

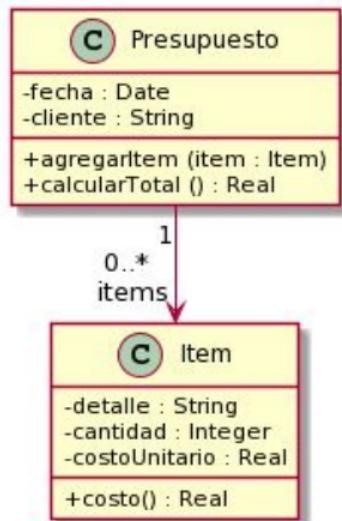


Apuntes semana 16/09

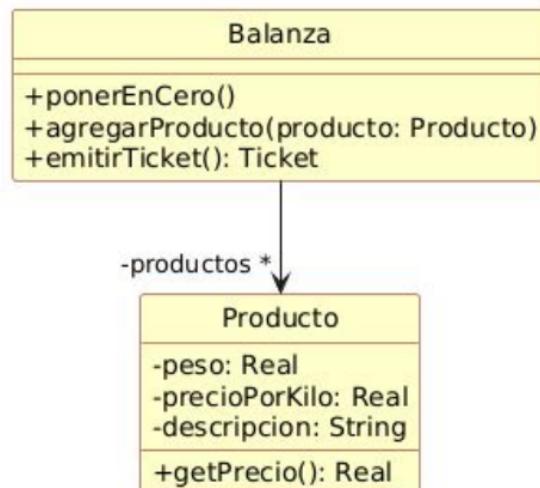
💡 Para entender Streams

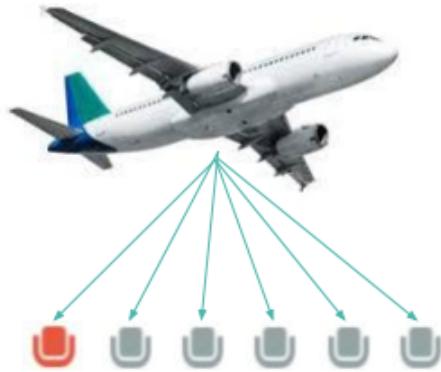
Ejercicios sobre colecciones

Ejercicio 3: Presupuestos

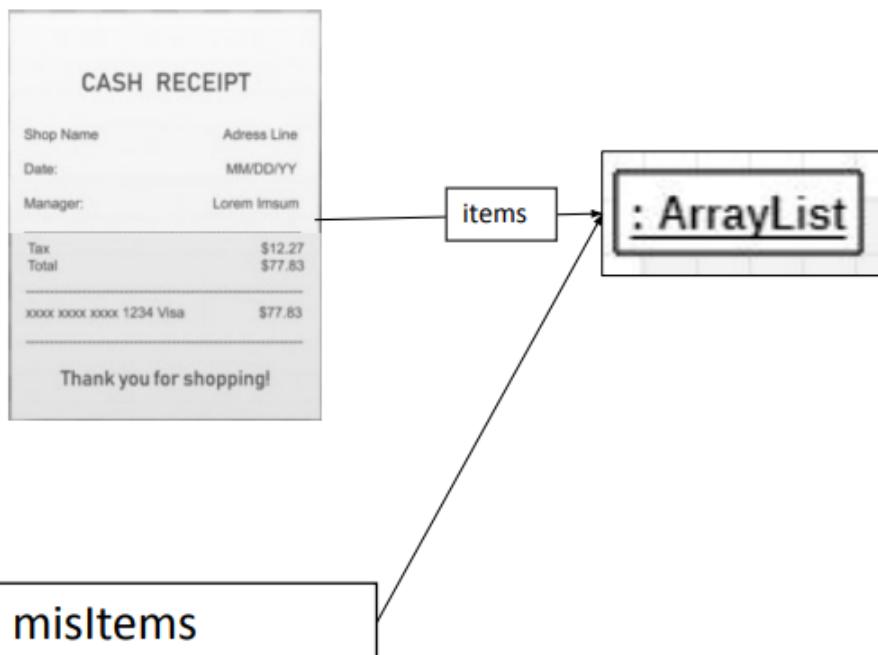


Ejercicio 4: Balanza mejorada





Colecciones como objetos



```
List<Item> misItems = ticket.getItems();
```

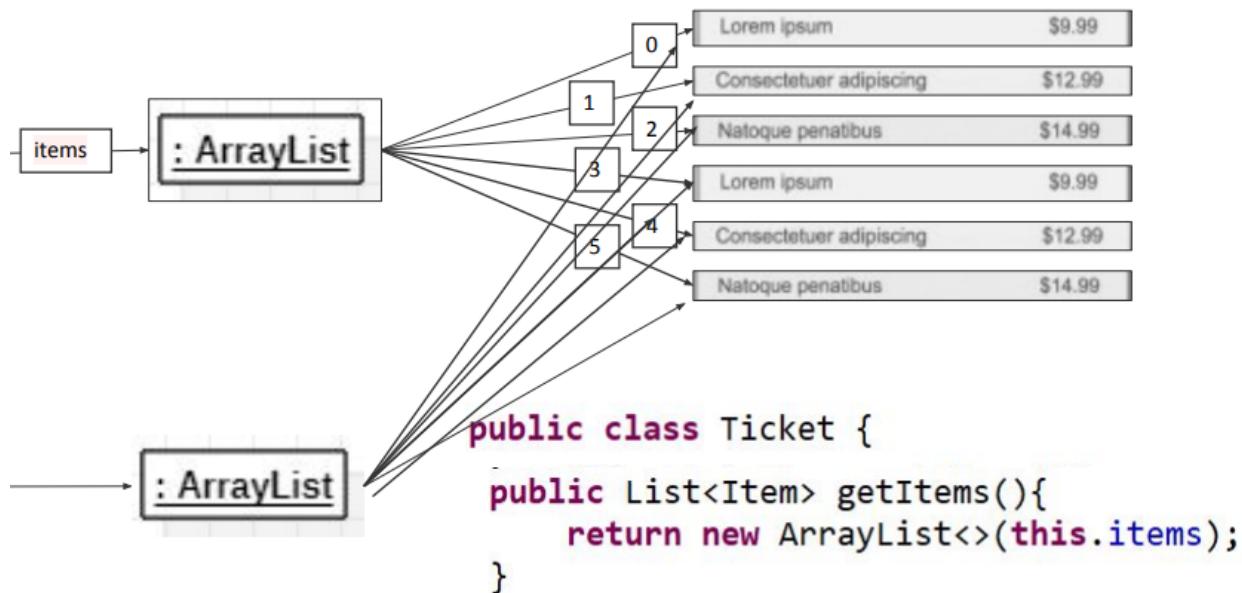
```
public class Ticket {
```

```

public List<Item> getItems() {
    return items;
}
}

```

“misItems” apunta a la misma lista que Items, lo cual significa que puede manipular y recorrer la lista como se le antoje, pero no se debe hacer



Precaución

- Nunca modifíco 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 obtengo



```
Ticket ticket = new Ticket(cliente);  
ticket.addItem(new Item(producto, 10));
```



