

# Resumen de Orientación a objetos 1

## Programa o Sistema Orientado a Objetos

Un conjunto de objetos que colaboran enviándose mensajes. Todo ocurre "dentro" de los objetos

Los sistemas están compuestos(solamente) por un conjunto de objetos que colaboran para llevar a cabo sus responsabilidades

Los objetos son responsables de:

- Conocer sus propiedades
- Conocer otros objetos(con los que colaboran)
- Llevar a cabo ciertas acciones

## Aspectos de interés en esta definición

- No hay un objeto “main”
- Cuando codificamos, describimos(programamos) clases
- Una jerarquía de clases no indica lo mismo que la jerarquía top-down
- Cuando se ejecuta el programa lo que tenemos son objetos que cooperan y que se crean dinámicamente durante la ejecución
- Podemos pensar la interacción usuario/software de la misma manera

- Este mismo modelo nos permite entender(al menos en parte) otros modelos de computación: viendo a los objetos como proveedores de servicios por ejemplo
- Este mismo modelo asume objetos localizados en el mismo espacio de memoria(pueden estar distribuidos)

## **Impacto de como “pensamos” el software**

- La estructura general cambia: en vez de una jerarquía: Main/procedures/sub-procedures tenemos una red de “cosas” que se comunican
- Pensamos en que “cosas” hay en nuestro software(los objetos) y como se comunican entre sí
- Hay un “shift” mental crítico en forma en el cual pensamos el software como objetos
  - Mientras que la estructura sintáctica es “lineal” el programa en ejecución no lo es

## **¿Qué es un objeto?**

- Es una abstracción de una entidad del dominio del problema. Ejemplos: Persona, Producto, auto...
- Puede representar también conceptos del espacio de la solución(Estructuras de datos, tipos “básicos”, archivos, ventanas, íconos...)

### **Características de los objetos**

Un objeto tiene:

- Identidad
  - Para distinguir un objeto de otro
- Conocimiento

- En base a sus relaciones con otros objetos y su estado interno
- Comportamiento
  - Conjunto de mensajes que un objeto sabe responder

## **Estado Interno**

- El estado interno de un objeto determina su conocimiento
- El estado interno esta dado por:
  - Propiedades básicas(intrínsecas) del objeto
  - Otros objetos con los cuales colabora para llevar a cabo sus responsabilidades
- El estado interno se mantiene en las variables de instancia(v.i.) del objeto
- Es privado del objeto. Ningún objeto puede accederlo.(¿Cuál es el impacto de esto?)

## **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. Cuáles son sus responsabilidades
- Se especifica a través del conjunto de mensajes que el objeto sabe responder: protocolo
- Ejemplo:



## Implementación

- La realización de cada mensaje(es decir, la manera en que un objeto responde a un mensaje) se especifica a través de un método
- Cuando un objeto recibe un mensaje responde activando un 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, éste responde activando el método asociado a ese mensaje(siempre y cuando exista)
- Como resultado del envío de un mensaje puede retornarse un objeto

## Métodos

### ¿Qué es un método?

- Es la contraparte funcional del mensaje
- Expresa la forma de llevar a cabo la semántica propia de un mensaje particular(el cómo)

### Un método puede realizar básicamente 3 cosas:

- Modificar el estado interno del objeto
- Colaborar con otros objetos (enviándole mensajes)
- Retornar y terminar

Y la entrada/salida de información?

- En un sistema diseñado correctamente, un objeto (programado por el desarrollador) no debería realizar ninguna operación vinculada a la interfaz (mostrar algo) o a la interacción (esperar un "input")
- En la mayoría de los entornos de desarrollo es hasta imposible hacerlo, y en nuestro caso lo será
- ¿Qué ganamos? Poder cambiar el estilo o el dispositivo de interacción sin necesidad de tocar el Código que pasa a ser independiente de la interfaz

## Formas de conocimiento

- Para que un objeto conozca a otro lo debe poder "nombrar". Decimos que se establece una ligadura (binding) entre un nombre y un objeto
- Podemos identificar tres formas de conocimiento entre objetos.
  - Conocimiento interno: Variables de instancia
  - Conocimiento externo: Parámetros
  - Conocimiento temporal: Variables temporales
- Además 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 del mundo exterior"

## **Características:**

- Esconde detalles de su implementación
- Protege el estado interno de los objetos
- Un objeto sólo 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 qué se publica
- Facilita modularidad y reutilización

## **Clases e instancias**

- Una clase es una descripción abstracta de un conjunto de objetos
- Las clases cumplen tres roles:
  - Agrupan el comportamiento común a sus instancias
  - Definen la forma de sus instancias
  - Crean objetos que son instancia de ellas
- En consecuencia todas las instancias de una clase se comportan de la misma manera
- Cada instancia mantendrá su propia 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
- Gráficamente

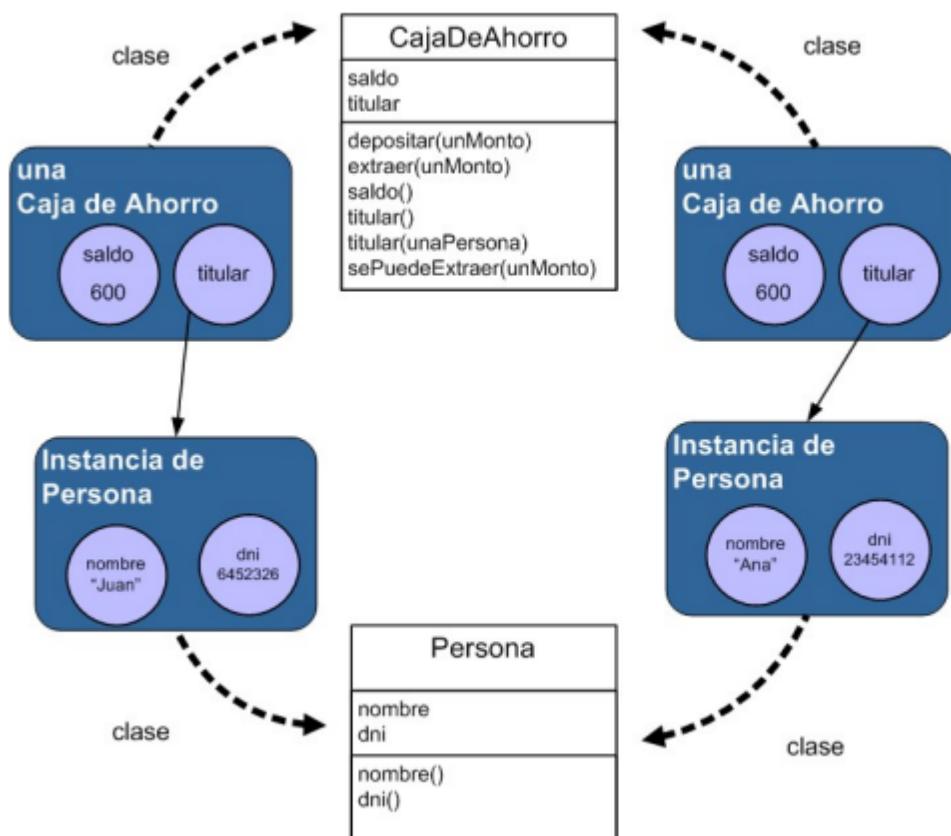
**Variables de Instancia**  
Los nombre de las v.i. se escriben en minúsculas y sin espacios

CajaDeAhorro
saldo
titular
depositar(unMonto)
extraer(unMonto)
saldo()
titular()
titular(unaPersona)
sePuedeExtraer(unMonto)

**Nombre de la Clase**  
Comienzan con mayúscula y no posee espacios

**Protocolo**  
Para cada mensaje se debe especificar como mínimo el nombre y los parámetros que recibe

## Ejemplo de clases e instancias



## Creación de objetos

- ¿Cómo creamos nuevos objetos?
- Instanciación
- Comúnmente se utiliza la palabra reservada new para instanciar objetos

### 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 una misma clase
  - Tendrán la misma estructura interna.
  - Responderán al mismo protocolo (los mismos mensajes) de la misma manera (los mismos métodos)

### Inicialización

- Para que un objeto esté listo para llevara a cabo sus responsabilidades hace falta inicializarlo
- Inicializar un objeto significa darle valor a sus variables

## Relaciones objetosas

Objetos que conocen a otros, identidad e igualdad, relaciones uno a muchos, delegación, polimorfismo y rol de los tipos e interfaces

## Relaciones entre objetos

- Un objeto conoce a otro porque
  - Es su responsabilidad mantener a ese otro objeto en el sistema(tiene un, conoce a)
    - Por ejemplo: cada cuenta conoce a su titular
  - Necesita delegarle un trabajo(enviarle mensajes)
    - Por ej: una cuenta recibe a otra como parámetro(destino) para pedirle que deposite
- Un objeto conoce a otro cuando
  - Tiene una referencia en una variable de instancia(rel. duradera)
  - Le llega una referencia como parámetro(rel. temporal)
  - Lo crea(rel. temporal/duradera)
  - Lo obtiene enviando mensajes a otros que conoce(rel. temporal)

## This(un objeto que habla solo)

- this(o en algunos lenguajes self) es una "pseudo-variable"
  - no puedo asignarle valor
  - Toma valor automáticamente cuando un objeto comienza a ejecutar un método
- this hace referencia al objeto que ejecuta el método(al receptor del mensaje que resultó en la ejecución del método)
- Se utiliza para:
  - Descomponer métodos largos(refinamiento top-down)
  - Reutilizar comportamiento repetido en varios métodos

- Aprovechar comportamiento heredado
- Pasar una referencia para que otros puedan enviarnos mensajes
- En algunos lenguajes (por ejem: java)
  - Puede obviarse (es implícito), aunque en OO1 preferimos no hacerlo
  - Para desambiguar referencia a las variables de instancia del objeto

Ejemplos:

```
Private void payThePrice(){
    this.disableShields();
    energy -=1;
}

Public void step(){
    locomotion.move(this);
    energySource.consumeBattery(this);
    armsSystem.fireArms(this);
    collector.collectArtifacts(this);
}
```

## Reutilizar comportamiento repetido

```
Public void translateBy(Point2D point){
    this.position.translateBy(point);
    this.disableShields();
    this.energy-=1;
}

public void fireUpon(Ship ship){
    ship.takeFireFrom(this.position);
    this.disableShields();
    this.energy -=1;
}
```

Vemos que ambos en las ultimas dos líneas hacen lo mismo. Podríamos crear el siguiente método y no repetir código

```
Private void payThePrice(){
    this.disableShields();
    energy -=1;
```

```
}
```

Quedando así ahora

```
Public void translateBy(Point2D point){  
    this.position.translateBy(point);  
    this.payThePrice();  
}  
  
public void fireUpon(Ship ship){  
    ship.takeFireFrom(this.position);  
    this.payThePrice();  
}
```

## Igualdad / el operador ==

- Las variables son punteros a objetos
- Más de una variable pueden apuntar a un mismo objeto
- Para saber si dos variables apuntan al mismo objeto utilizo “==”
- == es un operador, no puede redefinirse

## Igualdad / el método equals()

- Dos objetos pueden ser iguales - la igualdad se define en función del dominio

## Relaciones entre objetos y chequeo de tipos

- Java es un lenguaje, estáticamente, fuertemente tipado
  - Debemos indicar el tipo de todas las variables(relaciones entre objetos)

- El compilador chequea la correctitud de nuestro programa respecto a tipos
- Se asegura de que no enviamos mensajes a otros objetos que no los entienden
- Cuando declaramos el tipo de variable, el compilador controla que solo “enviamos a esa variable” mensajes acordes al tipo
- Cuando asignamos un objeto a una variable, el compilador controla que su clase sea “compatible” con el tipo de variable

## Tipos de lenguajes OO (simplificado)

- Tipo ↔ conjunto de firmas de operaciones/métodos (nombre, orden y tipos de argumentos)
- Algunos lenguajes diferencian entre tipos primitivos y tipos de referencias (objetos)
  - Nos enfocaremos principalmente en los segundos
- Cada clase en Java define “explícitamente” un tipo (es un conjunto de firmas de operaciones)
  - Puedo utilizar clases, para dar tipo a las variables
- Asignar un objeto a una variable, no afecta al objeto (no cambia su clase)
  - La clase de un objeto se establece cuando se crea, y no cambia más
- Pero ... las clases no son la única forma de definir tipos

## Interfaces

- Una clase define un tipo, y también implementa métodos correspondientes
- Una variable tipada con una clase solo “acepta” instancias de clase(\*)
- Una interfaz nos permite declarar tipos sin tener que ofrecer implementación (desacopla tipo e información)

- Puedo utilizar Interfaces como tipos de variables
- Las clases deben declarar explícitamente que 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)

## Objeto que conoce a muchos...

- Las relaciones de un objeto a muchos se implementan con colecciones
- Decimos que un objeto conoce a muchos, pero en realidad conoce a una colección, que tiene referencias a otros métodos
- Para modificar y explotar la relación, envío mensajes a la colección

Se dibuja así:



## ¿Envidia o delegación?

¿Cómo implementar getPrecio() en la clase oferta?

Envidia...

- Una clase oferta que envidiosa y egoísta que quiere hacer todo
- Responsabilidades poco repartidas(Producto es solo datos)

- Clases más acopladas y poco cohesivas

Delegación...

- Calculo del precio del producto está con los datos requeridos
- Oferta “delega” y se desocupa de cómo se hace el cálculo
- Clases más desacopladas y más cohesivas

## Method Lookup (recordamos)

- 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 sabemos lo que hará)

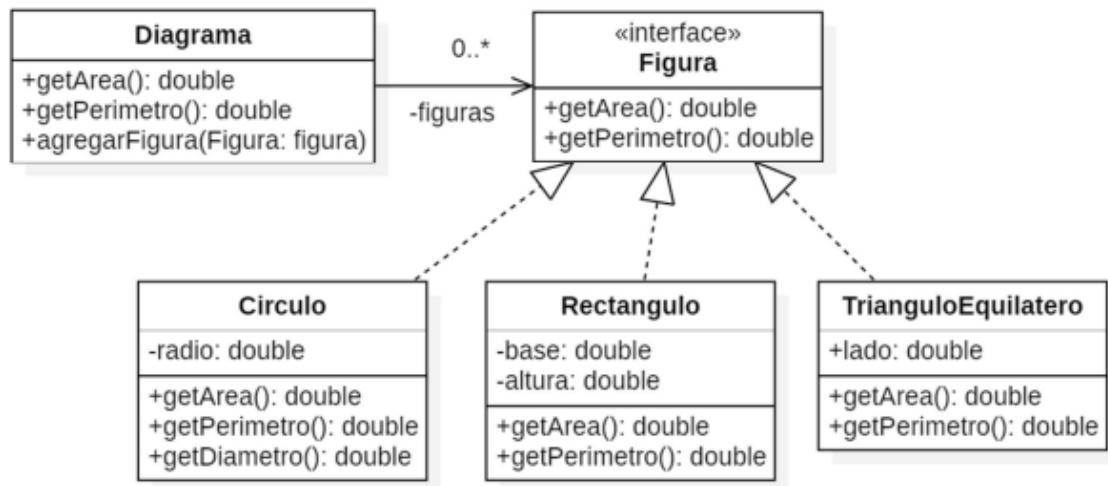


## Polimorfismo

- Objetos de distintas clases son polimórficos con respecto a un mensaje, si todos los entienden, aun cuando cada uno lo implemente de modo diferente
- Polimorfismo implica:

- Un mismo mensaje se puede enviar a objetos de distinta clase
- Objetos de distinta clase "podrían" ejecutar métodos diferentes en respuesta a un mismo mensaje
- Cuando dos clases Java implementan una interfaz, se vuelven polimórficas respecto a los de la matriz

Figuras polimórficas...



