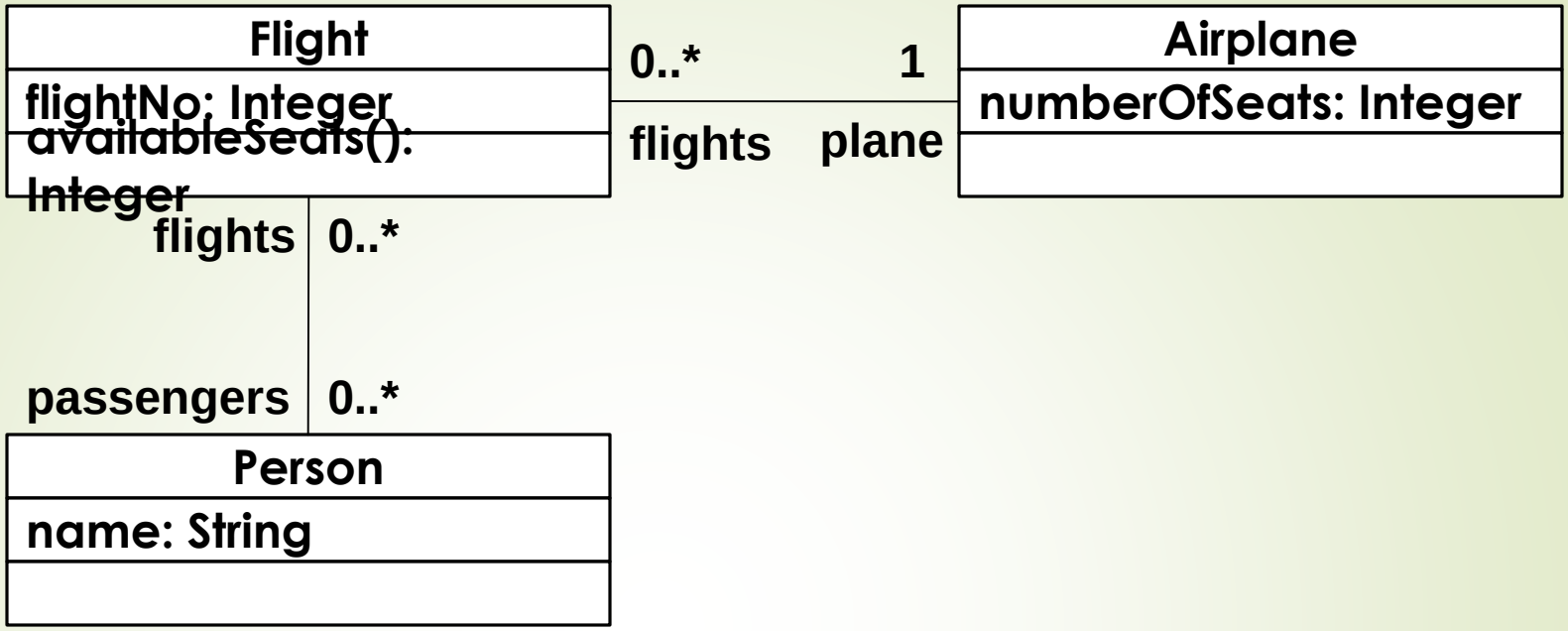
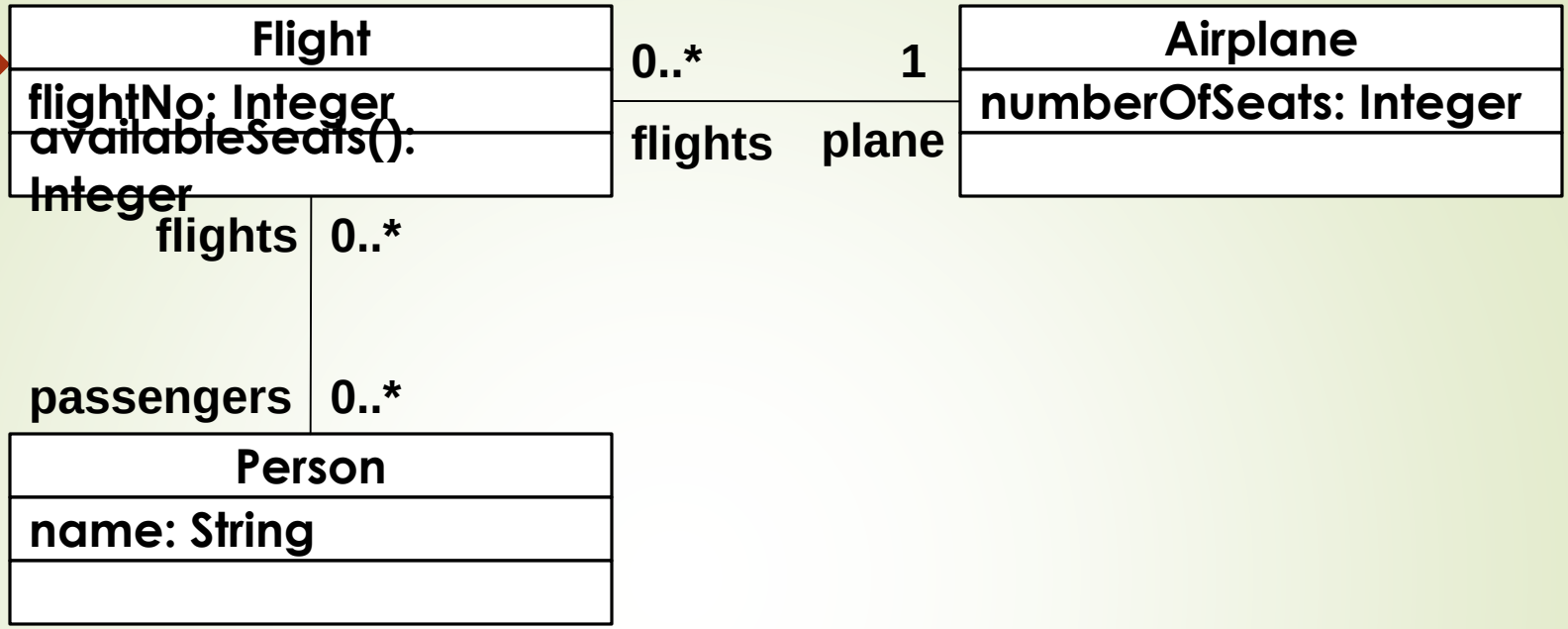
