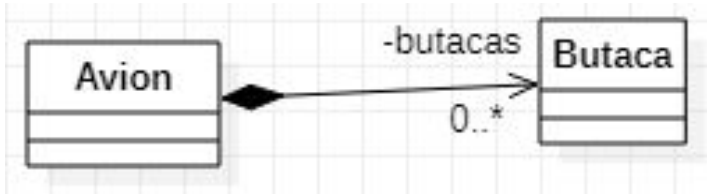
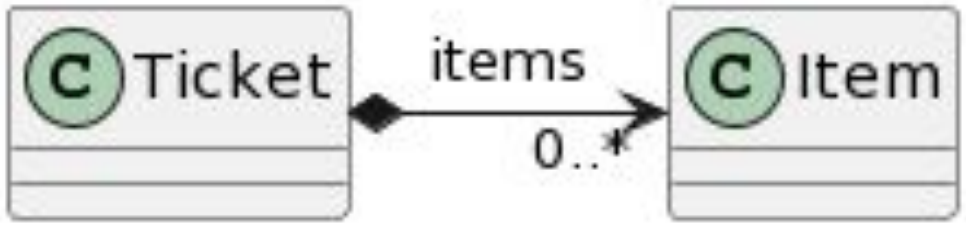
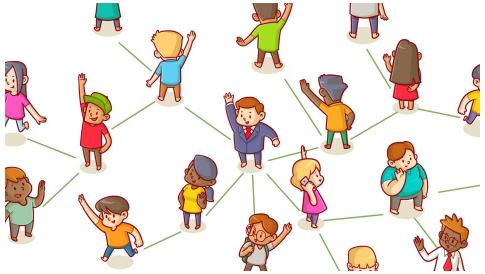
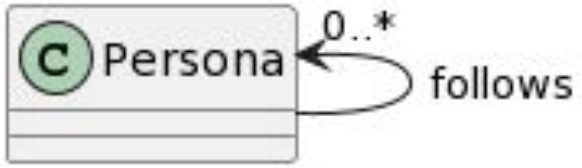
Colecciones