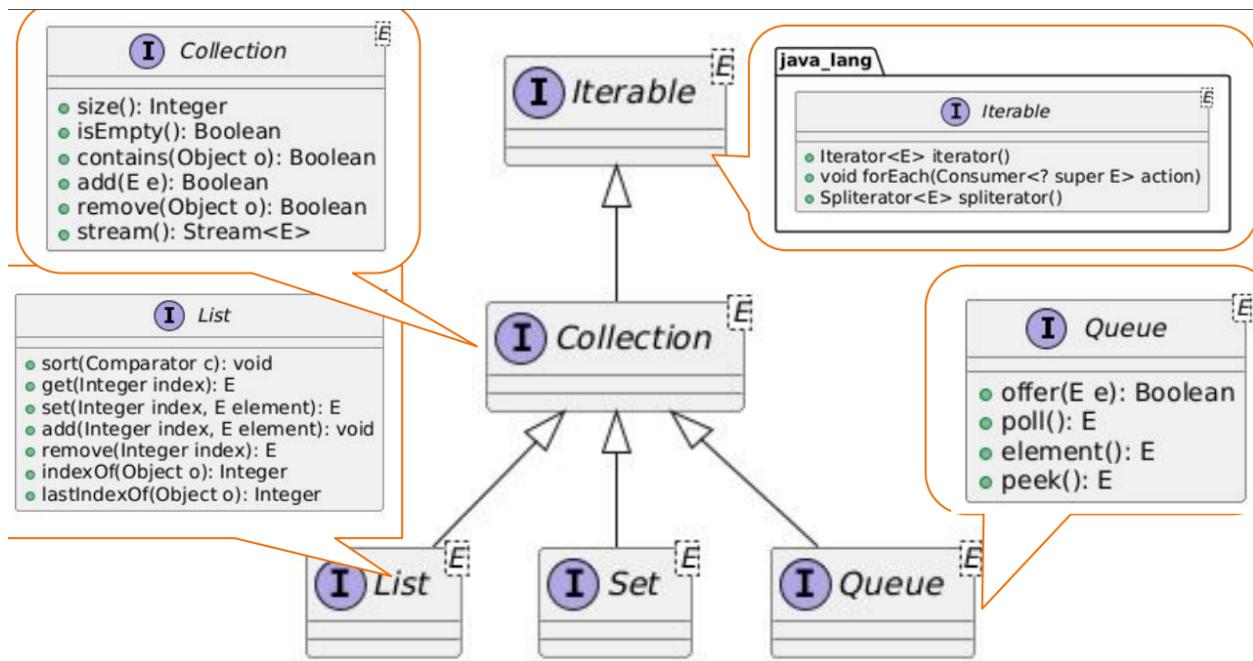
```
ticket.getItems().add(new Item(producto, 10));
```

Librería/framework de colecciones

- **Todos los lenguajes OO** ofrecen librerías de colecciones. Lo que buscan es abstracción, performance, reuso y productividad.
- Las **colecciones** permiten, generalmente, contenido heterogéneo en términos de clase, pero homogéneo en términos de comportamiento.
- La librería de colecciones de Java **se organiza en estos términos:**
 - Interfaces: representando la esencia de distintos tipos de colecciones
 - Clases abstractas: capturando aspectos comunes de implementación
 - Clases concretas: implementaciones concretas de las interfaces
 - Algoritmos útiles (implementados como métodos estáticos)

Tipos comunes de colecciones (Interfaces)

- List (java.util.List)
 - Admite duplicados y sus elementos están indexados por enteros de 0 en adelante.
- Set (java.util.Set)
 - No admite duplicados y sus elementos no están indexados, es 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.)

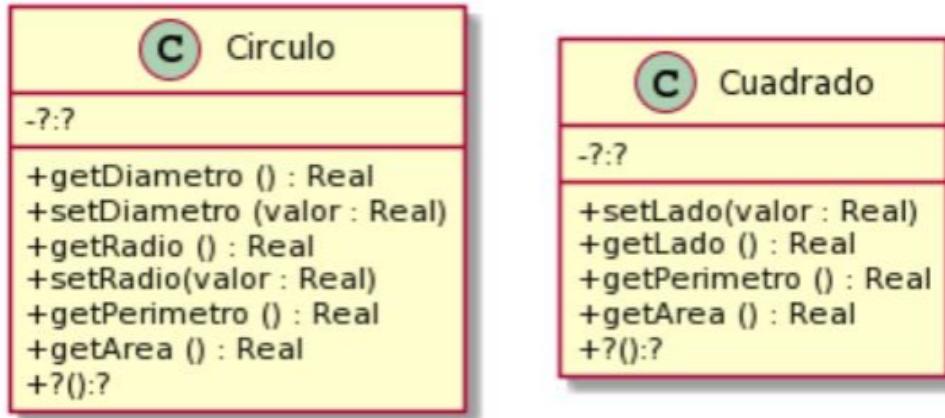


Generics y polimorfismo

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

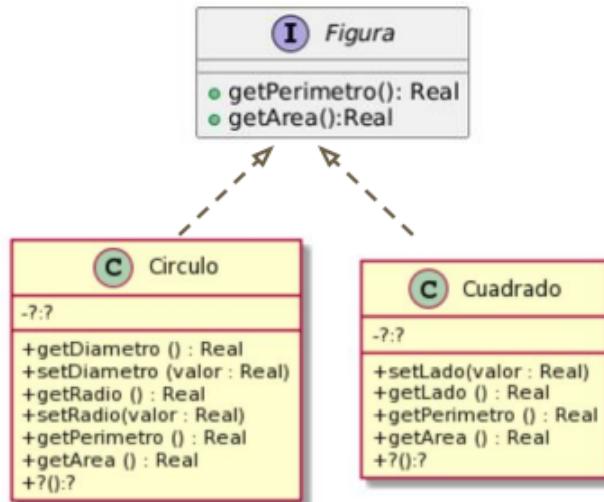
Contenido homogéneo da lugar al polimorfismo. Y al definir o instanciar una colección indico el tipo de su contenido.

ESTA MAL ESTO



```
ArrayList figuras = new ArrayList();
figuras.add(new Circulo());
figuras.add(new Cuadrado());
figura = figuras.get(0);
```

ESTA BIEN ESTO



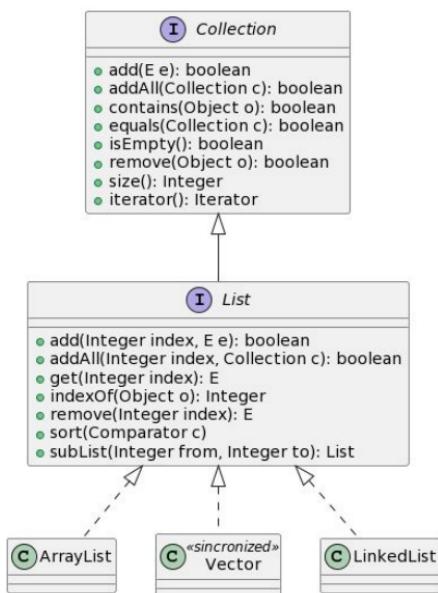
```

ArrayList<Figura> figuras = new ArrayList<Figura>();
figuras.add(new Circulo());
figuras.add(new Cuadrado());
Figura figura = figuras.get(0);

```



List



```

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

public class Presupuesto {
    private LocalDate fecha;
    private String nombreCliente;
    private List<Item> items;
}

public Presupuesto(String nombreCliente) {
    this.nombreCliente = nombreCliente;
    fecha = LocalDate.now();
    items = new ArrayList<>();
}

public void agregarItem(String detalle,
    int cantidad, double costoUnitario) {
    items.add(new Item(detalle, cantidad, costoUnitario));
}

public List<Item> getItems(){
    return items;
}

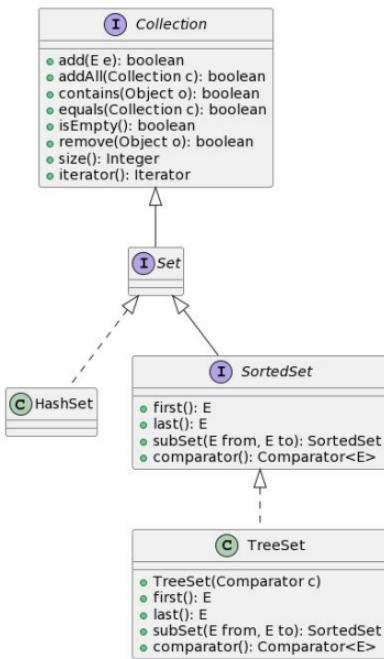
public int cantidadItems() {
    return items.size();
}

