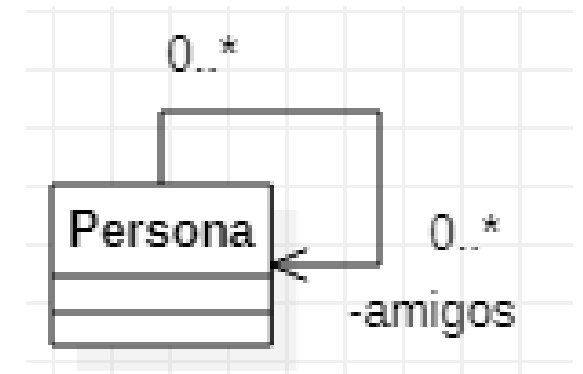
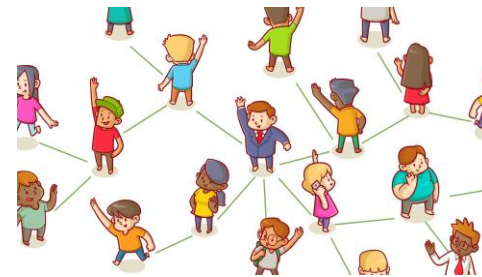
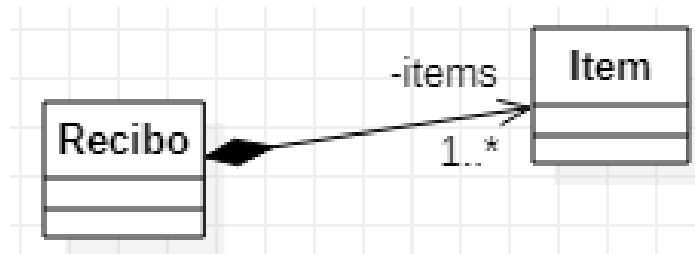
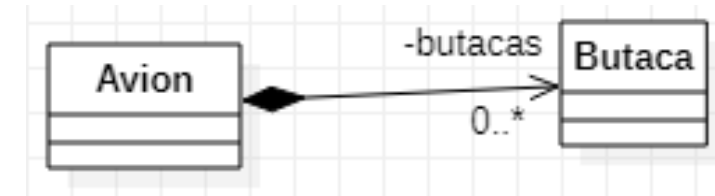




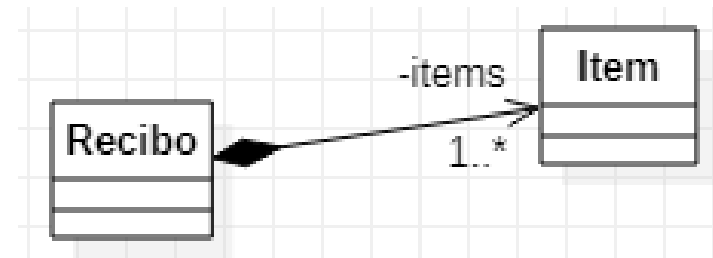
# Colecciones



# Relaciones 1 a muchos



# Colecciones como objetos

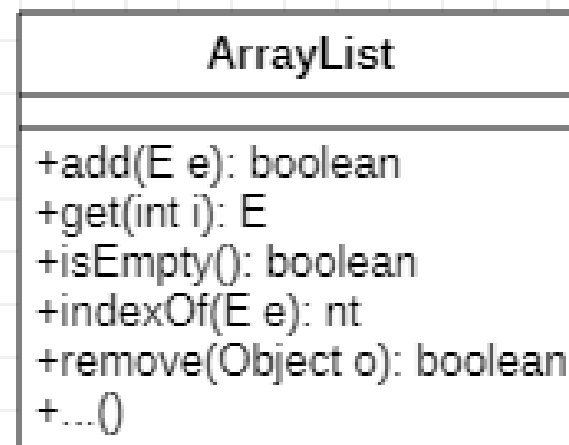


CASH RECEIPT	
Shop Name	Adress Line
Date:	MM/DD/YY
Manager:	Lorem Ipsum
<hr/>	
Tax	\$12.27
Total	\$77.83
<hr/>	
xxxx xxxx xxxx 1234 Visa	\$77.83
<hr/>	
Thank you for shopping!	

items

: ArrayList

1	→	1	→	>Lorem ipsum	\$9.99
2	→	2	→	Consectetuer adipiscing	\$12.99
3	→	3	→	Natoque penatibus	\$14.99
4	→	4	→	>Lorem ipsum	\$9.99
5	→	5	→	Consectetuer adipiscing	\$12.99
6	→	6	→	Natoque penatibus	\$14.99