Polimorfismo bien aplicado

- Permite repartir mejor las responsabilidades (delegar)
- Desacopla objetos y mejora la cohesión (cada cual hace lo suyo)
- Concentra cambios (reduce el impacto de los cambios)
- Permite extender sin modificar (agregando nuevos objetos)
- Lleva código más genérico y objetos reusables
- Nos permite programar por protocolo, no por implementación

# Herencia

Herencia, clases concretas, abstractas, generalización, especialización, this y super

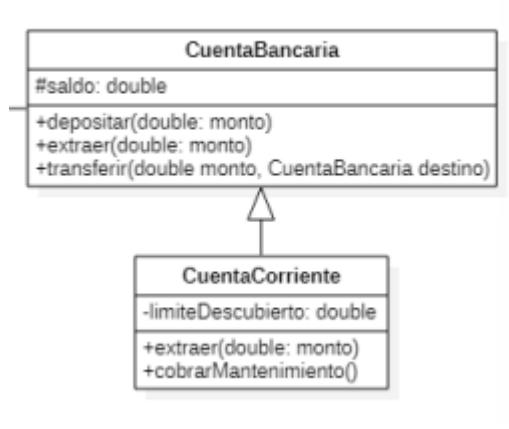
## Definición de herencia en programación:

- Mecanismo que permite a una clase “heredar” estructura y comportamiento de otra clase
  - Es una estrategia de reúso de código
  - Es una estrategia de reúso de conceptos/definiciones

En java solo tenemos herencia simple

Es una característica transitiva

Sigamos el siguiente ejemplo:



```
public class CuentaCorriente extends CuentaBancaria {
```



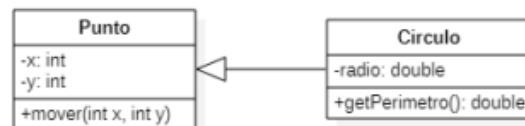
Vocabulario:

- CuentaCorriente

- es subclase de CuentaBancaria
- hereda de CuentaBancaria
- extiende cuentaBancaria
- CuentaBancaria
  - Es superclase de CuentaCorriente
- extaer() en CuentaCorriente
  - extiende extraer() de CuentaBancaria o redefine extraer() de CuentaBancaria

## La prueba “es un”

- 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
  - Una cuenta corriente es una cuenta, una ventana de texto es una ventana, un array es una colección...
- Un anti-ejemplo famoso (herencia para construcción)



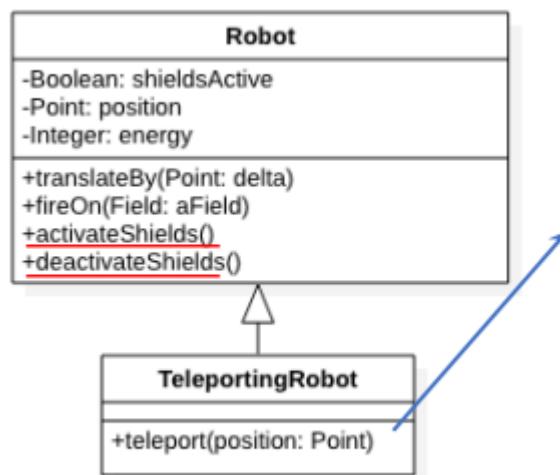
Un círculo NO-ES-UN punto, conoce o tiene un punto

## Method Lookup con herencia

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



## Aprovechar comportamiento heredado



La flecha celeste apunta al siguiente código de ejemplo:

```

public void teleport(Point2d position){
    this.activateShields();
    this.position = position;
    this.deactivateShields();
}
  
```

## Sobrescribir métodos(overriding)

- La búsqueda en la cadena de superclases termina tan pronto como encuentro un método cuya firma coincide con lo que busco
- Si heredaba un método con la misma firma, el mismo queda "oculto"(redefinir / override)
- No es algo que ocurra con frecuencia (puede dar mal olor)

## Extender métodos

¿Qué hacemos si queremos aprovechar(extender) ese método que heredábamos?

### Super

- Podemos pensar a super como una "pseudo-variable" (como this)
  - No puedo asignarle valor
  - Toma valor automáticamente cuando un objeto comienza a ejecutar un método
- En un método, "super y this" hacen referencia al objeto que lo ejecuta (receptor del mensaje)
- Utilizar super en lugar de this "solo cambia la forma en la que se hace el method lookup"
- Se utiliza para:
  - Solamente para extender comportamiento heredado  
(reimplementar un método e incluir el comportamiento que se heredaba para él)

## Super y el method lookup

Cuando super 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)

## Super() en los constructores

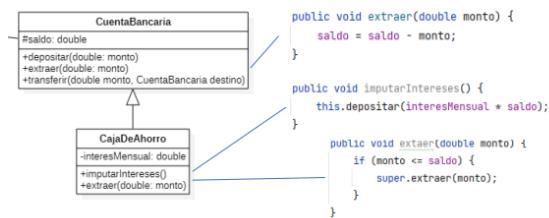
- Los constructores en java son subrutinas que se ejecutan en la creación de objetos - no se heredan
- Si quiero reutilizar comportamiento de otro constructor debo invocarlo explícitamente - usando super(...) al principio

```
public CuentaBancaria(Persona titular) {
    this.titular = titular;
}

public CuentaCorriente(Persona titular, double saldoInicial, double limiteDescubierto) {
    super(titular);
    saldo = saldoInicial;
    this.limiteDescubierto = limiteDescubierto;
}
```

## Especializar

- Especializar: crear una subclase especializando una clase existente

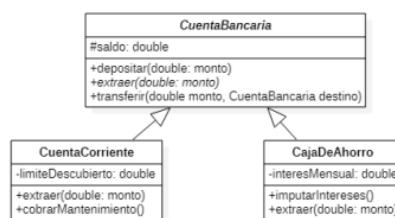


## Clase abstracta

- Una clase abstracta captura comportamiento y estructura que será común a otras clases
- Seguramente será especializada
- Puede declarar comportamiento abstracto y utilizarlo para implementar comportamiento concreto

Generalizar

Significa introducir una superclase que abstraiga aspectos comunes a otras — suele resultar una clase abstracta



## Situaciones de uso de herencia

- Subclasificar para especializar
  - La subclase extiende métodos para especializarlos
  - Ambas clases son concretas
- Herencia para especificar
  - La superclase es una combinación de métodos concretos y abstractos (clase abstracta)
  - La subclase implementa los métodos abstractos
- Subclasificar para extender
  - La subclase agrega nuevos métodos

## Situaciones de uso de herencia (feas)

- Heredar para construir
  - Heredo comportamiento y estructura pero cambio el tipo (no-es-un)
  - Se debe evitar aunque nos vamos a cruzar con ejemplos (en código de otros)
  - Si es posible, reemplazar por composición
- Subclasificar para generalizar (no es lo mismo que generalización)
  - La subclase reimplementa métodos para hacerlos más generales
  - Solo tiene sentido si no puedo reordenar la jerarquía (para especializar)
- Subclasificar para limitar
  - La subclase reimplementa un método para que deje de funcionar / limitarlo
  - Solo tiene sentido si no puedo reordenar la jerarquía (para especializar)
- Herencia indecisa (subclasificación por varianza)
  - Tengo dos clases con un mismo tipo, y algunos métodos compartidos
  - No puedo decidir cuál es la subclase y cual la superclase
  - Resolverlo buscando una superclase común (generalización)

## Tipos en lenguajes OO (simplificado)

Tipo ó Conjunto de firmas de operaciones/métodos (nombre, orden y tipos de los argumentos)

- Con eso en mente:
  - Cada clase en Java define “explícitamente” un tipo
  - Cada subclase define “explícitamente” un sub-tipo
  - Cada instancia de una clase A (o de cualquier subclase) “es del tipo” definido por esa clase

## Modificadores de visibilidad con herencia

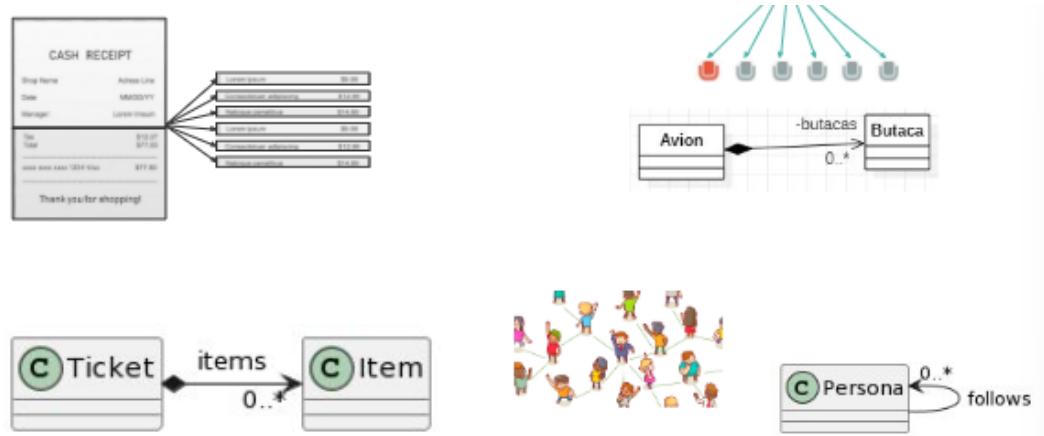
- Si declaro una variable de instancia como private en una clase A:
  - Las instancias de las de A tendrán esa variable de instancia
  - En los métodos de las subclases de A, no puedo hacer referencia a ella
- Si declaro un método "m()" como private en una clase A:
  - Las instancias de sus subclases entenderán el mensaje m()
  - En los métodos de las subclases de A no puedo enviar m() a "this"
- Si declaro variables y métodos como protected, puedo verlos en las subclases

## Clases abstractas e interfaces

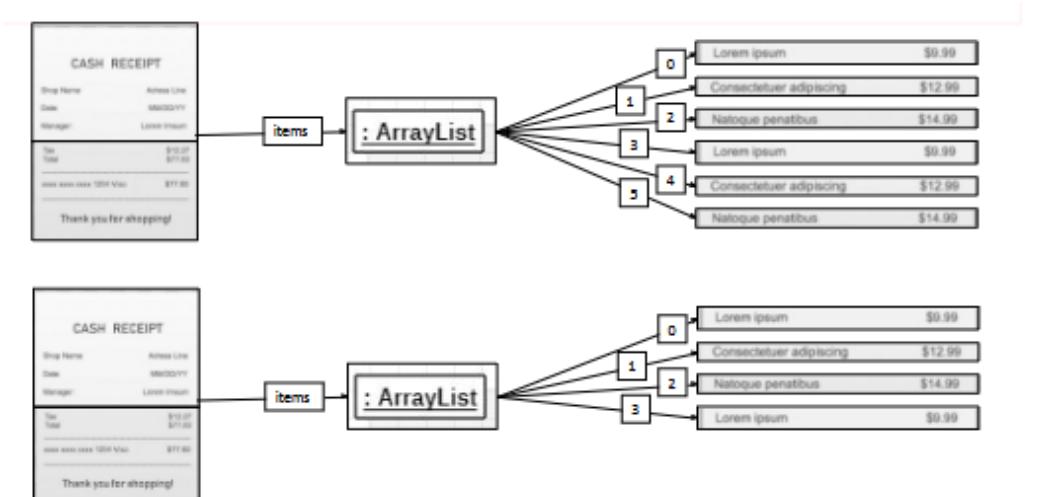
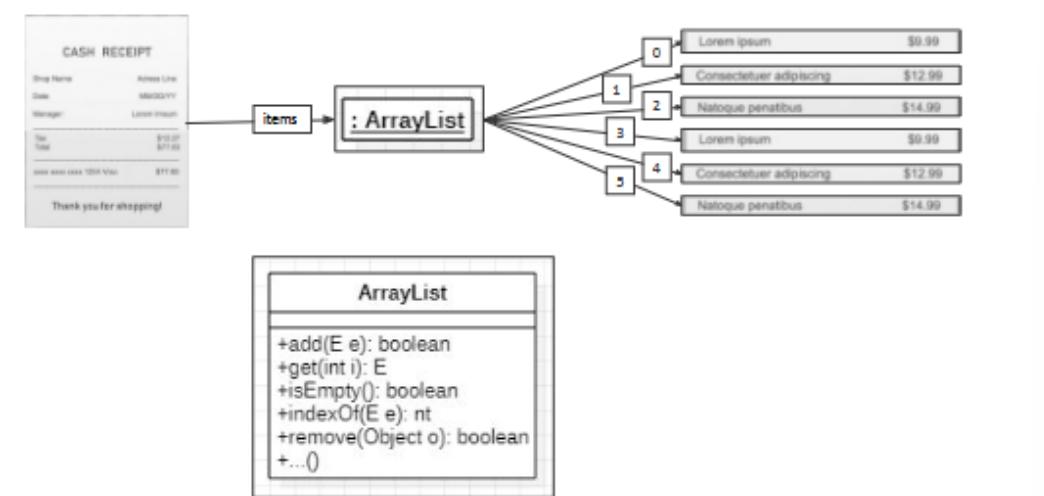
- Una clase abstracta "es una clase"
  - Puede usarla como tipo
  - Puede o no tener métodos abstractos
  - Ofrece implementación a algunos métodos
  - Sus métodos concretos pueden depender de sus métodos abstractos (p.e., si se define antes de sus subclases)
- Una interfaz define un tipo
  - Sirve como contrato
  - Puede extender a otras
- Se pueden implementar muchas interfaces pero solo se puede heredar de una clase