Relaciones 1 a muchos

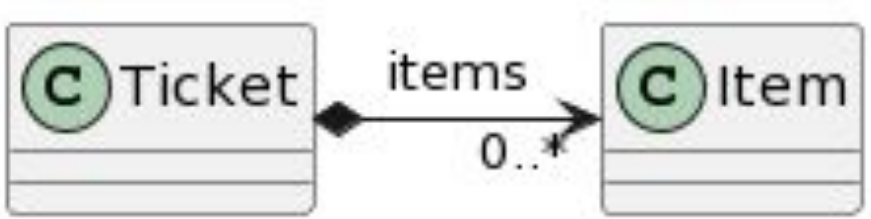


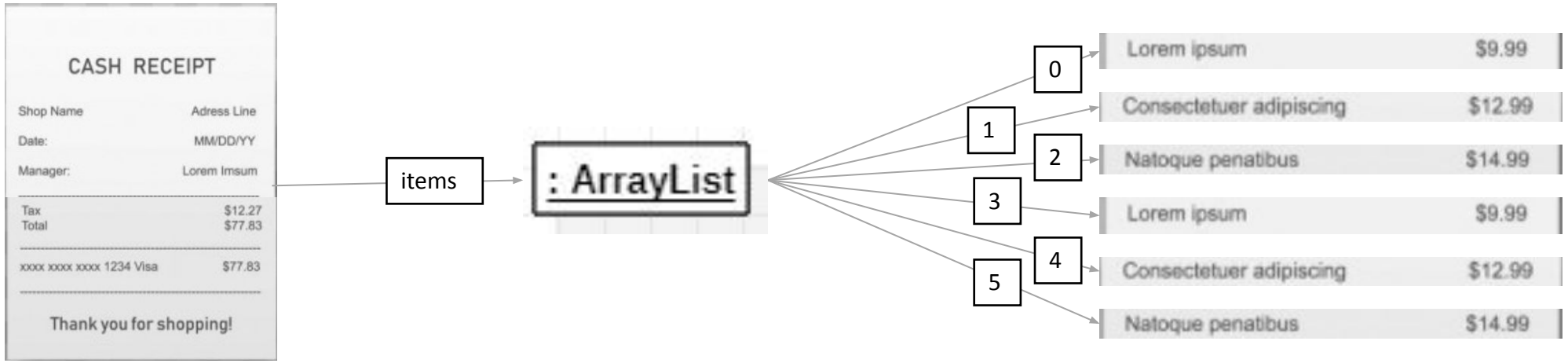


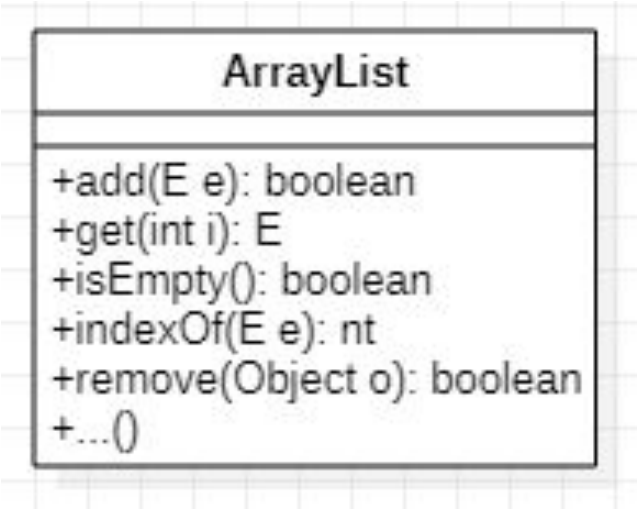




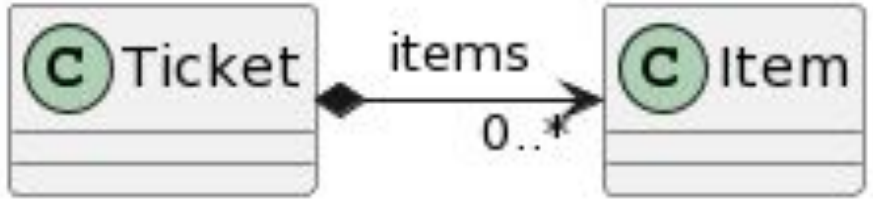
Colecciones como objetos

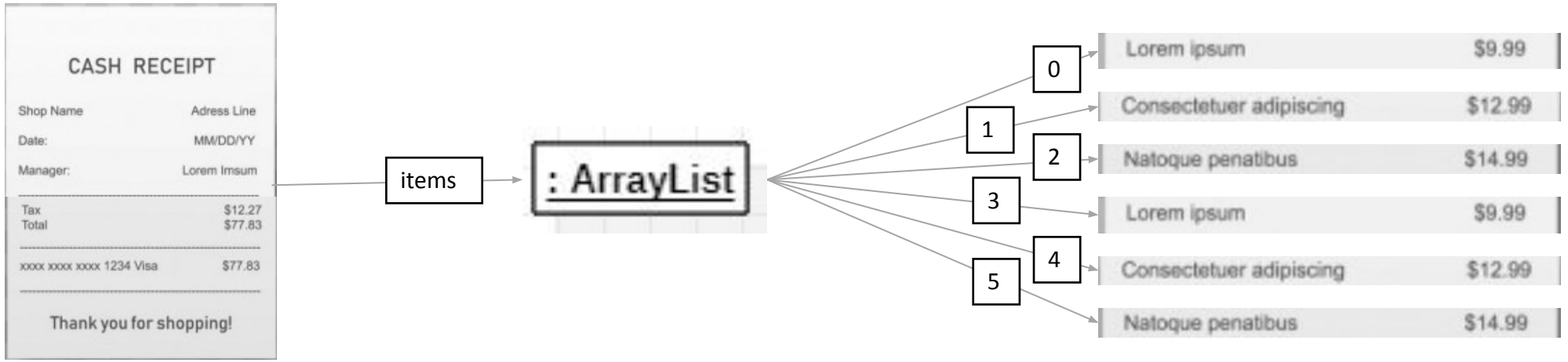


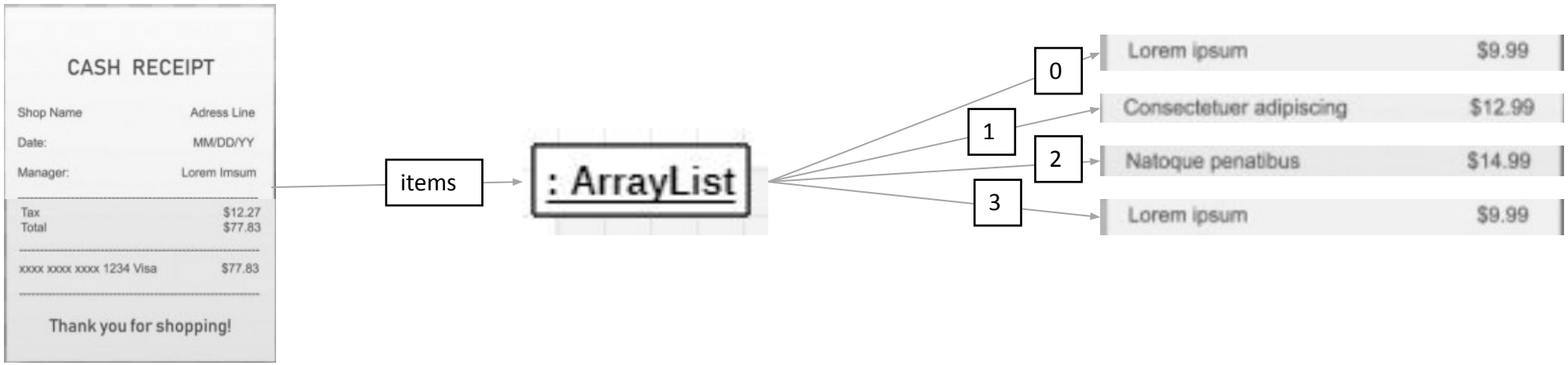




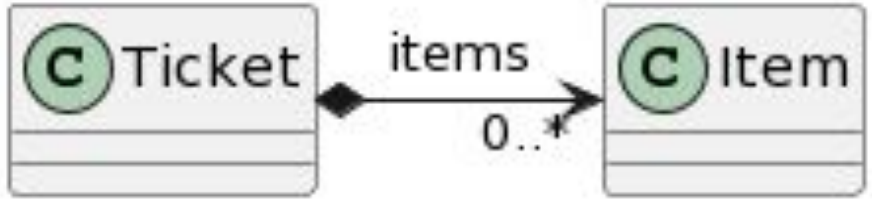
Colecciones como objetos

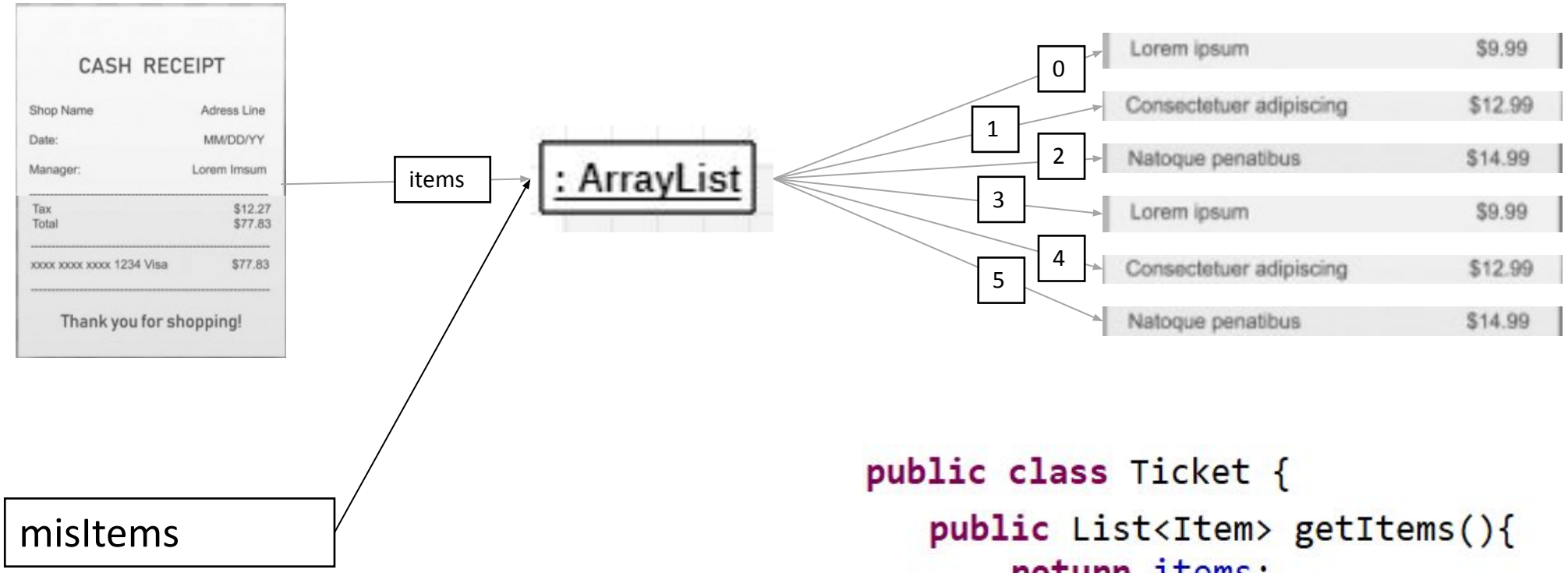






Colecciones como objetos

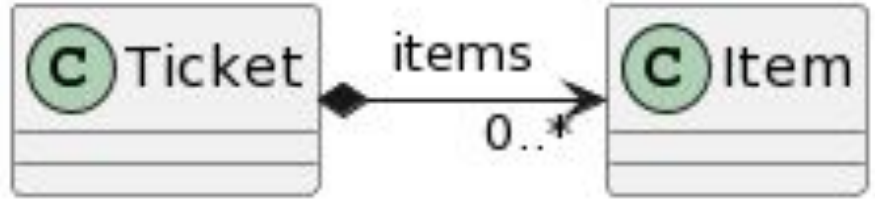


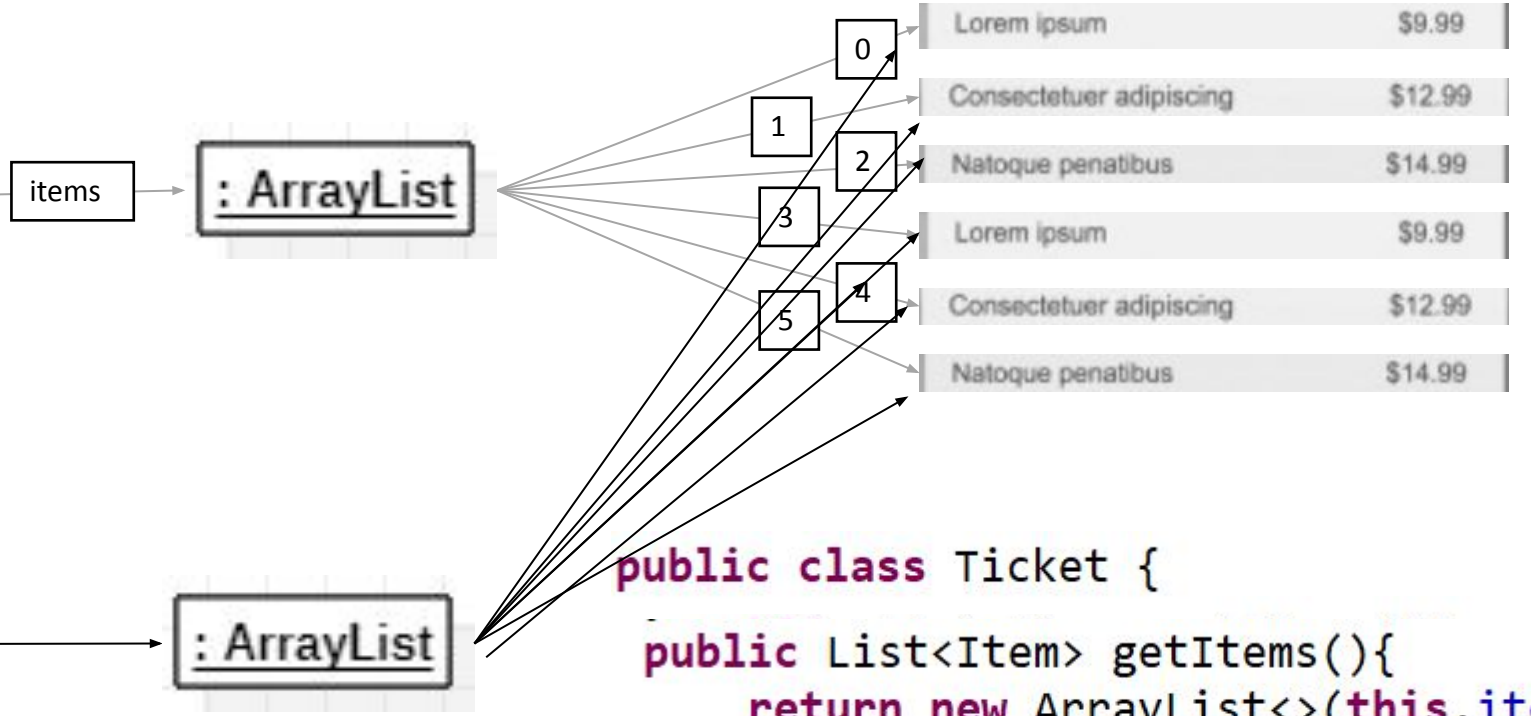



```
public class Ticket {
    public List<Item> getItems(){
        return items;
    }