```

Utilizamos la interfaz para darle el tipo a la variable

Inferencia de tipos: no es necesario indicarlo nuevamente

Set



```

import java.util.HashSet;
import java.util.Set;

public class Grupo {

    private String nombre;
    private Set<Persona> miembros;

    public Grupo() {
        miembros = new HashSet<>();
    }

    public boolean agregarMiembro(Persona nuevo) {
        return miembros.add(nuevo);
    }

    public boolean esMiembro(Persona alguien) {
        return miembros.contains(alguien);
    }

    public int cantidad() {
        return miembros.size();
    }
}

```

Prestar atención cuando los objetos de la colección cambian, y ese cambio afecta al `equals()` y al `hashCode()`

Operaciones sobre colecciones

Operaciones frecuentes

- Siempre (o casi) que tenemos colecciones repetimos las mismas operaciones:
 - Ordenar respecto a algun 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 comparable

producto_1

nombre = "Harina"
precio = 100

producto_2

nombre = "Aceite"
precio = 500

```
public class Producto implements Comparable<Producto>{
    @Override
    public int compareTo(Producto o) {
        return this.nombre.compareTo(o.getNombre());
    }
}
```

Orden natural

Negativo → el primero es menor

0 → son iguales

Positivo → el primero es mayor

```
Collections.sort(productos);
```

Ordenando colecciones - Comparator

- En java, para ordenar nos valemos de un comparador.
- Los TreeSet usan un comparador para mantenerse ocupados
- Para ordenar List, le enviamos el mensaje sort, con un comparador como parámetro.

Recorriendo colecciones

- Recorrer colecciones es algo frecuente

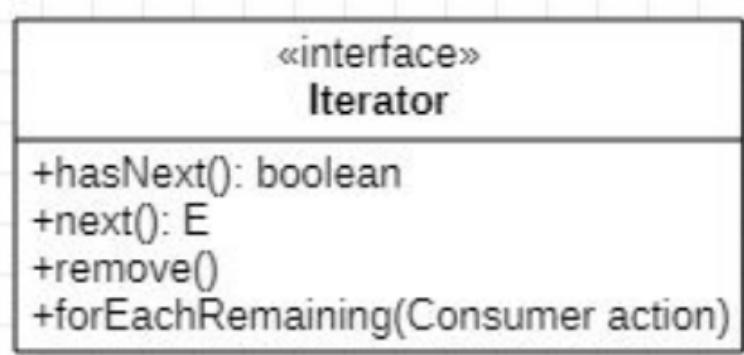
- El loop de control es el lugar donde mas se comete errores
- El código es repetitivo y queda atado a la estructura/tipo de la colección

```
for (int i=0; i < clientes.size(); i++ ) {  
    Cliente cli = clientes.get(i);  
    for (int j=0; j < productos.size(); j++ ) {  
        Producto prod = productos.get(j);  
        // hacer algo con los clientes y los productos  
    }  
}
```

Iterator (iterado externo)

- Todas las colecciones entienden iterator()
- Proporciona una manera de recorrer (o iterar) sobre los elementos de una colección de forma secuencial sin exponer su representación interna.
- Esto es útil para poder trabajar con diferentes tipos de colecciones (como listas, conjuntos y colas) de manera uniforme.
- Un iterador encapsula:
 - Como recorrer una colección particular
 - El estado de un recorrido
- No nos interesa la clase del iterador (son polimorficos)

```
for (Iterator<Cliente> it = clientes.iterator(); it.hasNext(); ) {  
    Cliente cli = it.next();  
    // hago algo con el cliente  
}
```



- **bucle for-each:** Ofrece una sintaxis mas limpia y menos propensa a errores. y nos permite modificar la colección. Este usa internamente un Iterator para iterar sobre los elementos de una colección.

```

for (TipoElemento elemento : colección) {
    // Operaciones con el elemento
}
    for (Cliente cli : clientes) {
        // hago algo con el cliente
}

```

Streams

Expresiones Lambda (clausuras / closures)

- Son métodos anónimos (no tienen nombre, no pertenecen a ninguna clase)
- Son útiles para:
 - Parametrizar lo que otros objetos deben hacer
 - Decirle a otros objetos que me avisen cuando pase algo (callbacks)

```
clientes.iterator().forEachRemaining(c -> c.pagarLasCuentas());  
  
clientes.forEach(c -> c.pagarLasCuentas());  
  
JButton button = new JButton("Click Me!");  
button.addActionListener(e -> this.handleButtonAction(e));
```

Expresiones Lambda - sintaxis



(parámetros, separados, por, coma) → { cuerpo lambda }

Ejemplos:

```
c -> c.esMoroso()  
  
(alumno1,alumno2) ->  
Double.compare(alumno1.getPromedio(), alumno2.getPromedio())
```

1. Parámetros:

- Cuando se tiene un solo parámetro los paréntesis son opcionales
- Cuando no se tienen parámetros, o cuando se tienen dos o mas, es necesario utilizar paréntesis.
- Opcionalmente se puede indicar el tipo, si no lo infiere.

2. Cuerpo del lambda

- Si el cuerpo de la expresión lambda tiene una única linea no es necesario utilizar las llaves y el return es implícito
- Si el cuerpo de la expresión lambda tiene varias líneas, es necesario usar llaves y debe escribirse el return.

```

//Calcular el total de deuda morosa
double deudaMorosa = 0;
for (Cliente cli : clientes) {
    if (cli.esMoroso()) {
        deudaMorosa += cli.getDeuda();
    }
}

//Generar facturas de pago para los deudores morosos
List<Factura> facturasMorosas = new ArrayList<>();
for (Cliente cli : clientes) {
    if (cli.esMoroso()) {
        facturasMorosas.add(this.facturarDeuda(cli));
    }
}

```

//Identificar el cliente moroso de mayor deuda
 Cliente deudorMayor;
 double deudaMayor = 0;
 for (Cliente cli : clientes) {
 if (cli.esMoroso()) {
 if (deudaMayor < cli.getDeuda()) {
 deudaMayor = cli.getDeuda();
 deudorMayor = cli;
 }
 }
}

- El iterador simplifica los recorridos pero ...

Iteración externa

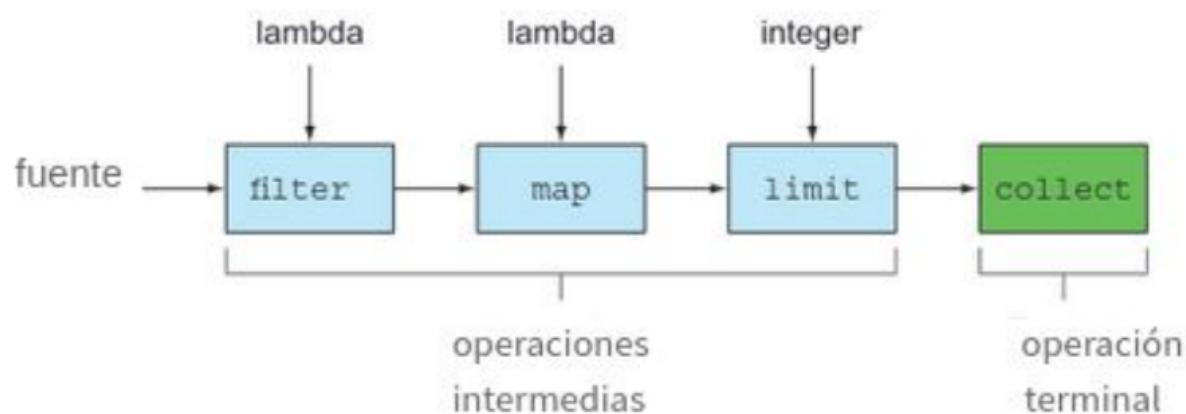
Stream

- Expresamos lo que queremos de una forma mas abstracta y declarativa → código mas fácil de entender y mantener
- Las operaciones se combinan para formar pipelines (tuberías)
- No almacenan los datos, sino que proveen acceso a una fuente de datos subyacente (colección, canal, etc)
- 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 mas frecuente de obtenerlos es via el mensaje stream() a una colección

