UML Diagrams

What is UML?

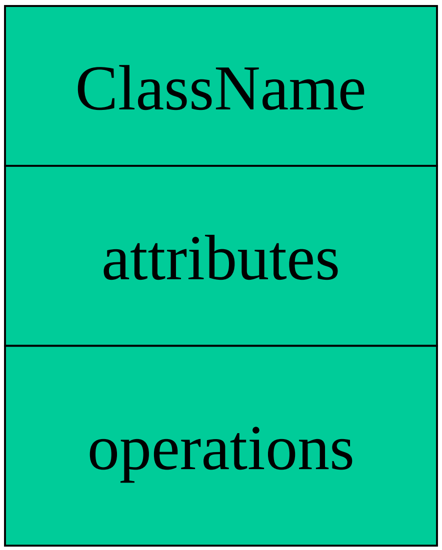
- Standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, business modeling and other non-software systems.
- The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.
- The UML is a very important part of developing object oriented software and the software development process.
- Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

UML Diagrams

- **Use-Case** *(relation of actors to system functions)*
- **Class** *(static class structure)*
- **Object** *(same as class - only using class instances – i.e. objects)*
- **State** *(states of objects in a particular class)*
- **Sequence** *(Object message passing structure)*
- **Collaboration** *(same as sequence but also shows context - i.e. objects and their relationships)*
- **Activity** *(sequential flow of activities i.e. action states)*
- **Component** *(code structure)*
- **Deployment** *(mapping of software to hardware)*

Class Diagrams

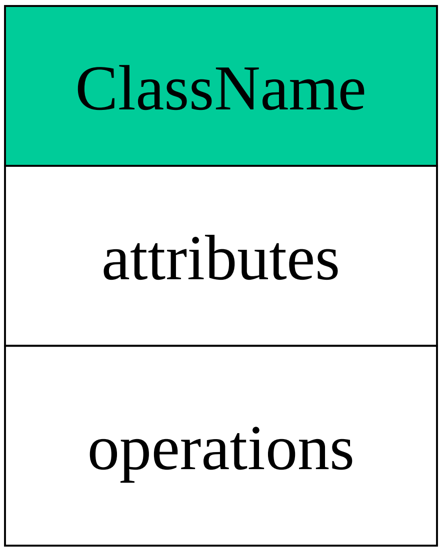
Classes



A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

Class Names



The **name of the class** is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.

Class Attributes

Person
name : String address : Address birthdate : Date ssn : Id

An ***attribute*** is a named property of a class that **describes the object being modeled**.

In the class diagram, attributes appear in the second compartment just below the name-compartment.

Class Attributes (Cont'd)

Person
name : String address : Address birthdate : Date / age : Date ssn : Id

Attributes are usually listed in the form:

attributeName : Type

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For example, a Person's age can be computed from his birth date. A derived attribute is designated by a preceding '/' as in:

/ age : Date

Class Attributes (Cont'd)

Person
+ name : String # address : Address # birthdate : Date / age : Date - ssn : Id

Attributes can be:

+ public
# protected
- private
/ derived

Class Operations

Person
name : String address : Address birthdate : Date ssn : Id
eat sleep work play

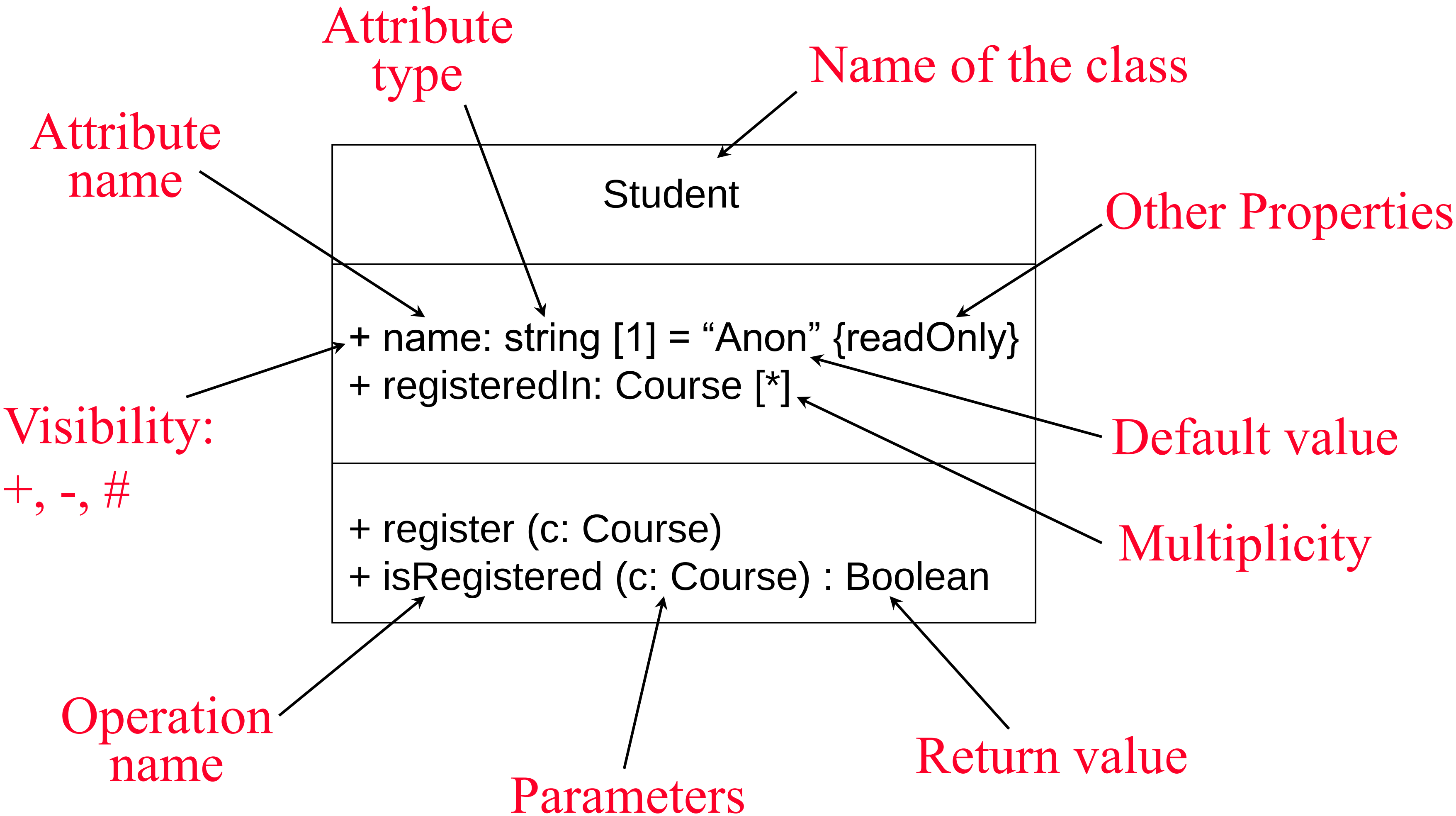
Operations describe the class behavior and appear in the third compartment.

