Object Constraint Language (OCL)

Types of expressions in OCL

- Expressions can be used in a number of places in a UML model:
  - To specify the initial value of an attribute or association end.
  - To specify the derivation rule for an attribute or association end.
  - To specify the body of an operation.
  - To indicate an instance in a dynamic diagram.
  - To indicate a condition in a dynamic diagram.
  - To indicate actual parameter values in a dynamic diagram.

Types of constraints in OCL

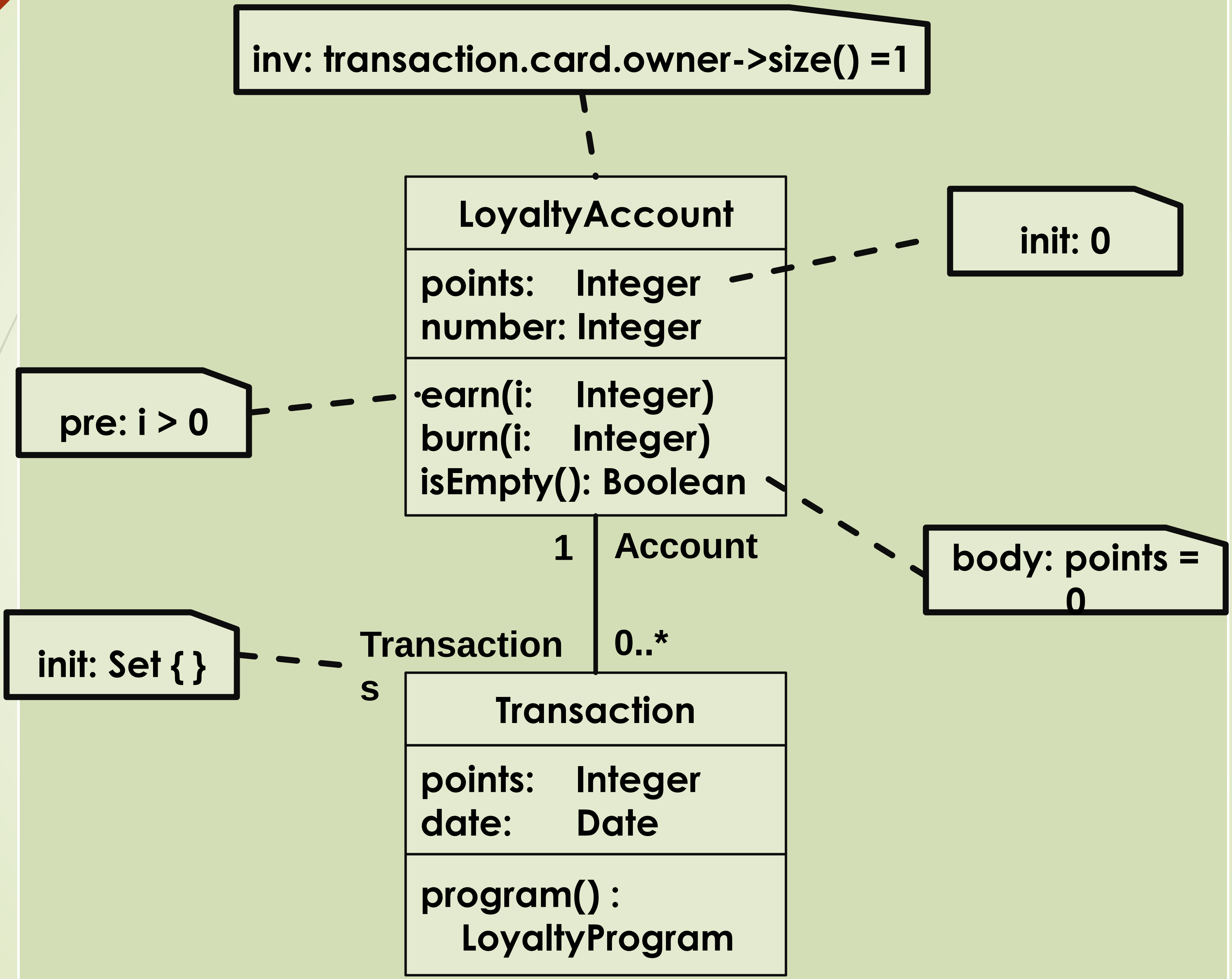
- There are four types of constraints:
  - An *invariant*
  - A *precondition*
  - A *postcondition*
  - A *guard* is a constraint that must be true before a state transition fires.

