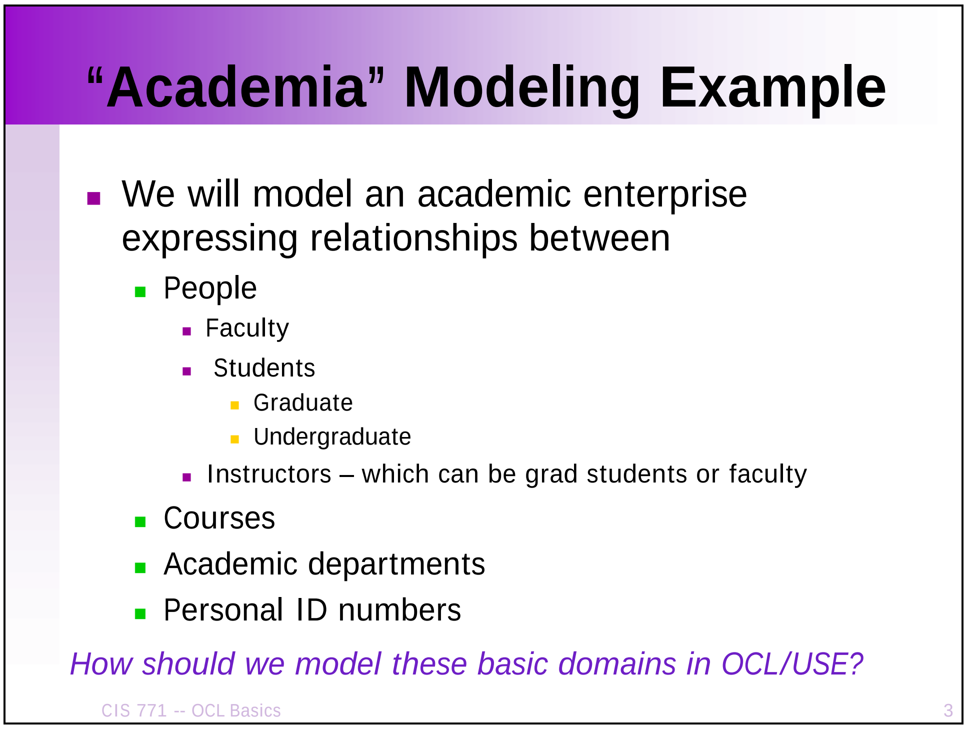
CIS 771: Software Specifications

Lecture 12: OCL Basics

Outline

- Review of how to declare class and association structures in USE
- OCL Types
- OCL Invariant Syntax
- Navigating across attributes and associations
- Collection and set operations

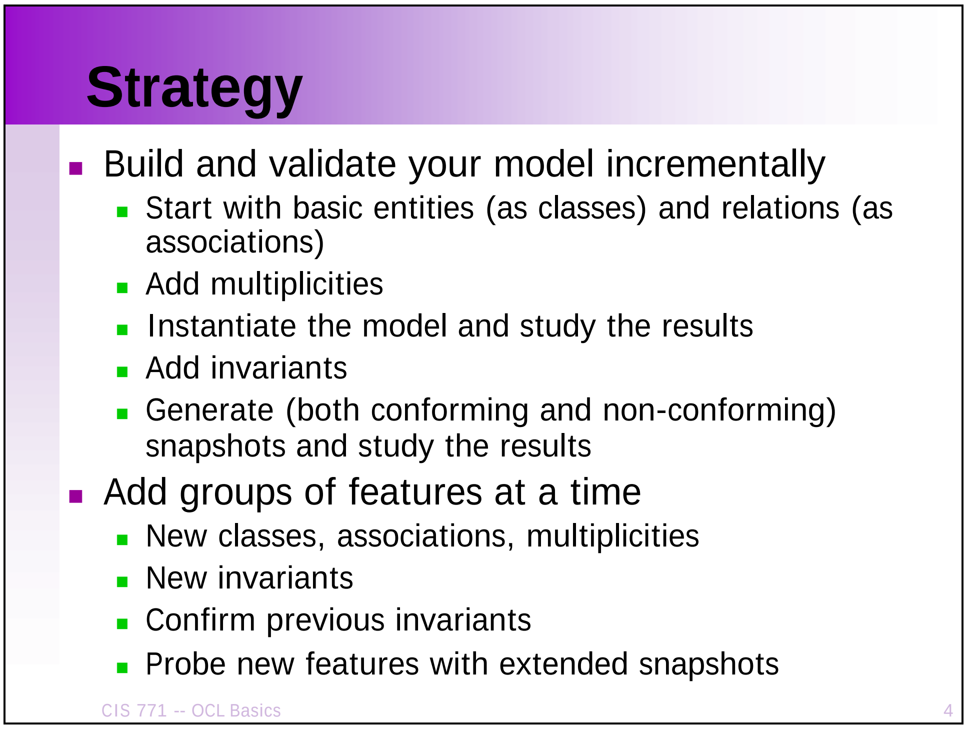
...with the Academia model as the running example.



“Academia” Modeling Example

- We will model an academic enterprise expressing relationships between
  - People
    - Faculty
    - Students
      - Graduate
      - Undergraduate
    - Instructors – which can be grad students or faculty
  - Courses
  - Academic departments
  - Personal ID numbers

How should we model these basic domains in OCL/USE?



Strategy

- Build and validate your model incrementally
  - Start with basic entities (as classes) and relations (as associations)
  - Add multiplicities
  - Instantiate the model and study the results
  - Add invariants
  - Generate (both conforming and non-conforming) snapshots and study the results
- Add groups of features at a time
  - New classes, associations, multiplicities
  - New invariants
  - Confirm previous invariants
  - Probe new features with extended snapshots

Basic Entities

- People
  - Students: Undergrads and Grads
  - Instructors: Faculty and Grads
- Courses

What we'll do now...

