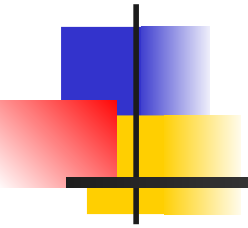
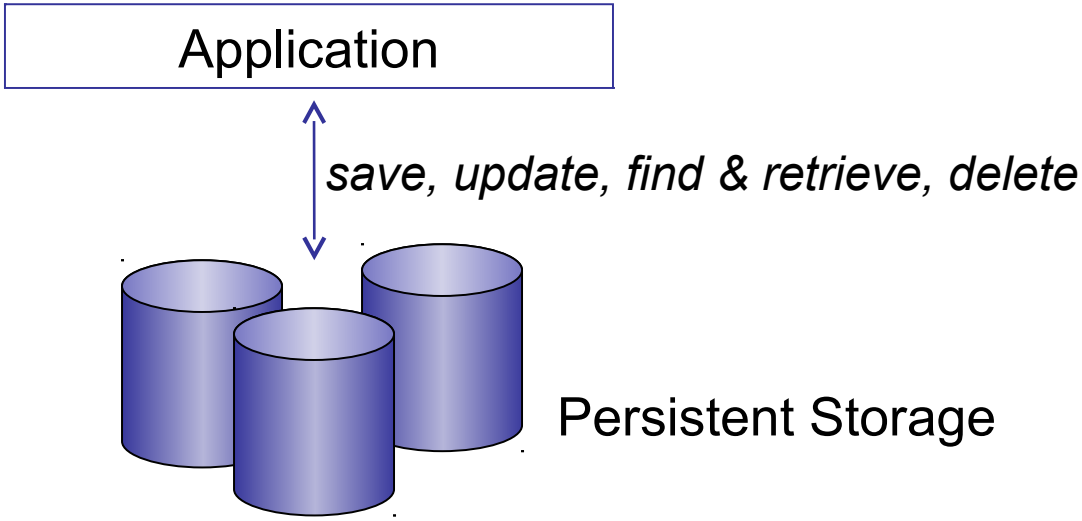


Persistence and Object-Relational Mapping

James Brucker

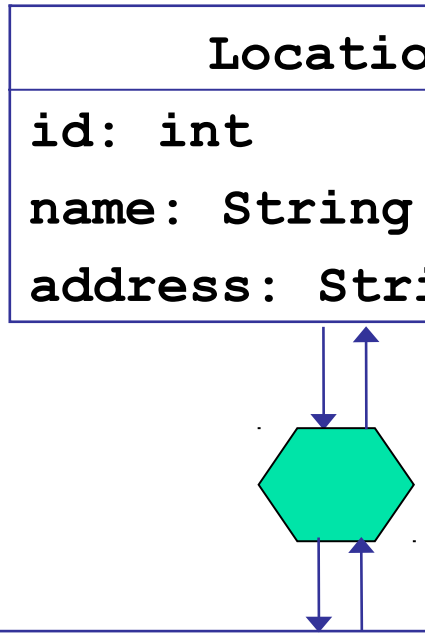
Goal

- Applications need to save data to *persistent storage*.
- Persistent storage can be database, directory service, files, spreadsheet, ...
- We want to ***abstract*** (hide) details of how data is being saved and restored.



Saving/Restoring Objects

An object's attributes are similar to the fields in a table.



Save object as row in a table, retrieve row of data and (re)create an object

LOCATIONS table		
id (PK)	name	address
101	Kasetsart	50 Ngamwongwang Rd, ...
102	Pizza Hut	44 Pahonyotin Rd, Jatujak,..

Object-Relational Mapping

Purpose

- save object as a row in a database table
- create object using data from a table
- save and recreate *associations* between objects