The Context of an OCL Expression

- The *context definition* of an OCL expression specifies the model entity for which the OCL expression is defined.
  - Usually this is a class, interface, datatype, or component. In terms of the UML standard, this is called a *Classifier*.
- Sometimes the model entity is an operation or attribute, and rarely it is an instance.
  - It is always a specific element of the model, usually defined in a UML diagram. This element is called the *context* of the expression.

The Context of an OCL Expression

- Next to the context, it is important to know the *contextual type* of an expression. The contextual type is the type of the context, or of its container.
- It is important because OCL expressions are evaluated for a single object, which is always an instance of the contextual type.
- To distinguish between the context and the instance for which the expression is evaluated, the latter is called the *contextual instance*.
- Sometimes it is necessary to refer explicitly to the contextual instance. The keyword *self* is used for this purpose.



inv: transaction.card.owner->size() =1

LoyaltyAccount

points: Integer
number: Integer

earn(i: Integer)
burn(i: Integer)
isEmpty(): Boolean

init: 0

pre: i > 0

body: points = 0

1

Account

init: Set { }

Transactions

0..*

Transaction

points: Integer
date: Date

program() : LoyaltyProgram

Invariants on attributes

- The simplest constraint is an invariant on an attribute.
- Suppose our model contains a class *Customer* with an attribute *age*, then the following constraint restricts the value of the attribute:

context Customer **inv**:

age >= 18

Invariants on associations

- One may also put constraints on associated objects.
- Suppose in our model contains the class *Customer* has an association to class *Salesperson*, with role name *salesrep* and multiplicity 1, then the following constraint restricts the value of the attribute *knowledgelevel* of the associated instance of Salesperson:

```
context Customer inv:
   salesrep.knowledgelevel >= 5
```

Collections of objects

- In most of the cases the multiplicity of an association is not 1, but more than 1.
- Evaluating a constraint in these cases will result in a collection of instances of the associated class.
- Constraints can be put on either the collection itself, e.g. limiting the size, or on the elements of the collection.
- Suppose in our model the association between Salesperson and Customer has role name *clients* and multiplicity 1..* on the side of the Customer class, then we might restrict this relationship by the following constraint.

```
context Salesperson inv:
    clients->size() <= 100 and
    clients->forAll(c: Customer | c.age >= 40)
```

Pre- and postconditions

- In pre- and postconditions the parameters of the operation may be used.
- Furthermore, there is a special keyword *result* which denotes the return value of the operation.
- It can be used in the postcondition only.
- As an example we have added an operation *sell* to the Salesperson class.

context Salesperson::sell(item: Thing): Real

pre: self.sellableItems->includes(item)

post: not self.sellableItems->includes(item) and result = item.price

Derivation Rules

- Models often define derived attributes and associations.
- A derived element does not stand alone.
- The value of a derived element must always be determined from other (base) values in the model.
- Omitting the way to derive the element value results in an incomplete model.
- Using OCL, the derivation can be expressed in a derivation rule.
- In the following example, the value of a derived element *usedServices* is defined to be all services that have generated transactions on the account:

context LoyaltyAccount::usedServices : Set(Services)

derive: transactions.service->asSet()

Initial Values

- In the model information, the initial value of an attribute or association role can be specified by an OCL expression.
- In the following examples, the initial value for the attribute *points* is *0*, and for the association end *transactions,* it is an empty set:

context LoyaltyAccount::points : Integer

init: 0

context LoyaltyAccount::transactions :
  Set(Transaction)

init: Set{}

Initial Values

- Note the difference between an initial value and a derivation rule.
  - A derivation rule states an invariant:
  - The derived element should always have the same value that the rule expresses.
  - An initial value, however, must hold only at the moment when the contextual instance is created. After that moment, the attribute may have a different value at any point in time.