Stream Pipelines

- Para construir una pipeline se encadenan envíos a mensajes
 - Una fuente, de la que se obtienen los elementos
 - Cero o mas operaciones intermedias, que devuelven un nuevo stream
 - Operaciones terminales, que retornan un resultado

- 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
limit de un stream obtengo otro de un numero especifico de elementos	collect anyMatch allMatch noneMatch retorna boolean
sorted de un stream obtengo otro ordenado	min max retorna Optional

Optional

- Se utiliza para poder representar un valor que podría estar presente o ausente en un resultado
- Son una forma de manejar la posibilidad de valores nulos de manera mas segura y explicita
- Algunos métodos en Streams como findFirst() o max() devuelven un Optional para representar el resultado.
- Luego, se puede utilizar métodos de Optional como ifPresent() or Else(), orElseGet(), entre otros para poder 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 or 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 (de mapeo) 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 un orden específico.
- Se puede 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 un objeto Collector
 - Podemos programar uno, pero solemos utilizar los que fabrica Collectors (Collectors.toList(), Collectors.counting(), ..)

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

Operación terminal: findFirst()

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

```
Alumno primerAlumnoNombreConLetraM = alumnos.stream()
    .filter(alumno -> alumno.getNombre().startsWith("M"))
    .findFirst()
    .orElse(null);
```

el flujo de elementos se detiene al encontrar el primero.

Filtrar, colectar, reducir, encontrar ...

```
//Calcular el total de deuda morosa
double deudaMorosa = 0;
for (Cliente cli : clientes) {
    if (cli.esMoroso()) {
        deudaMorosa += cli.getDeuda();
    }
}

//Generar facturas de pago para los deudores morosos
List<Factura> facturasMorosas = new ArrayList<>();
for (Cliente cli : clientes) {
    if (cli.esMoroso()) {
        facturasMorosas.add(this.facturarDeuda(cli));
    }
}

//Identificar el cliente moroso de mayor deuda
Cliente deudorMayor;
double deudaMayor = 0;
for (Cliente cli : clientes) {
    if (cli.esMoroso()) {
        if (deudaMayor < cli.getDeuda()) {
            deudaMayor = cli.getDeuda();
            deudorMayor = cli;
        }
    }
}
```

Usando iteradores externos

```
//Calcular el total de deuda morosa
double deudaMorosa = clientes.stream()
    .filter(cli -> cli.esMoroso())
    .mapToDouble(cli -> cli.getDeuda())
    .sum();
```

```
//Generar facturas de pago para los deudores morosos
List<Factura> facturasMorosas = clientes.stream()
    .filter(cli -> cli.esMoroso())
    .map(cli -> this.facturarDeuda(cli))
    .collect(Collectors.toList());
```

```
//Identificar el cliente moroso de mayor deuda
Cliente deudorMayor = clientes.stream()
    .filter(cli -> cli.esMoroso())
    .max(Comparator.comparing(cli -> cli.getDeuda()))
    .orElse( other: null);
```

Usando Streams

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 y saber cuando y como aplicarlas de manera efectiva. También reconocer cuando no son apropiadas.

Siempre que me sea posible voy a usar alguna construcción de más alto nivel, ya que 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 se mantenga ordenada por un criterio
- quiero eliminar los elementos que cumplen una condición
- cuando no requiero recorrer secuencialmente porque el algoritmo no lo requiere

UML (Lenguaje Unificado de Modelado)



UML (Lenguaje Unificado de Modelado) es un estándar para modelar sistemas de software.

- **Diagramas de clases:** Representan clases, atributos, métodos y relaciones entre ellas.
- **Diagramas de secuencia:** Muestran la interacción entre objetos en el tiempo.
- **Diagramas de casos de uso:** Describen las interacciones entre actores y el sistema.
- **Correspondencia UML y lenguajes de programación:** Los elementos UML se traducen a conceptos de lenguajes de programación como clases y relaciones.
- **Editores gráficos vs. Editores UML:** Los gráficos son visuales generales; los editores UML siguen estrictamente las reglas del lenguaje.

UML es un lenguaje de modelado visual que permite:

- **Especificar, visualizar, construir y documentar** artefactos de un sistema de software.
- Captura decisiones y conocimientos relacionados con el diseño del sistema.

Historia UML

Grady Booch:

- Método Booch

Jim Rumbaugh

- Object-Modeling Technique (OMT) de Rumbaugh

Ivar Jacobson

- OOSE (Object-Oriented Software Engineering)

OMG

Object Management Group es una organización internacional sin fines de lucro que se dedica a la estandarización y promoción de estándares de tecnología.

¿Qué puedo hacer con UML?

Lenguaje UML

Es un lenguaje de modelado visual utilizado para especificar, visualizar, construir y documentar artefactos de un sistema de software, permitiendo capturar decisiones y conocimientos.

Diagramas de estructura

- Diagrama de Clases
- Diagrama de Paquetes
- Diagrama de Componentes
- Diagrama de Objetos
- Diagrama de Despliegue

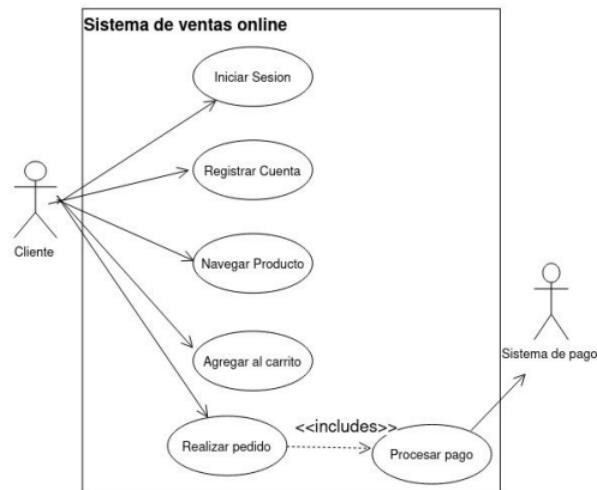
Diagramas de comportamiento

- Diagrama de Casos de Uso

- Diagramas de Interacción
 - Diagrama de Secuencia
 - Diagrama de Colaboración
- Diagrama de Máquinas de estado
- Diagrama de Actividades

Diagramas de Casos de Uso

Un caso de uso es una representación del comportamiento de un sistema desde la perspectiva de un usuario externo. 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” incluye a personas y otros sistemas que interactúan con el sistema.
- Elementos de un modelo de casos de uso:
 - Actores
 - Casos de uso
 - Relaciones
- **Actor:** Representa a un usuario externo, un sistema o una entidad que interactúa con el sistema modelado.
- **Caso de uso:** Representación de una funcionalidad específica.
- **Relaciones entre casos de uso:**

- **Extensión** (`<<extends>>`)
- **Inclusión** (`<<includes>>`)

Diagramas de Casos de Uso - Relaciones:

- Un caso de uso **incluye** a otro si el primero incorpora el comportamiento especificado en el segundo.



- Un caso de uso **extiende** a otro si puede ser mejorado o ampliado con funcionalidad adicional en ciertos escenarios, pero no siempre se aplica.