- We'll talk about some general issues regarding modeling using classes and generalization,
- I'll do a few examples,
- You'll do a few examples, and
- We'll assess the results.
- Then we'll go onto relations (associations)

Generalization

- When modeling, you often want to express that one type of entity is a special case of another
  - or symmetrically, that one type of entity generalizes another
- Note that Person generalizes Faculty
  - symmetrically, Faculty specializes/refines Person
- A faculty member has all the attributes of a person, but it refines person by adding additional attributes or more specialized behavior
- We sometimes say Faculty "is a" Person
- Using other common terminology, Faculty conforms to Person

Modeling Generalization in Alloy

Fragment of Alloy Academia model

```
model Academia {
  domain {Person, Course}

  state {
    partition Faculty, Student : Person
    …
  }
```

- Faculty “is a” Person is modeled by saying that the set of Faculty entities is a subset of the set of Person entities
- Thus, every Alloy constraint/relation that applies to Person also applies to Faculty

Modeling Generalization in UML/USE

Starting the USE Academia model

```
model Academia

class Person
attributes
  name : String
end
```

```
class Faculty < Person
end
```

- In UML, the notion of generalization is a primitive notion, and so we model generalization directly
  - In USE’s textual form, `Faculty < Person`
  - In diagrams using the arrow $\longrightarrow\!\!\rhd$

For You To Do…

- Pause the lecture
- Download the academia-1.use file
- Add to this file the remainder of the class declarations and generalization relationships for
  - Student
  - Grad student
  - Undergrad student
  - Instructor
  - Course
- Load the resulting model into USE (so that it can check your syntax)
- Create some system snapshots

Assessment

Dealing with Instructor…

- Recall that in Alloy, we did the following…

```
Instructor : Person
```

Note that we cannot specify here that Instructors can only be grad students or faculty. We will do that later in an invariant schema.

- Really, what we want to say is that…
  - Instructor “is a” Faculty, or
  - Instructor “is a” Grad

Assessment

Dealing with Instructor… considering multiple inheritance

- Note that UML allows multiple inheritance
- One Option…

```
class Instructor < Faculty, Grad
end
```

  - But this specifies that an instructor implements the behavior (conforms to, is-a) Faculty and a Grad

Assessment

Dealing with Instructor… considering multiple inheritance

- Another option…

```
class Instructor < Person        end
class Student < Person           end
class Faculty < Instructor       end
class Grad < Student, Instructor end
```

  - But this specifies that, e.g., every Grad is an Instructor

Assessment

Dealing with Instructor… several possible solutions

- In Java, we could have Faculty and Grad both implement an Instructor interface
  - USE doesn't provide modeling of interfaces
- We could also implement Instructor as an *adapter* class

```
class Instructor
attributes
  instructor:Person
end
```

Note that we cannot specify here that Instructors can only be grad students or faculty. We will do that later in an invariant.

Same situation as in Alloy!

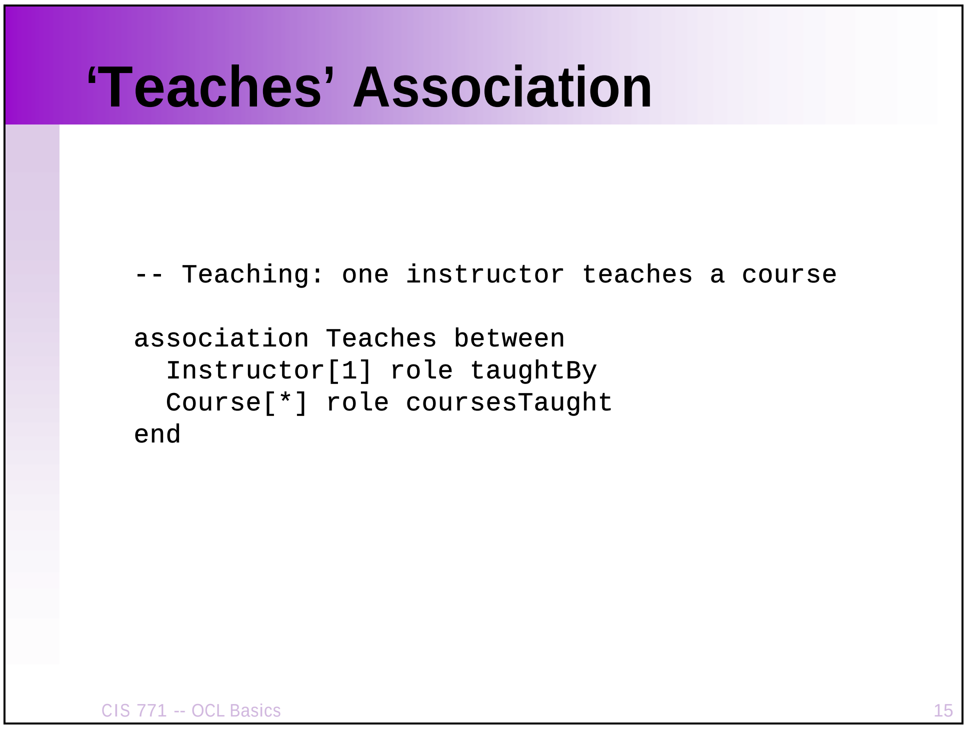
- We'll just use the adaptor approach

Basic Relations

- Relationships
  - One *instructor* *teaches* a *course*.
  - One or more *students* are *taking* a *course.*
  - *Students* can be *waitingfor* for *course*.
- Relations are modeled as *associations* in UML