Class Operations (Cont'd)

PhoneBook
newEntry (n : Name, a : Address, p : PhoneNumber, d : Description) getPhone (n : Name, a : Address) : PhoneNumber

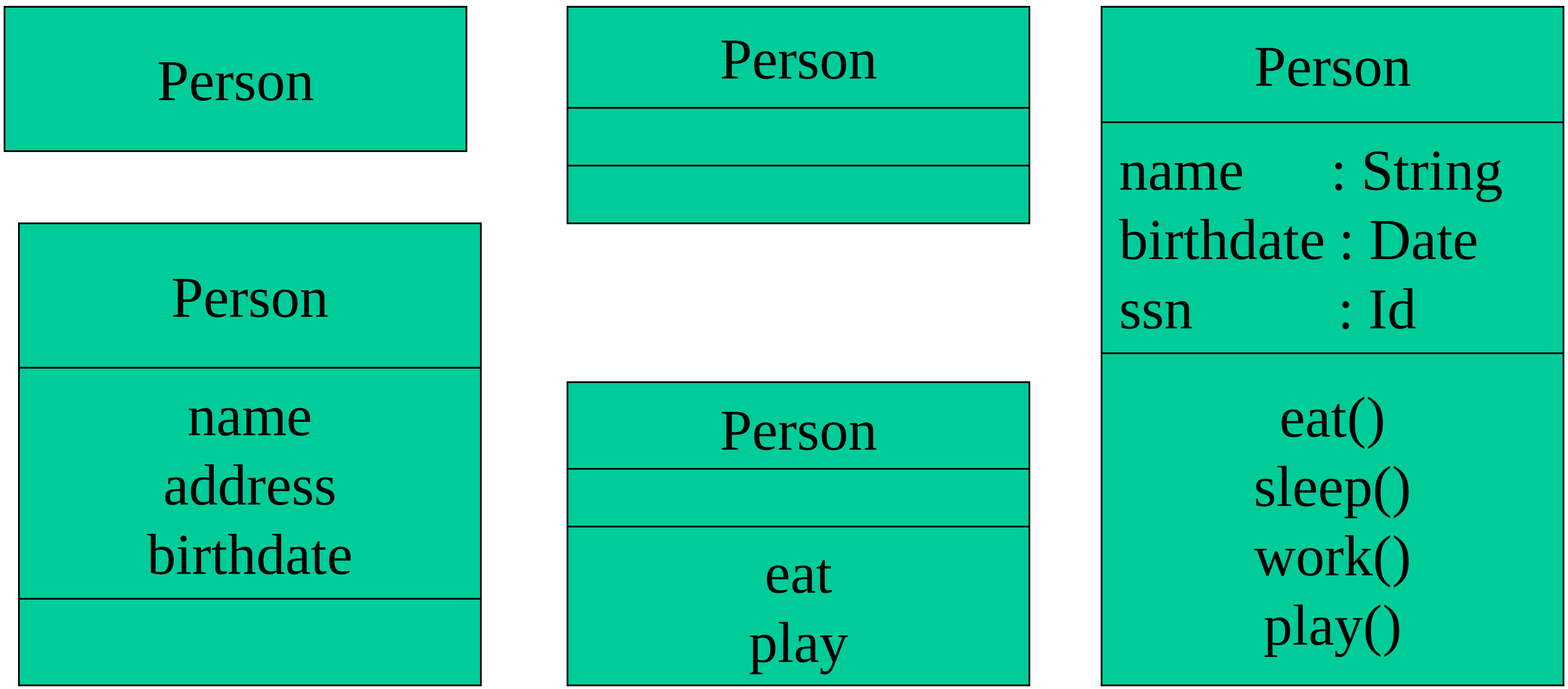
You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type.

The full notation…



Depicting Classes

When drawing a class, you needn’t show attributes and operation in every diagram.



UML Class-to-Java Example

```
Public class UNIXaccount
{
  public string username;
  public string groupname = "csai";
  public int filesystem_size;
  public date creation_date;
  private string password;
  static private integer no_of_accounts = 0
  public UNIXaccount()
  {
    //Other initialisation
    no_of_accounts++;
  }
  //Methods go here
};
```

UNIXaccount
+ username : string + groupname : string = “staff" + filesystem_size : integer + creation_date : date - password : string - no_of_accounts : integer = 0

Operations (Methods)

```
Public class Figure
{
  private int x = 0;
  private int y = 0;
  public void draw()
  {
    //Java code for drawing figure
  }
};
```

Figure
- x : integer = 0 - y : integer = 0
+ draw()

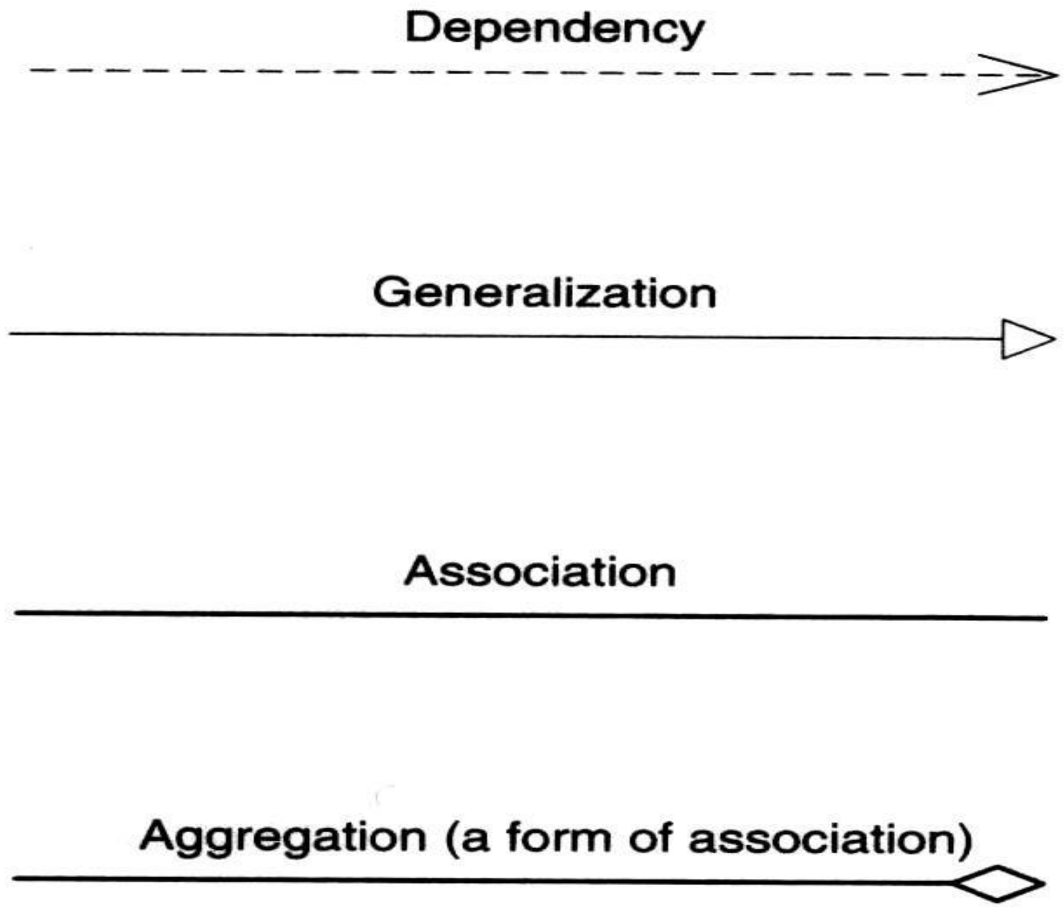
Relationships

In UML, object interconnections (logical or physical), are modeled as relationships.