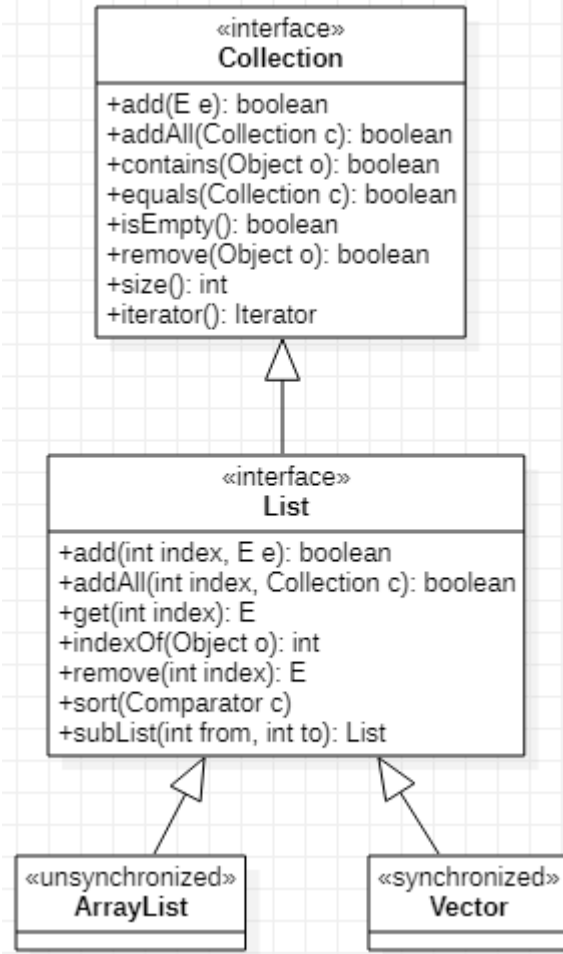
# Librería/framework de colecciones

- 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 (interfaces)

- List (`java.util.List`)
  - 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



```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
```

```
public class Ticket {
```

```
    private List<Item> items;
    private Date fecha;
    private Cliente cliente;
```

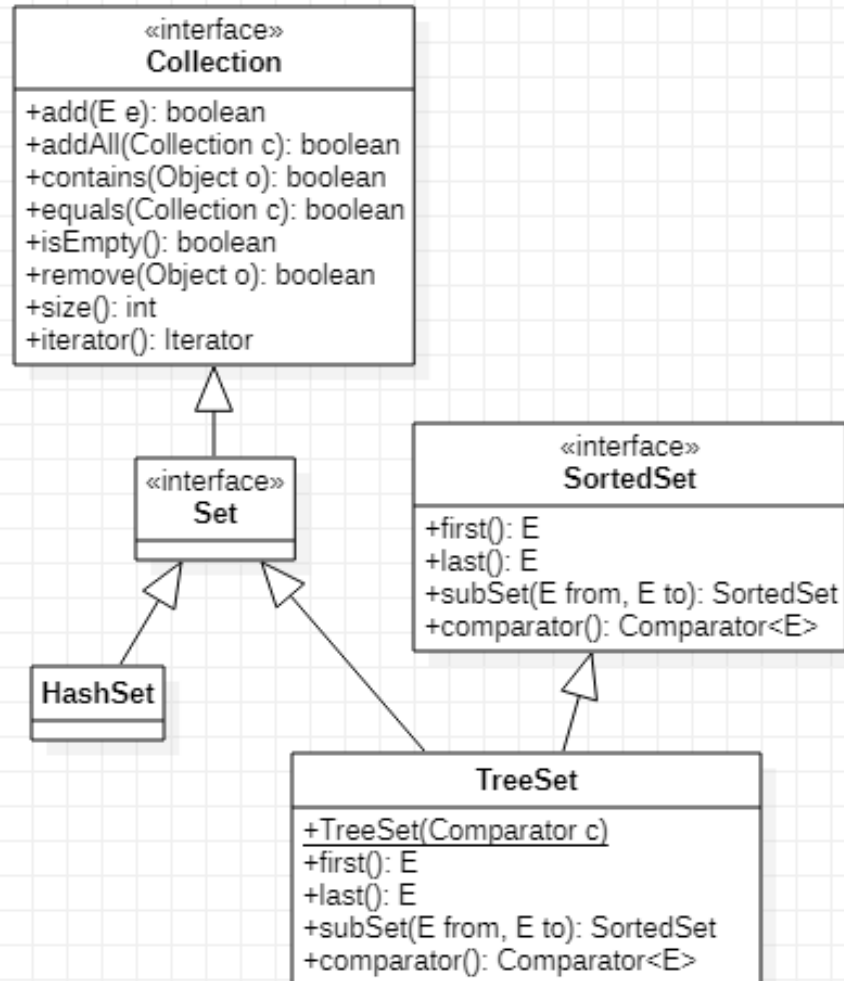
```
    public Ticket(Cliente cliente) {
        this.cliente = cliente;
        fecha = new Date();
        items = new ArrayList<Item>();
    }
```

```
    public void agregarItem(Producto producto, int cantidad) {
        items.add(new Item(producto, cantidad));
    }
```

