Ensuring integrity of state

Mario Alviano

Exercise

Realizzare un main creando un menù di scelta come descritto di seguito:

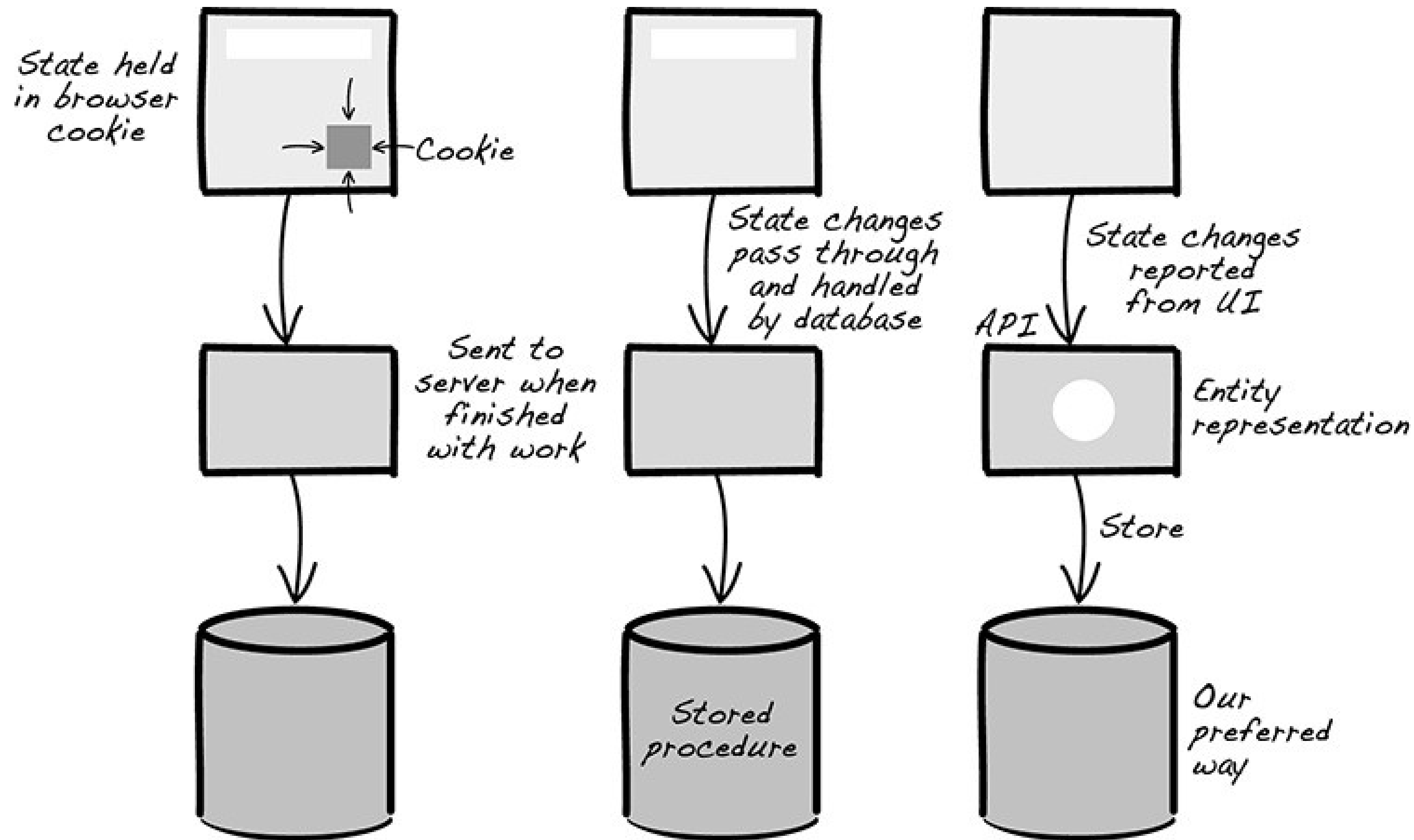
- Premi 1 per aggiungere un'automobile (che legge da input, usando lo Scanner, la targa, la casa produttrice, il nome e il prezzo, crea un'automobile con i dati inseriti e l'aggiunge alla Concessionaria usando il metodo aggiungiVeicolo);
- Premi 2 per aggiungere una moto (che legge da input, usando lo Scanner, la targa, la casa produttrice, il nome e il prezzo, crea una moto con i dati inseriti e l'aggiunge alla Concessionaria usando il metodo aggiungiVeicolo);
- Premi 3 per stampare (che stampa tutti i veicoli presenti nella Concessionaria)
- Premi 4 per ordinare per prezzo (che invoca il metodo di ordinamento per prezzo nella Concessionaria)
- Premi 5 per ordinare per nome (che invoca il metodo di ordinamento per nome nella Concessionaria)
- Premi 6 per stampare il valore (che invoca il metodo calcolaValore di Concessionaria e stampa il risultato)
- Premi 7 per salvare su file (che legge da input, usando lo Scanner, il nome di un file e invoca il metodo salvaSuFile di concessionaria)
- Premi 8 per leggere da file (che legge da input, usando lo Scanner, il nome di un file e invoca il metodo leggiDaFile di concessionaria)
- Premi 9 per uscire

