

Programación Orientada a Objetos

Encapsulamiento, `static` y records

Invariantes, paquetes, inmutabilidad y pruebas con JUnit

Gerson Garrido

Semana 3

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Encapsulamiento: acceso, invariantes y validación
2. Getters, setters y copias defensivas
3. `static`: lo que es de la clase
4. Paquetes
5. Inmutabilidad y `record`
6. Laboratorio: unit tests con JUnit

> **Lectura**
> Eck (v. 9): 5.1.3, 4.7.1, 8.2, 4.3.7, 4.2.4, 4.8.3, 4.6, 2.6.5, 5.3.1 y 7.4.

SECCIÓN 01

Encapsulamiento

01

¿Basta con `private`?

BankAccount.java · primera versión

```java
public class BankAccount {
    private long balance;

    public void deposit(long amount) {
        balance += amount;
    }

    public long balance() {
        return balance;
    }
}
```

- `account.balance = -500` no compila: el campo es `private`.
- Pero `account.deposit(-500)` sí compila…
- …y deja el saldo en −500, usando solo métodos públicos.
- Esconder el campo es el primer paso. El segundo: que los métodos no acepten cualquier cosa.

Encapsulamiento *(encapsulation)*

La idea

El objeto es el único responsable de su estado. Los demás le piden cosas a través de una **interfaz pública** pequeña, y la clase decide qué pedidos acepta.

- Si la clase está bien escrita, no importa qué haga el resto del programa: el objeto nunca queda en un estado imposible.
- `private` no es seguridad contra personas malintencionadas: es una herramienta para razonar.
- Todo lo que le puede pasar al saldo está en un solo archivo.

Modificadores de acceso

Modificador	Quién puede usar el campo o método
`private`	solo el código de la misma clase
(ninguno)	las clases del mismo **paquete** (acceso de paquete)
`public`	cualquier clase
`protected`	tiene que ver con herencia: semana 5

Regla práctica

Campos: `private`, siempre. Métodos: `public` si son parte de lo que la clase ofrece; `private` si son ayudantes internos.

Invariante de clase *(class invariant)*

> **Invariante**
> Una condición sobre los campos que vale siempre que alguien de afuera mira el objeto.

- El **constructor** lo establece.
- Cada **método público** lo preserva: si valía al entrar, vale al salir.
- A mitad de un método se puede romper un momento; importa al terminar.

- `BankAccount`: `balance >= 0`.
- `Fraction`: denominador positivo y fracción reducida.
  `2/4`, `-3/-6` y `1/2` se guardan igual: `1/2`.
- Escríbanlo en el comentario de la clase: es documentación.

El constructor valida

Fraction.java · nombres abreviados

```java
public Fraction(long num, long den) {
    if (den == 0) {
        throw new IllegalArgumentException(
                "denominador cero");
    }
    if (den < 0) {
        num = -num;
        den = -den;
    }
    long g = gcd(Math.abs(num), den);
    this.numerator = num / g;
    this.denominator = den / g;
}
```

- `throw` detiene el constructor: un objeto inválido nunca llega a existir.
- Lo que se puede arreglar se **normaliza**: signo y reducción.
- `IllegalArgumentException`: “no acepto ese argumento”. Lo demás, en la semana 9.

El error aparece donde se comete

```bash
$ java -cp build FractionDemo
2/4  -> 1/2
1/-3 -> -1/3
suma:     1/6
producto: -1/6
half sigue siendo 1/2
Exception in thread "main" java.lang.IllegalArgumentException: denominador cero
    at Fraction.<init>(Fraction.java:12)
    at FractionDemo.main(FractionDemo.java:11)
```

- Qué pasó: el mensaje. Dónde: `Fraction.<init>` (el constructor), línea 12, llamado desde `main`, línea 11.
- El `null` de la semana 2, en cambio, estalló dos líneas después.

Los métodos también validan

BankAccount.java

```java
public void deposit(long amount) {
    requirePositive(amount);
    balance += amount;
}

public boolean withdraw(long amount) {
    requirePositive(amount);
    if (amount > balance) {
        return false;
    }
    balance -= amount;
    return true;
}
```

- Monto ≤ 0: error de quien llama. Excepción.
- Saldo insuficiente: algo normal. Se informa con `false`, como `pay` en la semana 1.
- En los dos casos, el invariante se mantiene.
- `requirePositive` lanza la excepción; es `private`: un ayudante interno.