There are three kinds of relationships in UML:

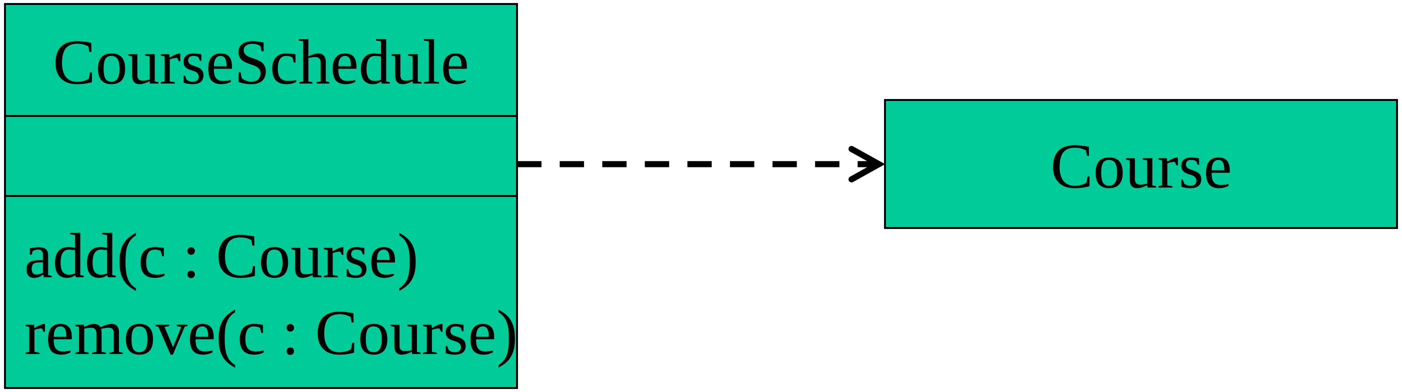
- dependencies
- generalizations
- associations

UML Relationships

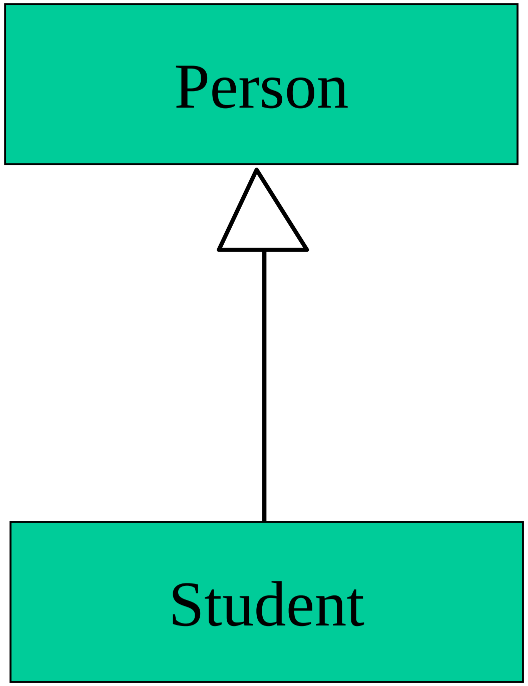


Dependency Relationships

A *dependency* indicates a semantic relationship between two or more elements. The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.



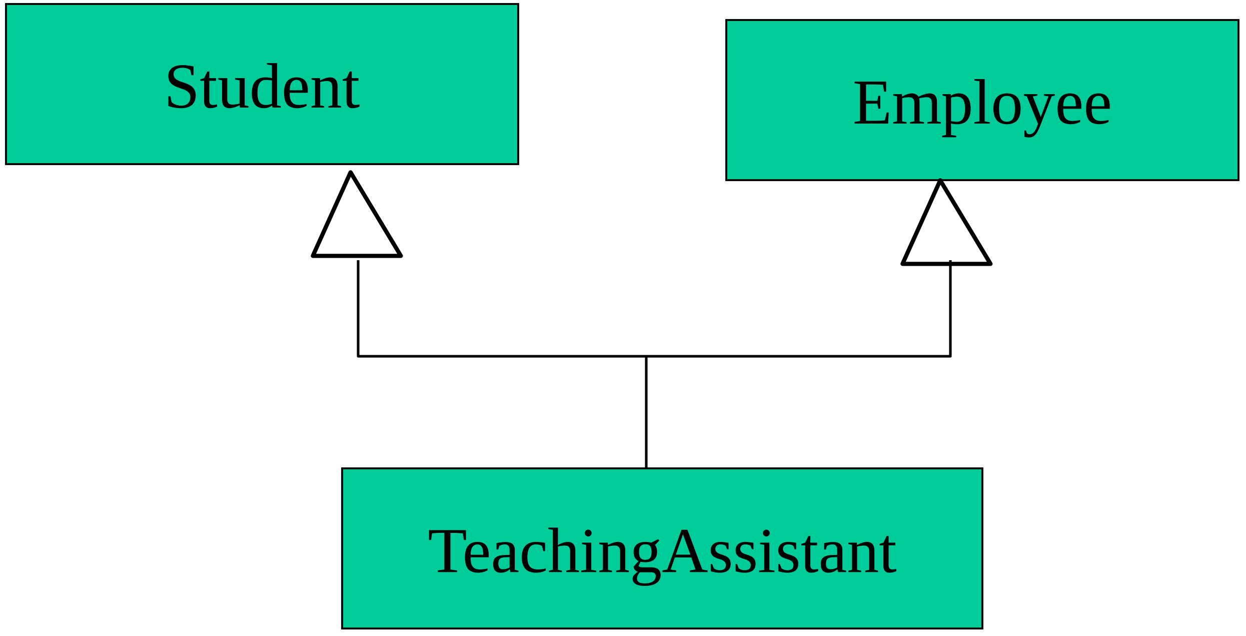
Generalization Relationships



A *generalization* connects a subclass to its superclass. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

Generalization Relationships (Cont'd)

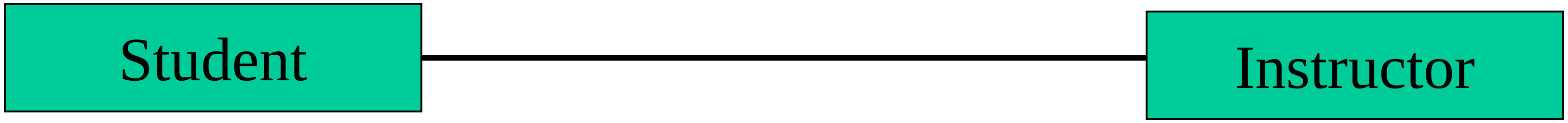
UML permits a class to inherit from multiple superclasses, although some programming languages (*e.g.*, Java) do not permit multiple inheritance.



Association Relationships

If two classes in a model need to communicate with each other, there must be link between them.

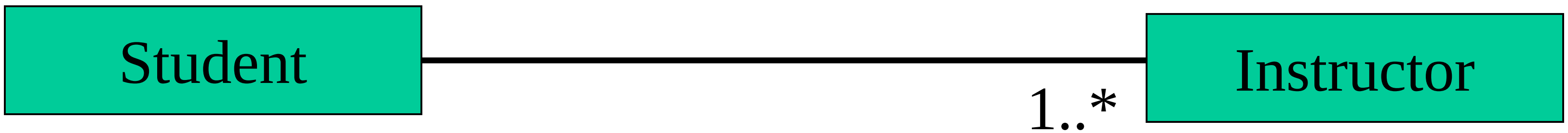
An *association* denotes that link.