dove si esegue l'operazione relativa quando l'utente sceglie il corrispondente numero.
Ogni operazione (a parte l'uscita) può essere svolta un numero arbitrario di volte.

Commentiamo la soluzione "base".
Ci sono problemi nel codice?
Possiamo gestire i menù come un dominio?

Not everything is immutable

- The state of the system is mutable
  - Items are added to the basket
  - Orders are paid
  - Items are shipped
- In DDD the mutable state is represented by entities
- How to create consistent entities?
  - Entities are more complex than primitive domains
  - Let’s use the **builder pattern**
- We also need to maintain consistency of entities



Several ways of handling the mutable state

Mixing different approaches makes difficult managing the state and exposes to vulnerability

Mutable state must be understood and modeled with entities

Consistency on creation

- An entity that is inconsistent with business logic rules is dangerous for the safety of the system
- Consistency must be present already on creation of the entity
  - We need a method to guarantee such consistency
- Do not underestimate the problem
  - How do I create a new BankAccount?
  - When do I specify the owner?
  - A bank account with no owner may imply that the bank loses its status
  - What do you think about **no-arguments constructor**?

```java
public class Account {
    private AccountNumber number;      ①
    private LegalPerson owner;     ①
    private Percentage interest;     ①
    private Money creditLimit;     ②
    private AccountNumber fallbackAccount;     ②

    public Account() {}     ③

    public AccountNumber getNumber() {
        return number;
    }

    public void setNumber(AccountNumber number) {
        this.number = number;
    }

    public LegalPerson getOwner() {
        return owner;
    }

    public void setOwner(LegalPerson owner) {
        this.owner = owner;
    }
    ...
}

class AccountService {

    void openAccount() {
        Account account= new Account();     ⑤
        account.setNumber(number);     ⑥
        account.setOwner(accountowner);     ⑥
        account.setInterest(interest);     ⑥
        account.setCreditLimit(limit);     ⑦
        ...
    }
}
```

No-arg constructor

1) Mandatory fields
2) Optional fields
5) The entity is inconsistent
6) Consistency after 3 setter
   (be careful to not forget anyone!)

Several issues

Inconsistent entities with
the promise to become consistent

If a mandatory field is added,
all usage of the constructor have to be checked
(compiler cannot report any problem)

ORM and no-arg constructors

- ORM needs no-arg constructors
- How to use ORM with safety?
  1) Conceptually separate the persistent model from the domain model
  2) Directly map domain objects to the persistent framework
- The first approach is safer, the second one is common, but…
  - Use private no-arg constructors
  - Annotate fields instead of providing getter and setter
  - ORM frameworks usually use reflection!