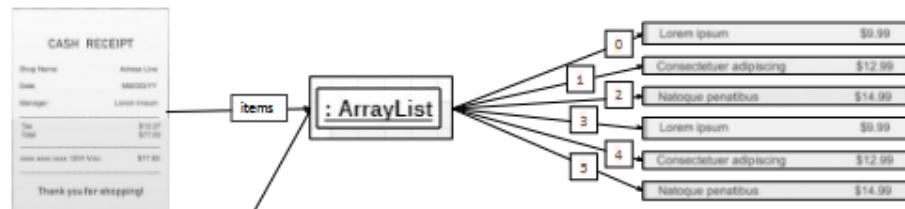
## Colecciones

## Relaciones 1 a muchos



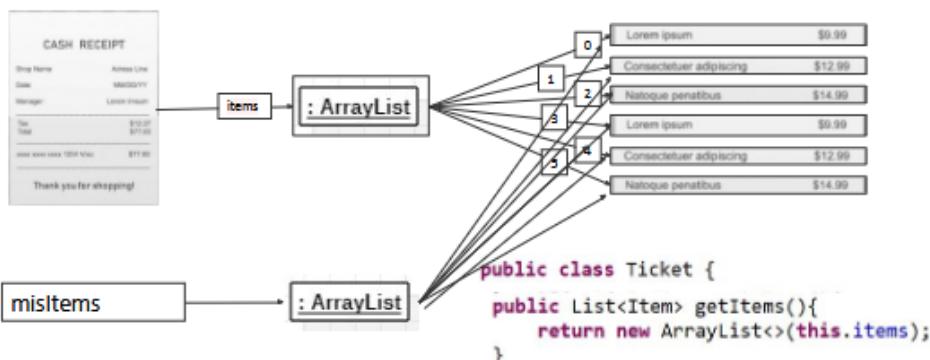
## Colecciones como objetos





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

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



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

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

## 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

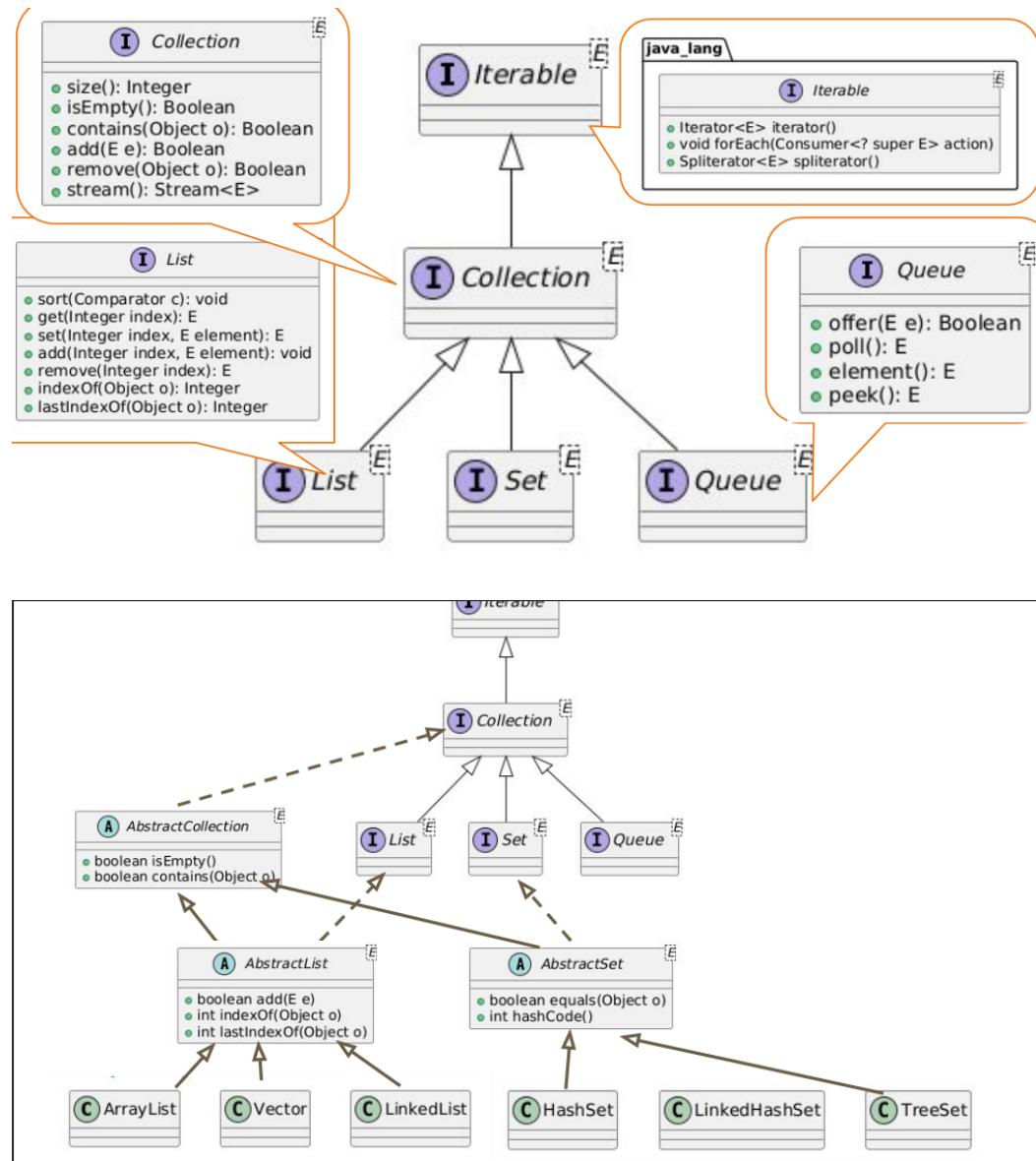
```
Ticket ticket = new Ticket(cliente);
ticket.addItem(new Item(producto, 10)); // Correcto
ticket.getItems().add(new Item(producto, 10)); // Incorrecto
```

## Libería/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 de colecciones (interfaces)

- 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)



## 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 su tipo de contenido

# 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

## Comparator

- 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

## 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

## Iterator

- Todas las colecciones tienen 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 trabajar con diferentes tipos de colecciones (listas, conjuntos y colas) de manera uniforme
- Un iterador encapsula:
  - Como recorrer una colección en particular
  - El estado de un recorrido
- No nos interesa la clase del iterador (son polimórficos)

## Streams

- Expresamos lo que queremos de una forma más abstracta y declarativa  
→ código más 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 I/O, etc)
- Cada operación produce un resultado, pero sin modificar 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

## Expresiones Lambda (clausuras / closures)

- Son métodos anónimos
- Útiles para:
  - parametrizar lo que otros objetos deben hacer
  - decirle a otros objetos que me avisen cuando pase algo(callbacks)

## **Expresiones Lambda - sintaxis**

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

Ejemplos:

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

### 1. Parámetros:

- a. Cuando se tiene solo un parámetro los paréntesis son opcionales
- b. Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis
- c. Opcionalmente se pueden indicar el tipo, sino lo infiere

### 2. Cuerpo de Lambda

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

## **Stream Pipelines**

- Para construir un pipeline se encadenan 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 retornar un resultado
- La operación terminal guía el proceso. Las operaciones intermedias son Lazy: se calcular y procesan solo cuando es

necesario, es decir, cuando se realiza una operación terminal que requiere resultado

### **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` 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

### **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

### **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

### **Operación intermedia: 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())

### **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 cuándo y cómo aplicarlas de manera efectiva
- Reconocer cuándo son apropiadas

Siempre que sea posible voy a utilizar 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

- Quiero ordenar una colección y que se mantenga ordenada por un criterio
- Quiero eliminar los elementos que cumplen una condición
- Cuando no quiero recorrer secuencialmente porque el algoritmo no lo requiere

# UML (Lenguaje Unificado de Modelado)

Es un lenguaje de modelado visual que nos permite

- Especificar
- visualizar
- construir
- documentar

artefactos de un sistema de software

Permite capturar decisiones y conocimientos

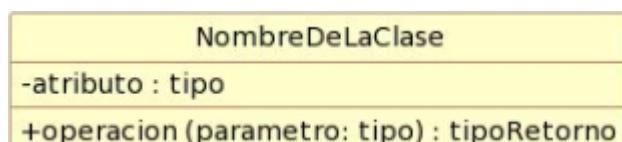
## Diagramas de estructura

- Diagrama de clases
- Diagrama de Paquetes
- Diagrama de Componentes
- Diagrama de Objetos
- Diagrama de Despliegue

### Diagrama de Clases

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

Una clase es representada gráficamente por cajas de tres compartimentos



Nombre de la clase

Singular

Comenzar con Mayúscula

estilo CamelCase

Si la clase es abstracta  
cursiva  
estereotipo <>abstract><

### **Atributos:**

Visibilidad:

- privada ( - )
- protegida ( # )
- pública ( + )
- paquete ( ~ )

Nombre:

- estilo camelCase
- comienza con minúscula

tipo:

- tipos UML: Integer, Real, Boolean, String
- valor por defecto

### **Operaciones:**

visibilidad

- privada (-)
- protegida (#)
- pública (+)
- paquete (~)

nombre

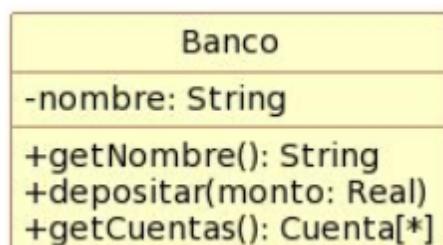
- estilo camelCase
- comienza con minúscula

Parámetros

- nombre: estilo camelCase

tipo de retorno

- Si no retorna nada, no se especifica
- Si retorna un objeto, se indica de qué 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><



### Relaciones:

Asociaciones →

Herencia (Generalización) →

Implementación →

Dependencia →

Asociación:

Navegabilidad

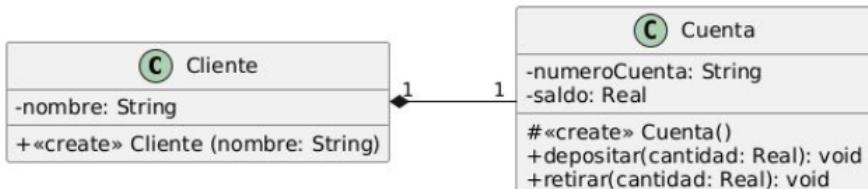


```

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

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

```



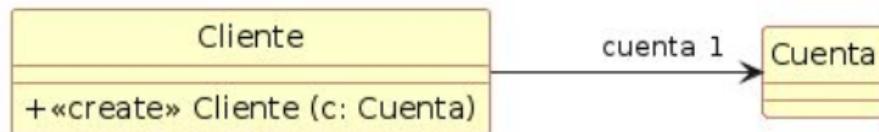
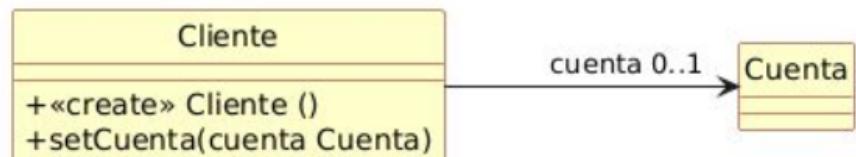
```

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

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

```

## Multiplicidad



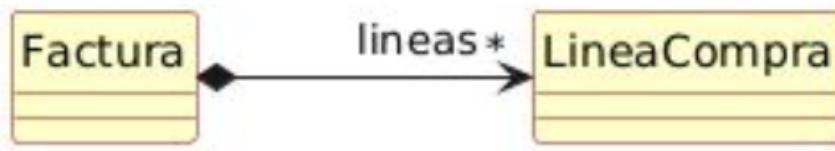
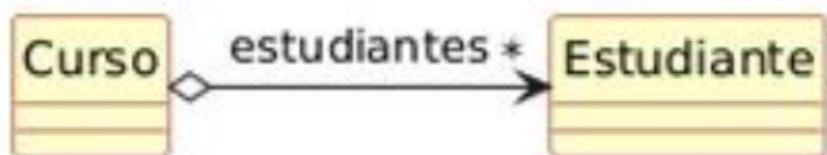
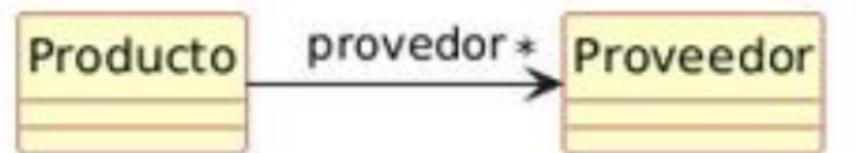
nombre de rol

tipo:

simple

agregación

composición



## 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 de la interfaz precedido por el estereotipo <<interface>>.

Dentro del rectángulo, se listan las operaciones o métodos que la interfaz define.



### Implementación y herencia

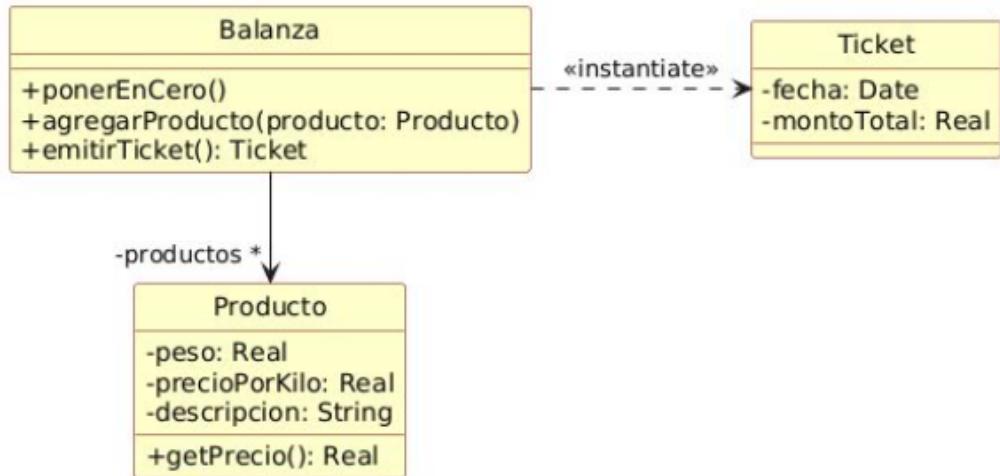
Una clase concreta implementa una interface. Esto significa que la clase proporciona una implementación concreta para todos los métodos (operaciones) definidos en esa interfaz.

Si una clase concreta extiende a una clase abstracta, significa que debe proveer las implementación de los métodos abstractos.