Association Relationships (Cont'd)

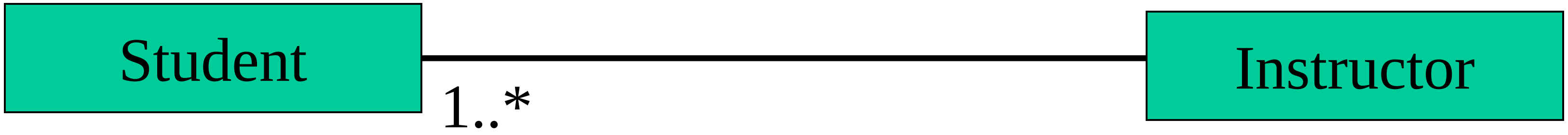
We can indicate the ***multiplicity*** of an association by adding *multiplicity adornments* to the line denoting the association.

The example indicates that a *Student* has one or more *Instructors*:



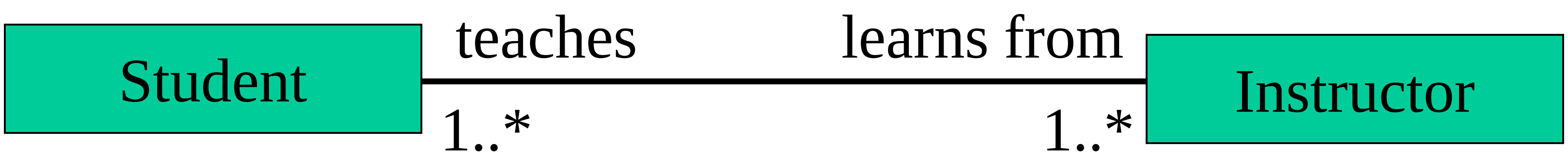
Association Relationships (Cont'd)

The example indicates that every *Instructor* has one or more *Students*:



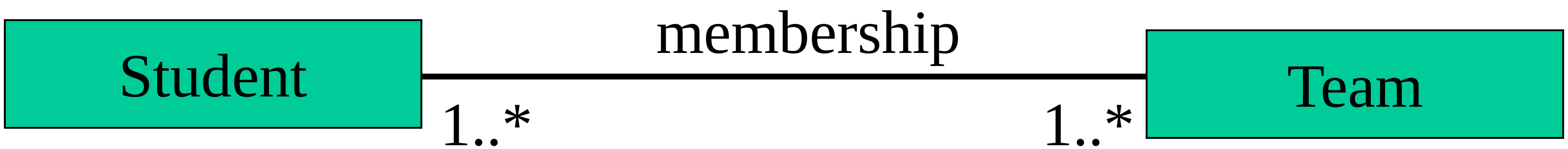
Association Relationships (Cont'd)

We can also indicate the behavior of an object in an association (*i.e.*, the *role* of an object) using *rolenames*.



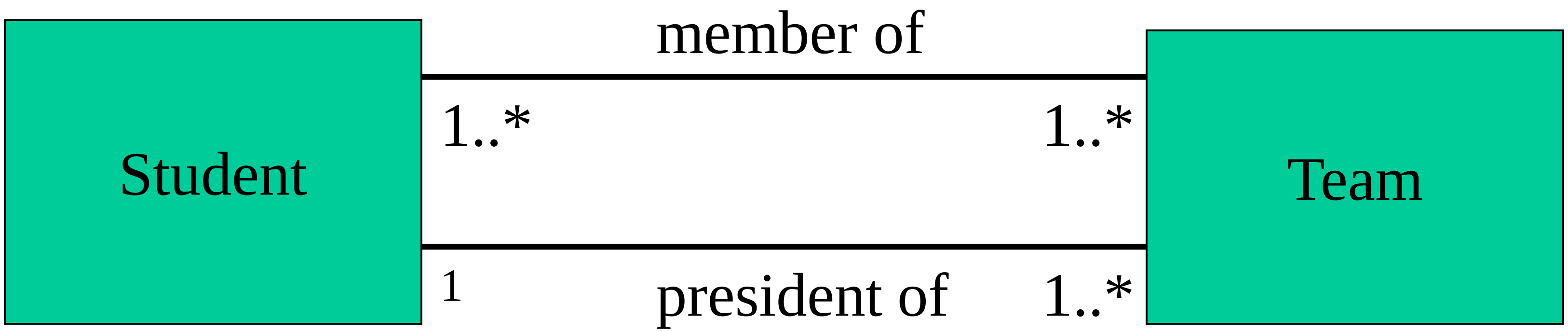
Association Relationships (Cont'd)

We can also name the association.



Association Relationships (Cont'd)

We can specify dual associations.

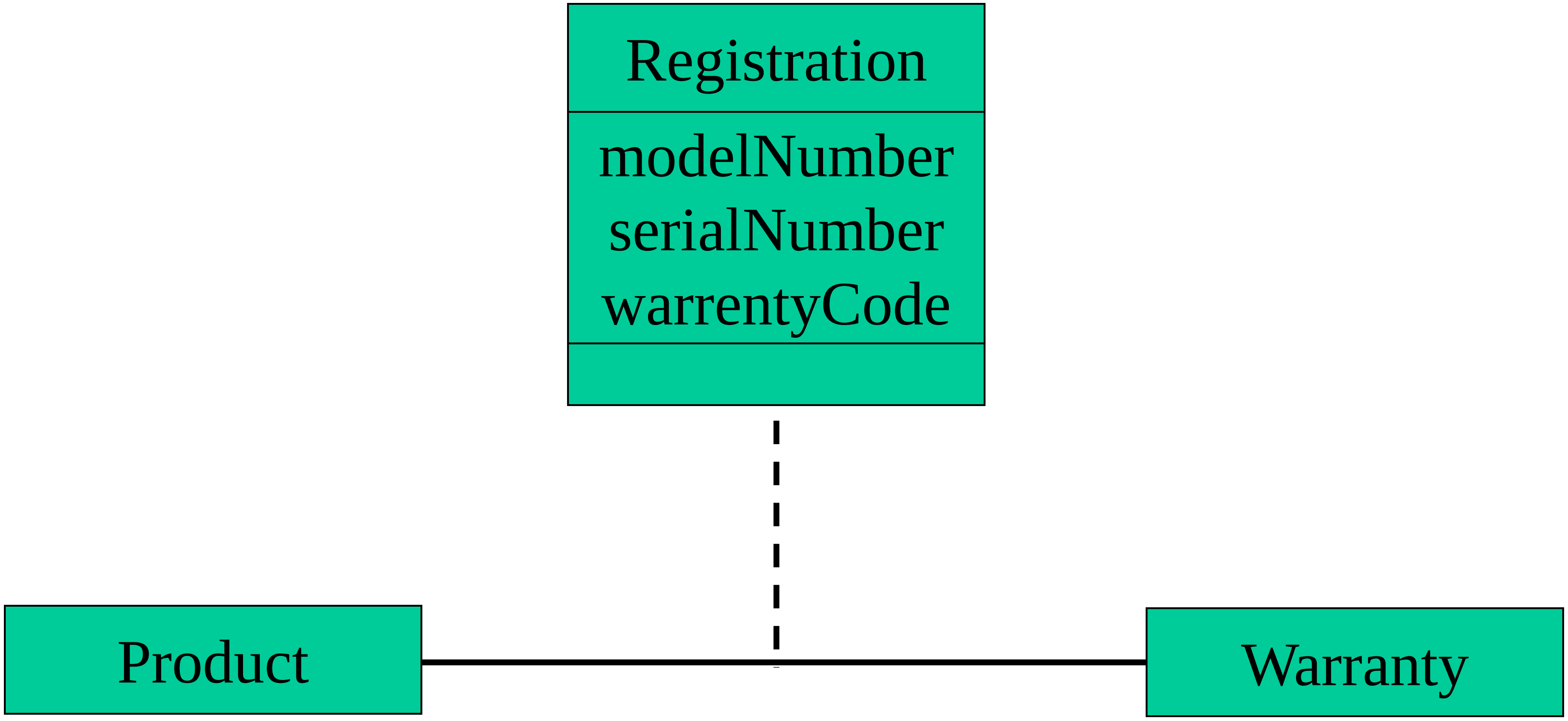


Association Relationships (Cont'd)