Body of Query Operations

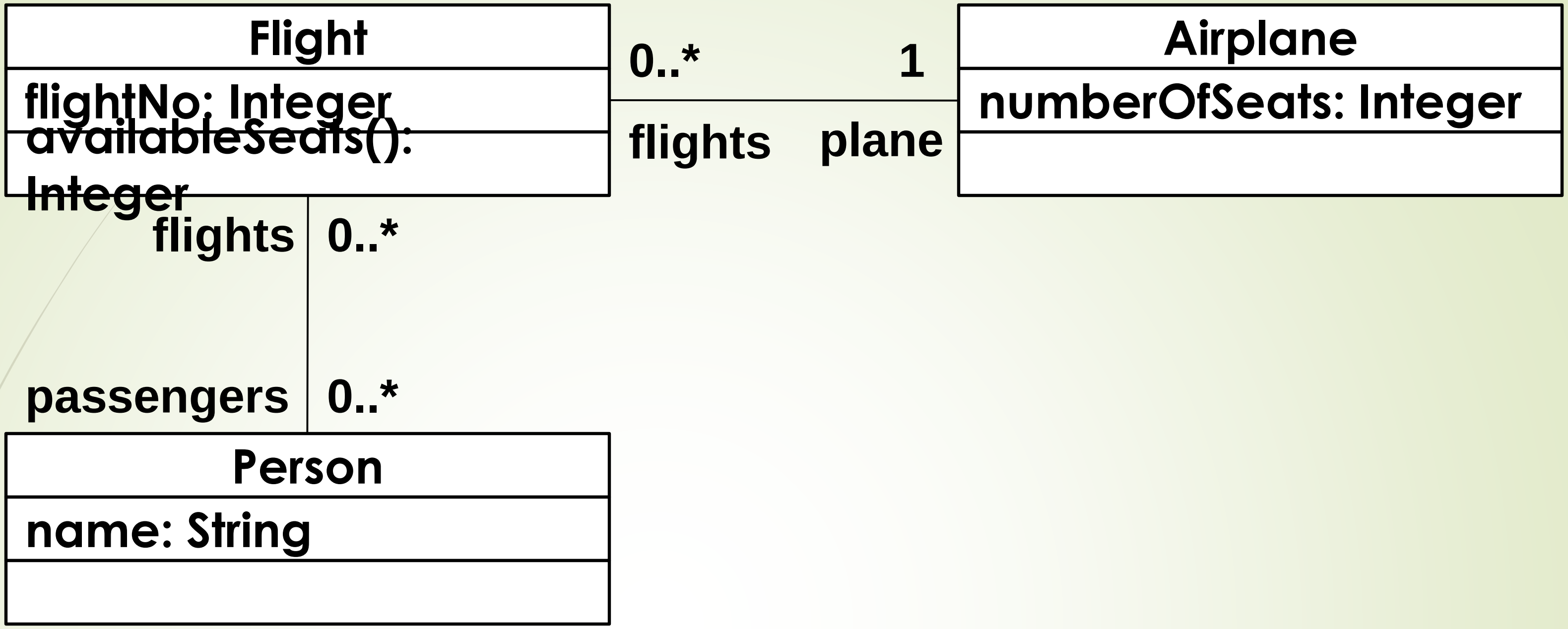
- The class diagram can introduce a number of query operations.
- Query operations are operations that have no side effects, i.e., do not change the state of any instance in the system.
- Execution of a query operation results in a value or set of values, without any alterations in the state of the system.
- Query operations can be introduced in the class diagram, but can only be fully defined by specifying the result of the operation.
- Using OCL, the result can be given in a single expression, called a *body expression*.
- In fact, OCL is a full query language, comparable to SQL. The use of body expressions is an illustration thereof.
- The next example states that the operation *getCustomerName* will always result in the name of the card owner associated with the loyalty account:

context LoyaltyAccount::getCustomerName() : String

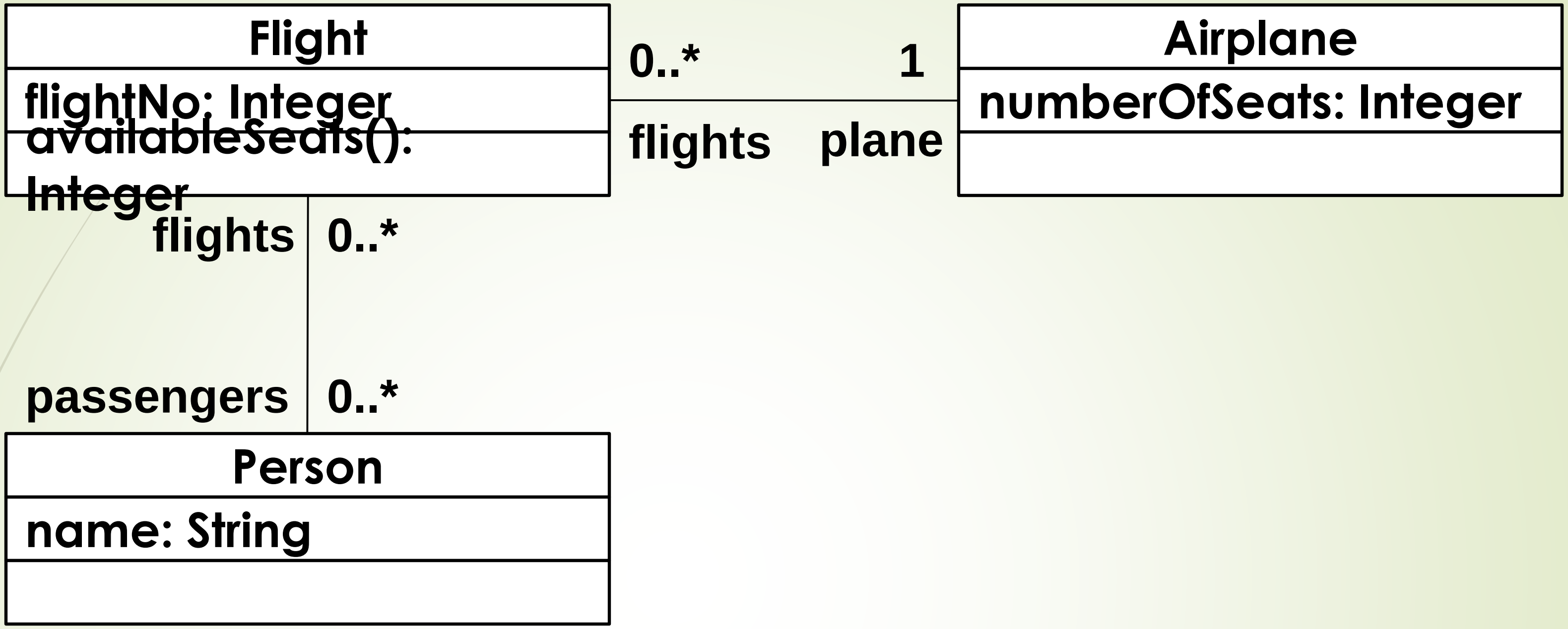
body: Membership.card.owner.name

Broken constraints

- Evaluating a constraint does not change any values in the system.
- A constraint states "this should be so".
- If for a certain object the constraint is not true, in other words, it is broken, then the only thing we can conclude is that the object is not correct, it does not conform to our specification.
- Whether this is a fatal error or a minor mistake, and what should be done to correct the situation is not expressed in the OCL.