### Dependencia

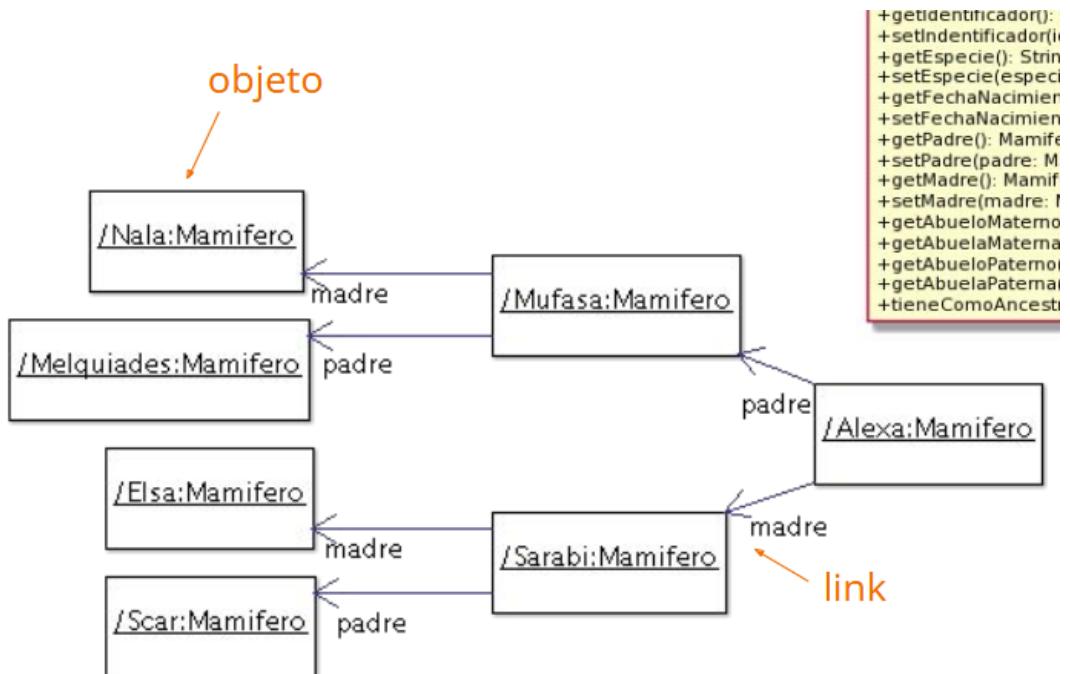
Indica que un cambio en la especificación de una clase puede afectar a otra clase que depende de ella.

Esta relación se utiliza para mostrar que una clase o componente necesita de otra 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.

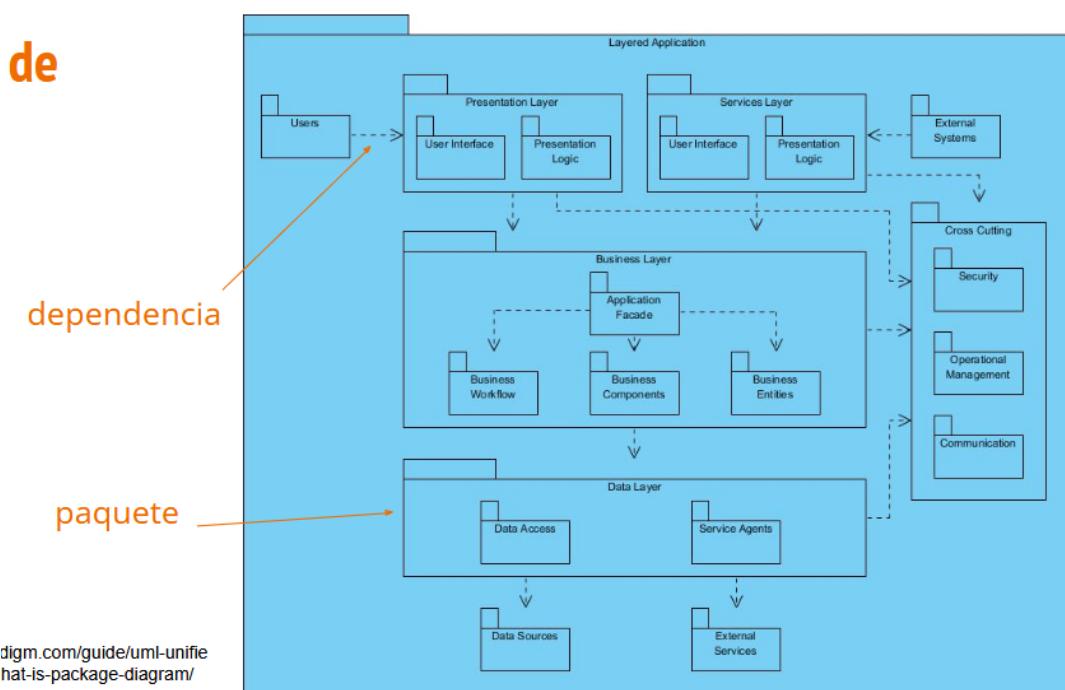


## Diagrama de Paquetes

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

Se requiere:

- Alta cohesión dentro de un paquete. Los elementos dentro de un paquete están relacionados
- Poco acoplamiento entre ellos (exportando sólo aquellos elementos necesarios e importando solo lo necesario)



## Diagramas de comportamiento

- Diagrama de casos de Uso
- Diagrama de Interacción
  - Diagrama de Secuencia
  - Diagrama de Colaboración
- Diagrama de Máquinas de Estado
- Diagrama de Actividades

### 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
- 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 uso son:
  - Actores
  - Casos de Uso
  - Relaciones

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

**Caso de Uso:** representación de una funcionalidad específica

**Relaciones:**

**Include:** un caso de uso incluye a otro si el primero incorporará el comportamiento especificado en el segundo

**Extend:** un caso de uso extiende a otro si puede ser mejorado o ampliado con funcionalidad adicional en ciertos escenarios, pero no siempre aplica

## Diagrama de secuencia

Un diagrama de secuencia es un tipo de diagrama de interacción porque describe cómo — y en qué orden — colabora un grupo de objetos.

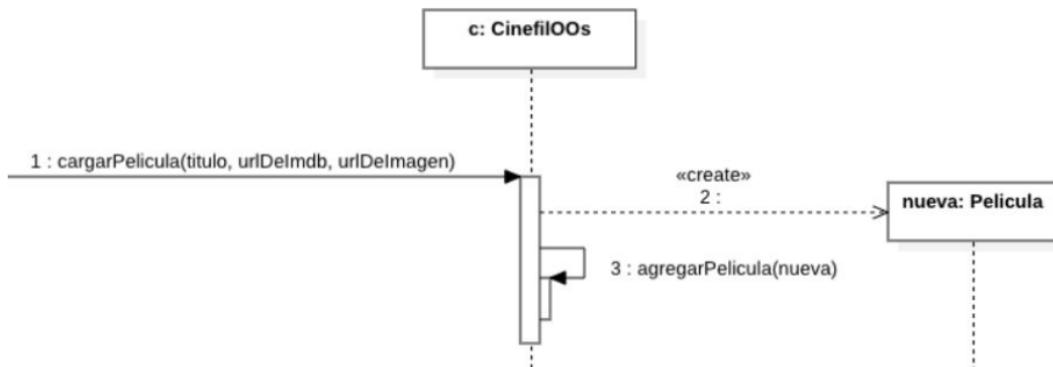
- Muestra claramente cómo interactúan distintos objetos en un sistema a lo largo del tiempo.

En un diagrama de secuencia

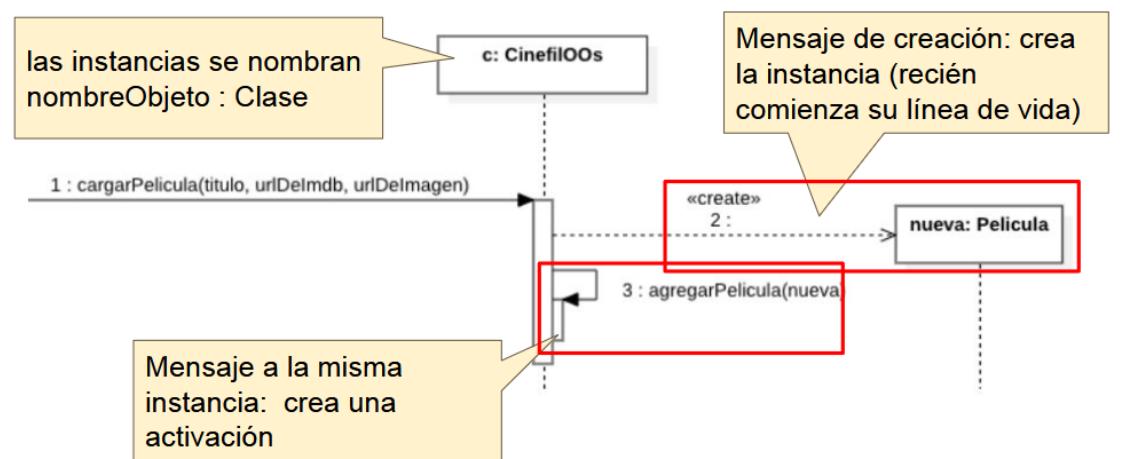
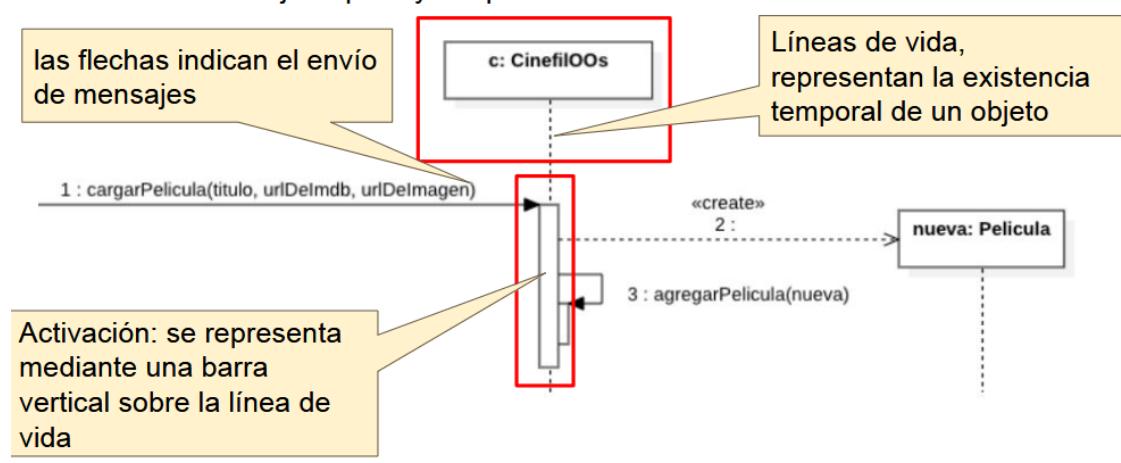
- los objetos se representan en la parte superior del diagrama

- el tiempo avanza de arriba hacia abajo

Diagrama de secuencia del mensaje cargarPelícula de CinefilOOs



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



Sintaxis del mensaje :

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

Los corchetes

indican que  
es opcional

## **CombinedFragment**

Un fragmento combinado permite representar la lógica y las condiciones en la interacción entre objetos.

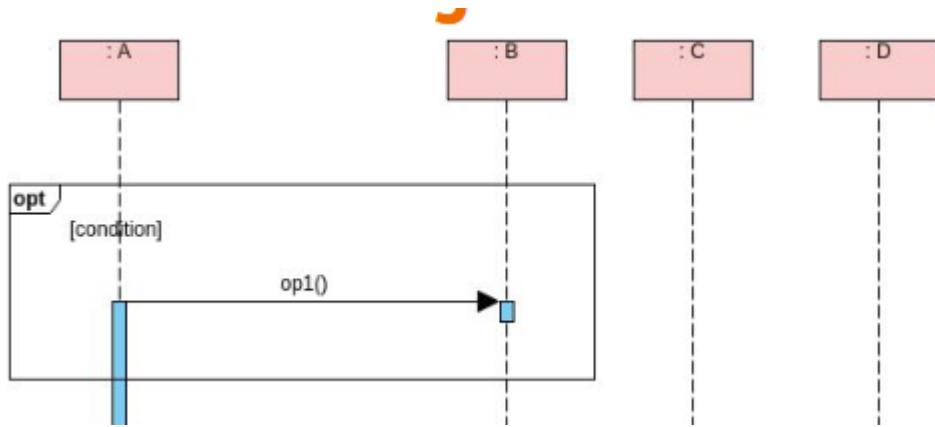
A través de ellos se pueden especificar bloques para repetición, opcionales y alternativos, entre otros.

Fragmentos más utilizados:

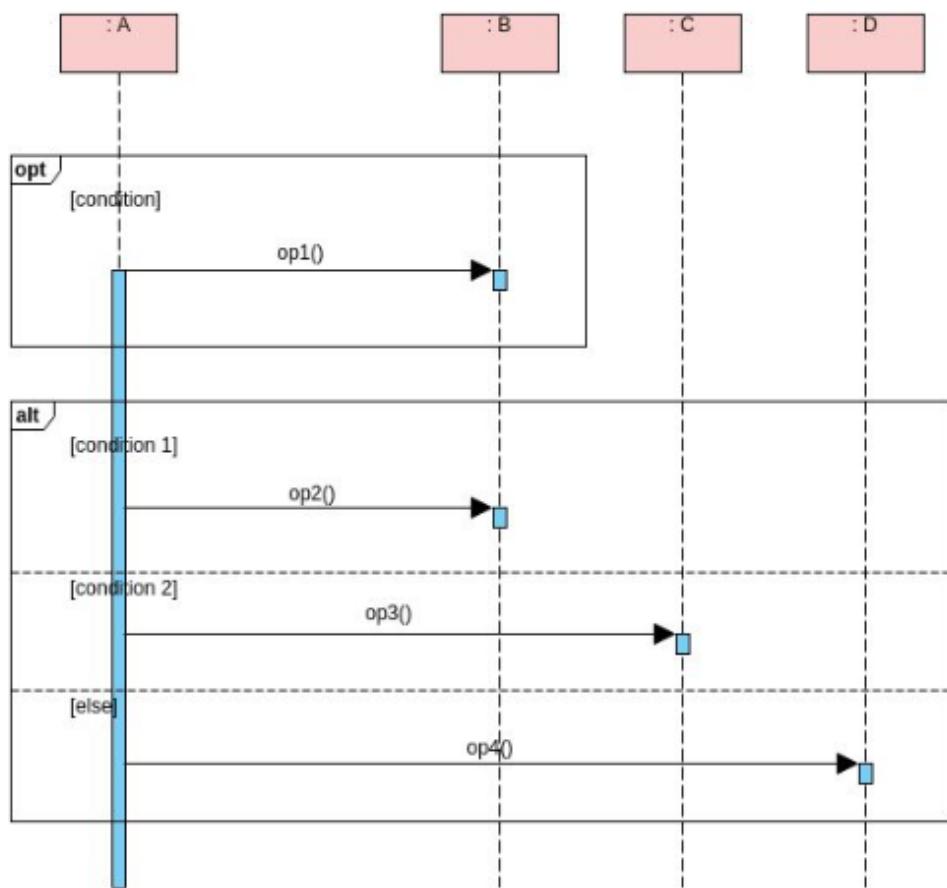
- opt: opcional
- alt: alternativa
- loop: bucle
- ref

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

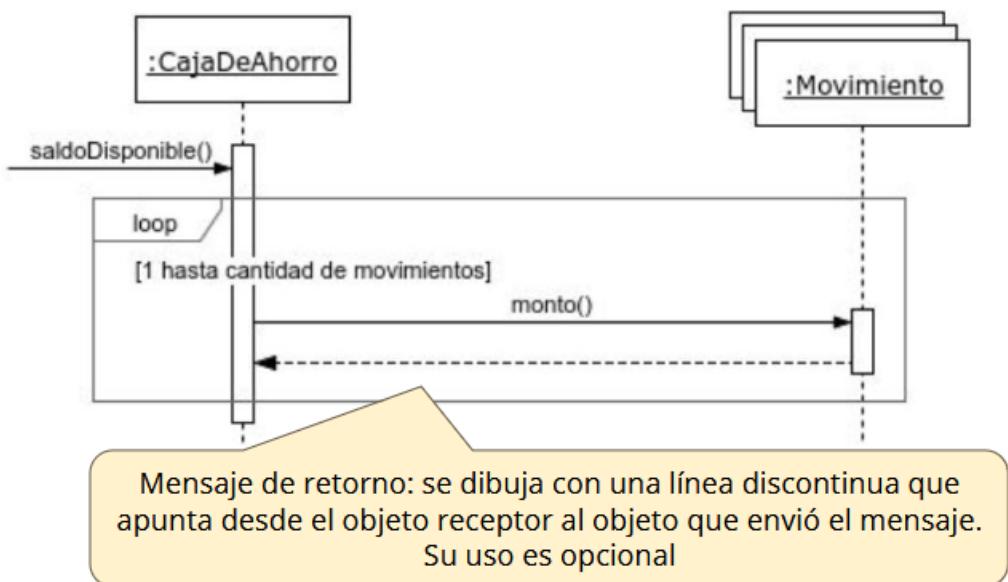
**opt (Opcional)**: Representa una parte de la secuencia de interacción que puede o no ejecutarse, dependiendo de una condición booleana. Si la condición es verdadera, se ejecuta la parte opcional; de lo contrario, se omite.