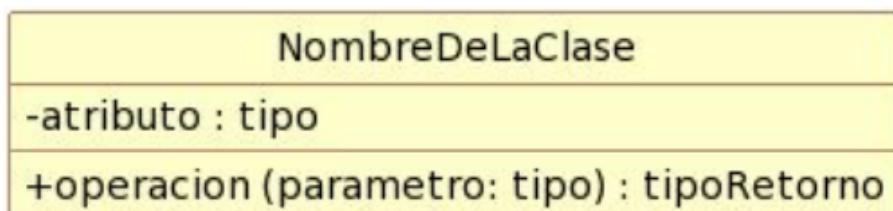


Diagramas de Casos de Uso - conversación

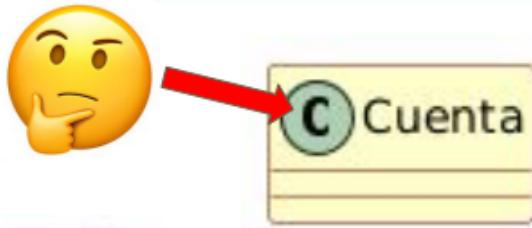
Comprar producto			
1-	Se indica el producto que se quiere comprar		
		2-	Se agrega el producto al carrito de compras
3.	Se procede a terminar la compra		
		4-	Se calcula el total
		5-	Se muestran las opciones de pago
6	Se elige una opción de pago		
		7-	Se crea una orden de compra, registrando el pago

Diagrama de clases

Una clase es una descripción de un conjunto de objetos que comparten atributos, operaciones, métodos, relaciones y semántica. Gráficamente, se representa con cajas divididas en tres compartimientos.



- Nombre de la clase
 - singular
 - debe comenzar con Mayúscula,
 - estilo CamelCase



Si la clase es abstracta

- cursiva
- estereotipo <>abstract>

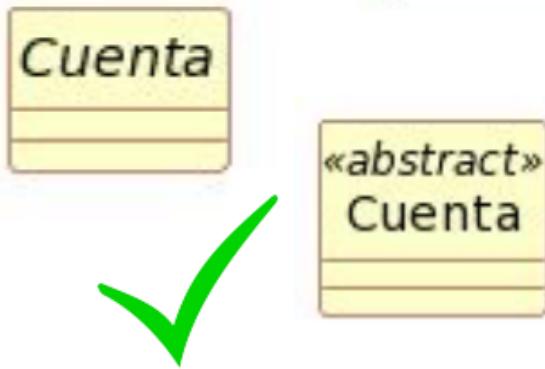
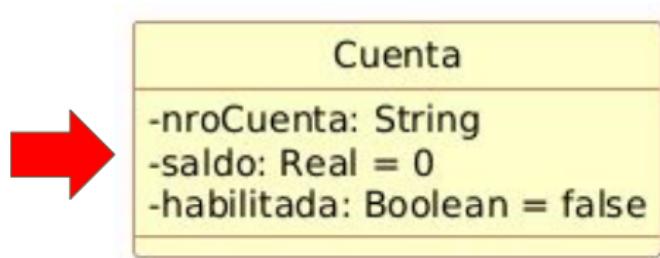


Diagrama de Clases - Atributos y Operaciones

- visibilidad
 - privada (-)
 - protegida (#)
 - pública (+)
 - paquete (~)
- nombre
 - estilo camelCase
 - comienza con minúscula
- tipo



- tipos UML: Integer, Real, Boolean, String
- valor por defecto
- Parametros
 - nombre: estilo camelCase
- Tipo de retorno
 - Si no retorna nada, no se especifica
 - Si retorna un objeto, se indica de que clase
 - Si retorna una colección, se indica el nombre de la clase [*]
- Si el método es abstracto
 - cursiva
 - con estereotipo <>abstract<>
- Si el método es un constructor
 - con estereotipo <>create<>

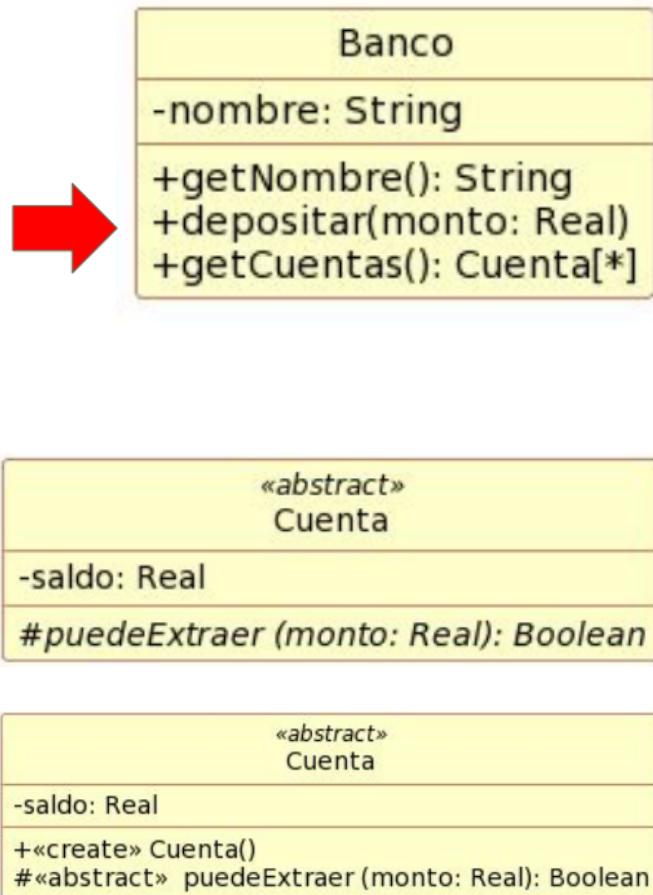
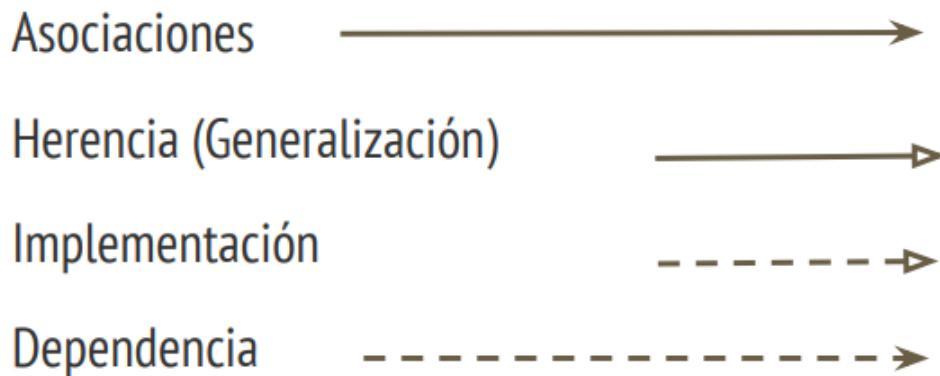
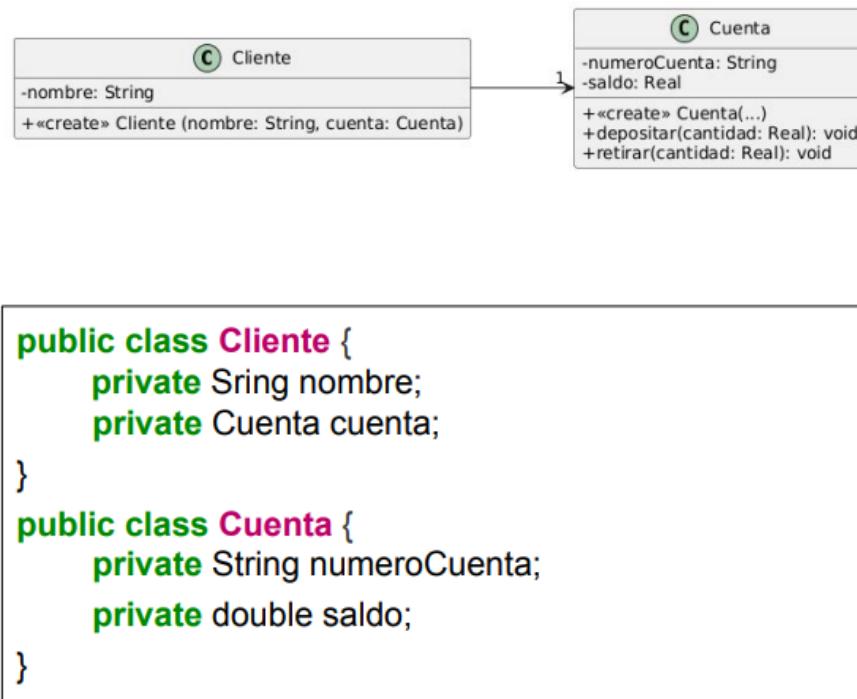