Precondiciones, postcondiciones, invariante

Precondición

Lo que el método exige a quien lo llama.

`amount > 0`

Postcondición

Lo que el método promete al terminar.

el saldo aumentó en `amount`

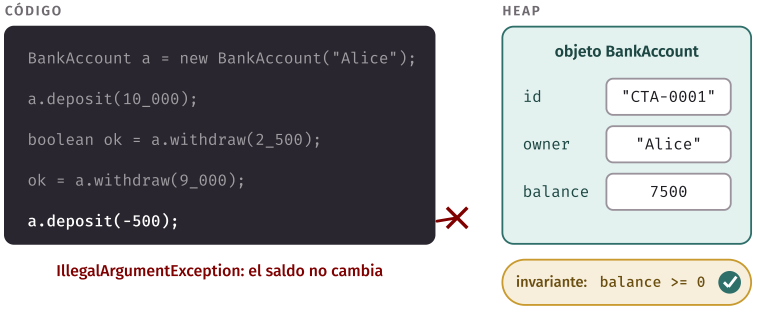
Invariante

Postcondición del constructor; pre- y postcondición de cada método público.

Lectura

Eck 4.7.1 *Preconditions and Postconditions* y 8.2.2–8.2.3.

La vida de una cuenta



5. un pedido inválido se rechaza antes de cambiar el estado

Getters sí, setters con cuidado

no lo hagan

```java
public void setBalance(long b) {
    balance = b;
}
```

- `setBalance(-500)` = `balance = -500`.
- Un setter por campo rompe el encapsulamiento igual que un campo público.
- Y no dice *por qué* cambia el saldo.

Métodos con intención

`deposit` y `withdraw` hablan el lenguaje del problema, y cada uno sabe qué reglas aplicar.

Tell Don't Ask

No: `if (a.balance() >= x) a.setBalance(a.balance() - x)`

Sí: `a.withdraw(x)`

Error común: entregar el arreglo interno

LeakyScores.java

```java
private final int[] values;

public LeakyScores(int[] values) {
    // ... valida cada nota ...
    this.values = values;
}
public int[] values() {
    return values;
}
```

CopyDemo.java

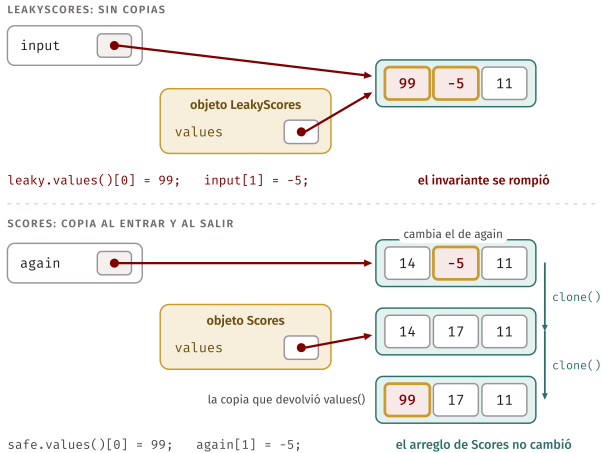
```java
int[] input = {14, 17, 11};
LeakyScores leaky =
        new LeakyScores(input);
leaky.values()[0] = 99;
input[1] = -5;
```

bash · salida recortada

```
$ java -cp build grades.CopyDemo
LeakyScores: máxima 99, ...
```

Notas de 0 a 20, pero queda un 99. El campo es `private`, pero el arreglo es un objeto: el alias de la semana 2.

Sin copias y con copias



Copia defensiva *(defensive copy)*

Scores.java

```java
public Scores(int[] values) {
    // ... valida ...
    this.values = values.clone();
}
public int[] values() {
    return values.clone();
}
```

Copia al entrar y al salir. `CopyDemo` con `Scores`: máxima 17, primera 14, segunda 17.

- `final int[] values` no evitó nada: impide cambiar la flecha, no los elementos.
- Una referencia `final` no hace inmutable al objeto.
- Con `String` no hace falta copiar: no se puede modificar.