What we'll do now…

- We'll review the syntax for defining associations
- We'll see that UML has richer notation for multiplicities
- We'll talk about role names and how to use them to navigate across associations
- You'll do some examples from the Academia model

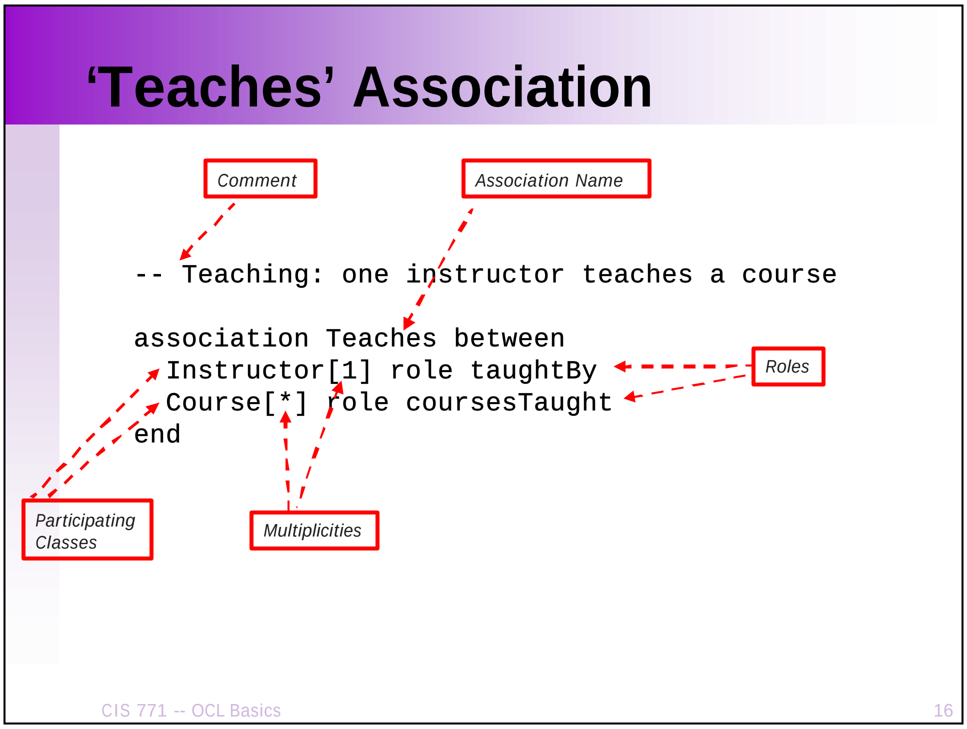


‘Teaches’ Association

```
-- Teaching: one instructor teaches a course

association Teaches between
  Instructor[1] role taughtBy
  Course[*] role coursesTaught
end
```

CIS 771 -- OCL Basics 15

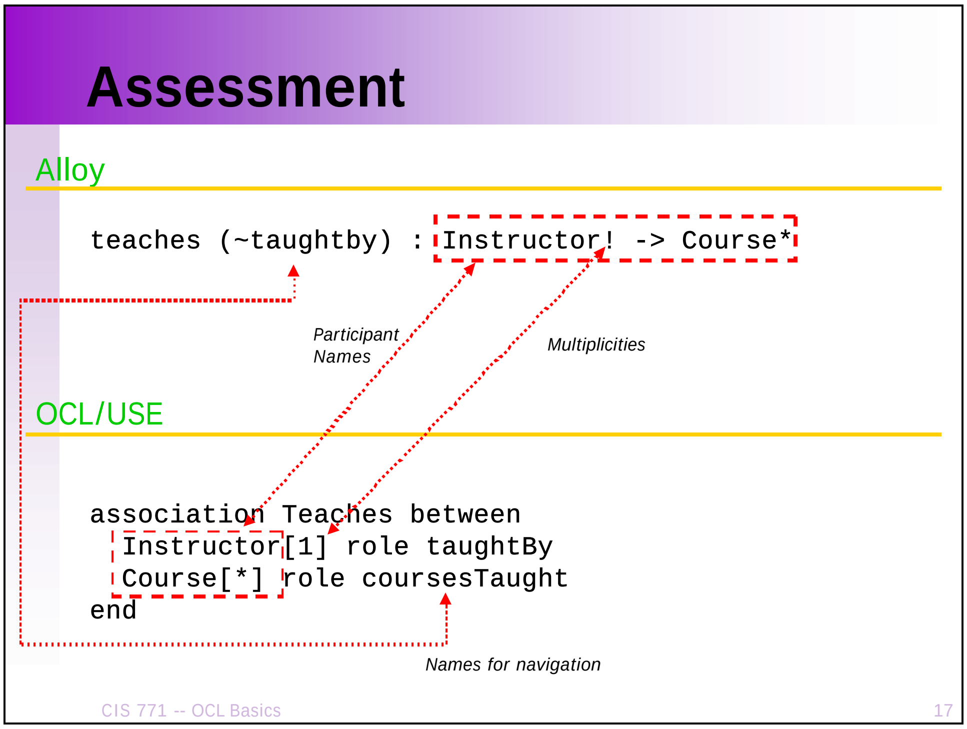


‘Teaches’ Association

```
-- Teaching: one instructor teaches a course

association Teaches between
  Instructor[1] role taughtBy
  Course[*] role coursesTaught
end
```