Diagrama de clases - Relaciones

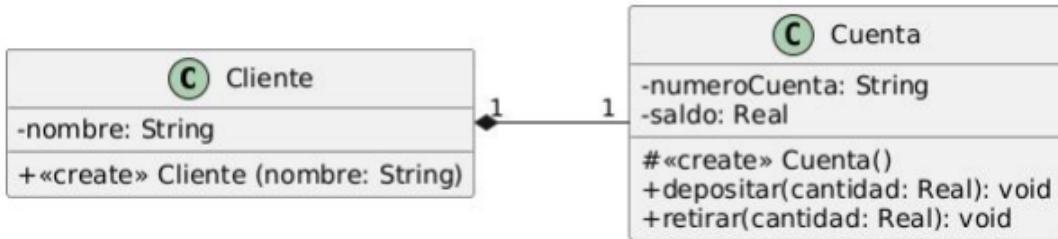


Asociación

- navegabilidad
- multiplicidad
- nombre de rol
- tipo:
 - simple
 - agregación
 - composición



Otro caso / Navegabilidad



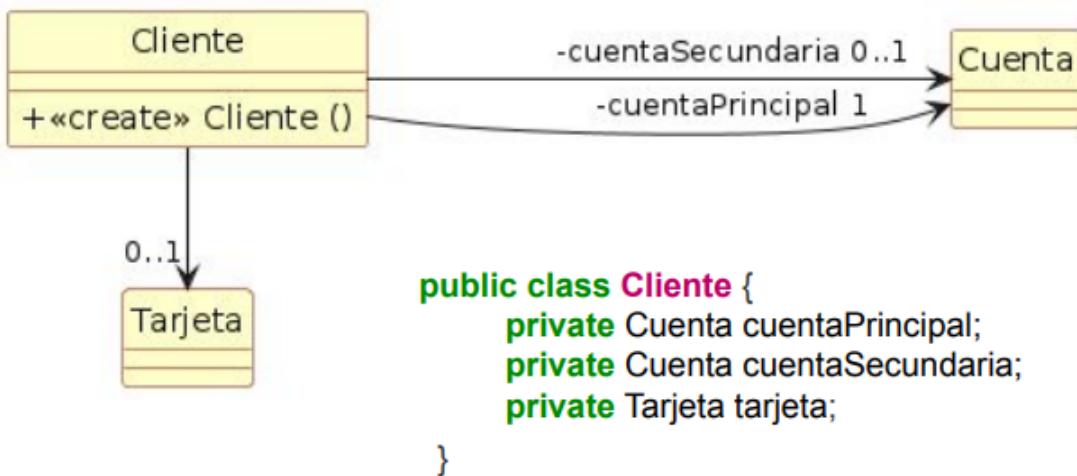
```

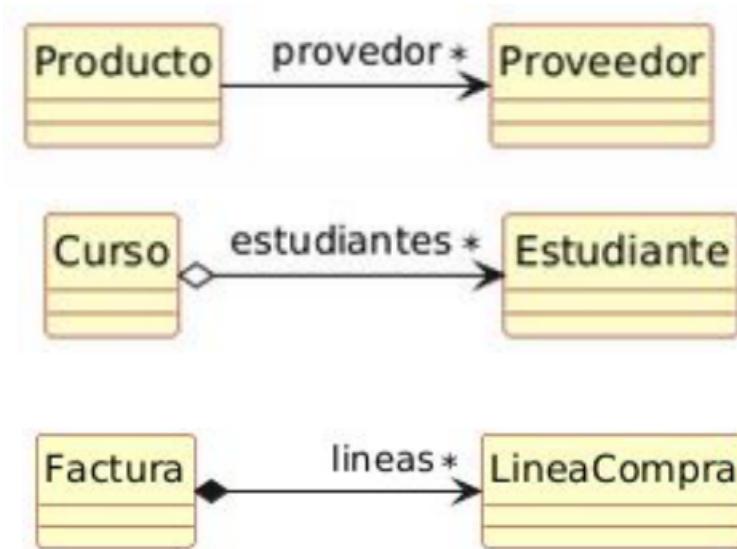
public class Cliente {
    private String nombre;
    private Cuenta cuenta;
}

public class Cuenta {
    private String numeroCuenta;
    private double saldo;
    private Cliente cliente;
}

```

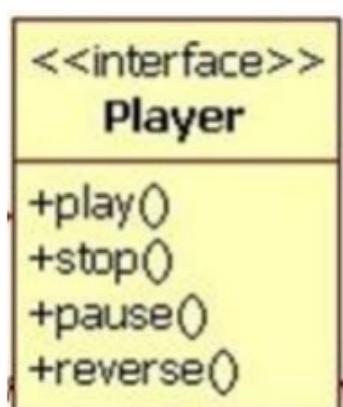
la notación "0..1" se utiliza para representar una **multiplicidad** en una asociación entre dos clases. En este caso, indica que un objeto de la clase "Cliente" puede estar asociado con **cero o un** objeto de la clase "Cuenta".





El asterisco (*) indica que un objeto de la clase en el origen de la relación puede estar asociado con **cero o más** objetos de la clase en el destino. En otras palabras, la multiplicidad es de **muchos**.

Diagrama de Clases - Interfaces



Una interfaz define un conjunto de operaciones que una clase debe implementar. En UML, se representa con un rectángulo y el estereotipo <<interface>>. Las clases que implementan la interfaz son responsables de definir sus métodos.