```

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

Colecciones como objetos






```java
public class Ticket {

    public List<Item> getItems(){
        return new ArrayList<>(this.items);
    }
```

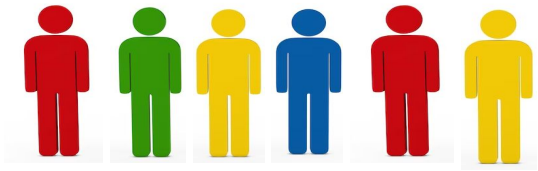
```java
List<Item> misItems = ticket.getItems();
```

Librería/framework de colecciones

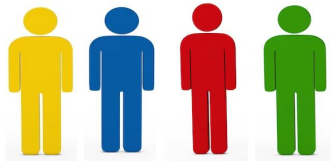
- Todos los lenguajes OO ofrecen librerías de colecciones
  - Buscan abstracción, interoperabilidad, performance, reuso, productividad
- Las colecciones admiten, generalmente, contenido heterogéneo en términos de clase, pero homogéneo en términos de comportamiento
- La librería de colecciones de Java se organiza en términos de:
  - Interfaces: representa la esencia de distintos tipos de colecciones
  - Clases abstractas: capturan aspectos comunes de implementación
  - Clases concretas: implementaciones concretas de las interfaces
  - Algoritmos útiles (implementados como métodos estáticos)

Algunos tipos populares (intefaces)

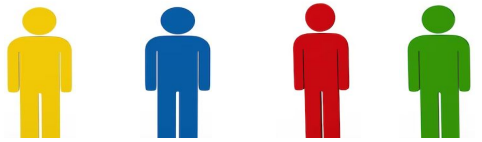
- List (java.util.List)



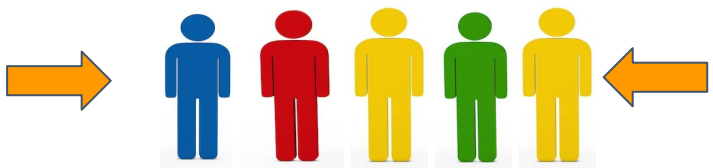
  - Admite duplicados
  - Sus elementos se están indexados por enteros de 0 en adelante (su posición)
- Set (java.util.Set)