CIS 771 -- OCL Basics 16



Assessment

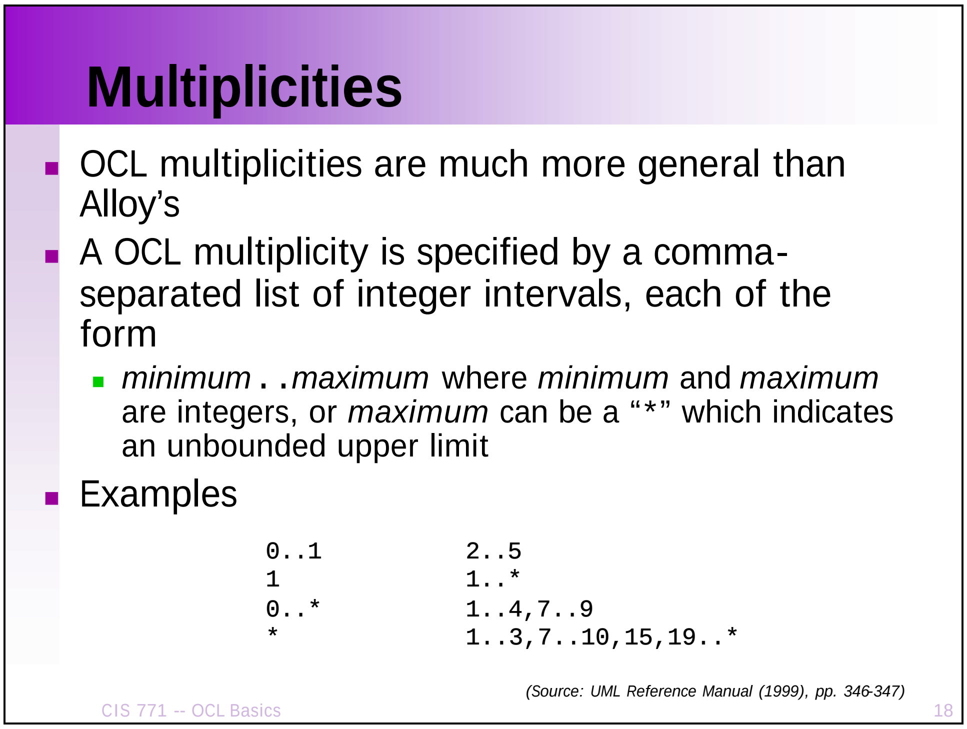
Alloy

```
teaches (~taughtby) : Instructor! -> Course*
```

OCL/USE

```
association Teaches between
  Instructor[1] role taughtBy
  Course[*] role coursesTaught
end
```

Participant Names
Multiplicities
Names for navigation



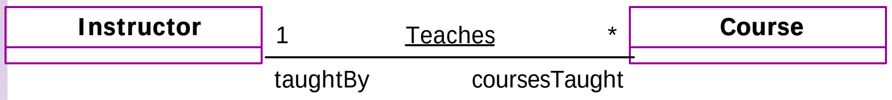
Multiplicities

- OCL multiplicities are much more general than Alloy's
- A OCL multiplicity is specified by a comma-separated list of integer intervals, each of the form
  - *minimum*`..`*maximum* where *minimum* and *maximum* are integers, or *maximum* can be a "`*`" which indicates an unbounded upper limit
- Examples

```
0..1          2..5
1             1..*
0..*          1..4,7..9
*             1..3,7..10,15,19..*
```

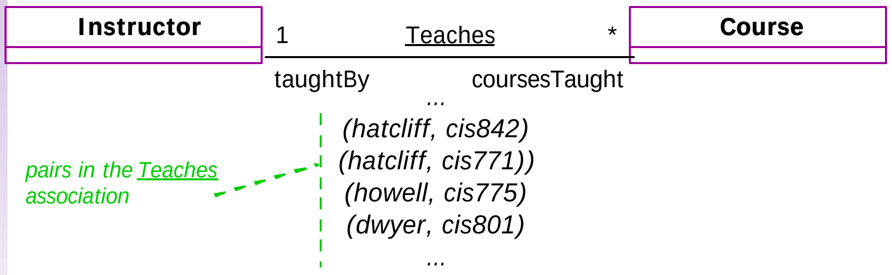
(Source: UML Reference Manual (1999), pp. 346-347)

Role Names



- Role names are used to navigate between classes
- Role names are the names of “pseudo-attributes” whose values are the sets of objects that result from navigating across the association
- For `c:Course, c.taughtBy` yields the instructor object that teaches the course
- For `i:Instructor, i.coursesTaught` yields the *set* of course objects representing courses taught by `I`
- If explicit role names are omitted, the default is the name of the participating class (using lowercase), e.g., `c.instructor`

Role Names



- For `c:Course, c.taughtBy` yields the instructor object that teaches the course, e.g.,
  - cis771.`taughtBy` = hatcliff
- For `i:Instructor, i.coursesTaught` yields the *set* of course objects representing courses taught by `I,` e.g.,
  - hatcliff.`coursesTaught` = {cis842, cis771}

For You To Do…

- Pause the lecture
- Download the academia-2.use file
- Add to this file the association declarations (with multiplicities) for the relations
  - teaches,
  - taking, and
  - waitingfor.
- Load the resulting model into USE
- Create some system snapshots
  - some that satisfy your multiplicity constraints
  - some that violate your multiplicity constraints

Some Hints on Using USE

- Type 'help' at the USE command line to see all the things that you can do with it.
- Notice that one of the options is the `read` command which reads from a file a sequence of commands for creating snapshots.
  - You should use the option for the rest of the lecture to re-populate your model with interesting instances after each revision of your model.