Constructor must specify all mandatory fields

Avoid such a situation

Prefer this approach

```
public class Account {
    private AccountNumber number;
    private LegalPerson owner;
    private Percentage interest;
    private Money creditLimit;
    private AccountNumber fallbackAccount;

    public Account(AccountNumber number,
                   LegalPerson owner,
                   Percentage interest) {        ①
        this.number = notNull(number);         ②
        this.owner = notNull(owner);         ②
        this.interest = notNull(interest);      ②
    }

    protected Account() {}     ③

    public AccountNumber number() { ... }     ④

    public LegalPerson owner() { ... }

    public void changeInterest(
                   Percentage interest) {     ⑤
        notNull(interest);     ⑥
        this.interest = interest;
    }

    public Money creditlimit() { ... }

    public void changeCreditLimit(
                   Money creditLimit) {     ⑦
        notNull(creditLimit);
        this.creditLimit = creditLimit;
    }
```

```
    public void changeFallbackAccount(AccountNumber
fallbackAccount) {
        notNull(fallbackAccount);
        this.fallbackAccount =
Validate.notNull(fallbackAccount);
    }

    public void clearFallbackAccount() {
        this.fallbackAccount = null;
    }
}

class AccountService {
    void openAccount() {
        AccountNumber number = ...
        LegalPerson accountowner = ...
        Percentage interest = ...
        Money limit = ...     ⑧
        Account account =
            new Account(number,
                        accountowner,
                        interest);     ⑨
        account.changeCreditLimit(limit);     ⑩
        accountRepostitory.registerNew(account);
    }
}
```

1-2) Constructor with all mandatory fields and validation
3) If need ORM or the like, private no-arg constructor
4) Access methods are domain-friendly
5-7) Possibility to change mandatory and optional fields
9) Entity is consistent on creation
10) Optional fields are added later

How to handle many fields

- Avoid constructors with 20 argoments
  - Likely, some arguments can be grouped in a domain primitive or entity
- Pay attention also to constructors with null arguments or entities with several combinations of arguments
- Before introducing the builder pattern, let’s have a look at the **fluent interface pattern**
  - The idea is to have code that looks like a fluent text in natural language
  - We can use it to set optional fields
  - The trick is to return a reference to the entity

```java
public class Account {
    private final AccountNumber number;
    private final LegalPerson owner;
    private Percentage interest;
    private Money creditLimit;
    private AccountNumber fallbackAccount;

    public Account(AccountNumber number,
        LegalPerson owner,
        Percentage interest) {
        ...     ①
    }

    public Account withCreditLimit(Money creditLimit) {
        this.creditLimit = creditLimit;
        return this;    ②
    }

    public Account withFallbackAccount(AccountNumber
fallbackAccount) {
        this.fallbackAccount = fallbackAccount;
        return this;    ②
    }
}
```

Methods **with*** return **this**

```java
class AccountService {

    void openAccount() {
        Account account = new Account(number,
                                      accountowner,
                                      interest)    ③
                                 .withCreditLimit(limit)
.withFallbackAccount(fallbackAccount);
        ...
    }
}
```

Account is created
with credit limit = limit

Caution! We loose the
command-query separation:
commands should change the state and
return nothing;
queries should return an answer
without changing anything

