        .map(fact -> fact.getCuit())
        .collect(Collectors.toSet());
```

Operación intermedia: sorted()

Se usa para ordenar los elementos de la secuencia en orden específico. Se pueden usar para ordenar elementos en orden natural si son comparables o se debe proporcionar un comparador personalizado para especificar cómo se debe realizar la ordenación.

```
List<Alumno> alumnosOrdenados = alumnos.stream()
        .sorted((a1, a2)->
Double.compare(a1.getPromedio(), a2.getPromedio()))
        .collect(Collectors.toList());
```

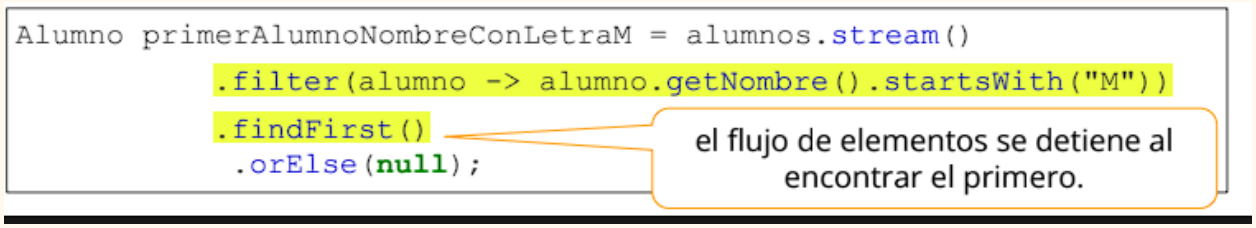
Operación terminal: collect()

El mensaje collect() es una operación terminal. Es un reductor que nos permite obtener un objeto o colección de objetos a partir de los elementos del stream. Recibe como parámetro el objeto Collector.

```
List<Factura> facturas = this.getFacturas();
long aConsumidorFinal = facturas.stream()
        .filter(fact -> fact.esConsumidorFinal())
        .collect(Collectors.counting());   // podria ser count()
```

Operación terminal: findFirst()

El mensaje findFirst() es una operación terminal. Devuelve un Opcional con el primer elemento del Stream si existe. Luego puedo usar: **orElse()** que devuelve el valor contenido en el Opcional si está presente. Si el Optional está vacío, entonces orElse() devuelve el valor predeterminado proporcionado como argumento.



Cuando no es necesario usar streams

Los lenguajes de programación proporcionan nuevas características que están disponibles para ser utilizadas. Es esencial estar dispuesto a explorar y aprender estas funcionalidades, saber cómo y cuándo aplicarlas de manera efectiva y reconocer cuando no son apropiadas.

Siempre que sea posible, voy a usar alguna construcción de más alto nivel. Son más concisas y están optimizadas y probadas.

Sin embargo, es importante reconocer que en algunos casos no es la elección óptima y es necesario considerar otras soluciones más adecuadas a la situación específica: quiero ordenar una colección y que mantenga un criterio de ordenamiento, quiero eliminar los elementos que cumplen cierta condición, cuando no requiero recorrer secuencialmente porque el algoritmo no lo requiere.