Design Goals

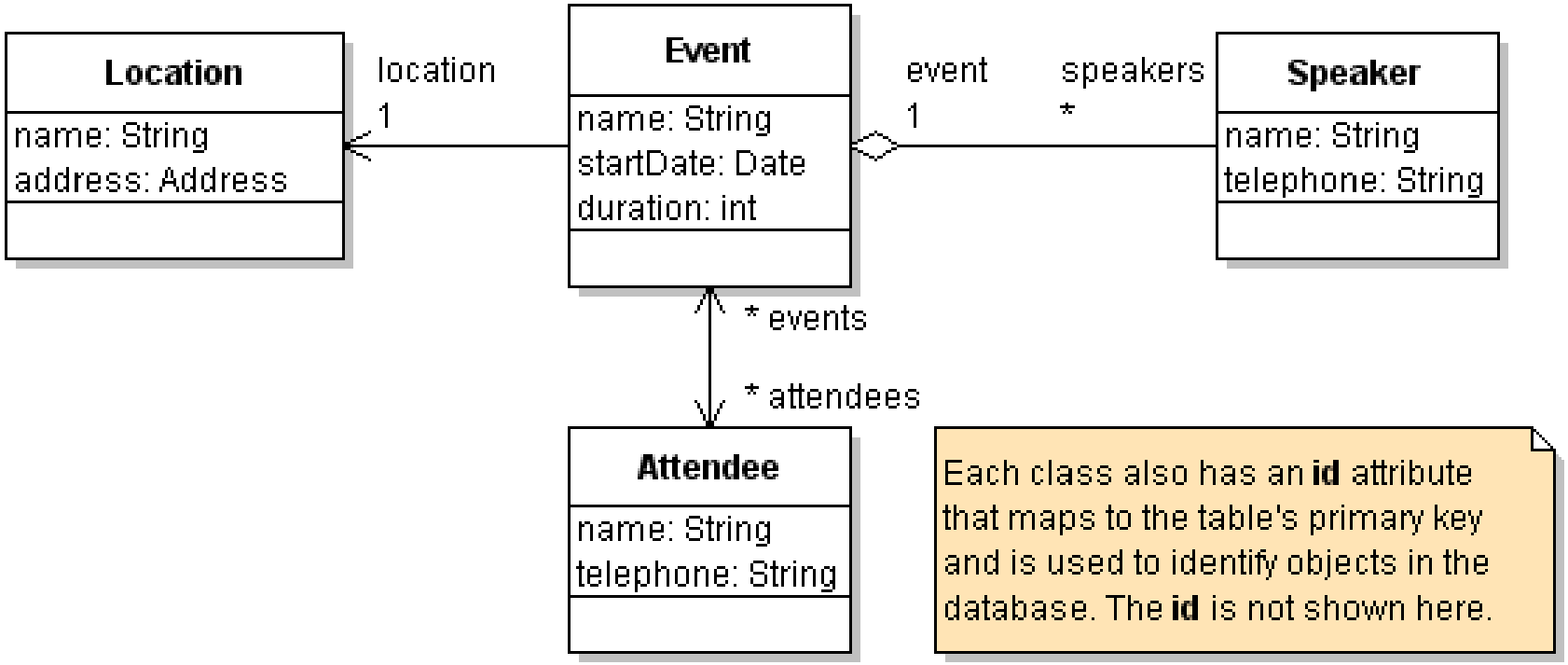
- **separate** the O-R mapping service from our application
- **abstract details** of how its done -- app just calls save()
- localize the impact of change in the database.

Object-Relational Mismatch

- Database structure isn't the same as objects.
- Objects have associations and collections
  databases have relations between tables.
- Objects are unique.
  Can be hard to preserve uniqueness when an object is saved & restored multiple times.