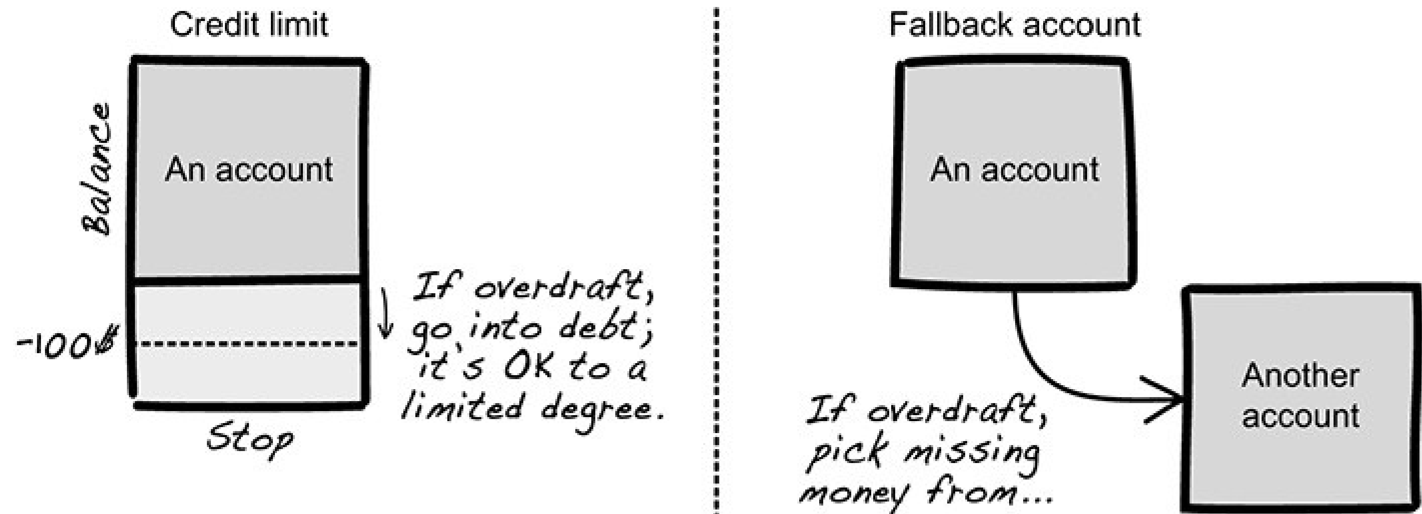
Nonfluent fluent interface

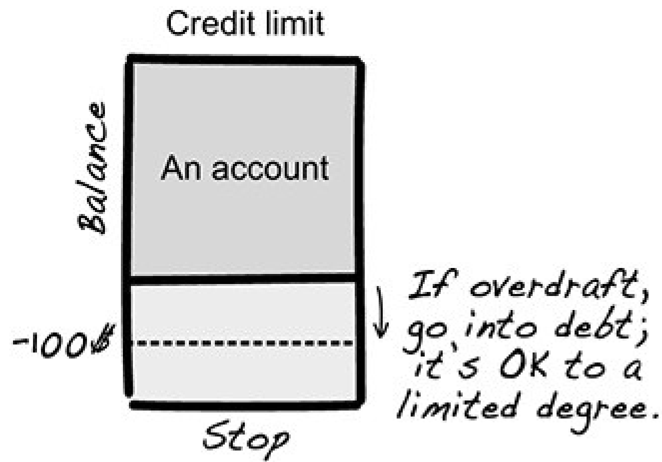
- Do not return this in setter
- A fluent interface must result in code that can be read fluently
- What about the following code?

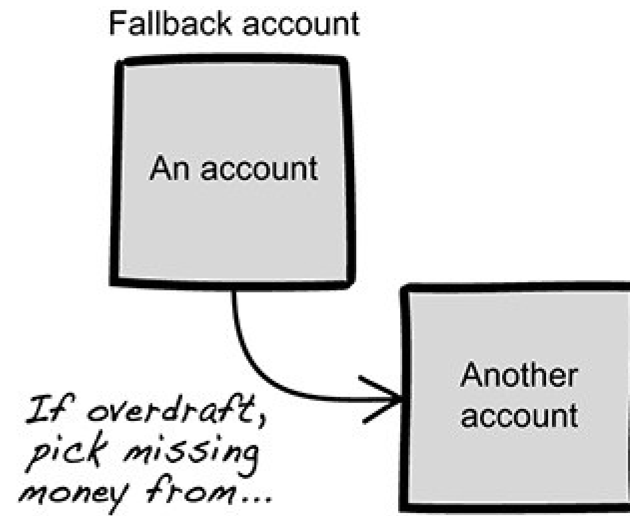
```
Person p = new Person()
    .setFirstName("Deve")
    .setLastName("Loper")
    .setProfession("Developer");
```