Diagrama de Clases - Implementacion y herencia

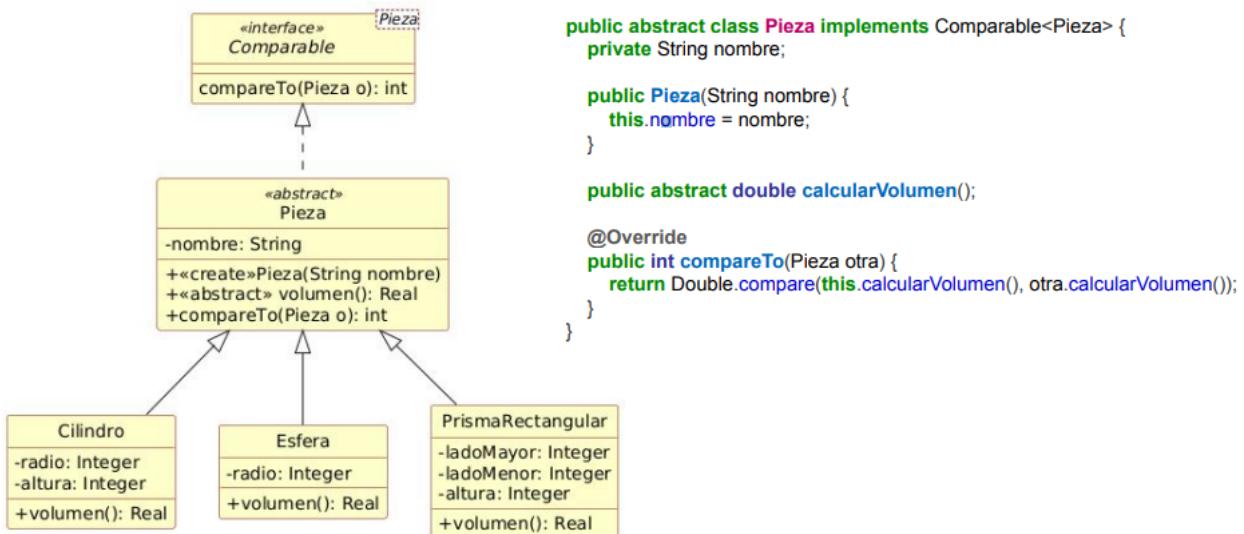
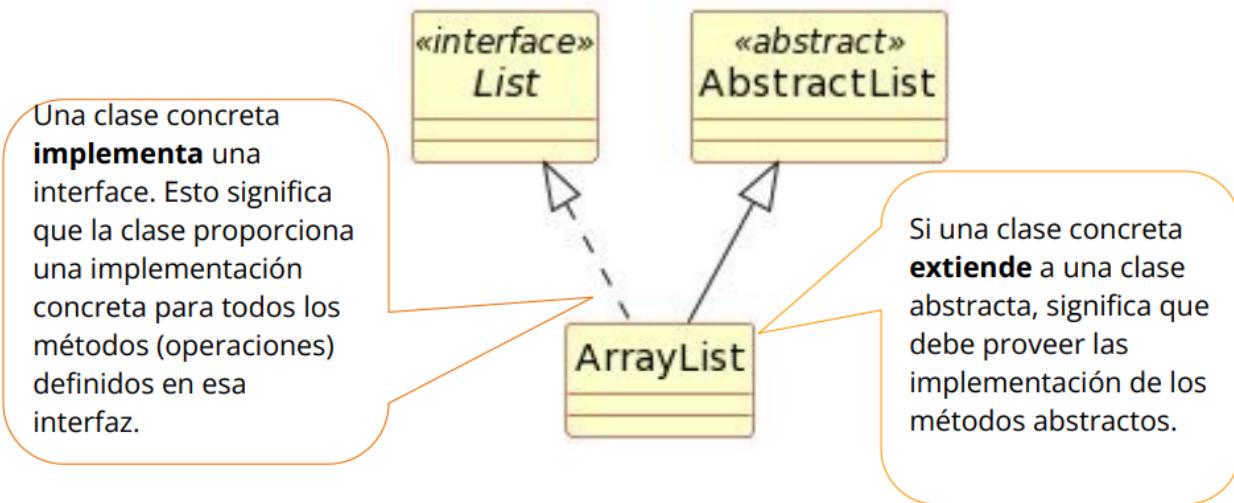
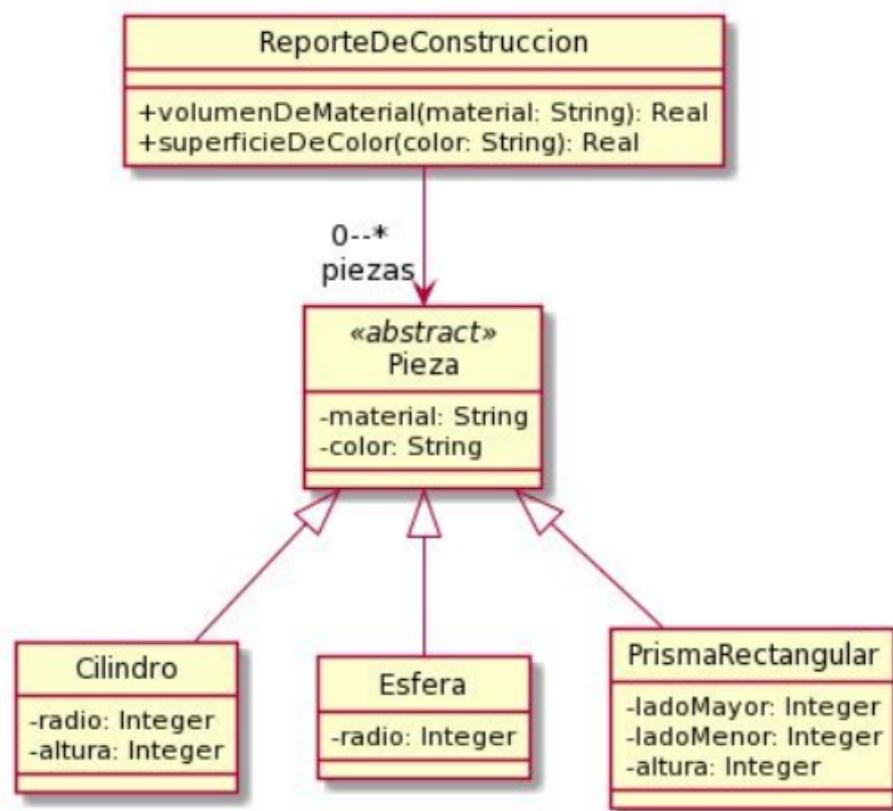
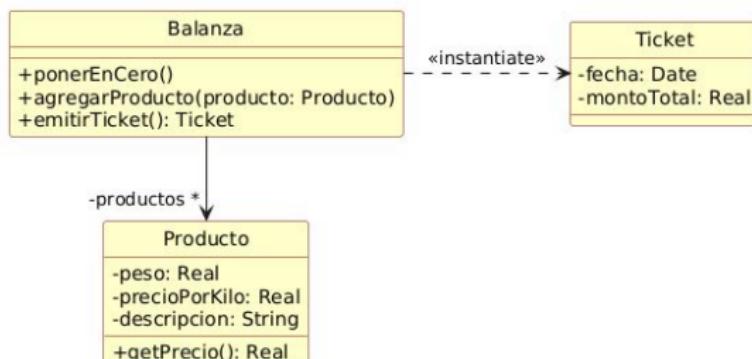


Diagrama de Clases - Dependencia

Un cambio en la especificación de una clase puede afectar a otra que depende de ella.

Esta relación muestra que una clase utiliza o necesita otra para funcionar, sin mantener una referencia permanente.



"0--*" significa que puede haber **cero o más** instancias de la clase **Pieza** asociadas a una instancia de la clase **ReporteDeConstruccion**. Esto

describe una relación en la que un reporte puede tener ninguna, una o muchas piezas asociadas.

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

Los diagramas de paquetes agrupan clases y muestran la organización del sistema. Se busca alta cohesión dentro de un paquete y bajo acoplamiento entre ellos, exportando e importando solo lo necesario.