We can constrain the association relationship by defining the *navigability* of the association. Here, a *Router* object requests services from a *DNS* object by sending messages to (invoking the operations of) the server. **The direction of the association indicates that the server has no knowledge of the *Router*.**

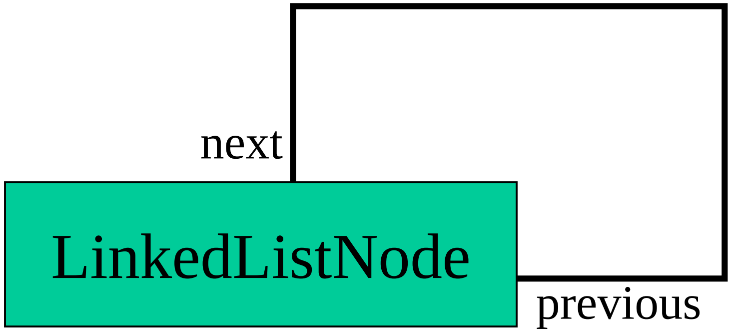
Association Relationships (Cont'd)

Associations can also be objects themselves, called *link classes* or an *association classes*.



Association Relationships (Cont'd)

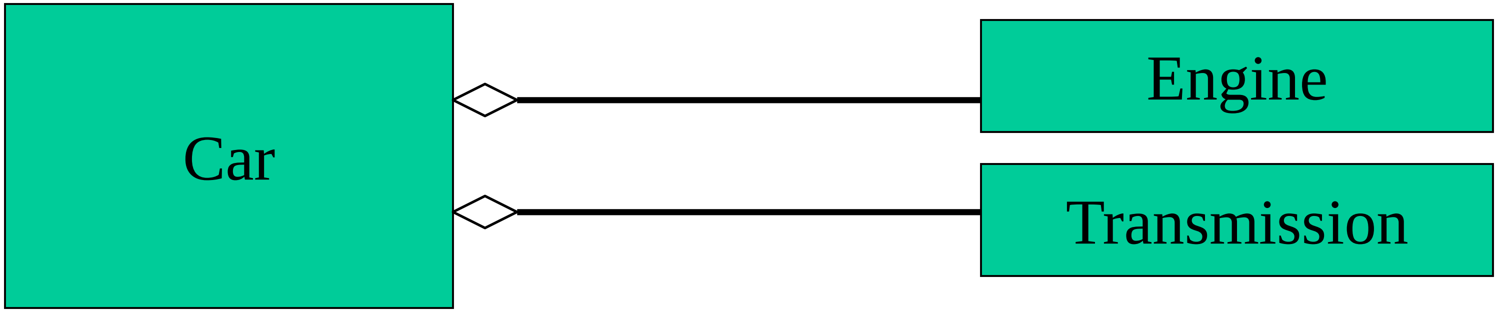
A class can have a *self association*.



Association Relationships (Cont'd)

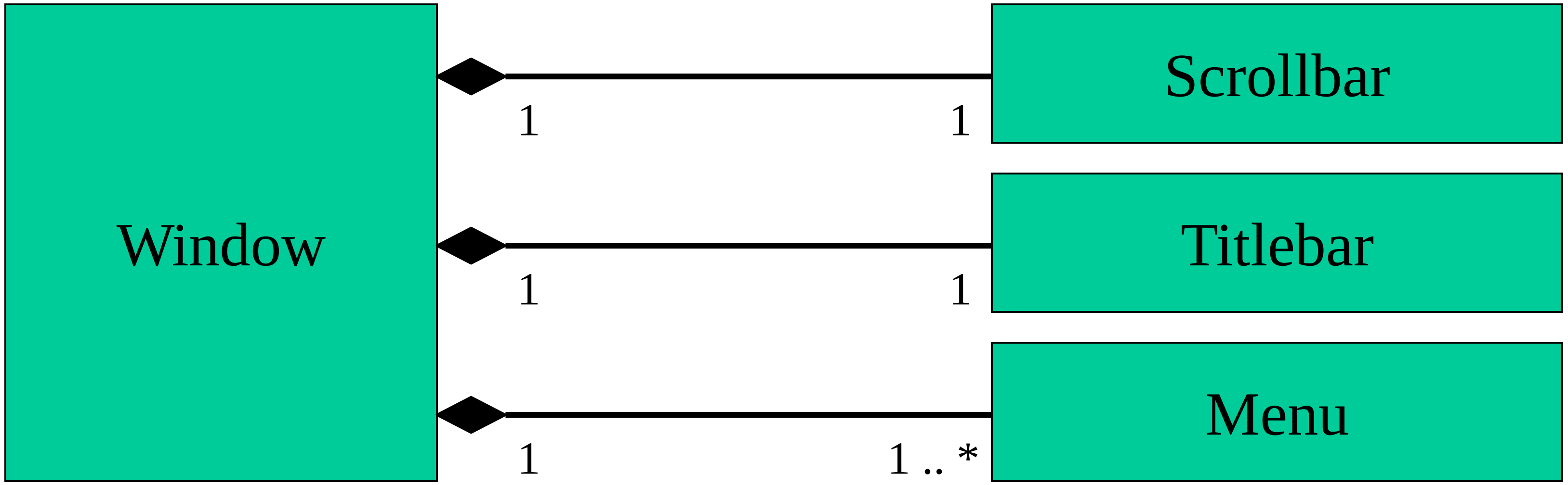
We can model objects that contain other objects by way of special associations called *aggregations* and *compositions*.

An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, **where the part can exist independently from the aggregate**. Aggregations are denoted by a hollow-diamond adornment on the association.



Association Relationships (Cont'd)

A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.*, they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.



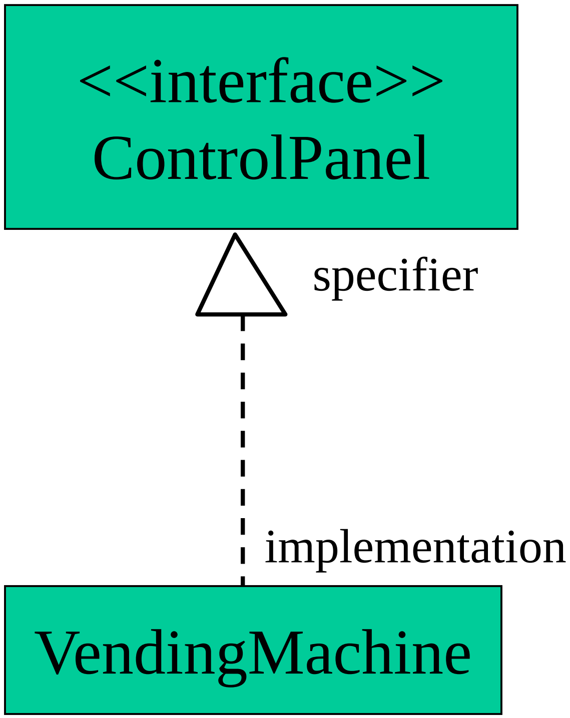
Interfaces

<<interface>>
ControlPanel

An *interface* is a named set of operations that specifies the behavior of objects without showing their inner structure.