Advanced constraints

- They involve several fields at the same time
- Example: a bank account must have either an overdraft or a fallback account







Account must have either overdraft safeguard mechanism but can't have both.

```
private void checkInvariants()
    throws IllegalStateException {

    validState(fallbackAccount != null
                    ^ creditLimit != null);
}
```

```
    public void changeToFallbackAccount(AccountNumber
fallbackAccount) {
        this.creditLimit = null;     ①
        this.fallbackAccount = fallbackAccount;     ②
        checkInvariants();     ③
    }
```

The invariant can be violeted while the method is executed (1)

It is important to restore consistency before returning the control (2)

Better to verify complex invariants before returning the control (fail fast)

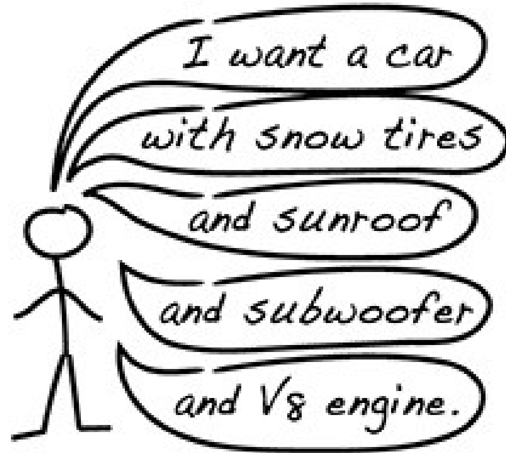
The builder pattern

- The idea is to obtain a full object, satisfying all constraints, before other portions of the code have access to it
- The complexity of the construction of the entity is hidded by another object, the builder
- Who uses the builder doesn’t need to see the partially constructed object (and cannot see it)

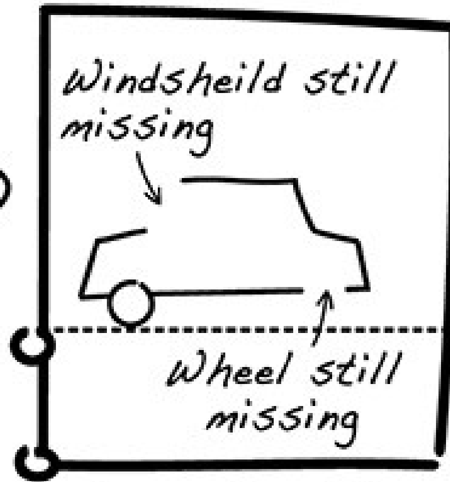
Example of builder

The basic idea of builder pattern

1) Tell the builder what you want.

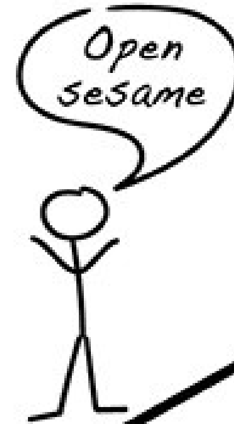


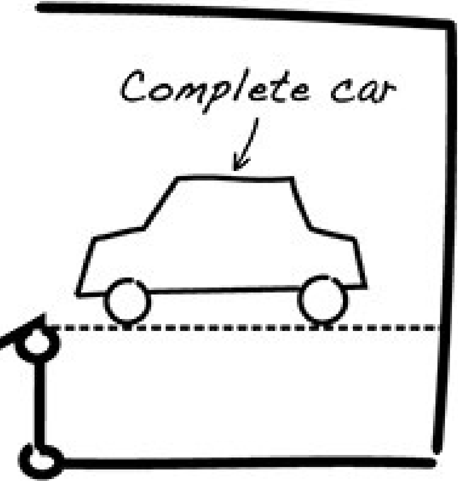
Car builder



Assembling the car in secrecy

2) Ask for the finished car.