**alt (Alternativa):** Este tipo de fragmento 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.



## ¿Cómo se define un UML?

Documento de especificación de UML

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

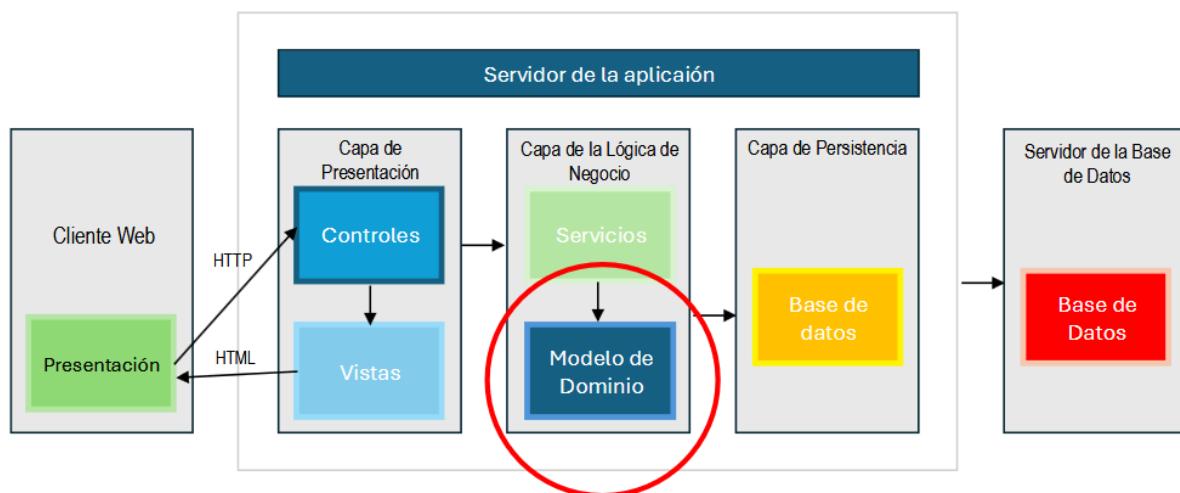
Las herramientas de modelado UML pueden ser tanto gráficas como basadas en el metamodelo UML

Herramienta gráfica: permite dibujar elementos con la misma imagen que UML

Herramientas basadas en el metamodelo UML:

Estas herramientas se centran más en la manipulación directa de los elementos del metamodelo UML.

## Modelo del dominio



## Identificación de clases conceptuales

La tarea es identificar las clases conceptuales vinculadas al escenario con el que se trabaja.

### Consejos

- Usar nombres del dominio del problema, no de la solución.
- Omitir detalles irrelevantes.
- No inventar nuevos conceptos (evitar sinónimos).
- Descubrir conceptos del mundo real.

### Estrategias

- Utilización de una lista de categorías de clases conceptuales.

- Identificar clases nominales.

Categorías:

Categoría de Clase Conceptual	Ejemplos
Objeto físico o tangible	Libro impreso
Especificación de una cosa	Especificación del producto, descripción
Lugar	Tienda
Transacción	Compra, pago, cancelación
Roles de la gente	cliente
Contenedor de cosas	Catálogo de libros, carrito, Avion
Cosas en un contenedor	Libro, Artículo, Pasajero
Otros sistemas	Sistema de facturación de AFIP
Hechos	cancelación, venta, pago
Reglas y políticas	Política de cancelación
Registros financieros/laborales	Factura/ Recibo de compra
Manuales, documentos	Reglas de cancelación, cambios de categoría de cliente

## Frases nominales

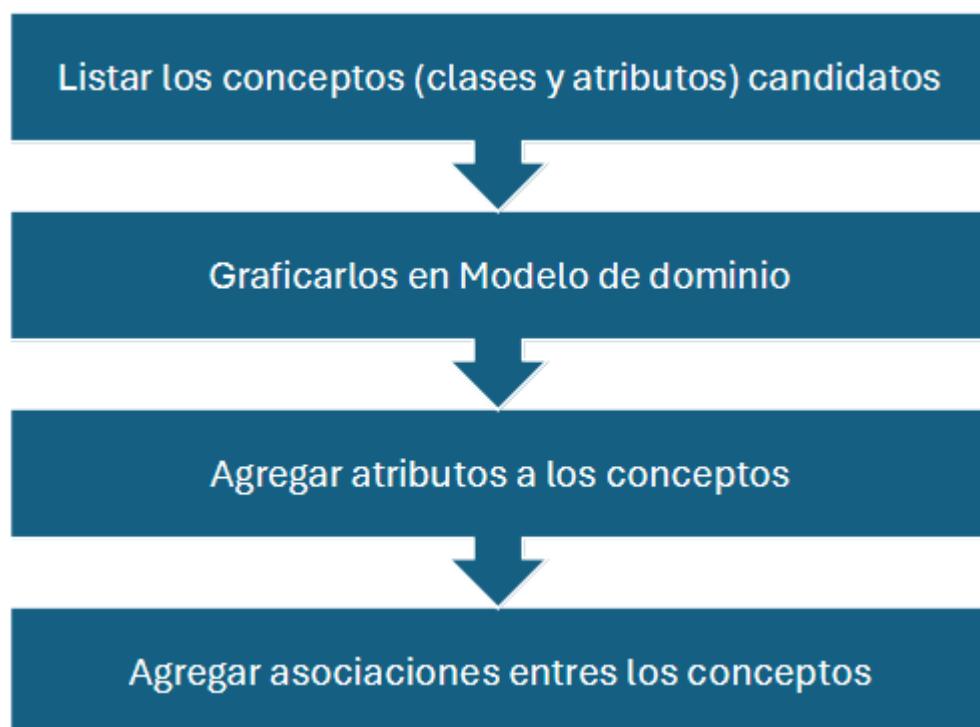
### ¿Qué son?

- Conjunto de palabras en una oración que funciona como un sustantivo.
- Este conjunto puede consistir de un único sustantivo (núcleo) o de más de una palabra en el que el sustantivo es el núcleo .

Algunas de las frases nominales son clases conceptuales candidatas, algunas podrían hacer referencia a clases conceptuales y algunas podrían ser atributos de las clases conceptuales.

## Construyendo el Modelo de Dominio

Un Modelo del Dominio es una representación visual de las clases conceptuales del mundo real en un dominio de interés



### Agregar atributos

- Se identifican los atributos que son necesarios para satisfacer los requerimientos de información de los casos de uso en desarrollo.
- Los atributos en un modelo deberían ser, preferiblemente, atributos simples o tipos de datos primitivos.

### ¿Cómo darse cuenta que algo pensado como atributo debe ser una clase conceptual?

- Está compuesto de secciones separadas.
- Tiene operaciones asociadas.
- Tiene otros atributos
- Es una cantidad con una unidad.
- Es una abstracción de uno o más tipos

## **Agregando asociaciones**

- Céntrese en aquellas asociaciones para las que se necesita conservar el conocimiento de la relación durante algún tiempo (asociaciones "necesito-conocer").
- Es más importante identificar clases conceptuales que identificar asociaciones.
- Demasiadas asociaciones tienden a confundir un modelo del dominio en lugar de aclararlo.
- Su descubrimiento puede llevar tiempo, con beneficio marginal.
- Evite mostrar asociaciones redundantes o derivadas.

## **Contratos**

Son una forma de describir el comportamiento en un sistema en forma detallada.

- Describen pre (antes) y post (después) condiciones.

## **Secciones de un contrato**

- Operación: se detalle el nombre de la operación y los parámetros
  - Pre-condiciones: Suposiciones relevantes sobre el estado del sistema o de los objetos del Modelo del Dominio, antes de la ejecución de la operación

- Post-condiciones: el estado del sistema o de los objetos del Modelo del Dominio, después de que se complete la ejecución de la operación.
  - Describen cambios que deben ocurrir. Por ejemplo: Creación de elementos, creación de asociaciones, destrucción de elementos o asociaciones.
  - Son declarativas: afirmaciones que deben ser verdaderas luego de la ejecución de la operación.

### **Precondiciones:**

- Suposiciones relevantes sobre el estado del modelo antes de la ejecución de la operación.
- No se validarán dentro de la operación, sino que se asumirán válidas.
- Son suposiciones no triviales.
- Ejemplo:
  - La cantidad de días de contratación de un servicio prolongado es mayor a 1.

### **Postcondiciones:**

- Describen cambios en los objetos del modelo de dominio.
- Modificación del valor de atributos.
- Creación o eliminación de instancias.
- Creación o ruptura de asociaciones.
- Son declarativas.
- Ejemplo
  - El cliente posee una nueva contratación

### **Test de Unidad:**

- Los contratos son una antesala a los test de unidad.
- Definen los requerimientos en términos de pre y post condiciones.
- Las pre-condiciones dan una idea del fixture del test.
- Las post-condiciones dan una idea de las verificaciones del test (los assert).

## **Del Análisis al Diseño**

Crear diagramas de interacción que muestran cómo los objetos se comunican con el objetivo de cumplir con los requerimientos capturados en la etapa de análisis.

A partir de los diagramas de interacción, diseñar diagramas de clases representando las clases que serán implementadas.

Crear diagramas de interacción requiere la aplicación de Principios o Heurísticas para la Asignación de Responsabilidades

## **Heurísticas para asignación de responsabilidades**

La habilidad para asignar responsabilidades es extremadamente importante para el diseño orientado a objetos.

La asignación de responsabilidades generalmente sucede durante la creación de los diagramas de secuencia.



## Responsabilidades de los objetos

- Hacer
  - Algo por si mismo
  - Iniciar una acción en otros objetos
  - Controlar o coordinar actividades de otros objetos
- Conocer (para hacer)
  - Conocer sus datos privados encapsulados
  - Conocer sus objetos relacionados
  - Cosas derivables o calculables

### Experto

- Asignar una responsabilidad al experto en información (la clase que tiene la información necesaria para realizar la responsabilidad)
- Expresa la intuición de que los objetos hacen cosas relacionadas con la información que tienen
- Para cumplir su responsabilidad, un objeto puede requerir información que se encuentra dispersa en diferentes clases expertos en información “parcial”

Ejemplo:

En el ejercicio de los servicios de limpieza y parquizaciones.

¿Quién tiene la responsabilidad de responder el monto a pagar de todos los servicios contratados?

El Cliente

## Creador

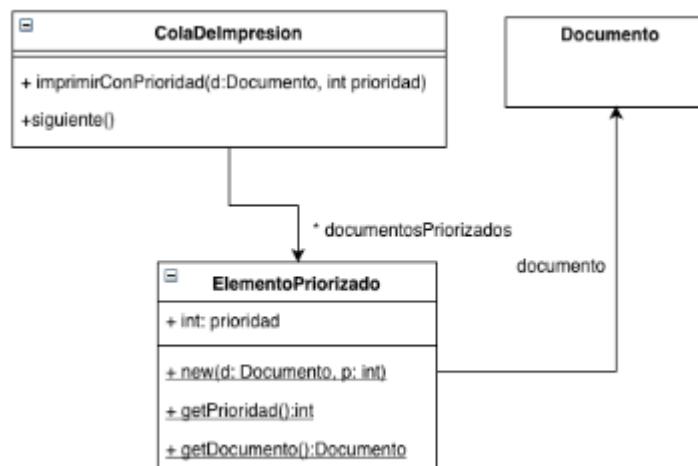
Asignar a la clase B la responsabilidad de crear una instancia de la clase A si:

- B usa objetos A en forma exclusiva
  - Ningún otro sabe de esos objetos
- B contiene objetos A (agregación, composición)
  - En una relación fuerte de composición, no es simplemente que los conoce
- B tiene los datos para inicializar objetos A
  - No necesita que se los pasen por parámetro

B usa objetos A en forma exclusiva

Ejemplo:

Una cola de impresión crea posicionadores



B contiene objetos A (agregación, composición).

Ejemplo:

Una clase instancia su colección de elementos

```
public class Presupuesto {  
    private List<Articulo> articulos;  
  
    public Presupuesto(){  
        this.articulos = new ArrayList<Articulo>();  
    }  
}
```

B tiene los datos para inicializar objetos A.

Ejemplo:

```
public void enviarEmail(){  
    String cuerpo = "Estimado " + cliente.getNombreyApellido()  
                  "su envío de " + items.size()  
                  + "ya se está enviando";  
    Email email = new Email(cliente.getDireccionEMail(),  
                            this.emailFrom(), "Aviso de envío",cuerpo);  
    this.getServidor().enviar(email);  
}
```

## Bajo acoplamiento

- El acoplamiento es una medida de dependencia de un objeto con otros. Es bajo si mantiene pocas relaciones con otros objetos.
- El alto acoplamiento dificulta el entendimiento y complica la propagación de cambios en el diseño.