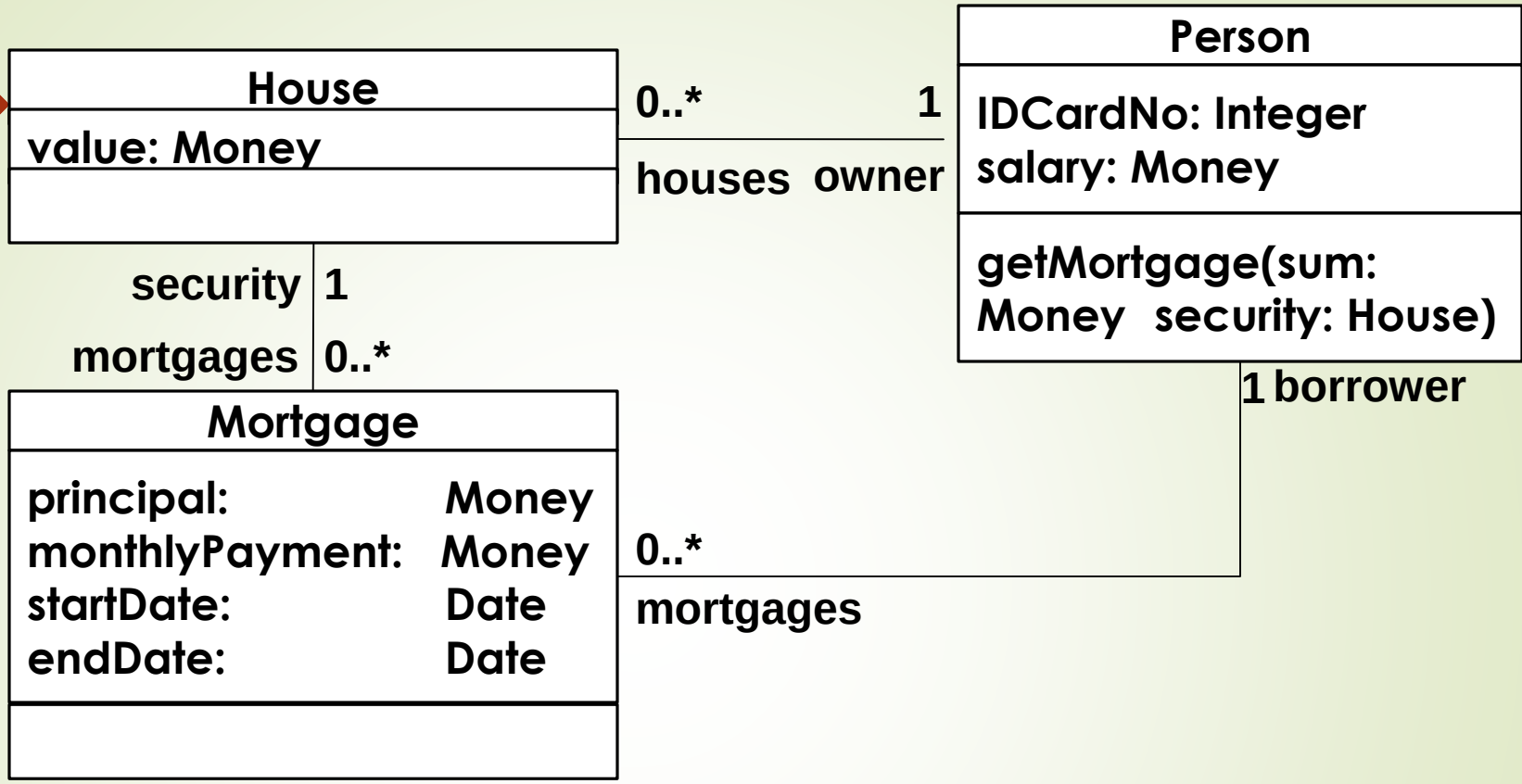