You don't need to see the car in its incomplete and inconsistent intermediary stages.

```
AccountBuilder accountBuilder =
        new AccountBuilder(number,     ②
                           accountOwner,
                           interest);
accountBuilder.withCreditLimit(limit);

Account account = accountBuilder.build();
```

Builder constructor specifies all mandatory fields

Optional fields are added with other methods

Method build() returns the constructed object (all constraints are checked in this method)

Builders are well suited for the use of a fluent interface

```
Account account =
        new AccountBuilder(number,     ①
                           accountOwner,
                           interest)
                .withCreditLimit(limit)
                .build();    ③
```

```java
public class Account {
    private final AccountNumber number;
    private final LegalPerson owner;
    private Percentage interest;
    private Money creditLimit;
    private AccountNumber fallbackAccount;

    private Account(AccountNumber number,
                    LegalPerson owner,
                    Percentage interest) {    ①
        this.number = notNull(number);
        this.owner = notNull(owner);
        this.interest = notNull(interest);
    }

    ...

    private void checkInvariants() throws
IllegalStateException {
        validState(fallbackAccount != null
                    ^ creditLimit != null);    ②
    }

    public static class Builder {    ③
        private Account product;

        public Builder(AccountNumber number,
                       LegalPerson owner,
                       Percentage interest) {    ④
            product = new Account(number, owner, interest);
        }

        public Builder withCreditLimit(Money creditLimit) {
            validState(product != null);    ⑤
            product.creditLimit = creditLimit;
            return this;    ⑥
        }
```

```java
        public Builder withFallbackAccount(AccountNumber
fallbackAccount) {
            validState(product != null);    ⑤
            product.fallbackAccount = fallbackAccount;
            return this;    ⑥
        }

        public Account build() {
            validState(product != null);
            product.checkInvariants();    ⑦
            Account result = product;
            product = null;    ⑧
            return result;    ⑨
        }
    }
}
```

Constructor of the entity is private

Builder is a static inner class of the entity (it can access private methods of the entity)

Method build() checks invariants (7) before returning the entity (9), auto-destroying (8) to avoid a second usage

Preserving integrity of entities

- Now we know how to create valid entities
- How do we maintain their validity?
  - Impossible if the entity releases a mutable field
  - Impossible if the entity provides setters without controls
- Hence, we have to control changes

```java
class Order {
    private CustomerID custid;
    private List<OrderLine> orderitems;
    private Addr billingaddr;
    private Addr shippingaddr;
    private boolean paid;     ①
    private boolean shipped;

    public void setPaid(boolean paid) {
        this.paid = paid;     ②
    }
    public boolean getPaid() { return paid; }
}
```

Field paid is private (1),
but the setter (2)
exposes it to arbitrary changes

Compiler blocks (3),
but (4) is permitted

```java
Order order = ...
order.paid = true;      ③
order.setPaid(true);    ④
```

```
class Order {
    private boolean paid = false;    ①
    private boolean shipped;

    public void markPaid() { this.paid = true; }    ②
    public boolean isPaid() { return paid; }
}
```

Encode only business rules

An unpaid order can become paid,
but the other direction is not possible

Method markPaid() implements such a business rule (2)

Do not share mutable objects

- Entities need to share their data
- The safest way to do it is by sharing primitive domains, which are immutable
- Sharing mutable objects opens the possibility for changes out of the control of the entity
  - Goodbye encapsulation!

```
class Person {
    private String name;
    private StringBuffer title;

    String name() {
        return name;    ①
    }

    StringBuffer title() {    ②
        return title;
    }
}

String personalizedLetter(Person p) {
    String greeting =
        p.name()
         .concat(", we'd like to make you an
offer");    ③
    String salute =
        p.title()
          .append(", we'd like to make you an offer")
          .toString();    ④
    ...
```

(3) doesn't change the entity

(4) change the entity

String is immutable

StringBuffer is mutable

Class java.util.Date is deprecated because mutable!

```java
class Person {

    private Date birthdate;

    Date birthdate() {
        return birthdate.clone();
    }
}
```

If you really need to return
a mutable object,
returns a copy!