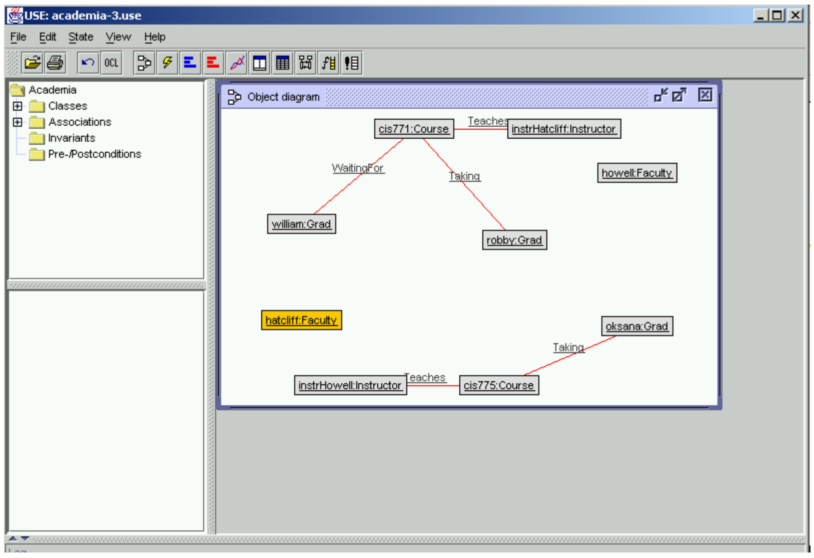
A Example Command Script

academia-basic-instantiation.cmd

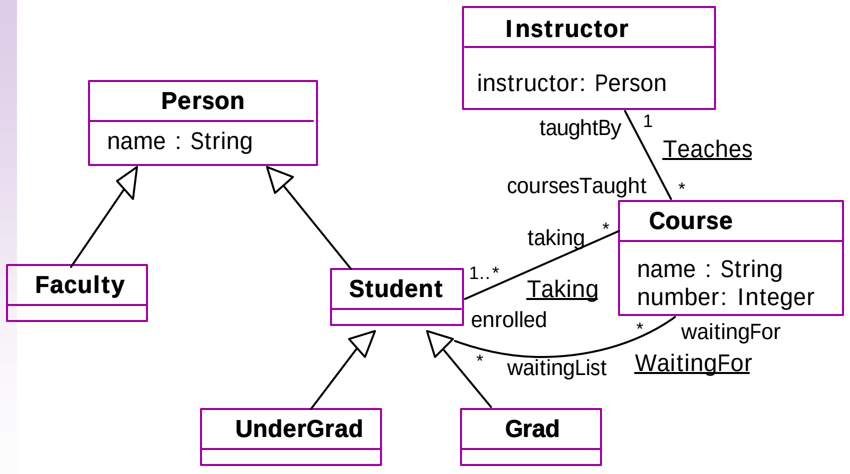
```
!create hatcliff:Faculty
!create howell:Faculty
!create robby:Grad
!create william:Grad
!create oksana:Grad
!create cis771:Course
!create cis775:Course
!create instrHatcliff:Instructor
!set instrHatcliff.instructor = hatcliff
!create instrHowell:Instructor
!set instrHowell.instructor = howell
!insert (instrHatcliff,cis771) into Teaches
!insert (instrHowell,cis775) into Teaches
!insert (robby,cis771) into Taking
!insert (william,cis771) into WaitingFor
!insert (oksana,cis775) into Taking
```

Notes: Load this into USE using the 'read' command at the command line (after completing the last 'For you to do' or opening academia-3.use When you get to this level, it's often best to turn off the 'check structure at every stage' option in the State menu. With this option off, you just force a check by e.g., giving the 'check' command at the command line. Note that at the end of this script, there are no structural violations. However, there are structural violations at intermediate steps (as you can see if you have the 'check structure at every stage' option turned on).

Screen Shot After Script



Class Diagram



Basic Invariants

- All instructors are either faculty or graduate students.
  - Couldn't be expressed in class definitions
- No one is waiting for a course unless someone is enrolled.
- No graduate student teaches a course that they are enrolled in.

How should we model these as invariants in OCL/USE?

Basic Invariants

What we'll do now...

- We'll go over the basic structure of invariants
  - there are several variations
- We'll look at the *types* of expressions that can be used in OCL invariants
- We'll cover just enough to allow us to specify the constraints on the previous slide
- There's a lot more to cover...
  - Read the OMG-OCL specification
  - We won't cover everything in the lectures
  - We will get into more details in the next lecture