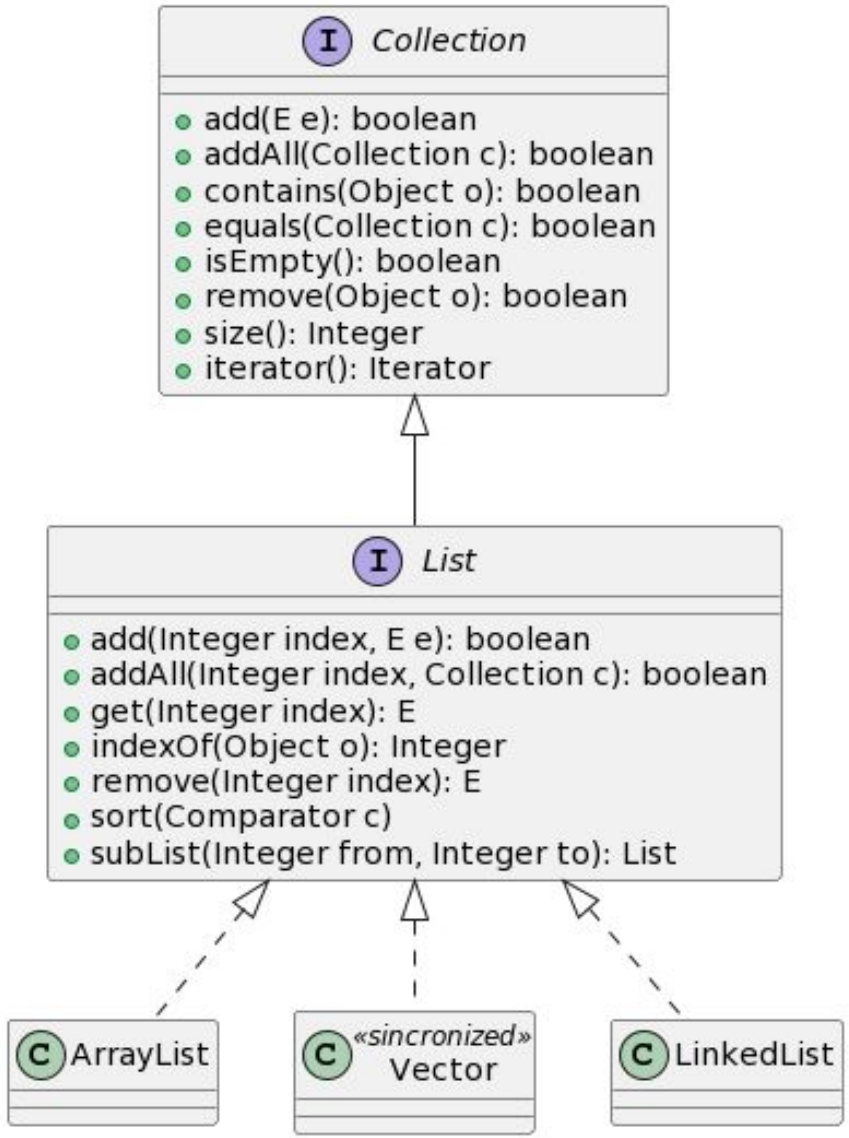
  - No admite duplicados.
  - Sus elementos no están indexados, ideal para chequear pertenencia
- Map (java.util.Map)
  - Asocia objetos que actúan como
  - claves a otros que actúan como valores



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



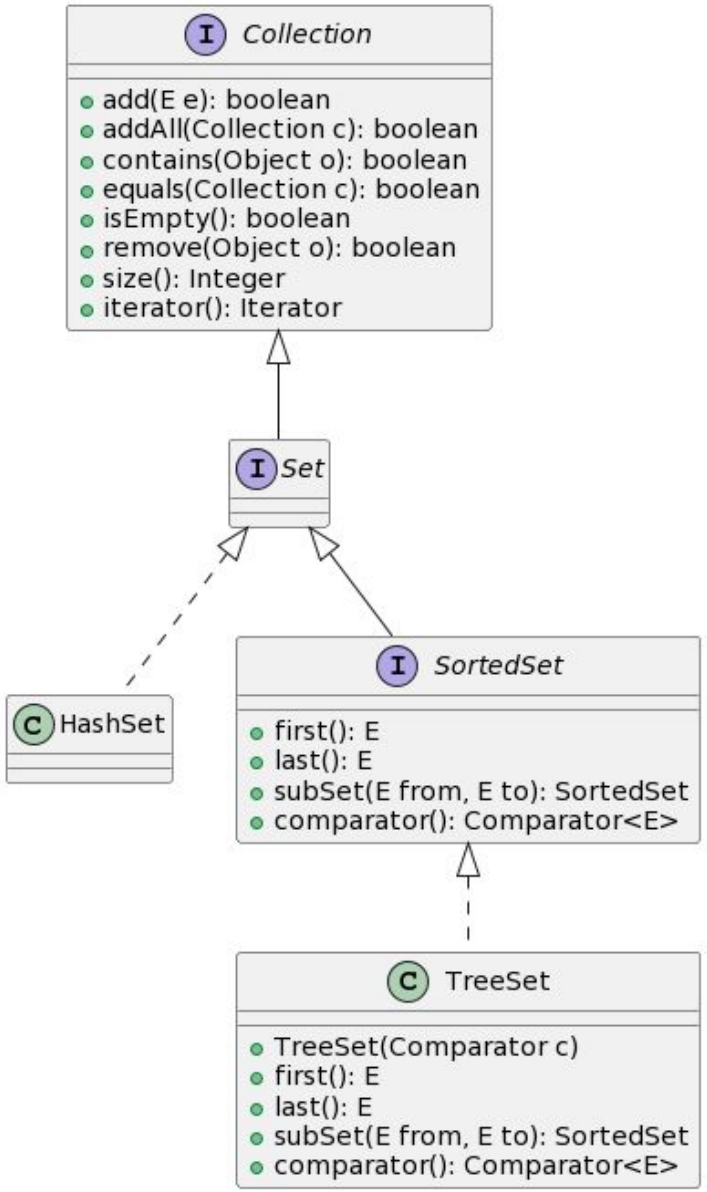
List




```java
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();
    }
```