It can be rendered in **the model by a one- or two-compartment rectangle**, with the *stereotype* <<interface>> above the interface name.

Interface Realization Relationship



A *realization* relationship connects a class with an interface that supplies its behavioral specification.

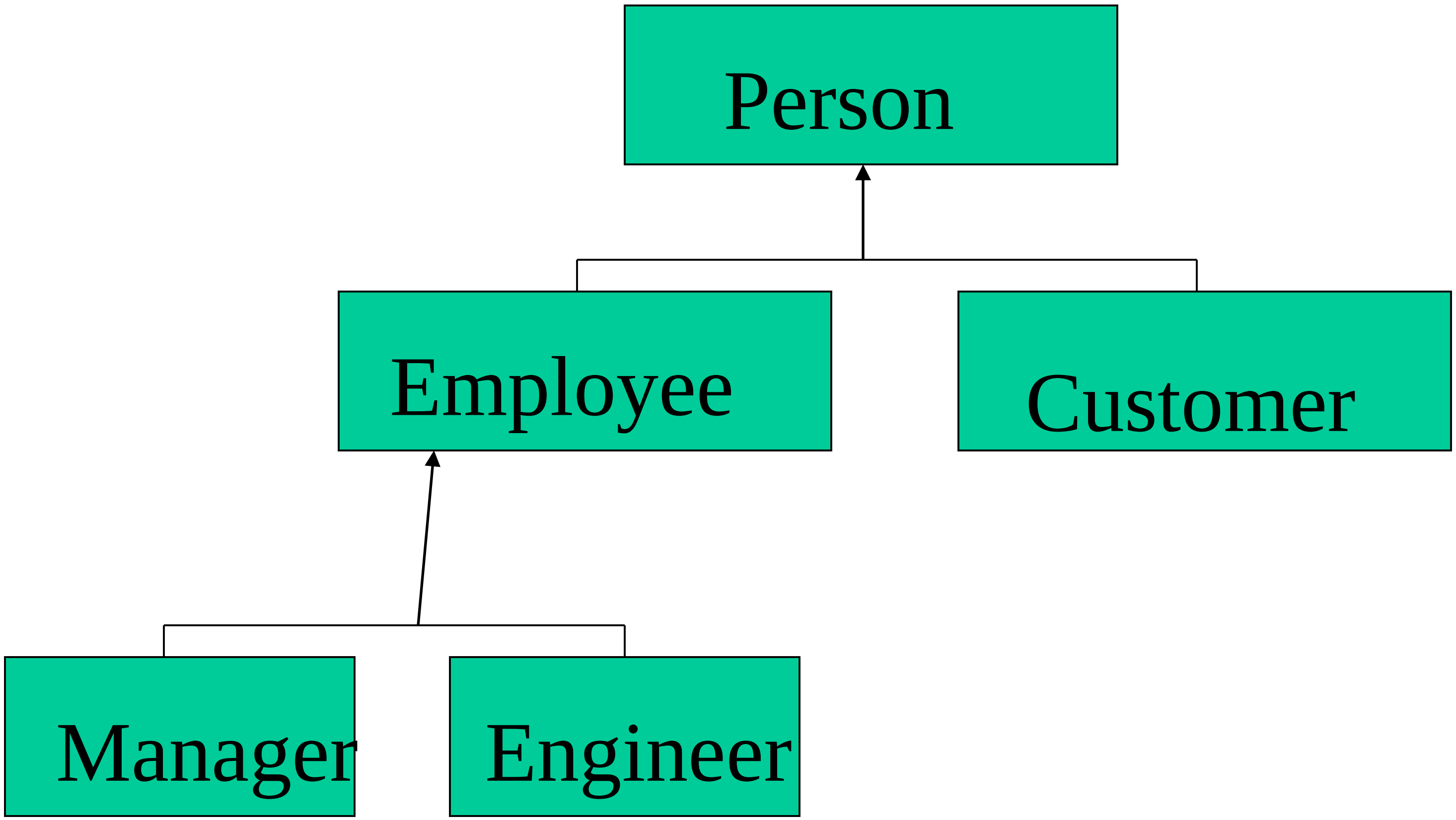
It is rendered by a dashed line with a hollow triangle towards the specifier.

Association (More Examples)

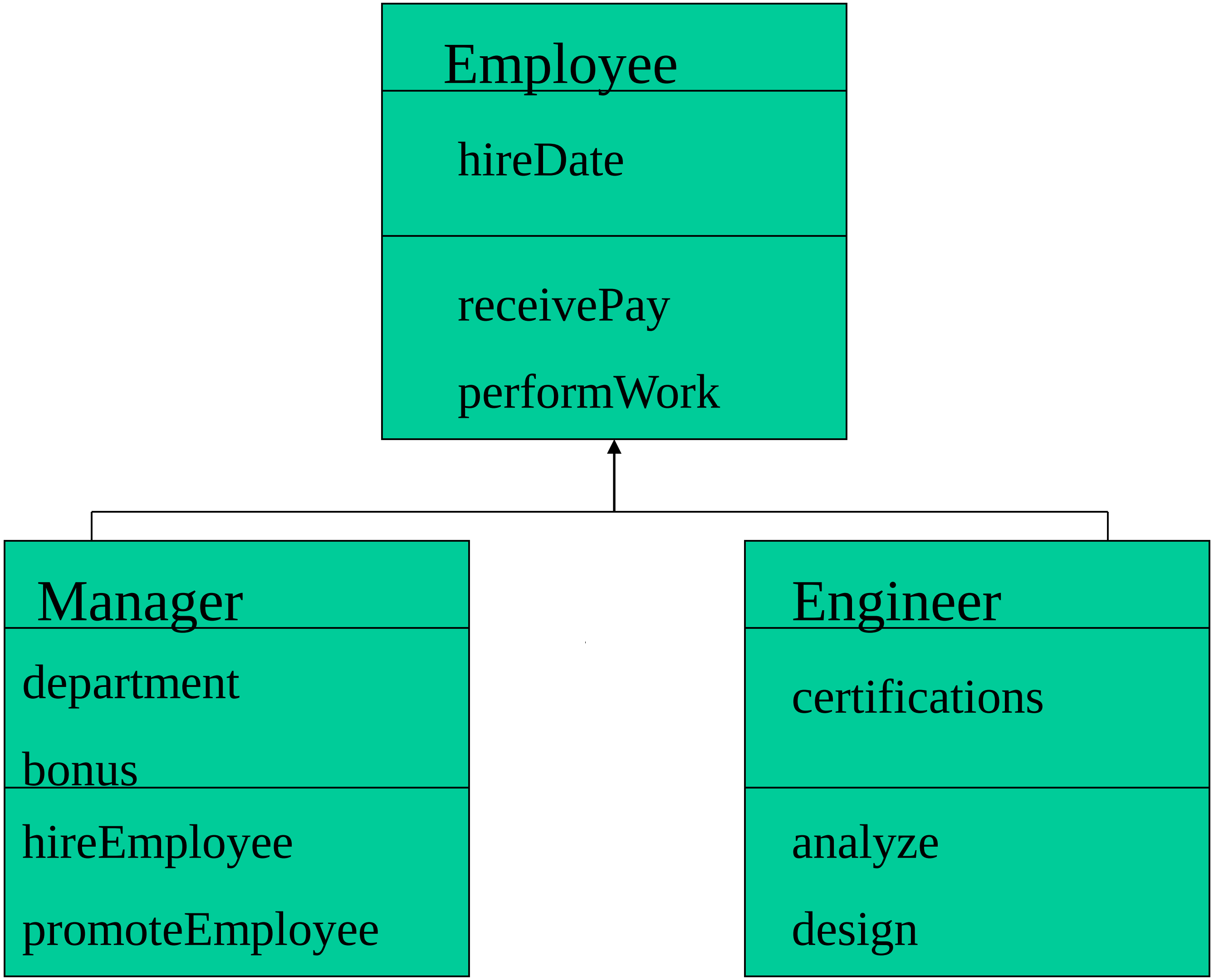
Association Relationship

- Name of relationship type shown by:
  - drawing line between classes
  - labeling with the name of the relationship
  - indicating with a small solid triangle beside the name of the relationship the direction of the association