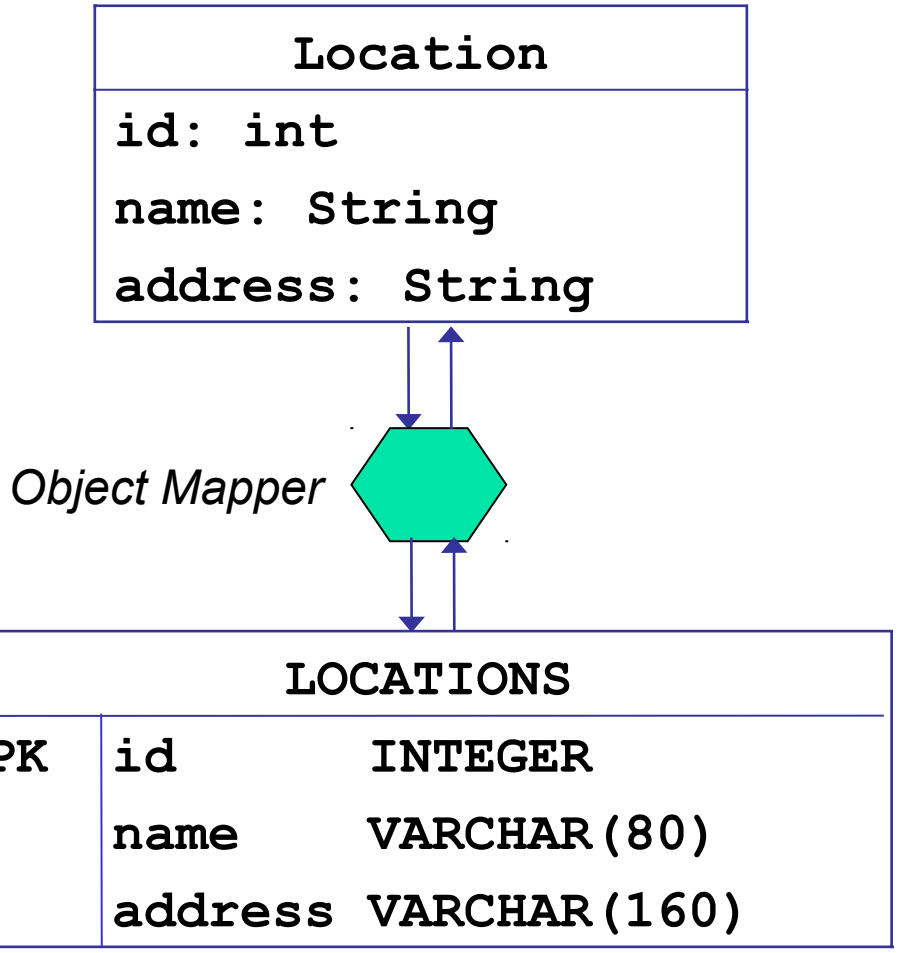
An Example

An Event Manager application with these classes:



Object-Relational Mapping

Map between an object and a row in a database table.