Inferencia de tipos: no es necesario indicarlo nuevamente

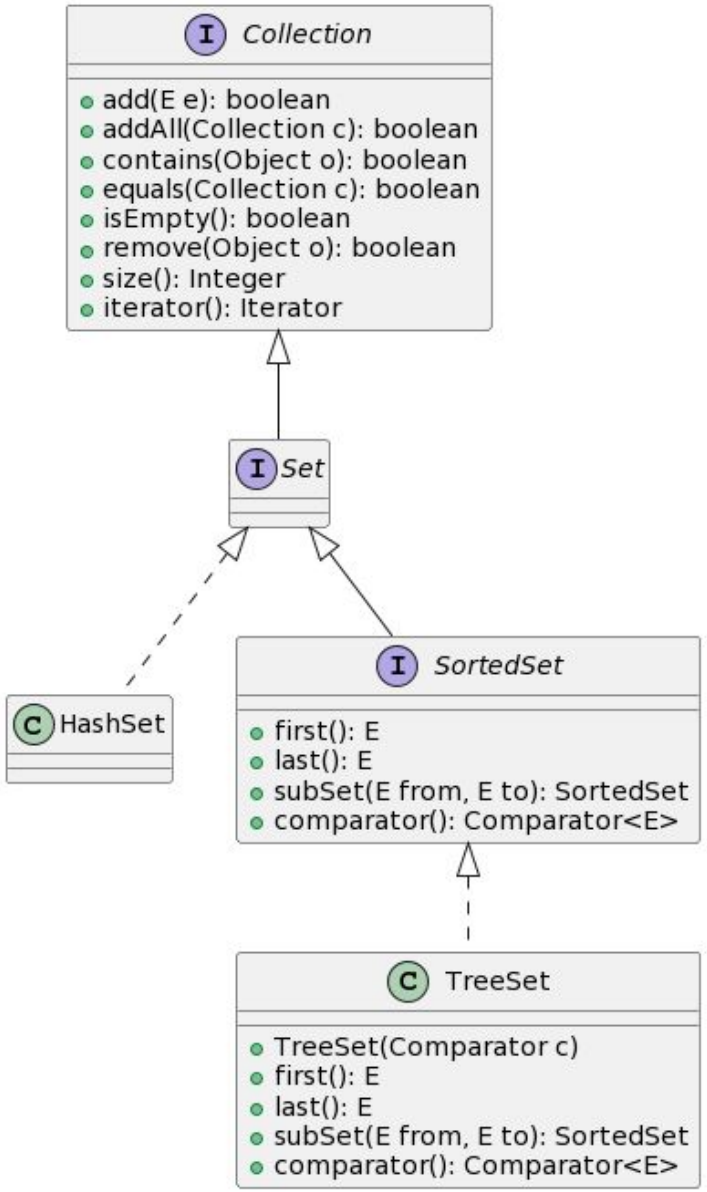
Set




```java
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()***

Set

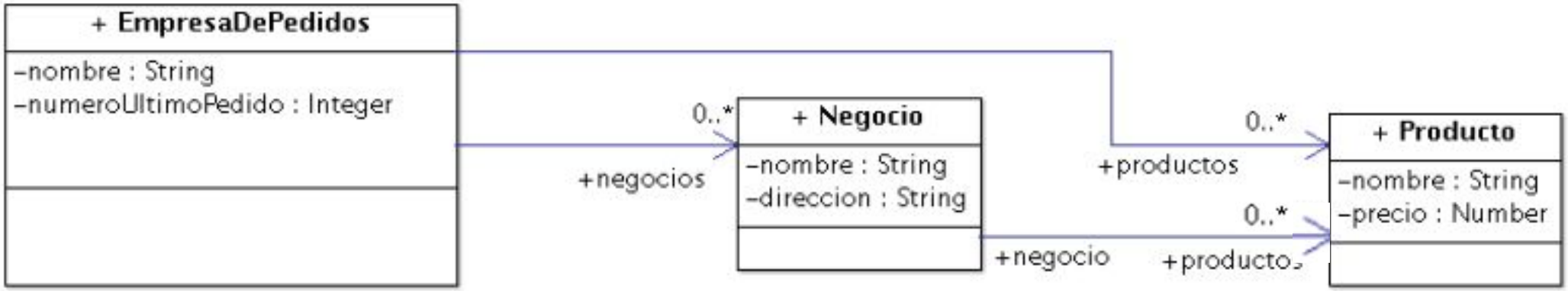