Generalization Relationship



Generalization Relationship



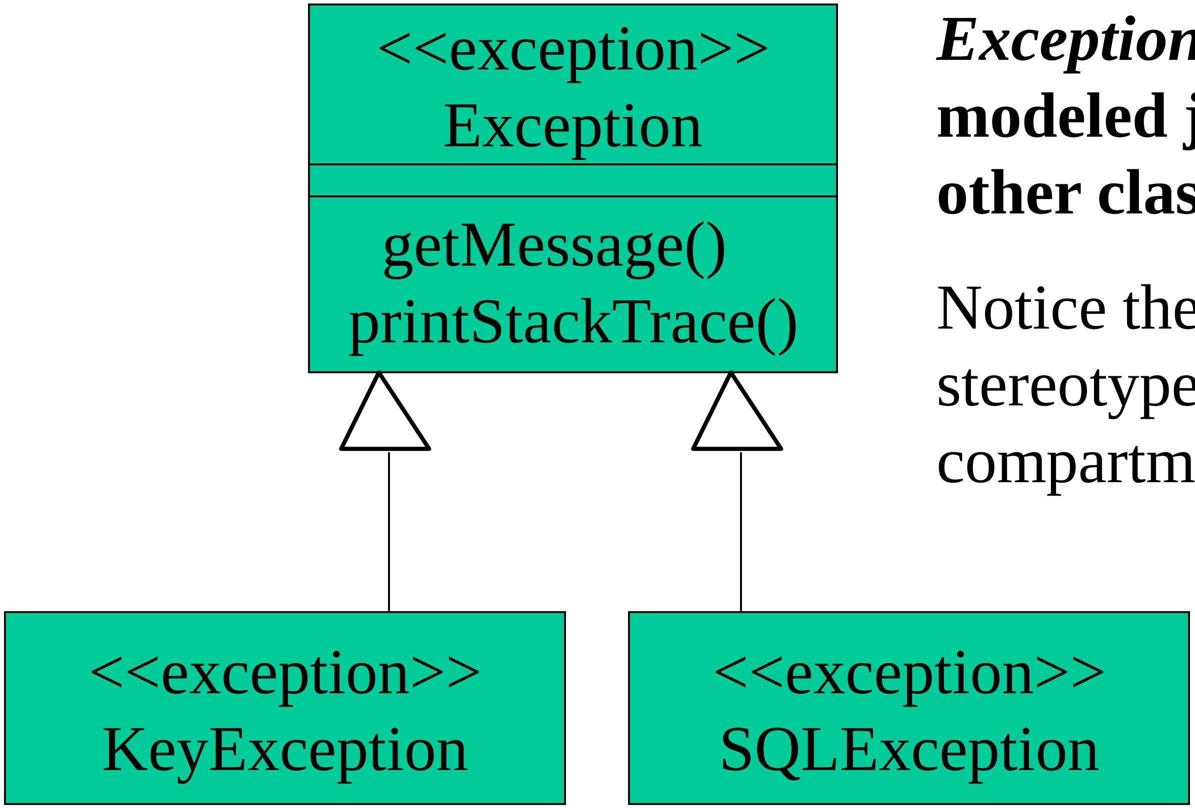
Aggregation Relationship

- Denoted by placing a diamond nearest the class representing the aggregation

Multiplicity

- Documents how many instances of a class can be associated with one instance of another class

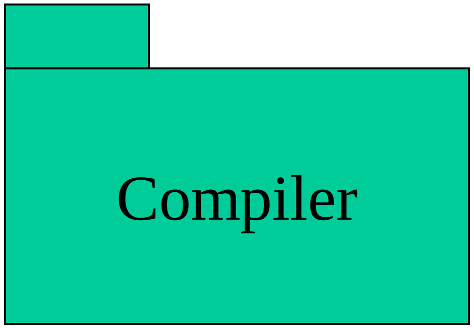
Exceptions



***Exceptions* can be modeled just like any other class.**

Notice the <<exception>> stereotype in the name compartment.

Packages



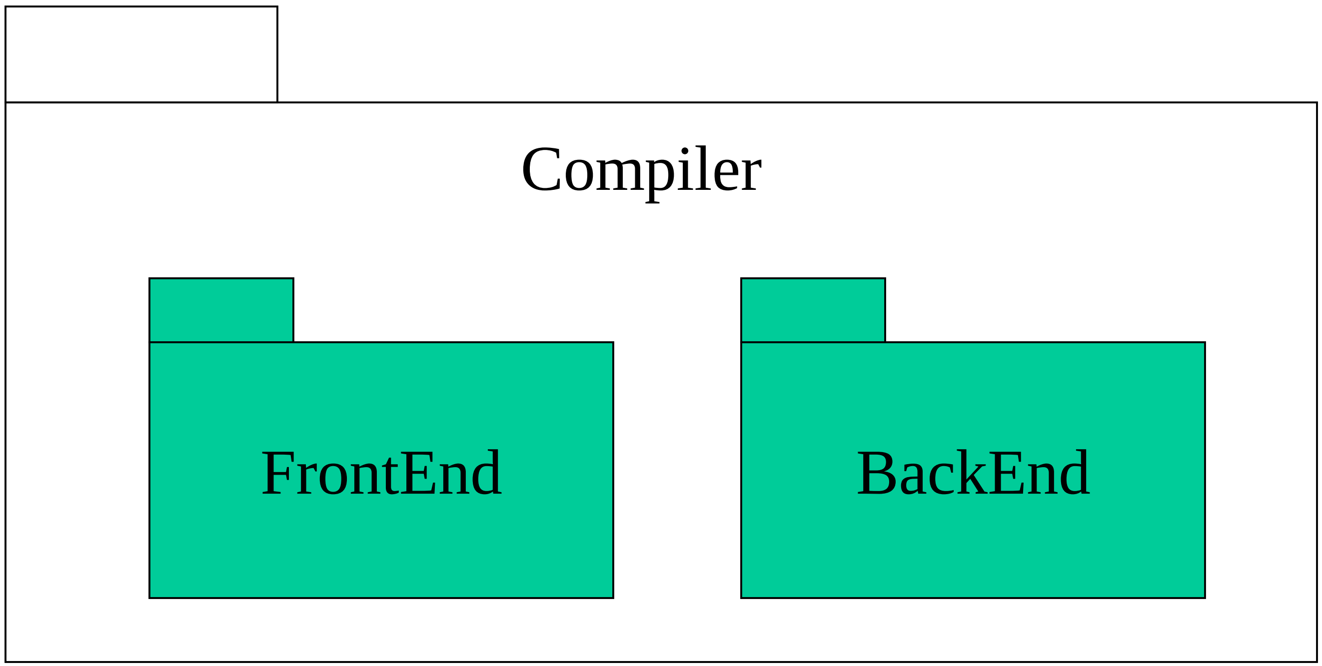
A *package* is a container-like element for organizing other elements into groups.