- No se puede considerar de manera aislada a otras heurísticas, sino que debe incluirse como principio de diseño que influye en la elección de la asignación de responsabilidad.
- Asignar responsabilidad de manera que el acoplamiento se mantenga lo más bajo posible

## **Alta cohesión**

- Asignar responsabilidades de manera que la cohesión se mantenga lo más fuerte posible
- La cohesión es una medida de fuerza con la que se relacionan las responsabilidades de un objeto, y la cantidad de ellas
- Ventaja: clases más fáciles de mantener, entender y reutilizar
- El nivel de cohesión no se puede considerar de manera aislada a otras responsabilidades y otras heurísticas, como Experto y Bajo acoplamiento

## **Modelo de Diseño**

### **Del Análisis al Diseño**

- Los casos de uso sugieren los eventos del sistema que se muestran en los diagramas de secuencia del sistema.
- En los contratos de las operaciones, utilizando conceptos del Modelo del dominio, se describen los efectos que dichos eventos (operaciones) producen en el sistema.
- Los eventos del sistema representan los mensajes que dan inicio a Diagramas de Secuencia del diseño, mostrando las interacciones entre los objetos del Sistema.

- Los objetos con sus métodos y relaciones se muestran en el Diagrama de Clases del Diseño (basado en el modelo del Dominio)

## Diagramas de Secuencia

- Cree un diagrama de secuencia por cada operación asociada al caso de uso.
- Si el diagrama queda complejo, sepárelo en diagramas menos complejos (uno por cada escenario).
- Use el contrato de la operación como punto de partida; piense en objetos que colaboran para cumplir la tarea (la mayoría de estos objetos están definidos en el modelo del dominio).
- Aplique las Heurísticas para Asignación de Responsabilidades (HAR) para obtener un mejor diseño.

Ejemplo:

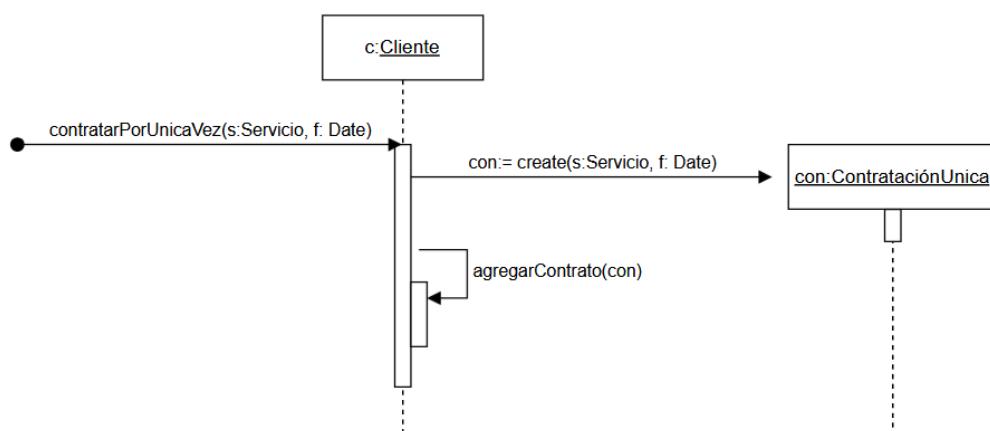
Operación: contratar servicio por única vez ( c: Cliente, fecha: Date, s: Servicio)

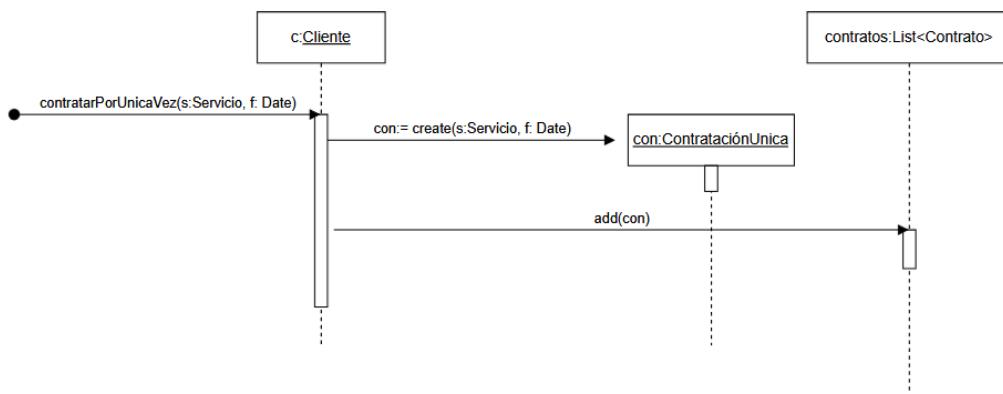
Pre-condición:

la fecha es una fecha válida para el contexto.

Post-condición

el cliente posee una contratación por única vez para el servicio s y la fecha indicada.





## Creación de Diagramas de Clases

- Identificar las clases que participan en los diagramas de interacción y en el Modelo del Dominio o Conceptual
- Graficarlas en un diagrama de clases
- Colocar los atributos presentes en el Modelo Conceptual
- Agregar nombres de métodos analizando los diagramas de interacción
- Agregar tipos y visibilidad de atributos y métodos
- Agregar las asociaciones necesarias
- Agregar roles, navegabilidad, nombre y multiplicidad a las asociaciones

## Transformación de los diseños en código

Mapear los artefactos de diseño a código orientado a objetos:

Clases	Clases
Atributos	Variables simples
Asociaciones (roles) ➔	Variables de referencia
Métodos	Métodos

## Extendiendo modelo conceptual

Nuevos conceptos pueden ser identificados y agregados al modelo.

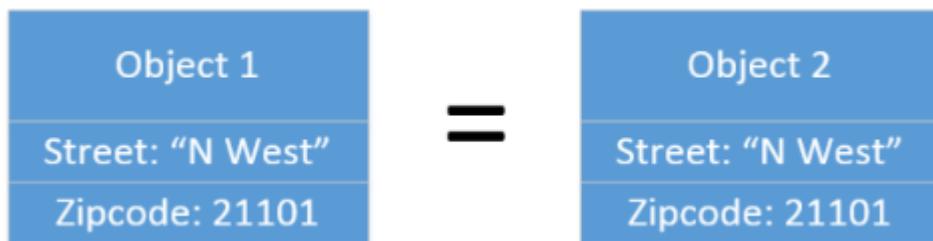
Descubriendo nuevas clases para evitar preguntar por el tipo o valor de un atributo. Polimorfismo

## Clases Conceptuales: "Entity vs. Value Object"

- Las entidades o clases del dominio de mi problema tienen un identificador, son modificables y comparables por identidad.

Value Object:

- Son comparables por contenido (igualdad estructural), no tienen identificador.



- No viven por si mismos, necesitan una entidad base, son intercambiables (un billete de 100 Pesos AR lo puedo cambiar por otro). Persisten adjunto a su base, no separadamente.
- Inmutables (no le defino setters).
- Si un value Object no es inmutable, entonces No es un value object
- ¿Cómo identificar un Value Object en el modelo?

Cuando necesitamos por ej. o delar Moneda, Fecha, Dirección, que puedan tener cierto comportamiento (getters..)

¿Cómo representar "Value Objects" en el modelo?



**<<Value Object>> delante del nombre**

## Heurísticas para Diseño “ágil” Orientado a objetos (Principios S O L I D)

### Principios para Diseño “ágil” Orientado a Objetos

#### Principios S O L I D

Relacionados a las HAR, para un buen estilo de DOO. Promueven Alta Cohesión y Bajo Acoplamiento

##### **S SRP: The Single-Responsibility Principle**

Principio de Responsabilidad única. Una clase debería cambiar por una sola razón.

Debería ser responsable de únicamente una tarea, y ser modificada por una sola razón (alta cohesión).

##### **O OCP: The Open-Closed Principle**

Entidades de software (clases, módulos, funciones, etc.) deberían ser “abiertas” para extensión, y “cerradas” para modificación.

Abierto a extensión: ser capaz de añadir nuevas funcionalidades.

Cerrado a modificación: al añadir la nueva funcionalidad no se debe cambiar el diseño existente.

## **L LSP: The Liskov Substitution Principle**

Los objetos de un programa deben ser intercambiables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa.

Es decir, que si el programa utiliza una clase (clase A), y ésta es extendida (clases B, C, D, etc...) el programa tiene que poder utilizar cualquiera de sus subclases y seguir siendo válido. Uso correcto de herencia (Is-a) y polimorfismo.

## **I ISP: The Interface-Segregation Principle**

Las clases que tienen interfaces "voluminosas" son clases cuyas interfaces no son cohesivas.

Las clases no deberían verse forzadas a depender de interfaces que no utilizan. Cuando creamos interfaces (protocolos) para definir comportamientos, las clases que las implementan, no deben estar forzadas a incluir métodos que no va a utilizar.

## **D DIP: The Dependency-Inversion Principle**

a. Los módulos de alto nivel de abstracción no deben depender de los de bajo nivel.

b. Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.

Módulos de alto nivel: se refieren a los objetos que definen qué es y qué hace el sistema.

Módulos de bajo nivel: no están directamente relacionados con la lógica de negocio del programa (no definen el dominio). Por ejemplo, el mecanismo de persistencia o el acceso a red .

Abstracciones: se refieren a protocolos (o interfaces) o clases abstractas.

Detalles: son las implementaciones concretas, (cuál mecanismo de persistencia, etc).

Ser capaz de «invertir» una dependencia es lo mismo que ser capaz de «intercambiar» una implementación concreta por otra implementación concreta cualquiera, respecto a la misma abstracción.

# Reuso de Código

## Herencia vs. Composición

### Herencia de Clases

- Herencia total: debo conocer todo el código que se hereda → Reutilización de Caja Blanca.
- Usualmente debemos redefinir o anular métodos heredados
- Los cambios en la superclase se propagan automáticamente a las subclases
- Herencia de Estructura vs. Herencia de comportamiento
- Es útil para extender la funcionalidad del dominio de aplicación

### Composición de Objetos

- Los objetos se componen en forma Dinámica → Reutilización de Caja Negra
- Los objetos pueden reutilizarse a través de su interfaz (sin conocer el código)
- A través de las relaciones de composición se pueden delegar responsabilidades entre los objetos

### Herencia vs. Composición

- Las clases y los objetos creados mediante herencia están estrechamente acoplados ya que cambiar algo en la superclase afecta directamente a la/las subclases.
- Las clases y los objetos creados a través de la composición están débilmente acoplados, lo que significa que se pueden cambiar más

fácilmente los componentes sin afectar el objeto contenedor.

### **Herencia de Clases**

- Herencia total: debo conocer todo el código que se hereda → Reutilización de Caja Blanca
- Usualmente debemos redefinir o anular métodos heredados
- Los cambios en la superclase se propagan automáticamente a las subclases
- Herencia de Estructura vs. Herencia de comportamiento
- Es útil para extender la funcionalidad del dominio de aplicación

### **Composición de objetos**

- Los objetos se componen en forma Dinámica → Reutilización de Caja Negra
- Los objetos pueden reutilizarse a través de su interfaz (sin conocer el código)
- A través de las relaciones de composición se pueden delegar responsabilidades entre los objetos

## **Cómo solucionar un mal uso de la herencia**

Composición, no herencia...

- Mecánicamente, heredar de ArrayList no cumple con el encapsulamiento; en cambio,
- componer con ArrayList para contener la colección de objetos de la pila es una opción de implementación que permite ocultarla públicamente.
- En este caso, en vez de heredar, el uso o composición permite reuso y mantiene el encapsulamiento!!

# Testing

## ¿Qué es un bug/error?

- El programa no hace algo que debería hacer
- El programa hace algo mal
- El programa falla (revienta)

## ¿Qué es testear?

Asegurarse de que el programa:

- hace lo que se espera
- lo hace como se espera
- no falla

## ¿Por qué no testeamos (o lo hacemos mal)?

- Lo dejamos para el final (¿para no trabajar de gusto?)
- Hay muchas combinaciones que considerar
- Requiere planificación, preparación y recursos adicionales
- Es una tarea repetitiva, y nos aparece poco interesante
- Creemos que es tarea de otro, nosotros programamos (;?)
- Creemos que alcanza con "programar bien"
- El objetivo de testear es encontrar bugs (¿Será que eso nos molesta?)

## Tipos de test

- Test funcionales
- Test no funcionales

- Test de unidad
- Test de integración
- Test de regresión
- Test de punta a punta
- **Test automatizados**
- Test de carga
- Test de performance
- Test de aceptación
- Test de UI
- Test de accesibilidad
- Alpha y beta test
- Test a/b
- ...

### **Test de unidad**

- Test que asegura que la unidad mínima de nuestro programa funciona correctamente, y aislada de otras unidades
  - En nuestro caso, la unidad de test es el método
- Testear un método es confirmar que el mismo acepta el rango esperado de entradas, y que retorna el valor esperado en cada caso
  - Tengo en cuenta parámetros
  - Estado del objeto antes de ejecutar el método
  - objeto que retorna el método
  - estado del objeto al concluir la ejecución del método

### **Test automatizados**

- Se utiliza software para guiar la ejecución de los test y controlar los resultados

- Requiere que diseñemos, programemos y mantengamos programas “tests”
  - En nuestro caso, esos programas serán objetos
- Suele basarse en herramientas que resuelven gran parte del trabajo
- Una vez escritos, los puedo reproducir a costo mínimo, cuando quiera
- Los tests son “parte del software” (y un indicador de su calidad)

## jUnit

- jUnit es una herramienta para simplificar la creación de tests de unidad y automatizar su ejecución y reporte
- Ayuda a escribir/programar tests útiles
- Cada test se ejecuta independientemente de otros (aislados)
- jUnit detecta, recolecta, y reporta errores y problemas
- xUnit es su nombre genérico; lo que aprendamos podemos llevarlo a otros lenguajes

## Anatomía de un test suite jUnit

- Una clase de test por cada clase a testear
- Un método que prepara lo que necesitan los tests (el fixture)
- Y queda en variables de instancia
- Uno o varios métodos de test por cada método a testear
- Un método que limpia lo que se preparó (si es necesario)

## Independencia entre tests

- No puedo asumir que otro test se ejecutó antes o se ejecutará después del que estoy escribiendo
- Por cada método de test (marcado con @Test):
  - Se crea una nueva instancia de nuestra clase de test
  - Se prepara (con el método marcado como @BeforeEach)
  - Se ejecuta el test y se registran errores y fallas

## Cobertura

- Cuando testeamos nos interesa saber cuan complejos/integrales son nuestros tests — podemos medirlo de diferentes formas
  - Clases cubiertas, métodos cubiertos, líneas cubiertas
  - Condicionales (ver que se ejecutaron con true y false)
  - Caminos/branches (ver si pasó por todos lados)
- Las herramientas modernas observan y reportan esos y otros valores
- Como escribir y mantener tests requiere esfuerzo no siempre maximizamos su cobertura
- Diseñar bien nuestros tests nos ayuda a enfocar el esfuerzo, optimizar el resultado y obtener un balance adecuado esfuerzo/cobertura

## ¿Por qué, cuándo y cómo testear?

- Testeamos para encontrar bugs
- Testeamos con un propósito (buscamos algo)
- Pensamos por qué testear algo y con qué nivel queremos hacerlo
- Testeamos temprano y frecuentemente
- Testeo tanto como sea el riesgo del artefacto
- No es necesario testear código de base que otros ya testearon (por ejemplo, partes del SDK, etc.)