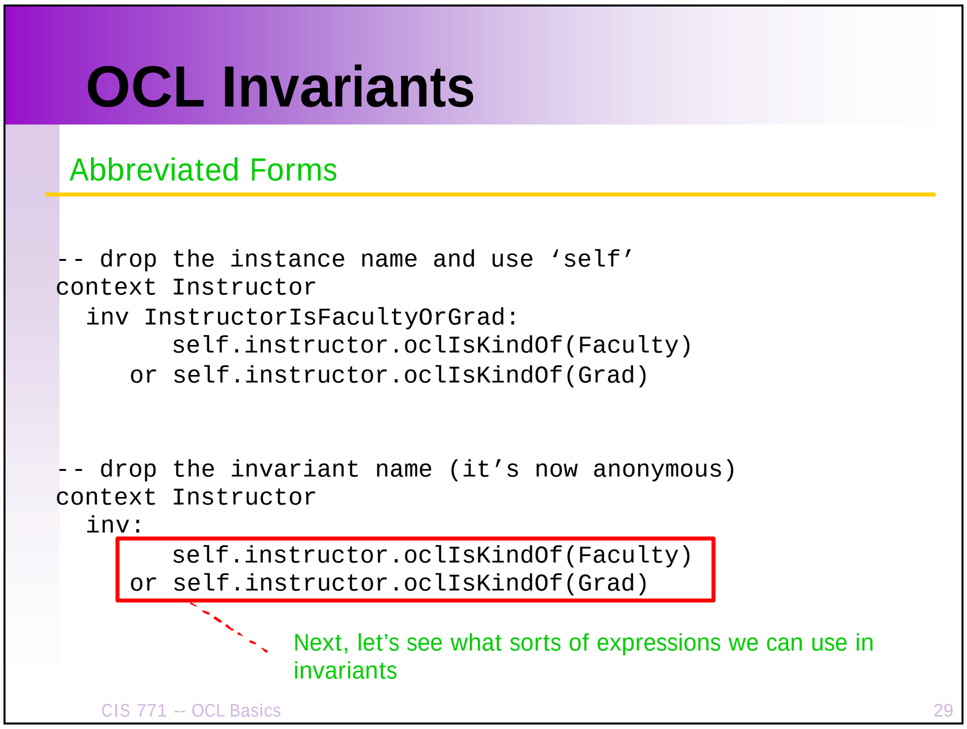
OCL Invariants

General form

`context` *id* : *Class*
  `inv` *name*: *bool-expr*   *--- first invariant for Class*
  …
  `inv` *name*: *bool-expr*   *--- nth invariant for Class*

Example

```
context i : Instructor
  inv InstructorIsFacultyOrGrad:
        i.instructor.oclIsKindOf(Faculty)
     or i.instructor.oclIsKindOf(Grad)
```



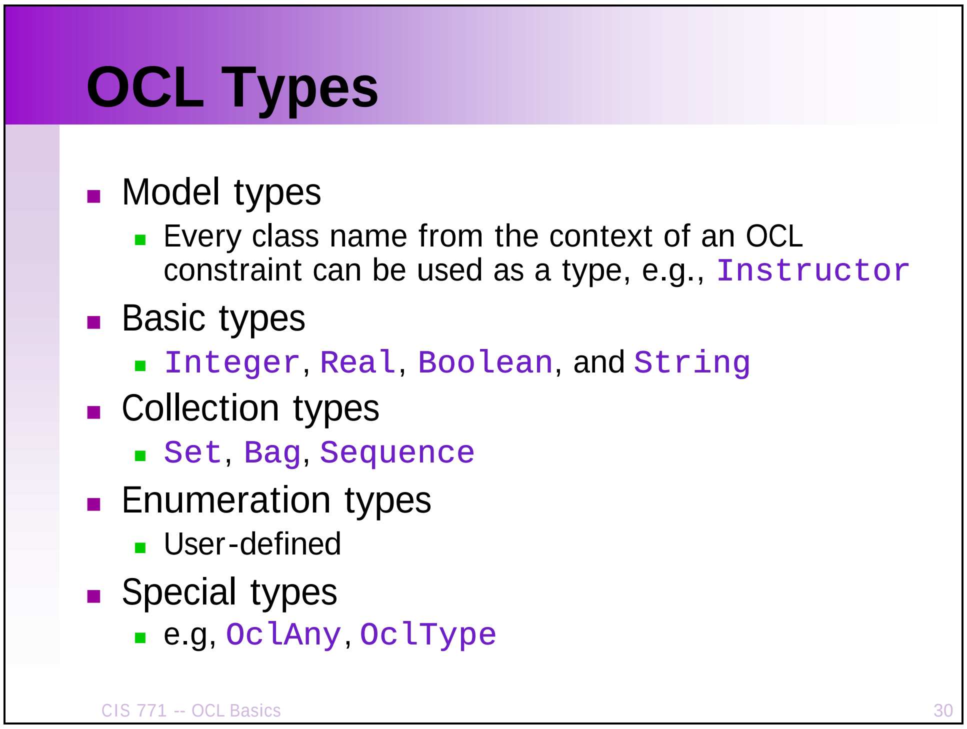
OCL Invariants

Abbreviated Forms

```
-- drop the instance name and use ‘self’
context Instructor
  inv InstructorIsFacultyOrGrad:
        self.instructor.oclIsKindOf(Faculty)
     or self.instructor.oclIsKindOf(Grad)


-- drop the invariant name (it’s now anonymous)
context Instructor
  inv:
        self.instructor.oclIsKindOf(Faculty)
     or self.instructor.oclIsKindOf(Grad)
```

Next, let’s see what sorts of expressions we can use in invariants



OCL Types

- Model types
  - Every class name from the context of an OCL constraint can be used as a type, e.g., `Instructor`
- Basic types
  - `Integer`, `Real`, `Boolean`, and `String`
- Collection types
  - `Set`, `Bag`, `Sequence`
- Enumeration types
  - User-defined
- Special types
  - e.g, `OclAny`, `OclType`

Basic Types and Values

- Boolean
  - `true`, `false`
- Integer
  - `1, -5, 2, 34, 26524`
- Real
  - `1.5, 3.14`
- String
  - `'To be or not to be…'` ← *Single quote*

Basic Types with Operations

- Boolean
  - `and, or, xor, not, implies, if-then-else`
- Integer
  - `*,+,-,/,abs`
- Real
  - `*,+,-,/,floor`
- String
  - `toUpper, concat`

Predefined Object Operations

- `o.oclIsKindOf(t : OclType) : Boolean`
  - true if o’s type is t or a subtype of t
- `o.oclIsTypeOf(t : OclType)  : Boolean`
  - true if o’s type is identical to t
- `o.oclAsType(t : OclType) : OclType`
  - analogous to an explicit type cast in Java
  - o’s static type becomes `OclType`
  - the expression evaluates to the denoted by object o if `o.oclIsKindOf(t : OclType)` is true,
  - the expression is undefined otherwise.