Class

*should have an **identifier** attribute*

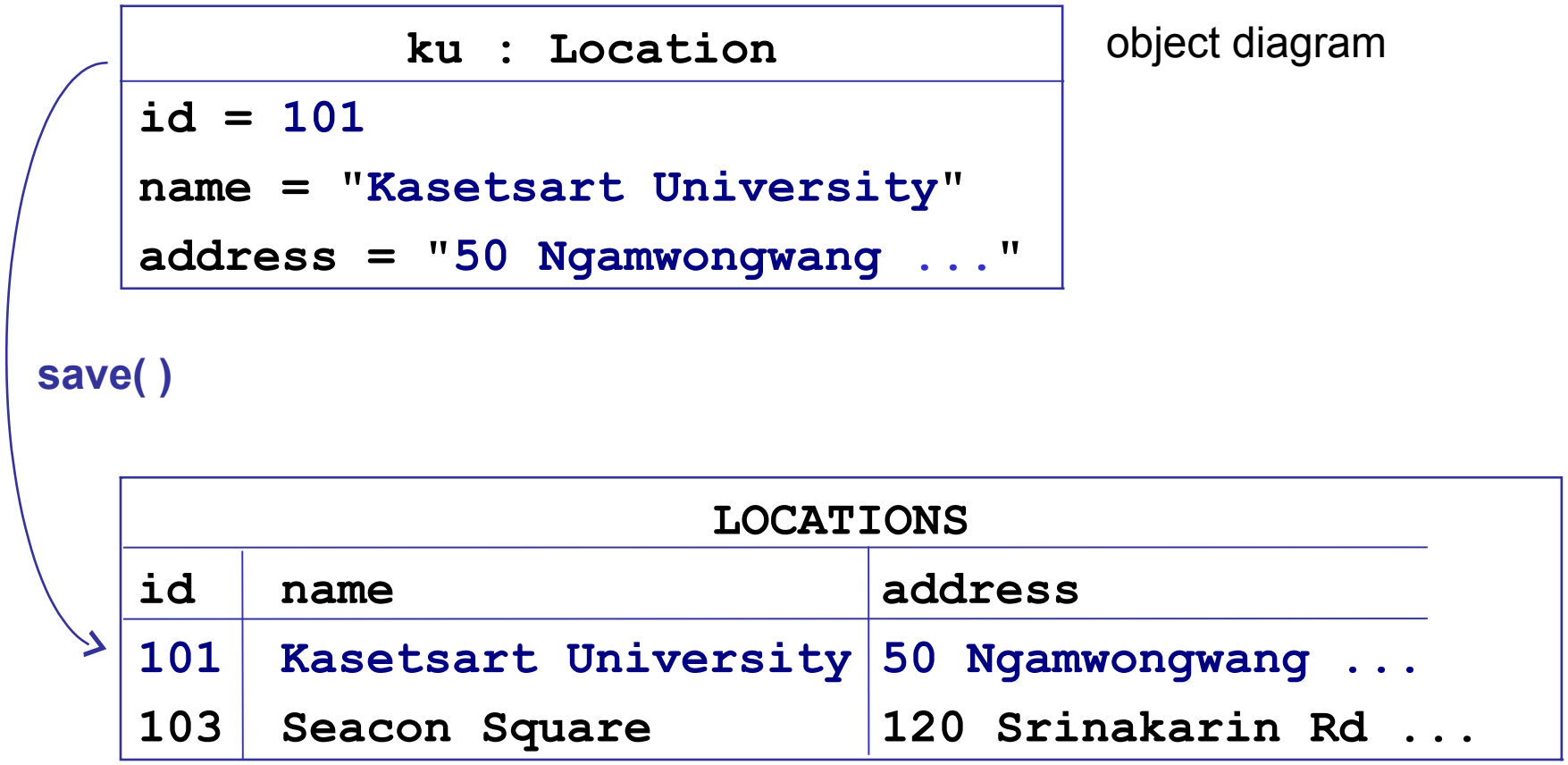
Object Mapper

save objects to rows in tables, restore data as objects

Database Table

identifier is usually the primary key of table

Mapping an Object



Code for ORM

```
ku = Location( "Kasetsart University",
               "50 Ngamwongwang Road, Bangkok" );
# save it!
object_mapper.save( ku )
# object_mapper assigns an id to object
print( ku.id )
101
```

Issues:

- mapper should choose a unique ID for each saved object
- what if same data (Kasetsart University) is already in the table?

Finding and Retrieving an Object

```
# find by id (only one match possible)
ku1 = object_mapper.find(id=101)
# find by name (may have many matches)
list = object_mapper.find(name="Kasetsart University")
```

Does object_mapper always return the same object?

```
ku1 = object_mapper.find(id=101)
ku2 = object_mapper.find(id=101)
```

ku1 == ku2 => true or false?

Object-Relational Operations: CRUD

Most Common persistence operations are:

Create save a new object in the database

Retrieve an object from the database

Update data for an object already saved in database

Delete object data from the database

Which one is most *Complex*?

Of the 4 CRUD operations, which do you think is the most complex case?

Create save a new object in the database

Retrieve an object from the database

Update data for an object already saved in database

Delete object data from the database

Providing CRUD

Simple:

Create orm.save(object)

Update orm.update(object)

Delete orm.delete(object) or orm.delete(object.id)

Complex:

Retrieve by id

Retrieve all

Retrieve using query expression: address contains "Bangkok" or city.population > 1000000

Retrieve first 10 objects, sorted by date

Try it in Django

```
cmd> python manage.py shell
>>> from polls.models import Question
>>> q = Question(question_text="Understand ORM?")
>>> q.pub_date = datetime.now()
>>> q.id
(nothing is printed)
>>> q.save( )
>>> q.id
6
>>> Question.objects.all( )
<QuerySet: [..., <Question: Understand ORM?>,...
```

Try it in Django

```
# Change something and update object in database
>>> q.question_text = "Next question?"
>>> q.save()
# Did it update the question in database?
>>> Question.objects.get( id=6 )
<Question: Next Question?>
# Can we delete it from database?
>>> q.delete( )
>>> Question.objects.get( id=6 )
DoesNotExist: Question matching query does not
              exist.
```

Design of Persistence Service

There are 2 Design Patterns for a persistence service (the "object mapper" in examples):

Data Access Objects - define a class that is responsible for persisting other objects (save, query, delete, ...)

Active Object Pattern - objects provide CRUD operations themselves.

- Behavior usually defined in a common superclass.