- an association between class *Flight* and class *Person*, indicating that a certain group of persons are the passengers on a flight, will have multiplicity many (0..*) on the side of the *Person* class.
- This means that the number of passengers is unlimited.
- In reality, the number of passengers 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.

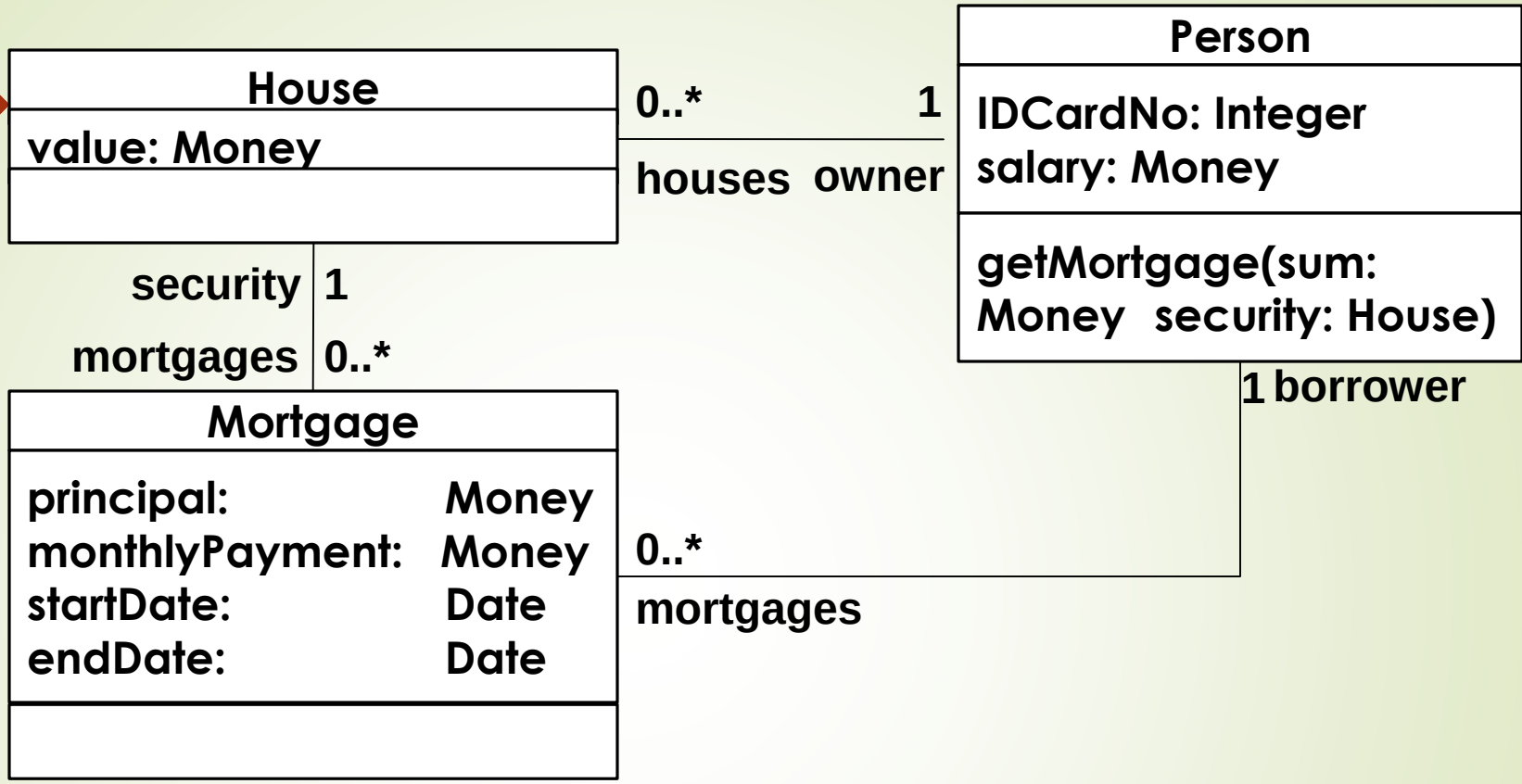


In this example, the correct way to specify the multiplicity is to add to the diagram the following OCL constraint:

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()

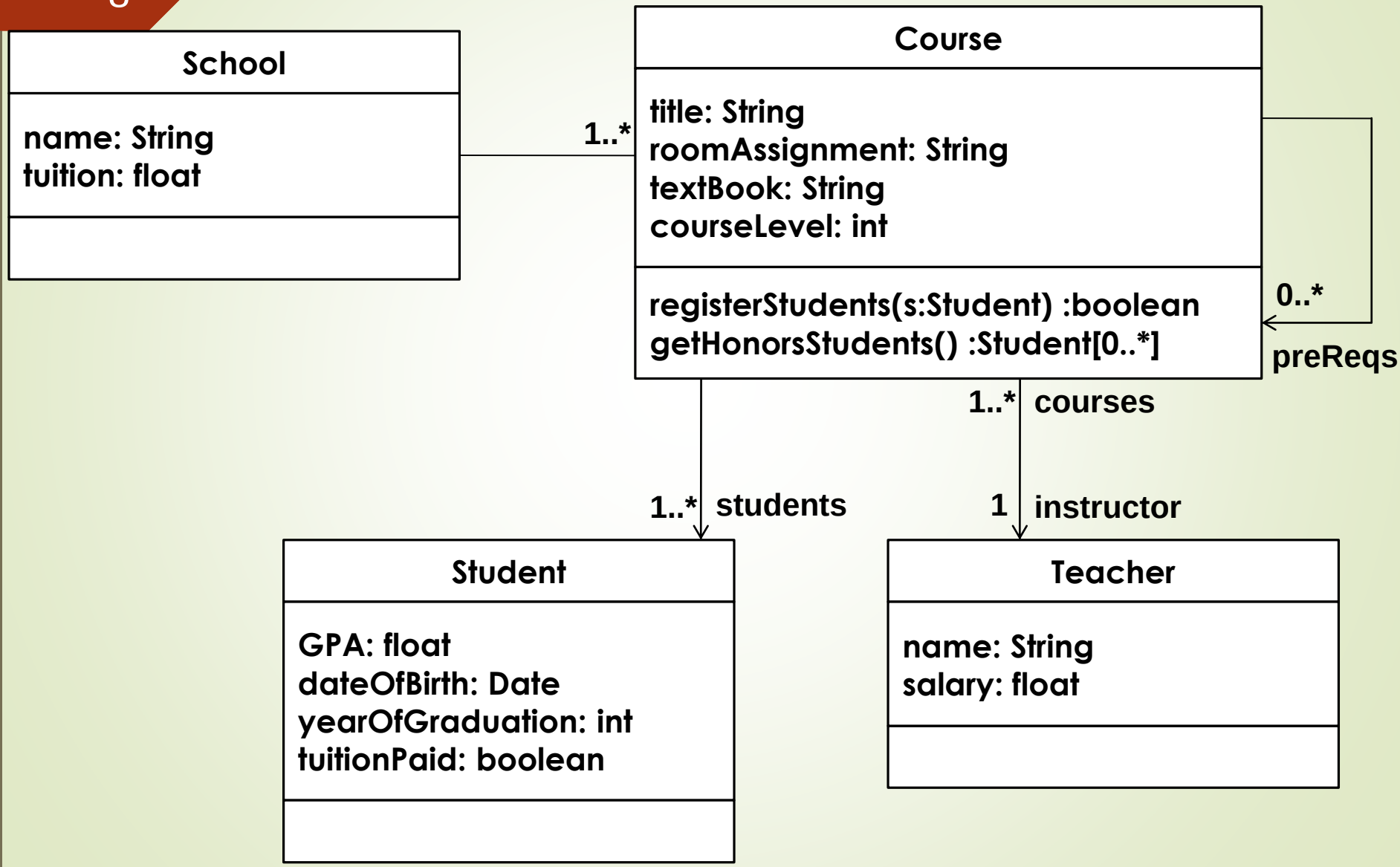
To remember about OCL:

- 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.