(see OMG-UML v1.3 Section 7.8.1.2 pp. 7.29-7.30)

Academia Invariant Revisited

Example

```
context i : Instructor
  inv InstructorIsFacultyOrGrad:
        i.instructor.oclIsKindOf(Faculty)
     or i.instructor.oclIsKindOf(Grad)
```

Examples

```
context s : Student
  inv: s.IsKindOf(Person)    -- is true
  inv: s.IsTypeOf(Person)    -- is false
  inv: s.IsKindOf(Course)    -- is false
  inv: s.IsKindOf(Grad)      -- is false
```

Collection Types

- Collection
  - An abstract type with the concrete types Set, Bag, and Sequence as subtypes
- Set
  - Analogous to the mathematical set (do duplicate elements)
- Bag
  - Like a set, but may contain duplicates
- Sequence
  - Like a bag, but the elements are ordered

Navigation Typing Rules

Consider a navigation expression of the form

object.rolename

- The value of the expression is the *set* of objects on the other side of the association, e.g.,
  - hatcliff.`coursesTaught` = {cis842, cis771}
- If the association end *rolename* has a maximum multiplicity of 1, then the navigation expression returns an *object*, e.g.,
  - cis771.`taughtBy` = hatcliff
- However, in the case above, we are also allowed to treat the result as a *set*,
  - cis771.`taughtBy->notEmpty`
- If the association on the Class Diagram is adorned with '{ordered}', the navigation results in a *sequence*

Collection Properties

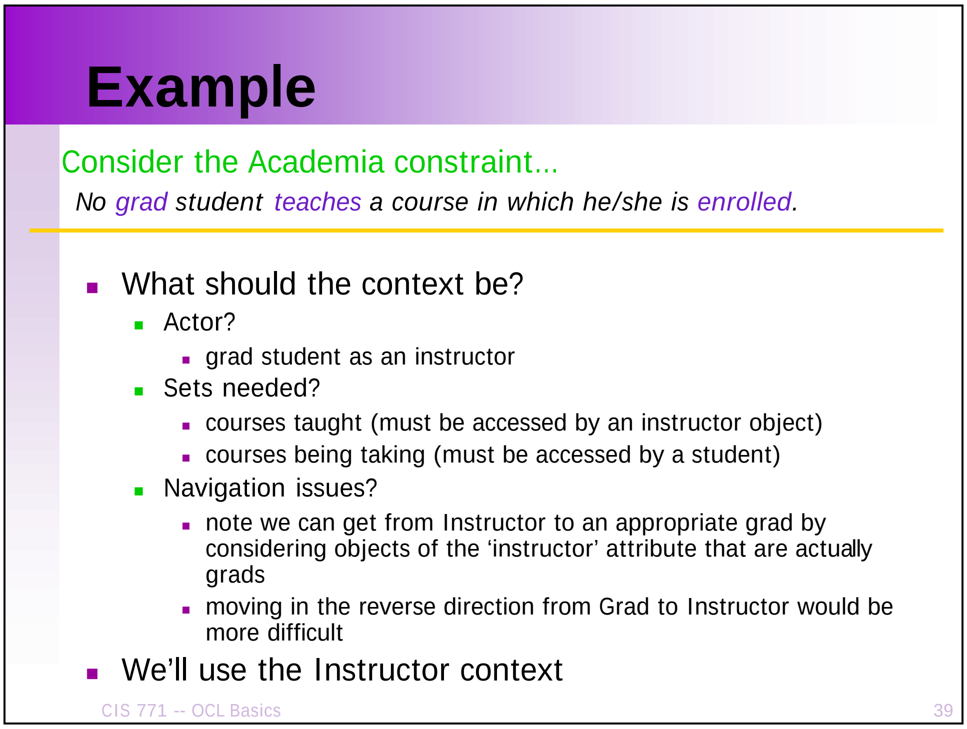
- A property of a collection is accessed using an arrow `->` followed by the name of the property
- Examples
  - *collection*`->notEmpty`
  - *collection*`->isEmpty`
  - *collection*`->size`
- There are many more...

(see OMG-UML v1.3 Section 7.8.2.1 pp. 7.37-7.40)

Set Properties

- *set*`->union`(set2: Set(T)) : Set(T)
- *set*`->intersection`(set2: Set(T)) : Set(T)
- *set* – (set2: Set(T)) : Set(T)
- There are many more...

(see OMG-UML v1.3 Section 7.8.2.2 pp. 7.40-7.42)