Data Access Object Pattern

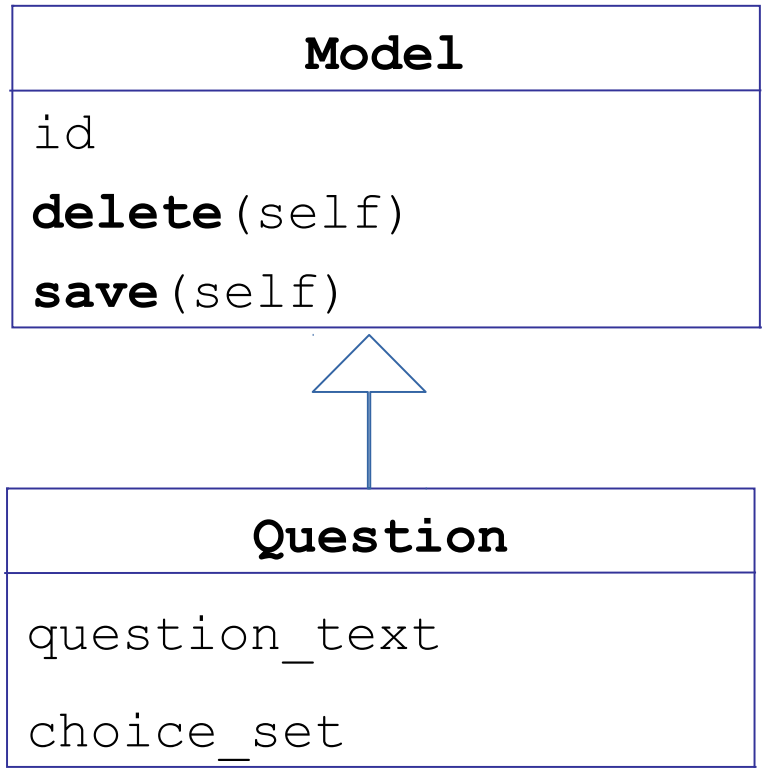
- A separate class provides persistence operations.
  - Append "Dao" to the class name, e.g. **`EventDao`**.

EventDao
`find( id ): Event`
`query( expression ): Event[*]`
`save( event )`
`update( event )`
`delete( event )`
`count( )`

This is like "object mapper" in previous slides.

Active Object Pattern

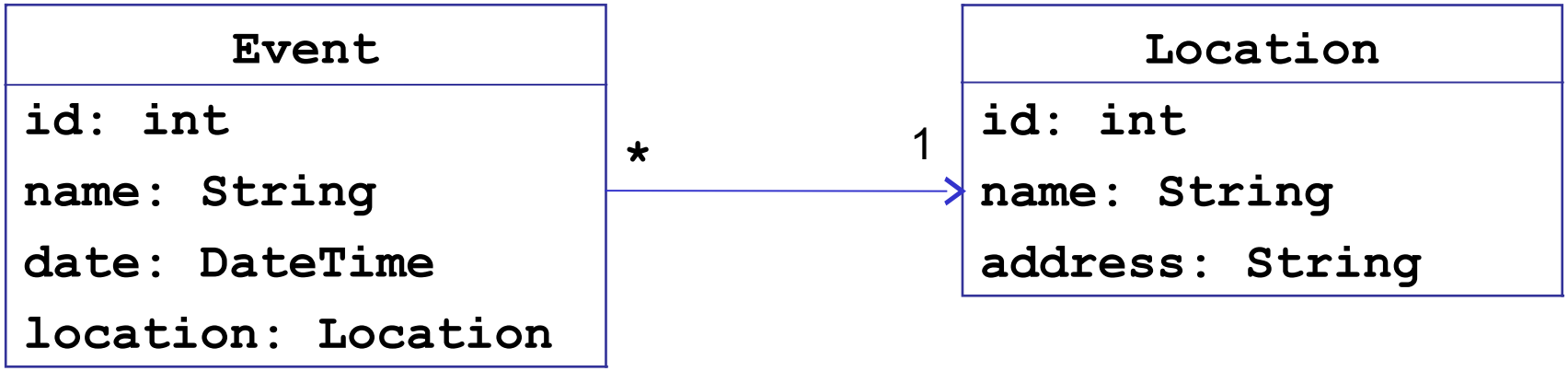
- Entity classes subclass a common super-class that defines persistence operations.
- Django uses this pattern.
- Object saves itself.
- The mysterious **`objects`** class attribute *should* be in Model, too.