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



# Set



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

```
public class Grupo {
```

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

```
    public Grupo() {
        setMiembros(new HashSet<Persona>());
    }
```

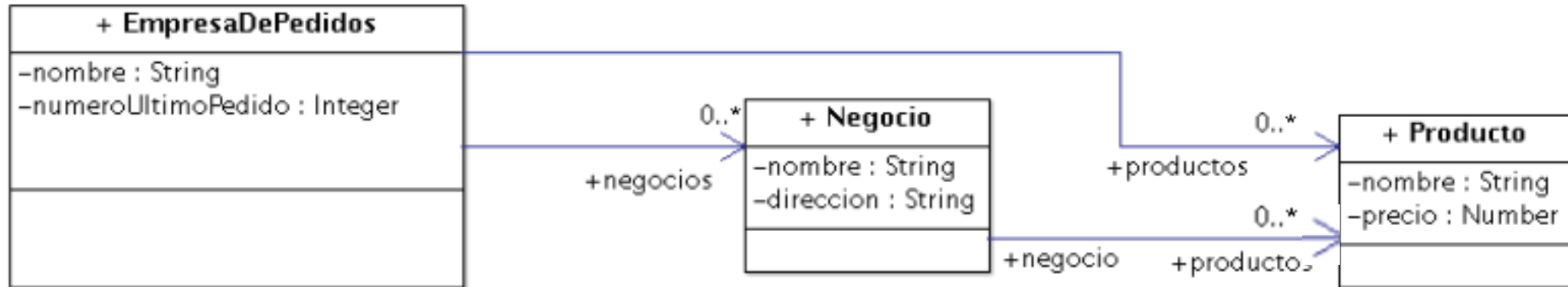
```
    public boolean agregarMiembro(Persona alguien) {
        return miembros.add(alguien);
    }
```

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

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

# 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

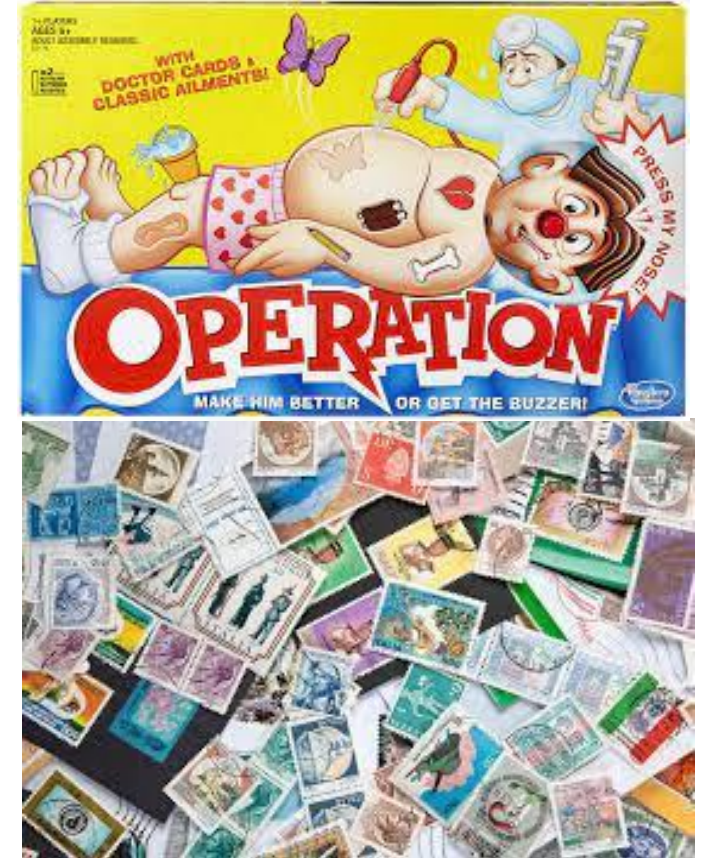
- Las colecciones admiten cualquier objeto en su contenido
- Cuanto mas sepa el compilador respecto al contenido de la colección, mejor podrá chequear lo que hacemos
- Contenido homogéneo da lugar a polimorfismo
- Al definir y al instanciar una colección indico el tipo de su contenido

```
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

- 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

```
figuras.sort(new ComparadorDeFiguras());
```

```
import java.util.Comparator;

public class ComparadorDeFiguras implements Comparator<Figura> {

    public int compare(Figura o1, Figura o2) {
        return Float.compare(o1.getSuperficie(), o2.getSuperficie());
    }