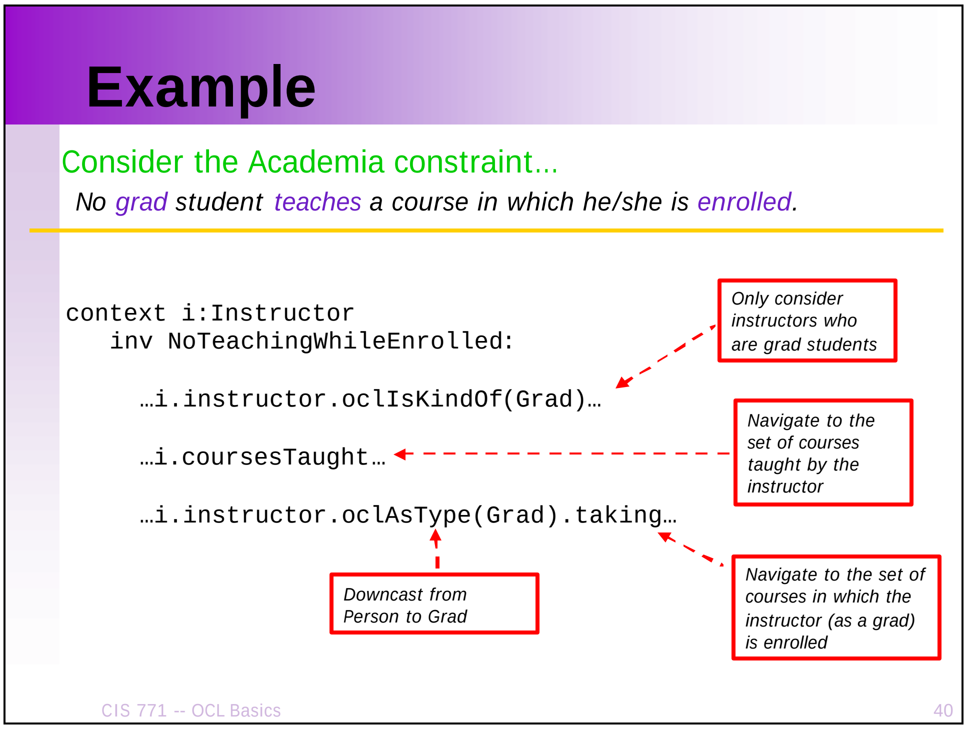


Example

Consider the Academia constraint…

No grad student teaches a course in which he/she is enrolled.

- What should the context be?
  - Actor?
    - grad student as an instructor
  - Sets needed?
    - courses taught (must be accessed by an instructor object)
    - courses being taking (must be accessed by a student)
  - Navigation issues?
    - note we can get from Instructor to an appropriate grad by considering objects of the 'instructor' attribute that are actually grads
    - moving in the reverse direction from Grad to Instructor would be more difficult
- We'll use the Instructor context



Example

Consider the Academia constraint…

No grad student teaches a course in which he/she is enrolled.

```
context i:Instructor
   inv NoTeachingWhileEnrolled:

     …i.instructor.oclIsKindOf(Grad)…

     …i.coursesTaught…

     …i.instructor.oclAsType(Grad).taking…
```

Only consider instructors who are grad students

Navigate to the set of courses taught by the instructor

Downcast from Person to Grad

Navigate to the set of courses in which the instructor (as a grad) is enrolled

Example

Consider the Academia constraint…

No grad student teaches a course in which he/she is enrolled.

```
context i:Instructor
   inv NoTeachingWhileEnrolled:
     i.instructor.oclIsKindOf(Grad)
       implies
     i.coursesTaught
        ->intersection(i.instructor.oclAsType(Grad).taking)
        ->isEmpty
```

Only consider instructors who are grad students.

Intersection of coursesTaught and courses being taken must be empty.

For You To Do…

- Pause the lecture
- Download the academia-3.use file
- Add to this file the invariant declarations for the three Academia constraints given earlier
  - All instructors are either faculty or graduate students.
  - No one is waiting for a course unless someone is enrolled.
    - This is the one we haven't done yet – but it is easy!
  - No graduate student teaches a course that they are enrolled in.
- Load the resulting model into USE
- Create some system snapshots
  - some that satisfy your invariants
  - some that violate your invariants

For You To Do (continued)...

- Read the OMG-OCL specification linked on the course web page
- Continuing from the additions you made to academia-3.use, see how much of the rest of the Academia model that you can encode (the description of extensions (in two more stages) appears on the following slides)
  - Coding the remaining class definitions and associations is easy
  - Coding some of the remaining constraints will require you to study the OCL specification (which you need to do before the next lecture!)
- As usual, create some system snapshots...
  - some that satisfy your model
  - some that violate your model

I. Academia State

- Add an unique ID attribute for students
  - since we have integers and strings in OCL, there is more than one way to model IDs
- Add student transcripts
  - a transcript gives a set of courses associated with a student (the courses that a student has completed)
- Add prerequisite structure for courses
  - relates a course to courses that are prerequisites for it

I. Academia Constraints

- Enforce the uniqueness constraint for IDs
- A student can only take a course for which they have already taken the prereqs
- A course does not have itself as a prerequisite
  - An even stronger requirement is that there are no cycles in the prerequisite structure
- Realism: that there exists a course with prerequisites that someone is enrolled in

II. Academia State

- Add Departments
  - Instructors per
  - Courses per
  - Required courses
  - Student majors
- Add Faculty-Grad student relationships
  - Advisor
  - Thesis committee

II. Department Associations

- Each *instructor* is in a single *department*.
  - Each *department* has at least one *instructor*.
- Each *department* has some *courses*
  - Courses are in a single department
- Each *student* has a single *department* as his/her *major*
  - i.e., a department

II. Faculty-Student Associations

- A *graduate* student has exactly one *faculty* member as an *advisor*.
- A *faculty* member serves on five or fewer *graduate* student *committees*.

II. Academia Constraints

- Advisors are on their student's committees
- Students are advised by faculty in their major
- Only faculty teach required courses
- Required courses are a subset of the courses for a major
- Students must take at least one course from their major each semester
- Realism: There are at least two departments and some required courses.

Acknowledgements

- Material for this lecture is based on the following sources
  - Mark Richters and Martin Gogolla. OCL - Syntax, Semantics and Tools. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, pages 43-69. Springer, Berlin, LNCS 2263, 2001.
  - Chapter 7 (the OCL chapter) of the OMG-UML specification (version 1.3 – March 2000)
  - *The Unified Modeling Language Reference Manual.* Rumbaugh, Jacobson, Booch. Addison-Wesley, 1999