SECCIÓN 02

static

02



Un dato de todas las cuentas

BankAccount.java

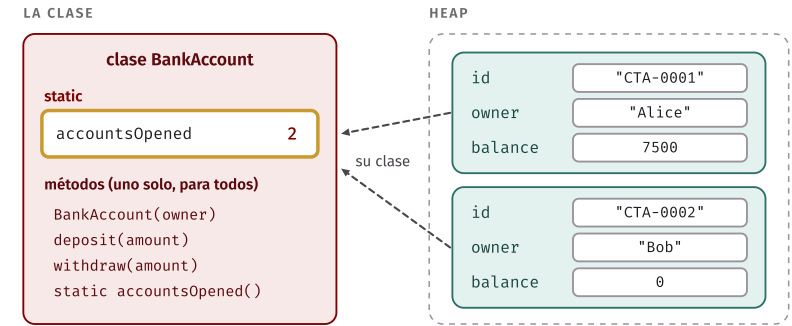
```java
private static int accountsOpened = 0;

private final String id;
private long balance;

public BankAccount(String owner) {
    // ... valida owner ...
    accountsOpened++;
    id = String.format("CTA-%04d",
                       accountsOpened);
}
public static int accountsOpened() {
    return accountsOpened;
}
```

- Un campo `static` existe una sola vez, aunque haya mil cuentas o ninguna.
- Un método `static` se llama con la clase: `BankAccount.accountsOpened()`
- Campos de instancia: uno por objeto.

Instancia y clase



Métodos `static`, constantes y `main`

- Ya los conocen: `Math.max`, `Math.abs`, `String.format`, `IO.println`. Todo lo que usan les llega por parámetro.
- El `gcd` de `Fraction` es `private static`: no usa ningún campo.
- Constantes: `static final`, en mayúsculas.

```
1  public static final int MAX_SCORE = 20;
```

```
el main largo de la semana 1
1  public static void main(String[] args)
```

- `static`: la JVM lo llama antes de que exista ningún objeto. Desde esta semana, las demostraciones se escriben así.

Un método `static` no tiene `this`

StaticError.java

```java
public class StaticError {
    private long balance;

    public static void main(String[] args) {
        balance = 100;
        IO.println(balance);
    }
}
```

bash · salida recortada

```
$ javac -Xlint:all
    StaticError.java
StaticError.java:5: error:
    non-static variable
    balance cannot be
    referenced from a static
     context
...
2 errors
```

- `balance` quiere decir `this.balance`. ¿De qué objeto? De ninguno.
- Al revés no hay problema: un método de instancia sí usa lo `static`.

Error común: `static` “para que compile”

StaticPitfall.java

```java
static class Tally {
    private static int count;
    void click() { count++; }
    int count() { return count; }
}
// ... en main ...
door.click();
door.click();
window.click();
```

bash

```
$ java -cp build StaticPitfall
door: 3, window: 3
```

- `count` es `static`: un solo contador para todos los objetos.
- La solución es crear objetos, no volver `static` lo que no lo es.
- Un `static` que cambia es una variable global. Úsenlo poco.

SECCIÓN 03

Paquetes

03



Paquetes *(packages)*

bank/BankAccount.java

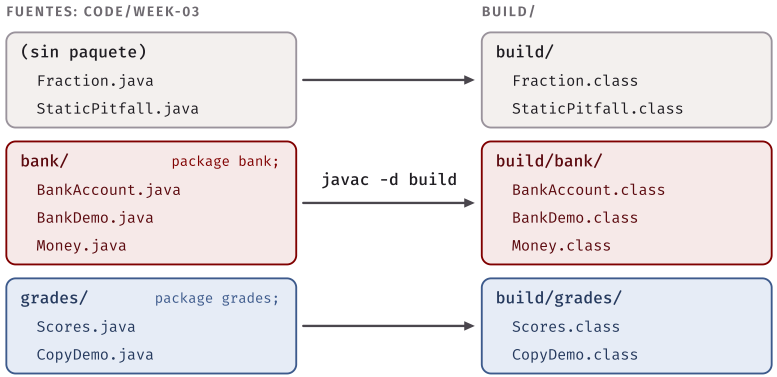
```java
package bank;

public class BankAccount {
    // ...
}
```

Como la biblioteca: `java.lang`, `java.util`, `java.time`.