Constraints on Classifiers

- Each OCL expression must have some sense of ***context*** *that an expression relates to.*
- *Often* the context can be determined by where the expression is written.
- For example, you can link a constraint to an element using a note.
- You can refer to an instance of the **context classifier** using the keyword self.
- For example, if you had a constraint on **Student** that their GPA must always be higher than 2.0, you can attach an OCL expression to **Student** using a note and refer to the GPA as follows:

self.GPA >= 2.0


School
name: String
tuition: float
1..*
Course
title: String
roomAssignment: String
textBook: String
courseLevel: int
registerStudents(s:Student) :boolean
getHonorsStudents() :Student[0..*]
0..*
preReqs
1..*
courses
1..*
students
1
instructor
Student
GPA: float
dateOfBirth: Date
yearOfGraduation: int
tuitionPaid: boolean
Teacher
name: String
salary: float

Constraints

- if you had a constraint on Student that their GPA must always be higher than 2.0, you can attach an OCL expression to Student using a note to refer to the GPA as follows:

  self.GPA > 2.0

- Note:
  - If you want to allow a GPA of less than 2.0 and send out a letter to the student's parents in the event such a low GPA is achieved, you would model such behavior using a UML diagram such as an activity or interaction diagram.

- The following invariant on Course ensures that the instructor is being paid:

  self.instructor.salary > 0.00

Business Rule

- The following expressions verify that a student's tuition was paid before registering for a course and that the operation registerStudent returned true:

context Course::registerStudent(s: Student): boolean

pre: s.tuitionPaid = true

post: result = true

Labels in OCL

- We can name pre and post conditions by placing a label after the pre or post keywords:

context Course::registerStudent(s: Student): boolean

pre hasPaidTuition: s.tuitionPaid = true

post studentHasRegistered: result = true

Use of previous value:

- Postconditions can use the **@pre** keyword to refer to the value of some element before an operation executes.
- The following expression ensures that a student was registered and the number of students in the course has increased by 1.

context Course::registerStudent(s: Student): boolean

pre hasPaidTuition: s.tuitionPaid = true

post studentHasRegistered: result = true AND

self.students = self.students@pre + 1

- We may specify the results of a query operation using the keyword **body**.
- Because OCL doesn't have syntax for program flow, we are limited to relatively simple expressions.
- The following expressions indicate that the honors students are students with GPAs higher than 3.5.

context Course::getHonorsStudents(s: Student): boolean

body: self.students->select(GPA > 3.5)

Conditionals

- OCL supports basic Boolean expression evaluation using the **if-then-else-endif** keywords.
- The conditional are used only to determine which expression is evaluated; they can't be used to influence the underlying system or to affect program flow.
- The following invariant enforces that a student's year of graduation is valid only if he/she has paid his/her tuition:

```
context Student inv:
if tuitionPaid = true then
  yearOfGraduation = 2005
else
  yearOfGraduation = 0000
endif
```

OCL's Logic Rules

- The boolean evaluation rules are:

1. **True OR-ed with anything is true.**
2. **False AND-ed with anything is false.**
3. **False IMPLIES *anything is True*.**

OCL's Logic Rules

- For example, the following expression enforces that if a student's GPA is less than 1.0, their year of graduation is set to 0. If the GPA is higher than 1.0, Rule #3 applies, and the entire expression is evaluated as true (meaning the invariant is valid).

context Student inv:

self.GPA < 1.0 IMPLIES self.yearOfGraduation = 0000

- OCL supports several complex constructs you can use to make your constraints more expressive and easier to write.
- You can break complex expressions into reusable pieces (within the same expression) by using the **let** and **in** keywords to declare a variable.
- You declare a variable by giving it a name, followed by a colon **(:)**, its type, an expression for its value, and the **in** keyword.
- The following example declares an expression that ensures a teacher of a high-level course has an appropriate salary:

```
context Course inv:
let salary : float = self.instructor.salary in
if self.courseLevel > 4000 then
salary > 100000.00
else
salary < 100000.00
endif
```