## Estrategia general

- Pensar que podría variar (que valores puede tomar) y que pueda causar un error o falla
- Elegir valores (de estados y parámetros) de prueba para maximizar las chances de encontrar errores haciendo la menor cantidad de pruebas posibles
  - Una combinación de valores es "un caso de prueba"
- Nos vamos a enfocar en dos estrategias:
  - Particiones equivalentes
  - Valores de borde

### **Test de particiones equivalentes**

- Partición de equivalencia: conjunto de casos que prueban lo mismo o revelan el mismo bug
  - Asumo que si un ejemplo de una partición pasa el test, los otros también lo harán. Elijo uno.
- Si se trata de casos en un conjunto, tomo un caso que pertenezca al conjunto y uno que no
  - Ej., debe tener entrada → Casos: una persona con entrada, una sin
- Si se trata de valores en un rango, tomo un caso dentro y uno por fuera en cada lado del rango
  - Ej., la temperatura debe estar entre 0 y 100 - > casos: -50, 50 , 150. Veremos que estos casos pueden mejorarse

### **Test con valores de borde**

- Los errores ocurren con frecuencia en los límites y ahí es donde los vamos a buscar
- Intentamos identificar bordes en nuestras particiones de equivalencia y elegimos esos valores
- Buscamos los bordes en estados/parámetros: velocidad, cantidad, posición, tamaño, duración, edad, etc.
  - También podemos buscar relaciones entre ellas (diferencias entre saldo y monto a extraer)

- Y buscar valores como: primero/último, máximo/mínimo, arriba/abajo, principio/fin, vacío/lleno, antes/después, junto a, alejado de , etc.

## Testing en OO1

- En el marco de OO1, testear es asegurarnos de que nuestros objetos hacen lo que se espera, como se espera
- Escribir tests de unidad (con JUnit) es parte “programar”
- Escribir tests nos ayuda a entender que se espera de nuestros objetos
- Con lo que sabemos hasta ahora encontraremos situaciones complejas de resolver
  - Ya veremos en OO2 estrategias para atacarlas
  - Por ahora el foco es testear con propósito, y diseñar bien los tests/casos

## Smalltalk

- Lenguaje OO puro — todo es un objeto (incluso las clases)
- Tipado dinámicamente
- Propone una estrategia exploratoria (construcciónista) al desarrollo de software
- El ambiente es tan importante como el lenguaje
  - Está implementado en Smalltalk
  - Ricas librerías de clases (fuentes de inspiración y ejemplos)
  - Todo su código fuente disponible y modificable
  - Tiene su propio compilador, debugger, editor, inspector, perfilador, etc

- Es extensible
- Sintaxis minimalista (con sustento en su foco educativo)
- Fuente de inspiración de casi todo lo que vino después (en OO)

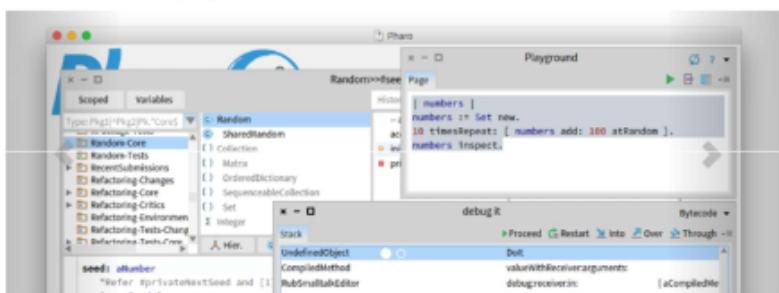
[Pharo](http://pharo.org)

News Features Download Documentation Community Contribute Stories About

# PharO

The immersive programming experience

Pharo is a pure object-oriented programming language *and* a powerful environment, focused on simplicity and immediate feedback (think IDE and OS rolled into one).



Discover      Download      Learn

Ejemplos:

```

"Asignación y terminación"
dificil := false .

"Mensajes unarios; solo el objeto receptor, sin parámetros"
3 squared .
'Hola' reversed .

"Mensajes binarios; objeto receptor y un parámetro"
'Hola', ' Manola' .
1 @ 10 .

"Mensajes de palabra clave; objeto receptor y n parámetros"
'Manola' includesSubstring: 'ola' .
'Hola' copyWithoutAll: 'ol'.
3 between: 1 and: 2 .

```

```

"Definición de clases e instanciación"

Object subclass: #Persona
instanceVariableNames: 'nombre apellido edad'
classVariableNames: ''
package: '001' .

Persona compile: 'initialize
edad := 0.' classified: 'initialization' .

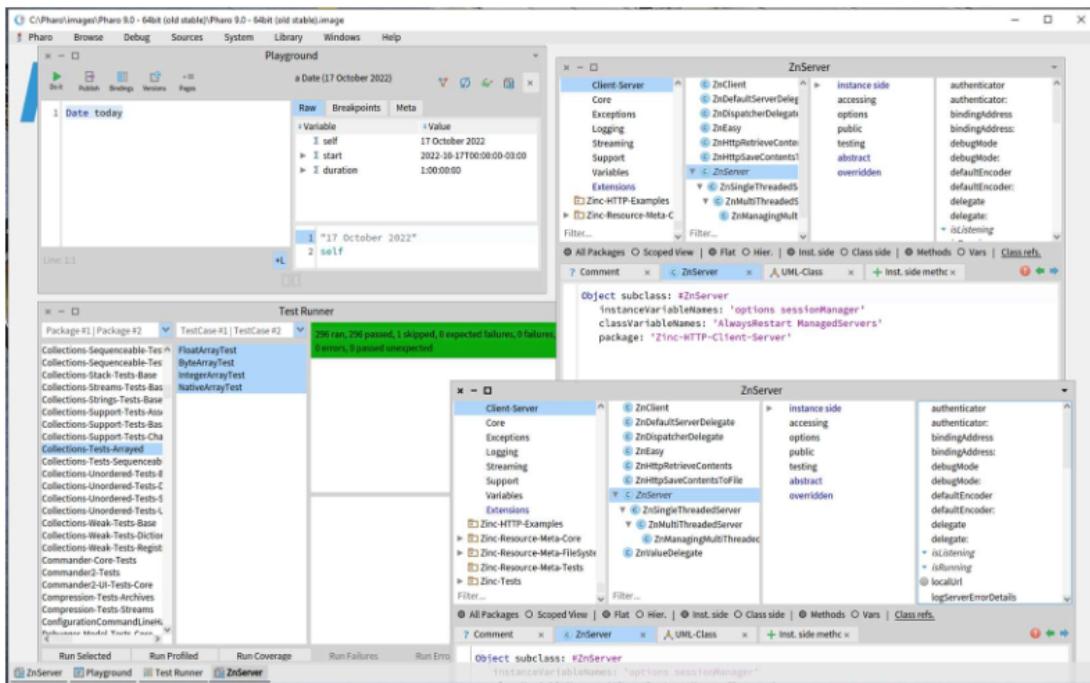
Persona compile: 'nombre: elNombre apellido: elApellido
nombre := elNombre.
apellido := elApellido.' classified: 'accessing' .

Persona compile: 'getNombreCompleto
^ nombre , ' ' ', apellido.' classified: 'accessing' .

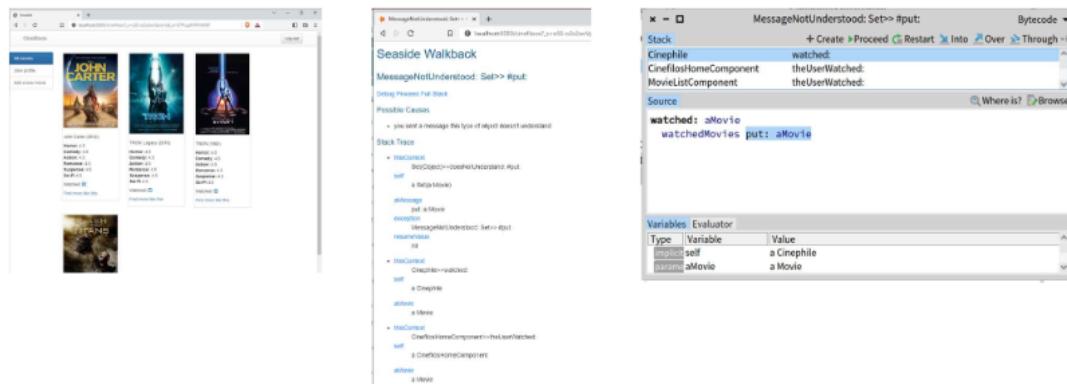
alguien := Persona new.
alguien nombre: 'Juan Carlos' apellido: 'Batman' .

```

## Ambiente:



Hot repair ... (objetos vivos, siempre)



## Clausuras (Closures)

```
nombre := 'Juan Gomez'.
aBlockClosure := [ Transcript show: 'Hola ', nombre; cr ].
aBlockClosure value.

button := PluggableButtonMorph new .
button label: 'Click me'.
button position: 400@10.
button actionBlock: aBlockClosure.
button openInWorld .
```

### Que devuelven...

```
[ 1 < 10 ] value.
```

```
aBlockClosure := [ | temp |
                    temp := OrderedCollection new.
                    temp add: (Random new next).
                    temp ].
aBlockClosure value.
aBlockClosure value.
```

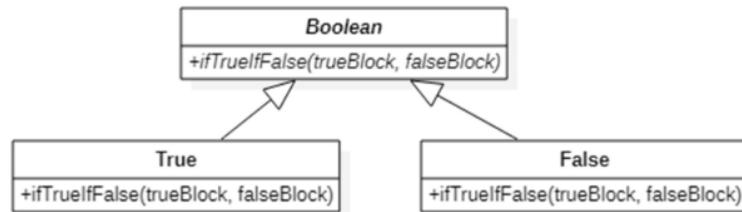
### Con parámetros

```
aBlockClosure := [ :algo | algo size ].
aBlockClosure value: 'hola'.
aBlockClosure value: Set new.
```

```
aBlockClosure := [ :a :b | a < b ].
aBlockClosure value: 1 value: 2.
```

## IF con objetos (puro polimorfismo)

```
(a < 100)
    ifTrue: [Transcript shown: 'a es MENOR a 100']
    ifFalse: [Transcript shown: 'a es MAYOR a 100' ]
```



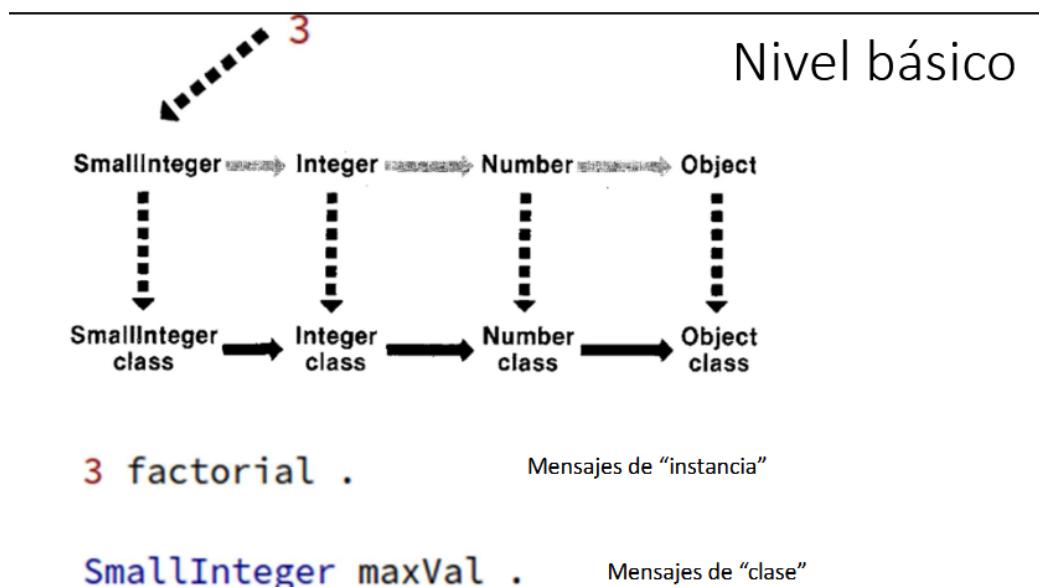
## Las clases son objetos

- Hay dos tipos de objetos: los que pueden crear instancias (de si mismos), y los que no
  - A los primeros les llamamos clases
- Si las clases entienden mensajes, tienen su propio conocimiento y comportamiento
- Esto (el metamodelo de Smalltalk) es uno de sus aspectos mas interesantes y desafiantes

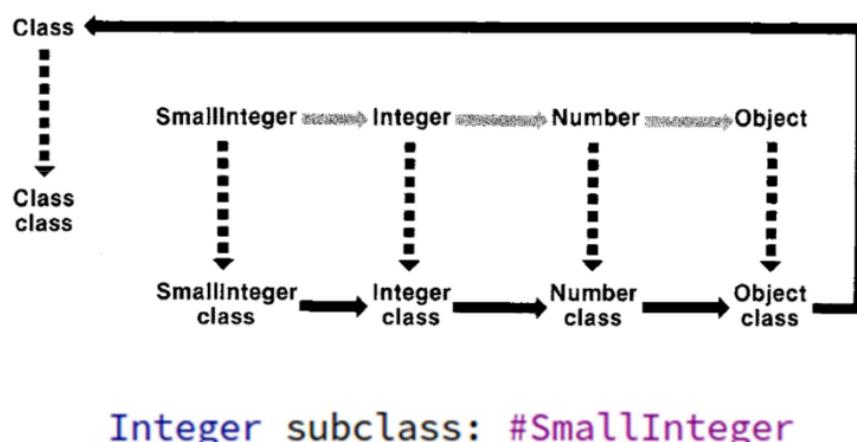
## Lo básico

- Hay objetos capaces de crear instancias y descubrir su estructura y comportamiento: las clases (p.e., `SmallInteger`).
- Todo objeto es instancia de una clase (p.e., `1` de `SmallInteger`)
- Las clases son instancias de una clase también (su metaclasses)
  - Por cada clase hay una metaclass (se crean juntas)