Diagrama de paquetes

<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-package-diagram/>

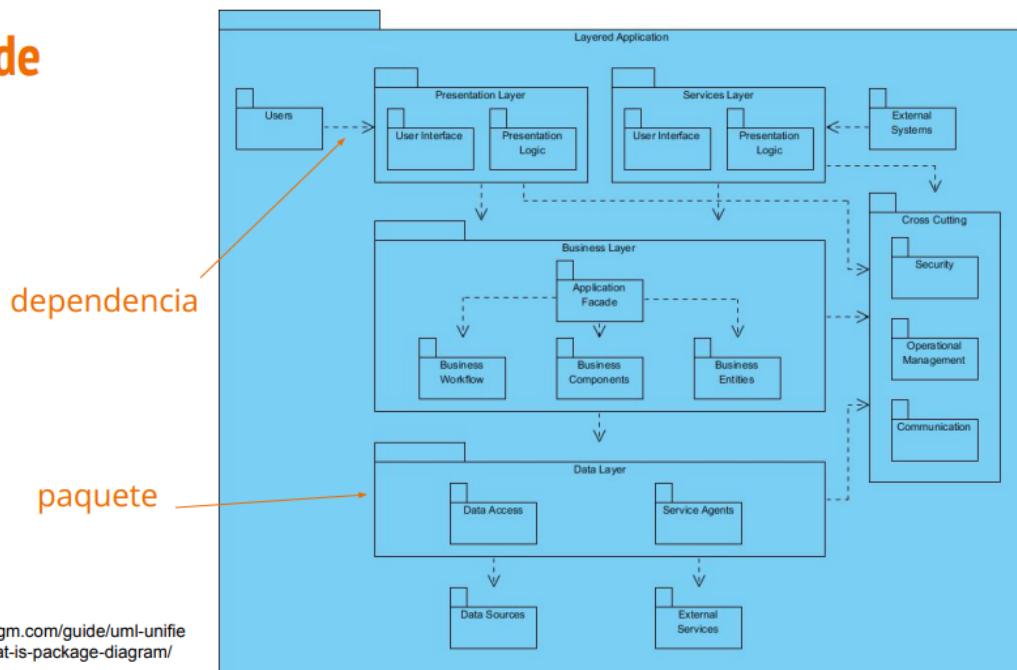
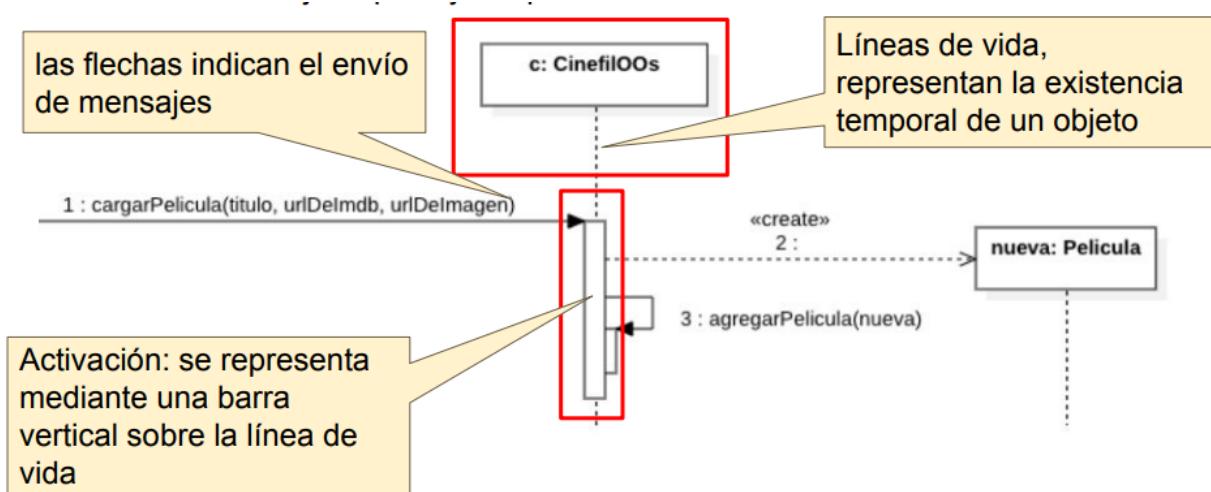


Diagrama de secuencia

Un diagrama de secuencia muestra cómo interactúan los objetos en un sistema y en qué orden. Los objetos se representan en la parte superior, y el tiempo avanza de arriba hacia abajo.

Las flechas horizontales muestran las interacciones entre los objetos, indicando quiéenvía un mensaje a quién y en qué orden.



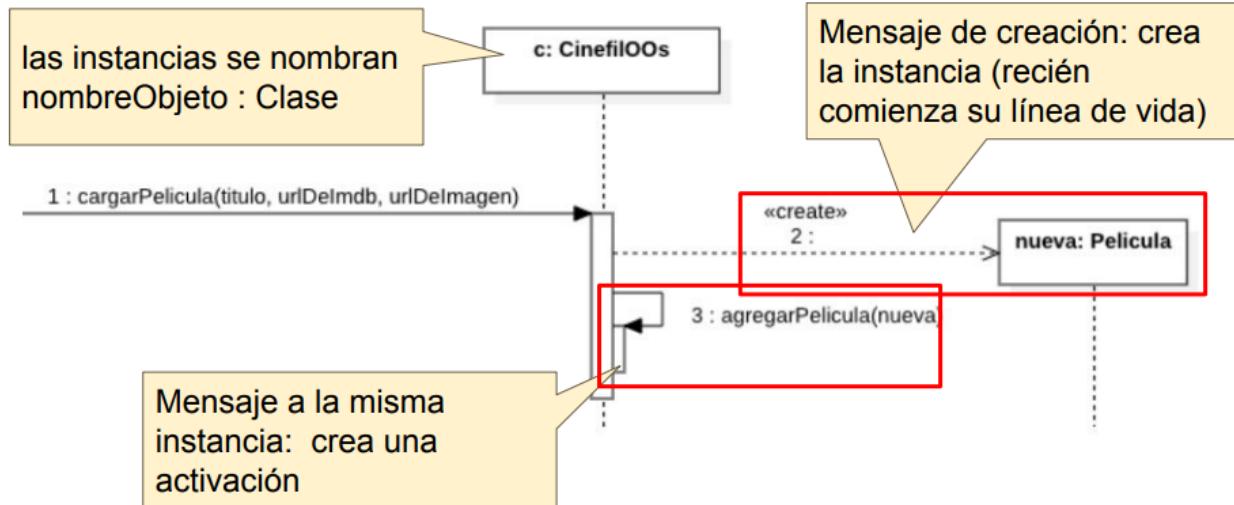


Diagrama de secuencia

Sintaxis del mensaje:

```
[atributo:= ] nombre del mensaje (parámetros) [:valor de retorno]
```

Ejemplos:

```
agregarPelicula (nueva)
```

```
hayMonto:= hayMontoSuficienteParaExtraer(monto)
```

Diagrama de secuencia - CombinedFragment

Un fragmento combinado en un diagrama de secuencia representa lógica y condiciones en la interacción entre objetos. Permite especificar bloques opcionales, alternativos y de repetición, entre otros. Los fragmentos más comunes son:

- **opt**: opcional
- **alt**: alternativa

- **loop**: bucle
- **ref**

Fragmento	Significado
alt	fragmento alternativo: tiene varias condiciones. Solo se ejecuta el fragmento cuya condición es verdadera.
opt	fragmento opcional: tiene un solo camino. Se ejecuta si esa condición es verdadera. Caso particular de alt.
loop	Bucle: el fragmento puede ejecutarse varias veces, y la condición indica la base de la iteración.
ref	Referencia: se refiere a una interacción definida en otro diagrama. El marco abarca las líneas de vida involucradas en la interacción.

Como se define UML?

UML se define como un lenguaje de modelado que estandariza la visualización, especificación y construcción de sistemas. El **meta modelo de UML** es una especificación que define los elementos y construcciones básicas para crear diagramas en UML, describiendo formalmente cómo se estructuran y relacionan esos elementos.

Herramientas de modelado

Las herramientas de modelado UML pueden ser gráficas o basadas en el metamodelo. Las herramientas gráficas permiten dibujar elementos con la imagen de UML, mientras que las basadas en el meta modelo se enfocan en la manipulación directa de los elementos del meta modelo UML.



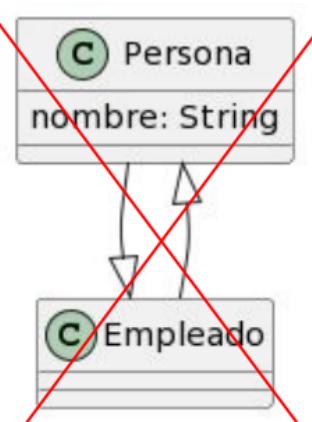
PlantUML

Herramienta que pasa de texto a imagen
no tiene chequeo del metamodelo.

Permite dibujar cosas ajenas a UML, o con errores semánticos

Versión online

En UML no es posible una jerarquía en ciclo.



Videos para refrescar la lectura

https://www.youtube.com/watch?v=E7s4_yLCyhs

<https://www.youtube.com/watch?v=fm4AfxJLnX4>

<https://www.youtube.com/watch?v=5e0QqZay6NM>