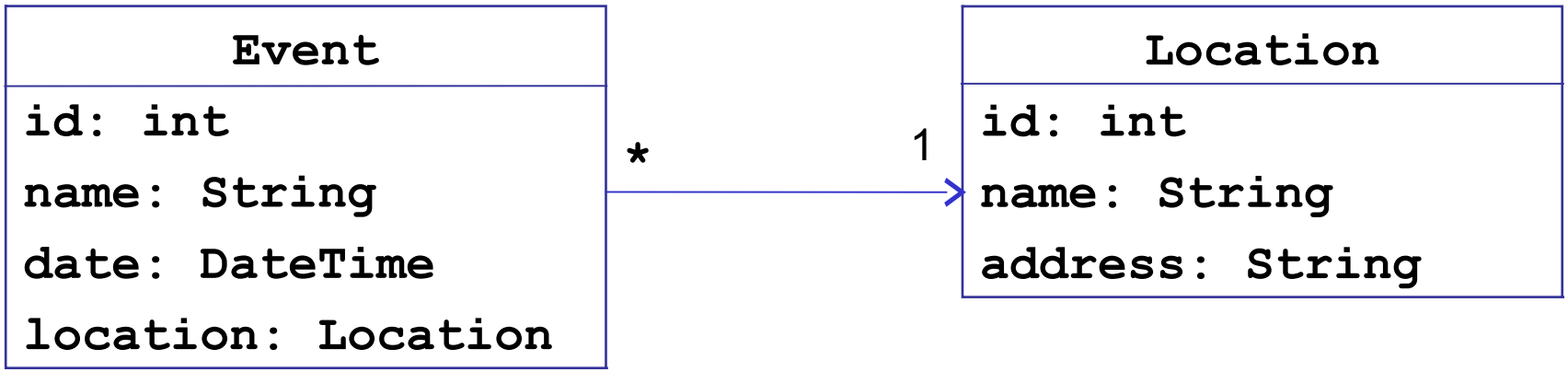
How to Save Associations?

Objects often have associations (references) to other objects. How can we save associations?

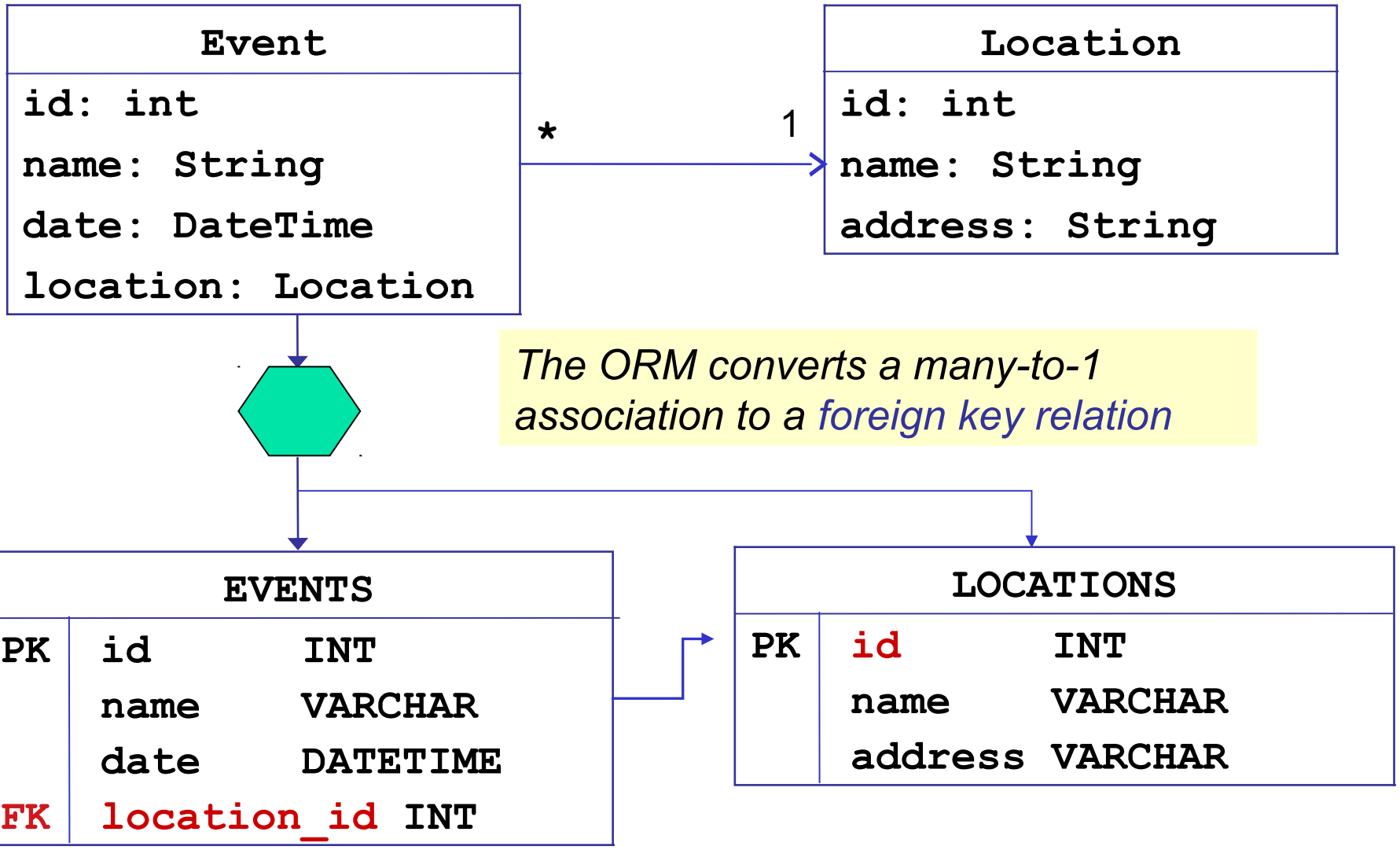
An Event has a Location:



O-R Mapping of n-to-1 Associations



O-R Mapping of n-to-1 Associations



n-to-1 association in Django

You specify only the related class, not the name of field in the database.

```
class Event(models.Model):
    name = models.CharField('name',max_length=80)
    date = models.DateTimeField('date')
    location = models.ForeignKey(Location)
```

Save What?

```
event = Event( "BarCamp 2019" )
ku = Location( "Kasetsart University", "..." )
# Yeah! Bar Camp is coming to KU!
event.set_location( ku )
event.set_date( datetime.date(2019, 11, 25) )
# save the event
object_mapper.save( event )
```

Did object mapper save the location, too?

Or do we have to save location ourselves?

Fetching an Event

```
# Retrieve the event
event2 = object_mapper.find( name="BarCamp 2019" )
# object mapper finds the event...
print( event2.name )
"BarCamp 2019"
# did it recreate the location, too?
print( event2.location.name )
???
```

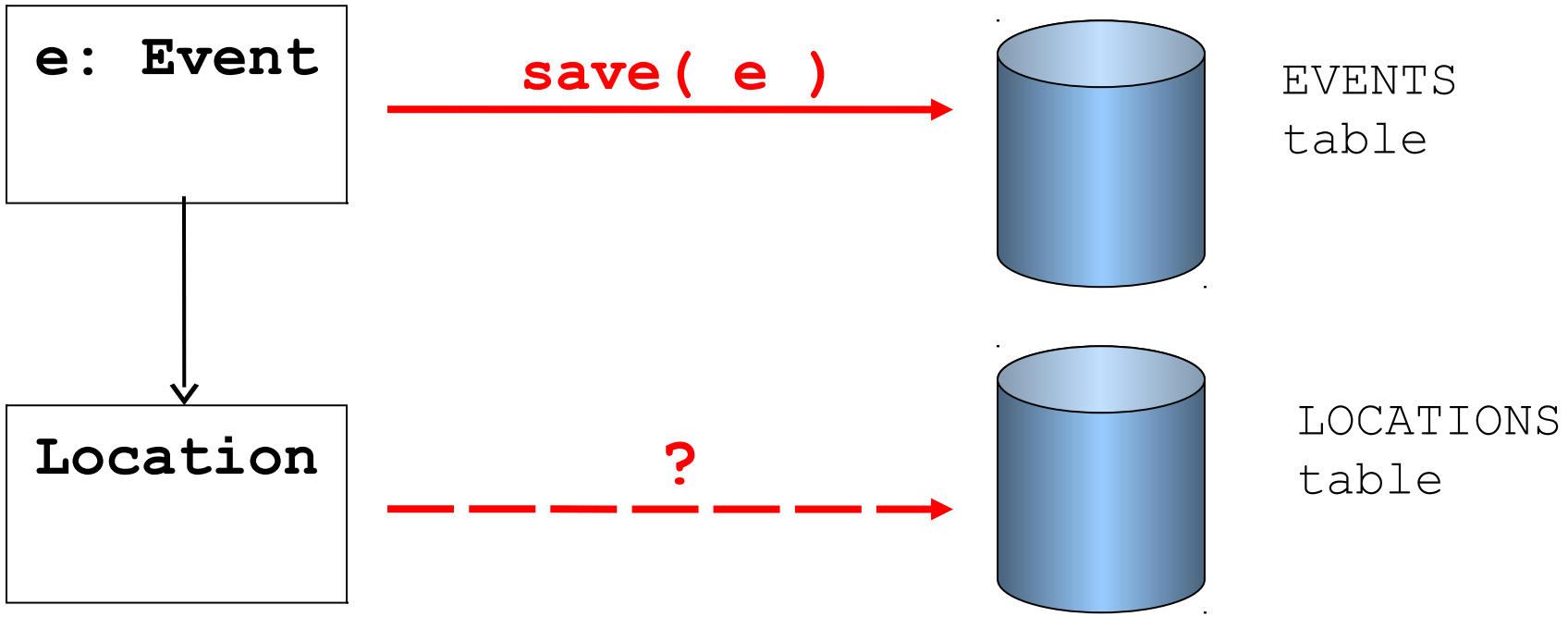
When we retrieve an event,

does the ORM retrieve the location object, too?

Cascading

When you save, update, delete an object in database...

are associated objects also saved/updated/deleted?



Cascading

Cascading means that an operation on one object should propagate (or cascade) to related objects.

Cascade = true: when you save an Event, save its Location, too (if necessary).

Cascade = false: when you save an Event, don't save its Location. Programming should save Location *first* so that Location has an id.

Frameworks Provide Cascading

In JPA, using annotations:

```
@Entity
class Event {

  @OneToMany(mappedBy="event", cascade=PERSIST)
  private List<Person> attendees;
```

NONE
PERSIST
REFRESH
REMOVE
ALL

Does Django do cascading save?

Try it with the polls app:

```
>>> c1 = Choice(choice_text="First Choice")