```java
import java.util.HashSet;
import java.util.Set;

public class Grupo {

    private String nombre;
    private Set<Persona> miembros;
```

```java
public class Persona {

    private String nombre;
    private String apellido;
    private String dni;

    public Persona(String nombre, String apellido, String dni) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.dni = dni;
    }
    @Override
    public int hashCode() {
        return this.dni.hashCode();
    }
    @Override
    public boolean equals(Object other) {
        ....
    }
```

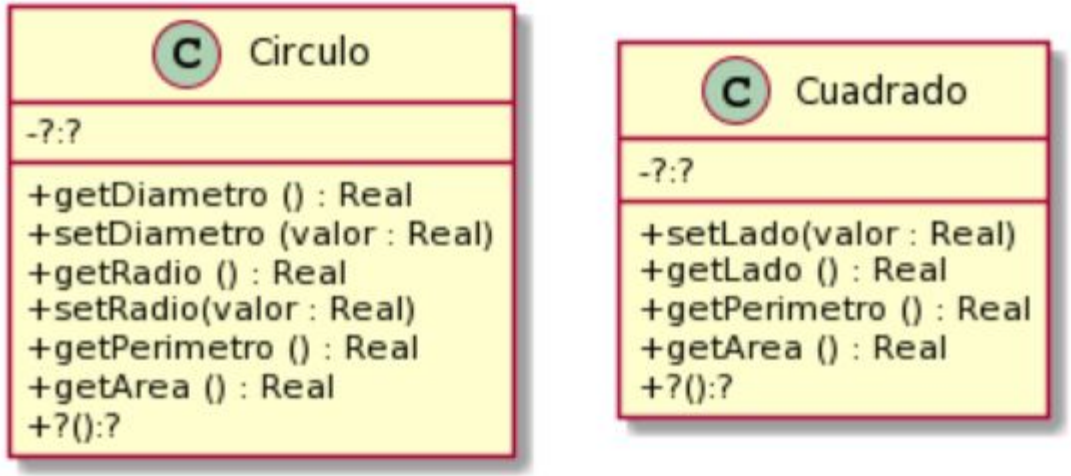
Colecciones en OO1

- El rol principal de las colecciones es mantener relaciones entre objetos
- En OO1, también vamos a utilizarlas como nuestros repositorios



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 a polimorfismo
- Al definir y al instanciar una colección indico el tipo de su contenido



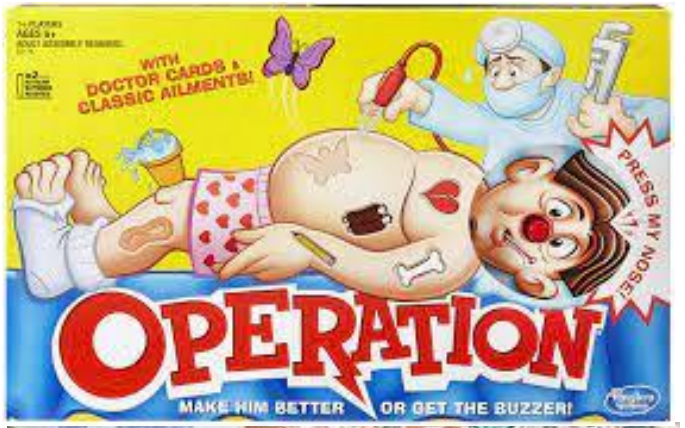
Generics y polimorfismo

- 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 a polimorfismo
- Al definir y al instanciar una colección indico el tipo de su contenido

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

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

Operaciones sobre colecciones



Operaciones frecuentes

- Siempre (o casi) que tenemos colecciones repetimos las mismas operaciones:
  - Ordenar respecto a algún criterio
  - Recorrer y hacer algo con todos sus elementos
  - Encontrar un elemento (max, min, DNI = xxx, etc.)
  - Filtrar para quedarme solo con algunos elementos
  - Recolectar algo de todos los elementos
  - Reducir (promedio, suma, etc.)
- Nos interesa escribir código que sea independiente (tanto como sea posible) del tipo de colección que utilizamos

Ordenando colecciones

producto_1
nombre = "Harina" precio = 100

producto_2
nombre = "Aceite" precio = 500

Quiero ordenar los productos por nombre

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

negativo -> el primero es menor
0 -> son iguales
positivo -> el primero es mayor

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

Método estático

Quiero ordenar los productos por precio (?)

Ordenando colecciones

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

```
public class ComparadorProductoNombre implements Comparator<Producto> {

    @Override
    public int compa
        return p1.ge
    }
```

```
public class ComparadorProductoPrecio implements Comparator<Producto> {

    @Override
    public int compare(Producto p1, Producto p2) {
        return Double.compare(p1.getPrecio(), p2.getPrecio());
    }
```

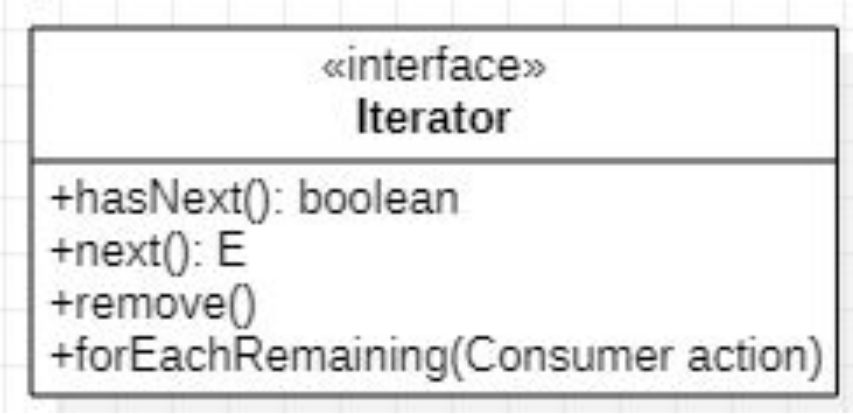
```
productos.sort(new ComparadorProductoNombre());
```

Recorriendo colecciones

- Recorrer colecciones es algo frecuente
- El loop de control es un lugar más donde cometer errores
- El código es repetitivo y queda atado a la estructura/tipo de la colección
  - ¿Qué hacemos con los Set?