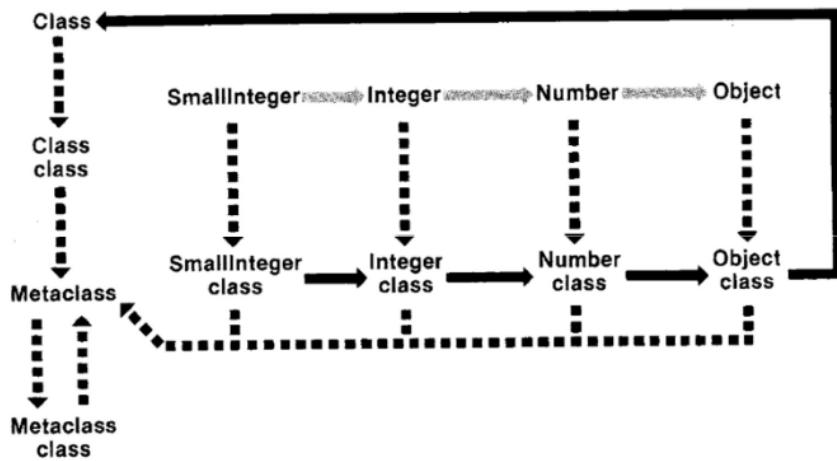
- SmallInteger es instancia de "SmallInteger class"
- Las metaclasses son instancias de la clase Metaclass
  - "SmallInteger class" es instancia de Metaclass



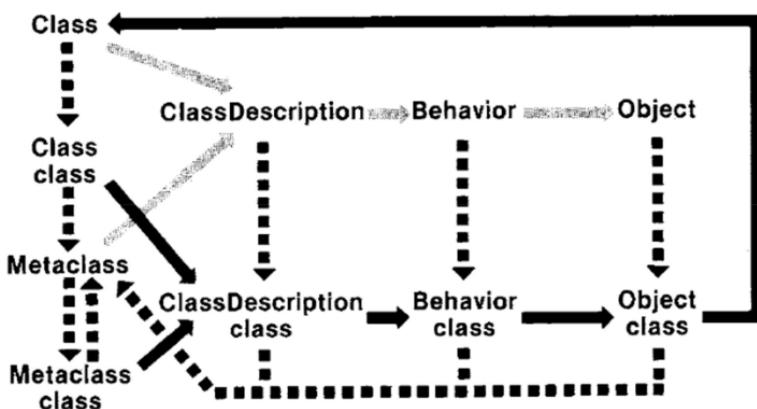
## Nivel avanzado



## Nivel experto



## Nivel super mega requete experto



## JavaScript (ECMAScript)

- Lenguaje de propósito general
- Dinámico
- Basado en objetos (con base en prototipos en lugar de clases)
- Multiparadigma

- Se adapta a una amplia variedad de estilos de programación
- Con una fuerte adopción en el lado del servidor (NodeJS)

## Un mínimo de sintaxis

```

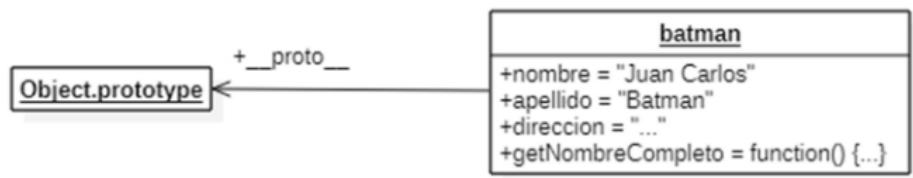
sumar = function(a,b) { return a + b }
sumar(1,2)
function restar(a, b) { return a-b }
batman = { nombre: 'Juan Carlos', apellido: 'Batman', edad: 50 }
batman.edad = 44
batman.direccion = "Baticueva 14, entre Gallos y Medianoche"
batman.getNombreCompleto = function()
    {
        return this.nombre
    }
batman.getNombreCompleto()

```

## Prototipos

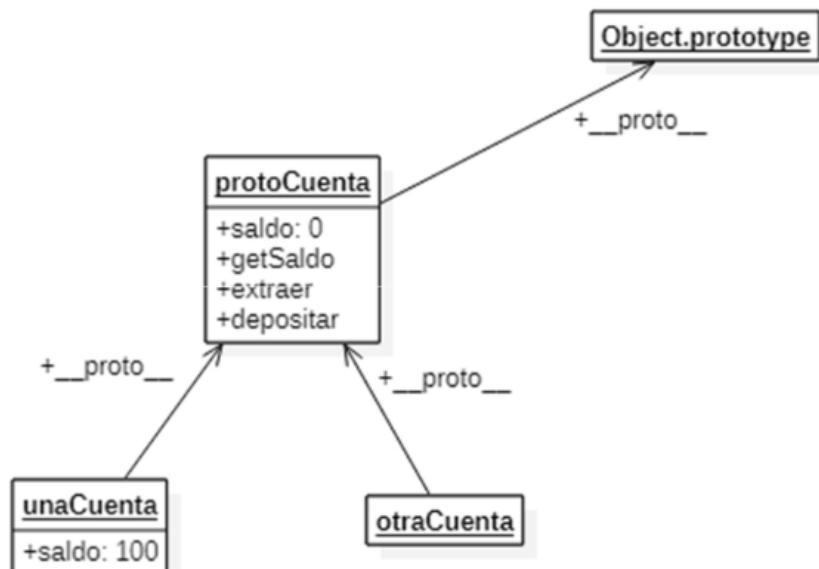
- En Javascript no tengo clases
- La forma mas simple de crear un objeto es mediante la notación literal (estilo JSON)
- Cada objeto puede tener su propio comportamiento (métodos)
- Los objetos heredan comportamiento y estado de otros (sus prototipos)
- Cualquier objeto puede servir como prototipo de otro
- Puedo cambiar el prototipo de un objeto (y así su comportamiento y estado)
- Término armando cadenas de delegación

**\_\_proto\_\_**



```

batman = { nombre: 'Juan Carlos', apellido: 'Batman',
           edad: 53}
batman.getNombreCompleto = function()
{ return this.nombre + " " + thi
robin = {}
robin.__proto__ = batman;
  
```

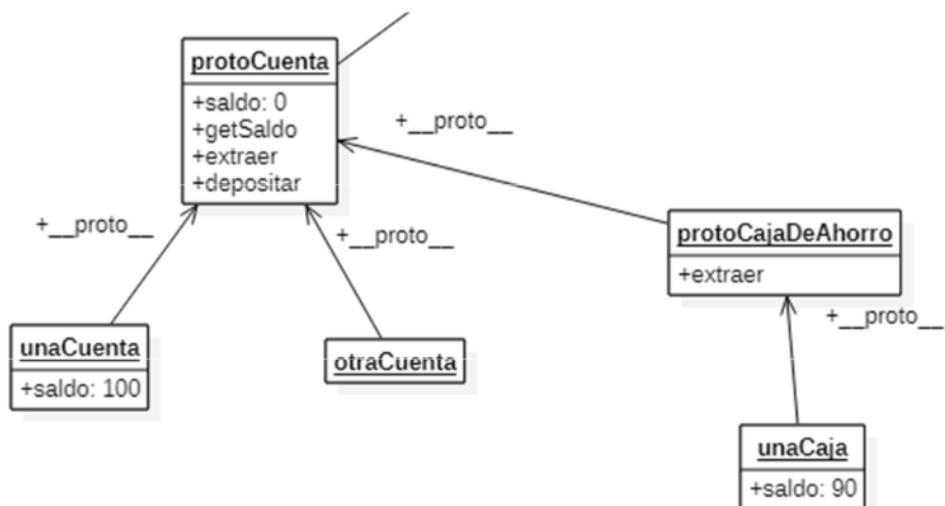


```

> protoCuenta = { saldo: 0}
< ↪ {saldo: 0}
> protoCuenta.getSaldo = function() { return this.saldo }
< ↪ f () { return this.saldo }
> protoCuenta.depositar = function(monto) { this.saldo = this.saldo + monto }
< ↪ f (monto) { this.saldo = this.saldo + monto }
> protoCuenta.extraer = function(monto) { this.saldo = this.saldo - monto }
< ↪ f (monto) { this.saldo = this.saldo - monto }
> unaCuenta = Object.create(protoCuenta)
< ↪ {}
> unaCuenta.getSaldo()
< 0
> unaCuenta.depositar(100)
< undefined
> unaCuenta.getSaldo()
< 100

```

## Prototipos



```

> protoCajaDeAhorro = Object.create(protoCuenta)
< > {}
> protoCajaDeAhorro.extraer = function(monto) {
    if (monto < this.saldo) {
        this.saldo = this.saldo - monto;
    }
}
< f (monto) {
    if (monto < this.saldo) {
        this.saldo = this.saldo - monto;
    }
}
> unaCajaDeAhorro = Object.create(protoCajaDeAhorro)
< > {}
> unaCajaDeAhorro.getSaldo()
< 0
> unaCajaDeAhorro.extraer(10)
< undefined
> unaCajaDeAhorro.getSaldo()
< 0
> unaCajaDeAhorro.depositar(100)
< undefined
> unaCajaDeAhorro.extraer(10)
< undefined
> unaCajaDeAhorro.getSaldo()
< 90

```

## Prototipos y “herencia”

## Funciones que son objetos y constructores

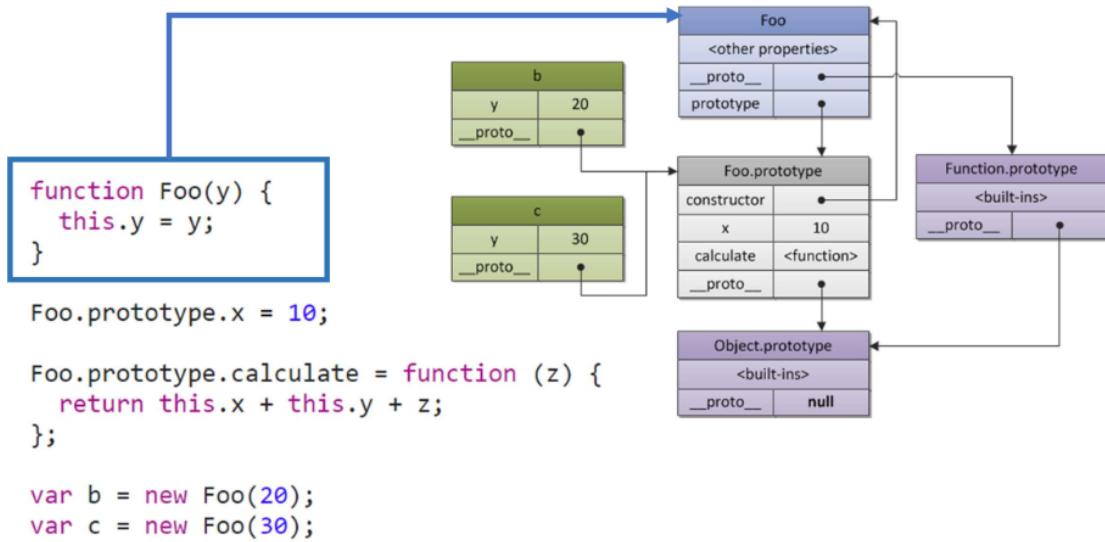
```

function Persona(nombre, apellido){
    this.apellido = apellido,
    this.nombre = nombre
}
juan = new Persona("juan", "gomez")

```

¿Que prototipo tienen esas personas?

¿Cómo lo uso para agregar datos comportamiento que hereden todas?



## Objetos y Persistencia

### Persistencia

#### ¿Por qué / donde guardo los objetos?

Problemas posibles:

- ¿Dónde puedo guardar objetos?
  - Memoria/Imagen: no permite distribución
  - Archivos: poco soporte y utilidades (ej. no hay índices, transacciones)
  - BBDD (Relacionales): diferencia de impedancia con sistemas OO
  - BDOO: pocas opciones, poca experiencia/recursos

### En memoria

En memoria los objetos pueden guardarse.

En ciertos casos hasta tiene sentido - hoy no existen BDD que se usan como cache y durabilidad.

Hay dos problemas que no se resuelven:

- durabilidad
- distribución, y con ella la escalabilidad

## **Archivos**

Archivos:

- Solución simple y durable
- Se podría incluso distribuir

Pero

- No hay integración con OO, hay que desarrollarla
- Poco soporte para performance y distribución (ej. no hay índices, transacciones)

Estas soluciones pueden servir en casos específicos pero no escalan

Sirven para ver limitaciones y motivos de utilizar otras soluciones de persistencia

Veamos las más realistas

## **Bases de Datos**

RDBMS

Bases de Datos relacionales, se pueden usar y tienen ventajas:

- SQL es un estándar que permite migrar fácilmente
- Es un paradigma muy conocido y utilizado
- Hay muchos sistemas "legacy"

## **ORM**

Diferencia de Impedancia

## Diferencia de OO respecto de RDBMS

- OO no contempla normalmente manejo de tx
- Se representan grafos de objetos sin límite aparente

## Diferencias de RDBMS respecto de OO

- Cuando hay múltiples relaciones y recursión no es eficiente
- Recuperar datos dispersados incurre en varios JOINs
- No soporta Jerarquías - se traduce en más JOINs

Se podría pensar que estamos volviendo al paradigma procedural → separando comportamiento de los datos

No exactamente:

- Seguimos programando OO
- Buscamos olvidarnos de la BD subyacente

## **DEMO 01 (hibernate + MySQL)**

### Persistencia por alcance

1. Clase y Mapping
2. Main Save (naive)
3. Save - persistencia por alcance

Reduce el nro de saves

Ideal para independencia

## **Desafíos**

Las ORMs tienen varios desafíos

La diferencia de impedancia es solo uno de ellos

Hay otros, tanto conceptuales (ej., independencia) como técnicos (performance).

## **Principio de Independencia**

Si vemos un código típico (aunque antiguo) de Hibernate encontramos:

- H/SQL embebido
- Annotations
- TX explícitas
- “Saves”

Todo esto va contra el principio de independencia

### **Performance - Caching**

Ejemplo - Hibernate Cache Nivel 1

Cuando una entidad se carga o actualiza por primera vez en una sesión, se almacena en la caché a nivel de sesión.

Solicitudes posteriores de la misma entidad en la misma sesión se sirven desde la caché

Se minimizan así accesos a la BD

### **BBDD Orientadas a Objetos**

Resuelven casi todo lo ligado a la diferencia de impedancia en RDBMS

- Ejemplo: Gemstone/S

### **Familias NoSQL**

Clave-Valor (Redis, DynamoDB, Riak)

- estilo tablas hash, el valor suele ser libre - no da mucho lugar a queries
- útiles para guardar información básica (sin relaciones externas) con pocos updates
- casos de uso frecuentes: sesiones y cachés

Documentos

- agregaciones con estructura - permite queries
- sin schema fijo

### **Demo 02**

En esta demo vemos un mapeador OO → MongoDB

1. Proyecto

2. Clase (isVoRoot)
3. Persistencia por Alcance