1. El paquete corresponde a una carpeta: `bank/`. `pe.uni.bodega` estaría en `pe/uni/bodega/`.
2. El nombre completo incluye el paquete: `bank.BankAccount`. Desde otro paquete: `import bank.BankAccount;`
3. `java.lang` se importa solo: por eso `String` no necesita `import`.
4. Sin modificador = acceso de paquete: las clases que colaboran se ayudan sin agrandar su interfaz pública.

Paquetes y carpetas



Error común: olvidar el paquete

```bash
$ java -cp build bank.BankDemo
CTA-0001 Alice: retiro true, saldo 7500
...
$ java -cp build BankDemo
Error: Could not find or load main class BankDemo
Caused by: java.lang.ClassNotFoundException: BankDemo
```

- Sin el paquete, la JVM busca `build/BankDemo.class`, que no existe.
- Las clases sin `package` están en el **paquete sin nombre**: sirve para ejemplos, pero una clase de un paquete con nombre no puede importarlas.
- El `Makefile` de esta semana busca las fuentes en todas las carpetas y ejecuta `bank.BankDemo`. Es el que usaremos de aquí en adelante.

SECCIÓN 04

Inmutabilidad

04



Una clase inmutable *(immutable)*

Receta

Campos `private final`; ningún método modifica el estado; las operaciones devuelven un objeto nuevo; copias defensivas de lo mutable.

Fraction.java · en dos pasos

```java
public Fraction plus(Fraction other) {
    long n = numerator * other.denominator
           + other.numerator * denominator;
    long d = denominator * other.denominator;
    return new Fraction(n, d);
}
```

- `plus` no toca `this`; el constructor de la fracción nueva la reduce.
- Un método que asigne un campo `final` no compila: *cannot assign a value to final variable numerator*.

¿Qué se gana?

- Se puede compartir sin miedo al alias: nadie le cambia el objeto a nadie.
- Quien lo recibe puede guardarlo sin copiarlo. La clase sí copia los componentes mutables que recibe o expone.
- Lo que vale al crearlo vale para siempre: fácil de razonar.
- El precio: un objeto nuevo en cada operación. En Java, casi siempre barato.

Valores y entidades

Inmutables: los valores (una fracción, una cantidad de dinero, una fecha). `String`, `Integer` y `LocalDate` lo son.
Mutables: las entidades que de verdad cambian (una cuenta, un inventario).

SECCIÓN 05

Records

05

`record`: un dato con valor

bank/Money.java

```java
public record Money(long cents, String currency) {
    public Money {      // constructor compacto
        if (cents < 0) {
            throw new IllegalArgumentException(
                    "monto negativo: " + cents);
        }
        // ... valida currency ...
    }
    public static Money soles(long cents) {
        return new Money(cents, "PEN");
    }
    public Money plus(Money other) { ... }
}
```

- La primera línea declara los componentes.
- El **constructor compacto** valida; los campos se asignan solos al final.
- Métodos propios y `static`: `soles` es un **static factory method**.
- Desde Java 16 *(JEP 395)*.

Lo que escribe el compilador

LO QUE ESCRIBIMOS

```
record Money(long cents,
             String currency) {
  public Money { ... }
  static Money soles(long c)
  Money plus(Money other)
}
```

los componentes van entre paréntesis

el constructor compacto se ejecuta al principio del canónico; los campos se asignan al terminar

LO QUE GENERA EL COMPILADOR

```
private final long cents;
private final String currency;
Money(long cents, String currency)
long cents()
String currency()
boolean equals(Object o)
int hashCode()
String toString()
```

un record es inmutable: sus campos son final y no hay setters

Comprobarlo con `javap`

bash · salida recortada

```
$ javap -p -cp build bank.Money
public final class bank.Money extends java.lang.Record {
  private final long cents;
  private final java.lang.String currency;
  public bank.Money(long, java.lang.String);
  ...
  public final java.lang.String toString();
  public final int hashCode();
  public final boolean equals(java.lang.Object);
  public long cents();
  public java.lang.String currency();
}
```

`toString` y `equals`, generados

bash · salida recortada

```
$ java -cp build bank.BankDemo
...
Money[cents=1050, currency=PEN] + S/ 2.50 = Money[cents=1300, currency=PEN]
iguales: true
```

- `toString`: el nombre y los componentes. Mucho mejor que `TransitCard@31221be2`.
- `equals` compara componentes: `total` y `new Money(1_300, "PEN")` son dos objetos distintos, y son iguales.
- Comparen con los dos `Thermometer` de la semana 2: iguales en todo, y `equals` daba `false`.
- El contrato de `equals` y `hashCode`, y cómo escribirlos a mano: semana 5.