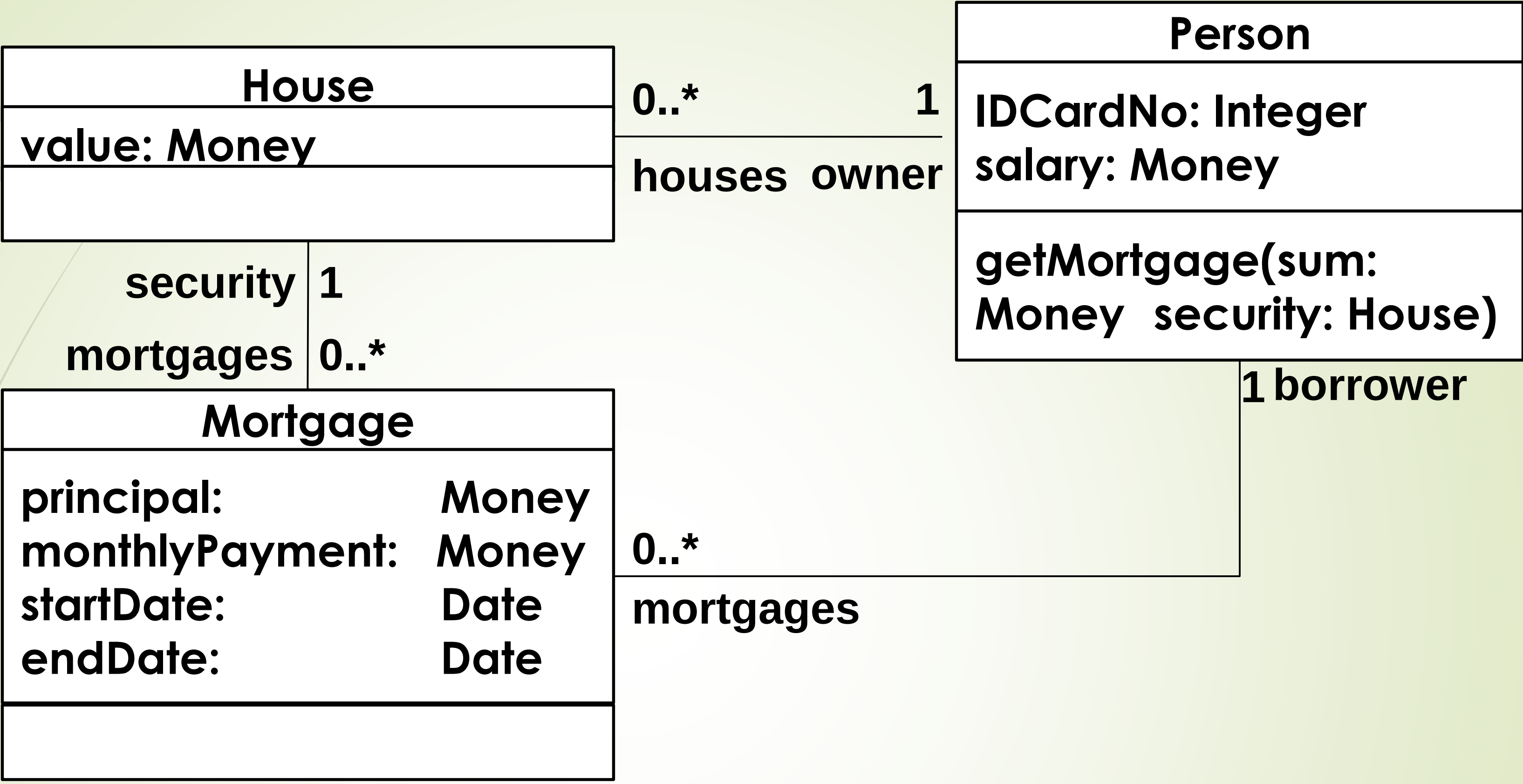


- an association between class *Flight* and class *Person*, indicating that a certain group of persons are the passengers on a flight, will have multiplicity many (0..*) on the side of the *Person* class.
- This means that the number of passengers is unlimited.
- In reality, the number of passengers will be restricted to the number of seats on the airplane that is associated with the flight.
- It is impossible to express this restriction in the diagram.