>>> q = Question(question_text="What's your choice?")
>>> q.choice_set.add( c1 )
TraceBack...
ValueError: <Choice: First Choice> isn't saved.
```

Looks like Django wants you to save associated objects yourself.

Other Kinds of Associations

There are other cases that ORM must handle:

- 1-to-many and many-to-many associations
- object containing an *ordered* collection, such as List.

Django invisibly handles all these.

For other ORM frameworks like SQLAlchemy (Python) or JPA (Java) it helps to understand how framework handles associations.
Especially cascading save/delete and lazy or eager fetching.

Django Query Methods

`Model.objects` provides many query methods and a simple query syntax.

Also a create( ) method to create & save in one step.

You should learn them.

Example of a Dumb Query

Find all poll questions containing the word "programming"

```
questions = Question.objects.all()
# Create a list of questions containing "programming"
qlist = [ q for q in questions
    if q.question_text.find("programming") >= 0 ]
```

Why is this inefficient?

Python Quiz:

what is `[q for q in questions if ...]` called?

Smarter Query

Let the database filter results for you:

```
questions = Question.objects.filter(
                question_text__icontains='programming')
# questions is a QuerySet.  Convert to a list
qlist = list(questions)
```

Why is this more efficient?

Django's query language uses __ (double underscore) to precede an operator.

```
# Find questions with pub_date >= date(2019,1,1)
Question.objects.filter(
      pub_date__gte=datetime.date(2019,1,1) )
```

Learn More

Making Queries in the Django documenation.

https://docs.djangoproject.com/en/2.2/topics/db/queries/

* You don't need the URL, of course -- because you already have the Django documentation on your own computer, right?