```
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 (iterador externo)



- Todas las colecciones entienden iterator()
- Un Iterator encapsula:
  - Como recorrer una colección particular
  - El estado de un recorrido
- No nos interesa la clase del iterador (son polimórficos)
- El loop for-each esconde la existencia del iterador

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

```
for (Cliente cli : clientes) {
    // hago algo con el cliente
}
```

Precaución

- Nunca modifico 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));
```

Streams

Expresiones Lambda (clausuras / closures)

- Son métodos anónimos (no tienen nombre, no pertenecen a ninguna clase)
- 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 no es necesario utilizar los paréntesis.
   - Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.
2. Cuerpo de lambda
   - Si el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves.

Filtrar, colectar, reducir, encontrar …

```java
//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));
    }
}
```

- El iterador simplifica los recorridos pero …

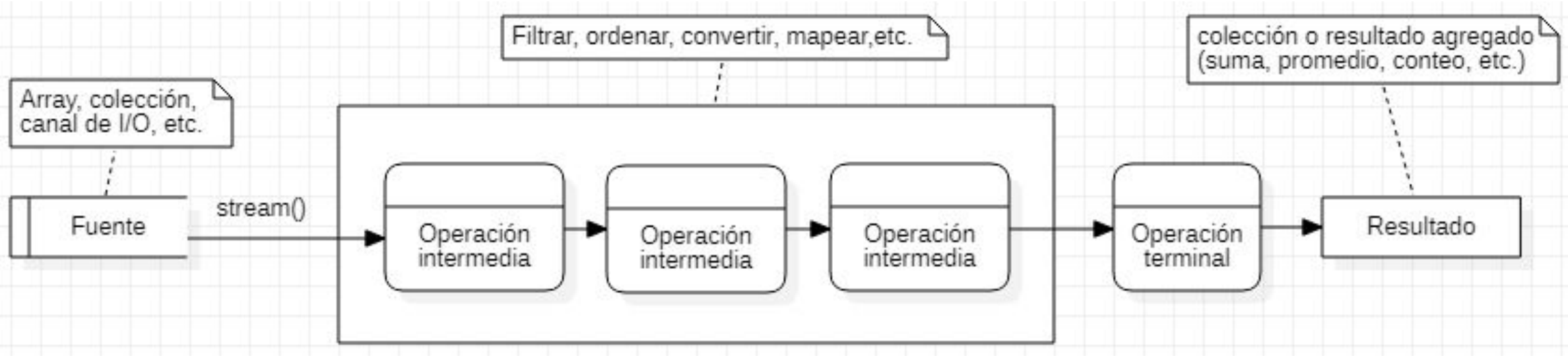
```java
//Identificar el cliente moroso de mayor deuda
Cliente deudorMayor;
double deudaMayor = 0;
for (Cliente cli : clientes) {
    if (cli.esMoroso()) {
        if (deudaMayor < cli.getDeuda()) {
            deudaMayor = cli.getDeuda();
            deudorMayor = cli;
        }
    }
}
```

Streams

- Objetos que permiten procesamiento funcional de colecciones
  - Las operaciones se combinan para formar pipelines (tuberías)
- Los streams:
  - No almacenan los datos, sino que proveen acceso a una fuente de datos subyacente (colección, canal I/O, 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 más frecuente de obtenerlos es vía el mensaje stream() a una colección

Stream Pipelines

- Para construir un pipeline encadeno envíos de mensajes
  - Una fuente, de la que se obtienen los elementos
  - Cero o más operaciones intermedias, 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	count | sum
map	average
limit	findAny | findFirst
sorted	collect
	anyMatch | allMatch | noneMatch
	min | max

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()`, devuelven un `Optional` para representar el 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 retorma 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 (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 está presente. Si el Optional está vacío, entonces orElse() devuelve el valor predeterminado proporcionado como argumento.

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

Filtrar, colectar, reducir, encontrar …

```java
//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;
        }
    }
}
```

```java
//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);
```

Cuando no 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 cuándo y cómo aplicarlas de manera efectiva
- reconocer cuándo no son apropiadas.

Siempre que me sea posible voy a usar alguna construcción de más alto nivel

- Son más concisas
- 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

Para llevarse

- Ojo con las colecciones de otro … son una invitación a romper el encapsulamiento
- Observar la estrategia de diseño de encapsular lo que varía/molesta
- Seguir investigando los protocolos de las colecciones, de los stream (no hablamos de reduce), y de Collectors
  - No queremos reinventar la rueda

Preguntas