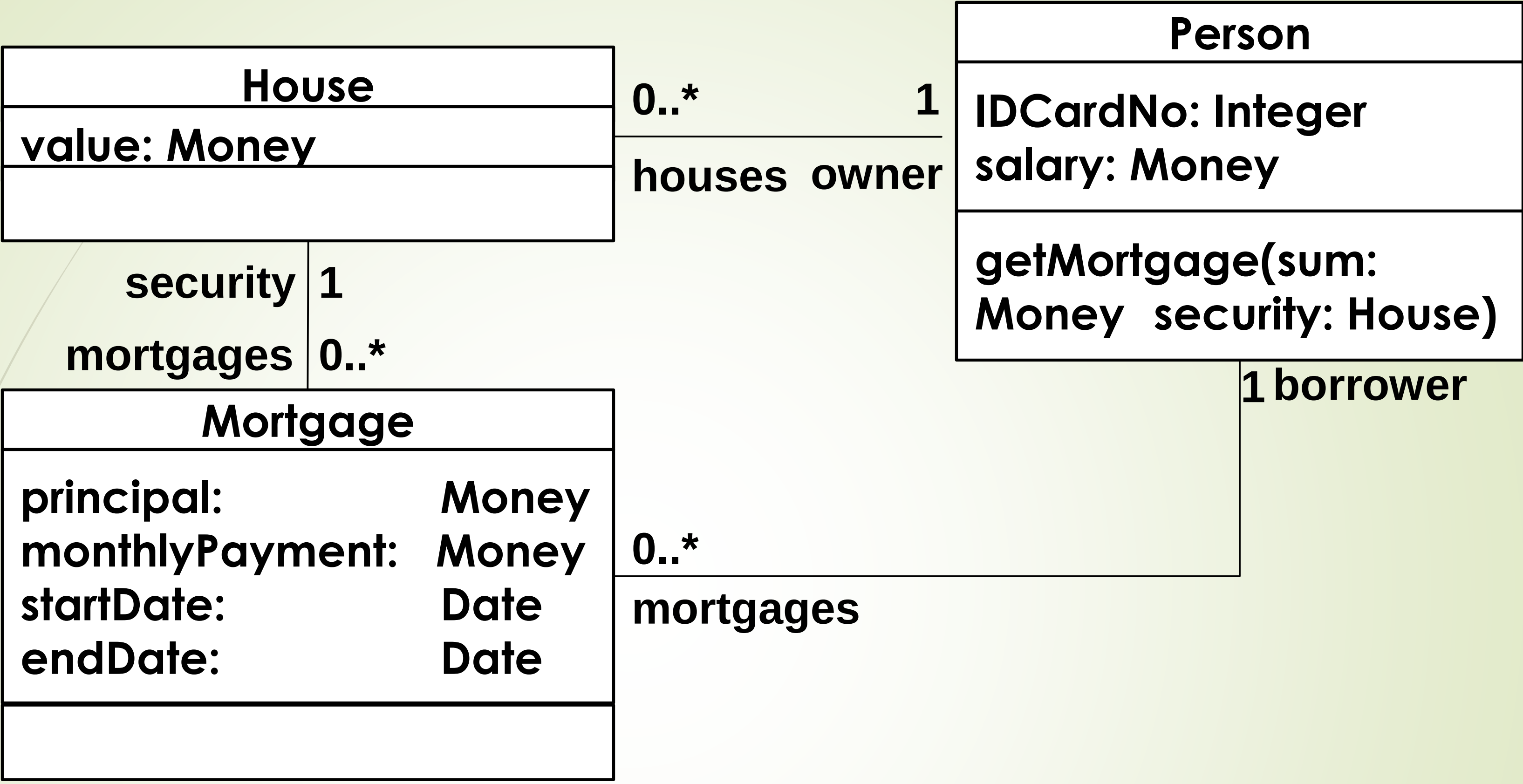


In this example, the correct way to specify the multiplicity is to add to the diagram the following OCL constraint:

context Flight
inv: passengers->size() <= plane.numberOfSeats



- A person may have a mortgage on a house only if that house is owned by him- or herself;
  - one cannot obtain a mortgage on the house of one's neighbor or friend.
- The start date for any mortgage must be before the end date.
- The ID card number of all persons must be unique.



- A new mortgage will be allowed only when the person's income is sufficient.
- A new mortgage will be allowed only when the counter-value of the house is sufficient.

Value Added by OCL

context Mortgage
inv: security.owner = borrower

context Mortgage
inv: startDate < endDate

context Person
inv: Person::allInstances()->isUnique(socSecNr)

context Person::getMortgage(sum : Money, security : House)
pre: self.mortgages.monthlyPayment->sum() <= self.salary * 0.30

context Person::getMortgage(sum : Money, security : House)
pre: security.value >= security.mortgages.principal->sum()

To remember about OCL:

- The Object Constraint Language is just a specification language.
- It obeys a syntax and has keywords.
- However, unlike other languages, it can't be used to express program logic or flow control.
- By design, OCL is a *query-only language; it can't modify the model (or executing system) in any* way.
- It can be used to express preconditions, postconditions, invariants (things that must always be True), guard conditions, and results of method calls.
- OCL can be used virtually anywhere in UML and is typically associated with a classifier using a note.
- When an OCL expression is evaluated, it is considered to be instantaneous, meaning the associated classifier can't change state during the evaluation of an expression.