Any subsequent change,
doesn’t change the entity
(but the copy)

Caution with collections

- private List<OrderLine> orderItems;
- public void setOrderItems(List<OrderLine> orderItems)
  - The argument is mutable, do not keep a reference in the entity
  - Do a copy of the list (expensive)
  - Better, encode only business logic, for example

    public void addOrderItem(OrderLine orderItem)

    public int numberOfItems()
- public List<OrderLine> orderItems()
  - Do not return a reference to a mutable field
  - Return a copy (expensive and it may confuse the called on the possibility to modify the list of the entity)
  - Return a read-only proxy (but pay attention to the objects in the list, are they mutable?)

```
void addFreeShipping(Order order) {
    if(order.value().greaterThan(FREE_SHIPPING_LIMIT) {
        List<OrderLine> orderlines = order.orderItems();
        orderlines.add(    ①
            new OrderLine(SHIPPING_VOUCHER, 1));
    }
}
```

Do not return references to mutable collections

```
class Order {
    private List<OrderLine> orderitems;    ①
    public List<OrderLine> getOrderItems() {
        return new ArrayList(orderitems);
    }
}
```

Return a copy using the copy constructor

```
class Order {
    private CustomerID custId;
    private List<OrderLine> orderitems;    ①
    public List<OrderLine> orderItems() {    ①
        return new
Collections.unmodifiableList(orderitems);
    }
}

List<OrderItem> items = order.orderItems();    ①
items.add(new OrderItem(SHIPPING_VOUCHER, 1));    ②
```

Better, return a read-only proxy

Trying to modify the list in (2) raises an exception

They are not immutable

- Even if the list is a copy or a read-only proxy, the entity may be externally mutated
- This may happen if the list contains mutable objects
- The solutions is to use lists of immutable objects (domain primitives)
- If you really need to return a mutable list, you have to do a in-depth copy (very expensive)

Google Form

- https://forms.gle/tWyjnj4Cz1pyE25T8

Exercise

Realizzare un main creando un menù di scelta come descritto di seguito:

- Premi 1 per aggiungere un’automobile (che legge da input, usando lo Scanner, la targa, la casa produttrice, il nome e il prezzo, crea un’automobile con i dati inseriti e l’aggiunge alla Concessionaria usando il metodo aggiungiVeicolo);
- Premi 2 per aggiungere una moto (che legge da input, usando lo Scanner, la targa, la casa produttrice, il nome e il prezzo, crea una moto con i dati inseriti e l’aggiunge alla Concessionaria usando il metodo aggiungiVeicolo);
- Premi 3 per stampare (che stampa tutti i veicoli presenti nella Concessionaria)
- Premi 4 per ordinare per prezzo (che invoca il metodo di ordinamento per prezzo nella Concessionaria)
- Premi 5 per ordinare per nome (che invoca il metodo di ordinamento per nome nella Concessionaria)
- Premi 6 per stampare il valore (che invoca il metodo calcolaValore di Concessionaria e stampa il risultato)
- Premi 7 per salvare su file (che legge da input, usando lo Scanner, il nome di un file e invoca il metodo salvaSuFile di concessionaria)
- Premi 8 per leggere da file (che legge da input, usando lo Scanner, il nome di un file e invoca il metodo leggiDaFile di concessionaria)
- Premi 9 per uscire

dove si esegue l'operazione relativa quando l'utente sceglie il corrispondente numero.
Ogni operazione (a parte l’uscita) può essere svolta un numero arbitrario di volte.

Implementiamo primitive di dominio ed entità per gestire i menù e scriviamo codice sicuro!

Fine della lezione