}
```

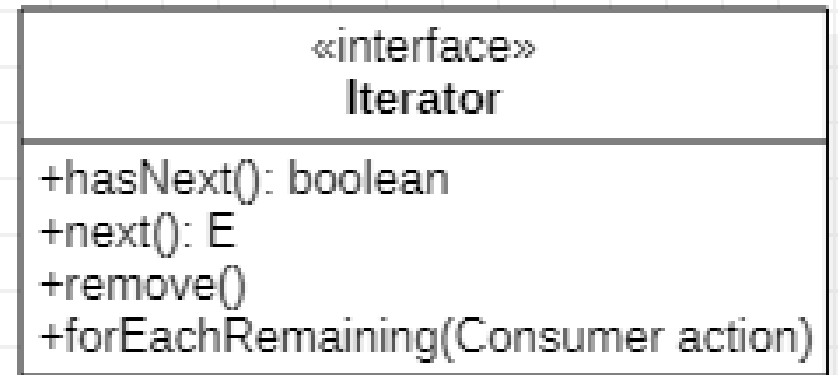
# Recorriendo colecciones

- 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  
    }  
}
```

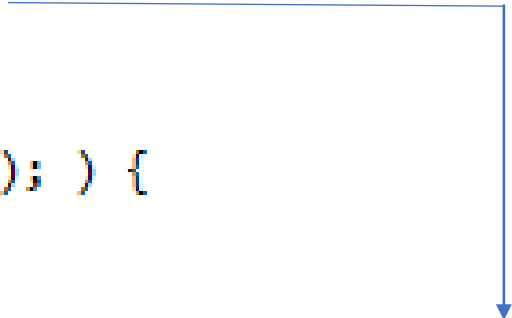
# Iterator (iterador externo)

- 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));
```

# Filtrar, coleccionar, 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));  
    }  
}
```

- El iterador simplifica los recorridos pero ...

```
//Identificar el cliente moroso de mayor deuda
```

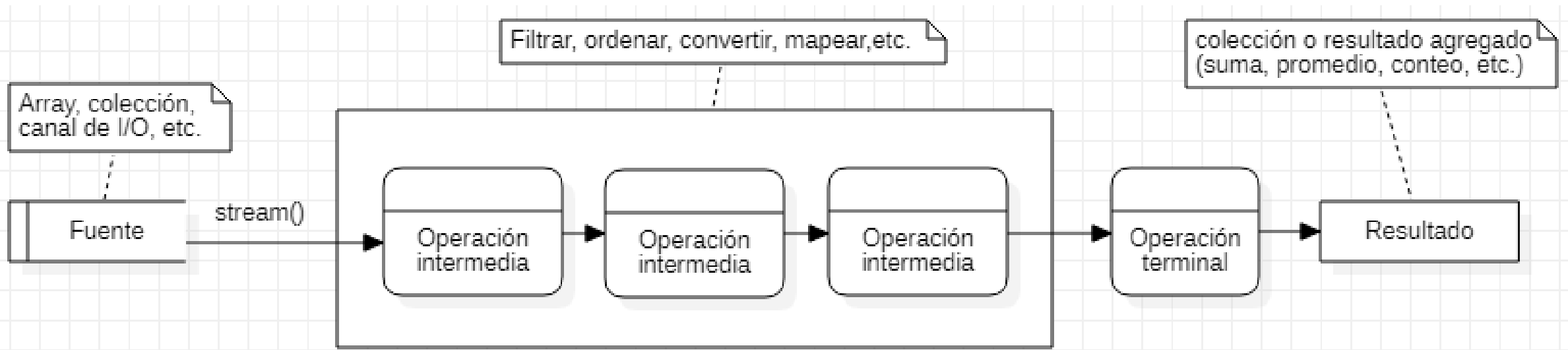
```
Cliente deudorMayor;  
double deudaMayor = 0;  
for (Cliente cli : clientes) {  
    if (cli.esMoroso()) {  
        if (deudaMayor < cli.getDeuda()) {  
            deudaMayor = cli.getDeuda();  
            deudorMayor = cli;  
        }  
    }  
}
```

# Streams

- Objetos que permiten procesamiento funcional de colecciones
  - Las operaciones se combinan para formar pipelines (tuberías)
- Los streams:
  - No almacenan sino que proveen acceso a una fuente (colección, canal I/O, etc.)
  - Cada operación produce un resultado, pero no modifica la fuente
  - Potencialmente sin final
  - Consumibles: cada elemento se visita una sola vez
- La forma más frecuente de obtenerlos es vía el mensaje `stream()` a una colección

# Stream Pipelines

- Para construir un pipeline encadenado 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



# 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<String> palabras = Arrays.asList("Monkey", "Lion", "Giraffe", "Lemur");  
List<String> conL = palabras.stream()  
    .filter(p -> p.startsWith("L"))  
    .collect(Collectors.toList());
```

# 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());
```



# 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()
```

# Filtrar, coleccionar, 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;
        }
    }
}
```

```
//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);
```

# Para llevarse

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