¿Record o clase?

`Money`: record

Un valor. Dos `Money(1300, "PEN")` son la misma cantidad.

`BankAccount`: clase

Saldo que cambia, invariante, identidad propia.

`Fraction`: podría

Valor inmutable. La escribimos a mano; `equals` en la semana 5.

Error común

Un record para algo que cambia y tiene reglas: eso es una clase con estado.

SECCIÓN 06

Laboratorio: pruebas con JUnit

06



¿Por qué pruebas?

Un programa de demostración

Alguien tiene que mirar la salida, saber qué debía decir, y volver a mirarla cada vez que el código cambia.

Un unit test

Un método que ejecuta un pedacito del programa y verifica él mismo el resultado. Pasa en silencio; si falla, dice qué esperaba y qué obtuvo.

- Con buenas pruebas, el programa se puede cambiar con confianza.
- **JUnit** (versión 6): el framework de pruebas más usado en Java. Viene en el contenedor.

Anatomía de una prueba

bank/BankAccountTest.java

```java
class BankAccountTest {
    @Test
    void depositIncreasesBalance() {
        // Preparar
        BankAccount account =
                new BankAccount("Alice");

        // Actuar
        account.deposit(10_000);

        // Verificar
        assertEquals(10_000, account.balance());
    }
}
```

- `XTest` junto a `X`, en el mismo paquete. Sin `public`.
- `@Test`: una **anotación**; JUnit la busca.
- El nombre dice qué se espera.
- **Preparar, actuar, verificar**, separados por líneas en blanco.
- `assertEquals` viene de un `import static`.

Assertions

Assertion	Pasa si
`assertEquals(esperado, real)`	son iguales (con `equals` si son objetos)
`assertNotEquals(no esperado, real)`	son distintos
`assertTrue(condición)`	la condición es `true`
`assertFalse(condición)`	la condición es `false`
`assertThrows(E.class, () -> código)`	el código lanza `E`

Error común: invertir `assertEquals`

Primero el **esperado**, después el **real**. Invertidos, la prueba falla igual, pero el mensaje dice lo contrario de lo que pasó.

Una prueba, un comportamiento

- `newAccountStartsAtZero`
- `depositIncreasesBalance`
- `withdrawWithEnoughBalance`
- `withdrawMoreThanBalanceChangesNothing`
- `rejectsNegativeDeposit`
- `rejectsZeroWithdrawal`
- `rejectsBlankOwner`
- `eachAccountGetsADifferentId`

- Leídos en voz alta, los nombres son la especificación de la clase.
- Si falla una prueba que verifica diez cosas, hay que leerla entera.
- Cada prueba crea sus propios objetos: ninguna depende de otra ni del orden.

make test

bash · árbol en ASCII, recortado

```
$ make test
...
+-- JUnit Jupiter [OK]
| +-- BankAccountTest [OK]
| | +-- depositIncreasesBalance() [OK]
| | +-- newAccountStartsAtZero() [OK]
| | +-- ...
| +-- ScoresTest [OK]
| '-- MoneyTest [OK]
...
[        18 tests successful      ]
[         0 tests failed          ]
```

- El *Console Launcher* busca en `build/` las clases con `@Test`.
- No corren en el orden en que se escribieron. No importa.
- *Jupiter* es el motor que ejecuta nuestras pruebas.

Cuando una prueba falla

bash · esperado cambiado a 1_000 a propósito; recortada

```
|   +-- depositIncreasesBalance() [X] expected: <1000> but was: <10000>
...
Failures (1):
  JUnit Jupiter:BankAccountTest:depositIncreasesBalance()
  => AssertionFailedError: expected: <1000> but was: <10000>
     ...Assertions.assertEquals(Assertions.java:673)
     bank.BankAccountTest.depositIncreasesBalance(BankAccountTest.java:29)
...
[         1 tests failed          ]
```

- Qué prueba, qué assertion, qué esperaba, qué obtuvo, en qué línea.
- Aquí el error estaba en la prueba. Casi siempre está en el código, y la prueba dice dónde buscar.

Probar que se rechaza lo inválido

bank/BankAccountTest.java

```java
@Test
void rejectsNegativeDeposit() {
    BankAccount account = new BankAccount("Alice");

    assertThrows(IllegalArgumentException.class, () -> account.deposit(-500));
    assertEquals(0, account.balance());
}
```

- `assertThrows`: la clase de la excepción esperada y el código que debería lanzarla.
- `() ->` arma un pedazo de código que `assertThrows` ejecuta cuando le toca: una **lambda** (semana 12). Por ahora, una receta.
- La segunda assertion: además de rechazar, el saldo no cambió.

Romper el invariante a propósito

Quitamos la validación de `deposit` y corremos `make test`:

bash · salida recortada

```
|   +-- rejectsNegativeDeposit() [X] Expected
        java.lang.IllegalArgumentException to be thrown,
        but nothing was thrown.
...
[         7 tests successful      ]
[         1 tests failed          ]
```

- Falla exactamente una prueba, y su nombre dice qué regla se perdió.
- Si ninguna falla: falta una prueba.
- Así se prueban las pruebas. Después, el código se deja como estaba.

Qué probar

Normales

El uso esperado: un depósito, un retiro que alcanza.

Bordes

Retirar todo el saldo, la lista vacía, las notas 0 y 20, un empate.

Inválidos

Lo que el invariante prohíbe: monto ≤ 0, titular en blanco, nota 21.

`ScoresTest` también prueba la copia defensiva: `changingTheReturnedArrayDoesNotChangeTheScores` fallaría con `LeakyScores`.

Ojo con lo `static` que cambia

`accountsOpened` se comparte entre todas las pruebas, en un orden que no controlamos: la prueba del código no espera `"CTA-0001"`.

El patrón de pruebas del curso

1. `XTest.java` junto a `X.java`, en el mismo paquete, sin `public`.
2. Imports estáticos de las assertions que se usan.
3. Un `@Test` por comportamiento, con un nombre que dice qué se espera.
4. Preparar, actuar y verificar, separados por líneas en blanco.
5. `assertEquals(esperado, real)`.
6. Normales, bordes e inválidos; los inválidos con `assertThrows(..., () -> ...)`.

Desde hoy

Cada clase del curso y de Bodega viene con sus pruebas, escritas así.

El laboratorio

1. Correr `make test` y leer la salida.
2. Escribir `FractionTest`: reducción, signo, denominador cero, `plus`, `times`, que no cambien la original.
3. Romper el invariante (p. ej. quitar la reducción) y ver qué prueba lo detecta.
4. Pasar `Fraction` al paquete `numbers`.

Sin `equals` todavía: comparen las partes.

```
1  assertEquals(1, f.numerator());
2  assertEquals(2, f.denominator());
```

bash · después de mover Fraction

```
./FractionDemo.java:4: error:
  cannot find symbol
  symbol:   class Fraction
```

¿Qué le falta a `FractionDemo`? ¿Y al `Makefile`?

Resumen

- **Encapsulamiento**: campos `private` y una interfaz pública pequeña. El **invariante** lo establece el constructor y lo preserva cada método público.
- Validar y rechazar con `throw new IllegalArgumentException(...)`. Métodos con intención, no setters. **Copias defensivas** para los campos mutables.
- `static`: de la clase, no del objeto. Sin `this`. Constantes `static final`.
- **Paquetes**: una carpeta por paquete; `java -cp build paquete.Clase`.
- **Inmutables**: campos `final`, operaciones que devuelven objetos nuevos. Un `record` los escribe por nosotros, con `equals`, `hashCode` y `toString`.
- **JUnit**: `@Test`, un comportamiento por prueba, preparar-actuar-verificar, `assertEquals`, `assertThrows`.

Para la próxima clase

- **P0** se entrega esta semana. **P1** (Inventario) se publica hoy: clases con invariantes, `ArrayList` y pruebas JUnit escritas por ustedes.
- Lecturas de la semana 4: Eck 3.8 (repaso), 7.1, 7.2.4 y 7.3.
- Terminar `FractionTest` y mover `Fraction` a su paquete.

Pregunta para pensar

Una clase guarda sus productos en un `ArrayList` y tiene un getter que lo devuelve. ¿Qué puede hacer un cliente con esa lista? ¿Cómo lo evitarían?