A package can contain classes and other packages and diagrams.

Packages can be used to provide controlled access between classes in different packages.

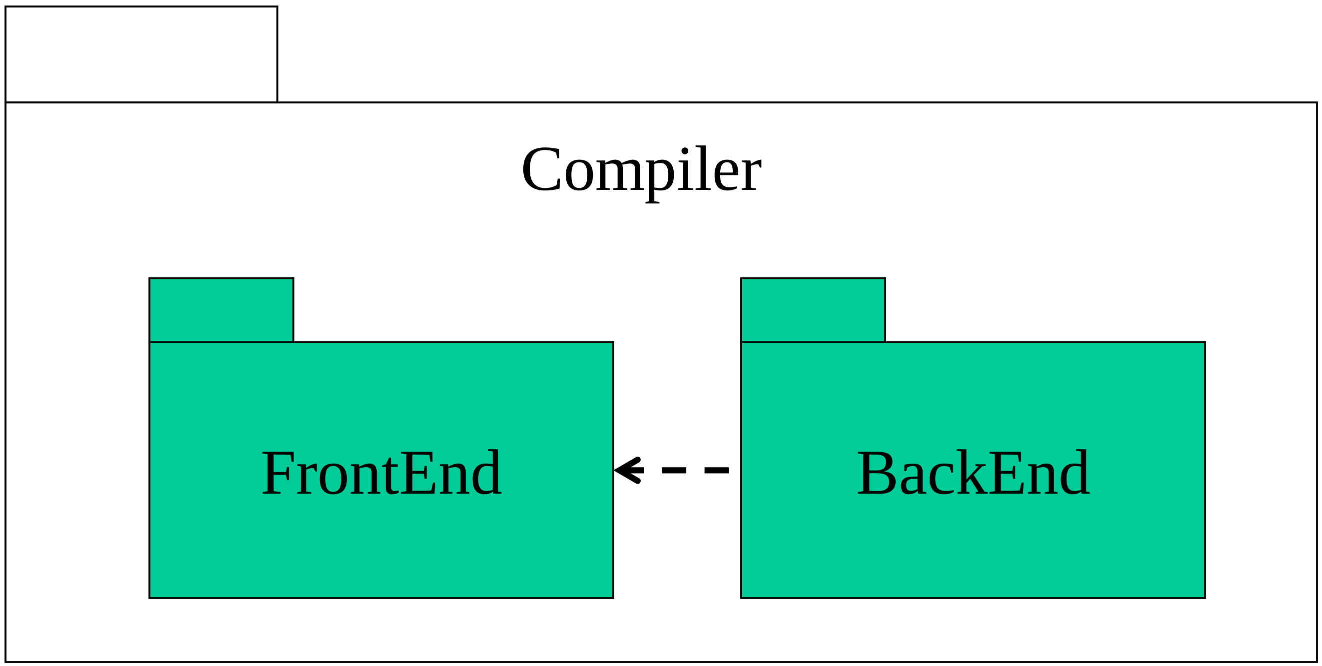
Packages (Cont'd)

Classes in the *FrontEnd* package and classes in the *BackEnd* package cannot access each other in this diagram.

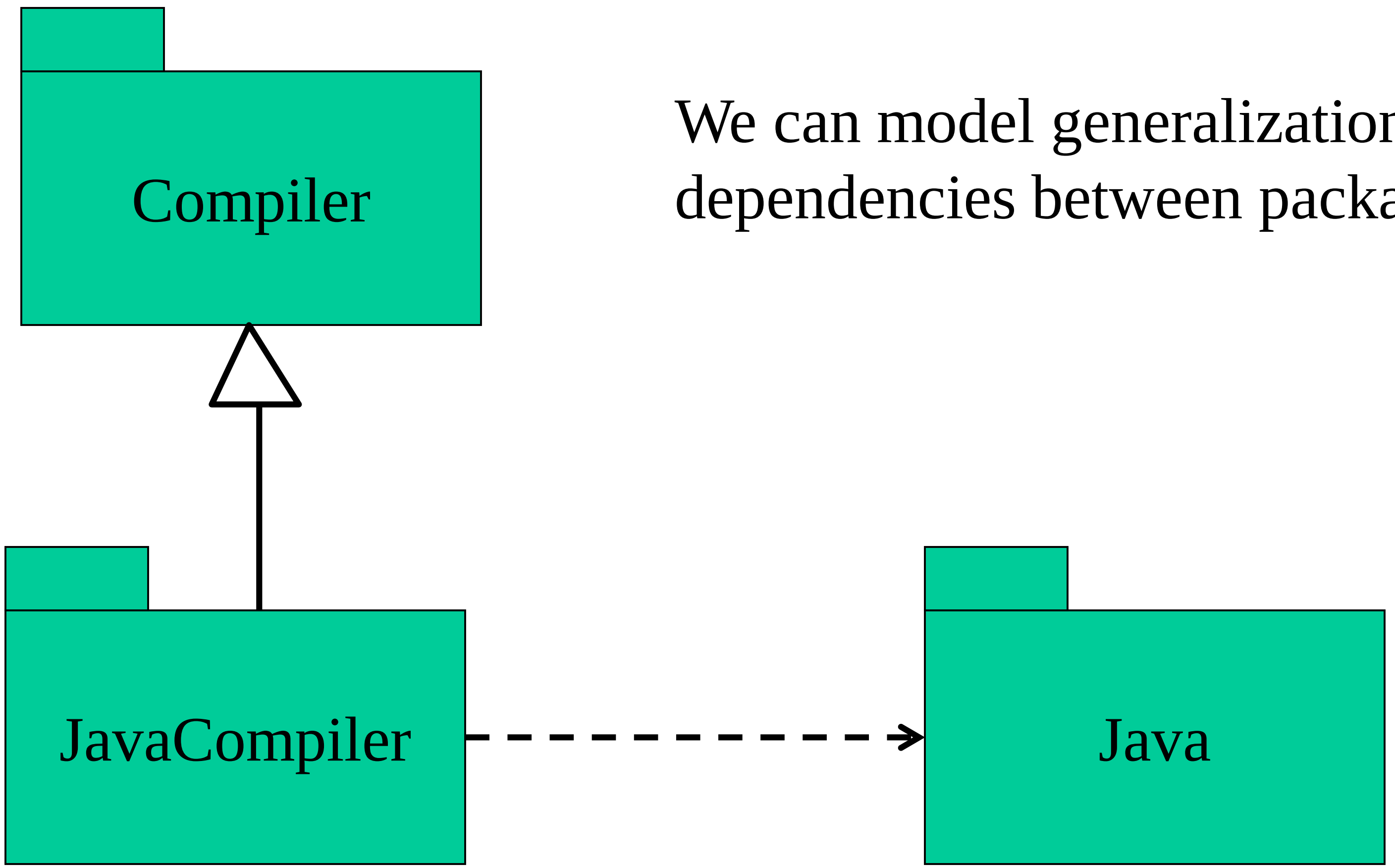


Packages (Cont'd)

Classes in the *BackEnd* package now have access to the classes in the *FrontEnd* package.



Packages (Cont'd)



We can model generalizations and dependencies between packages.

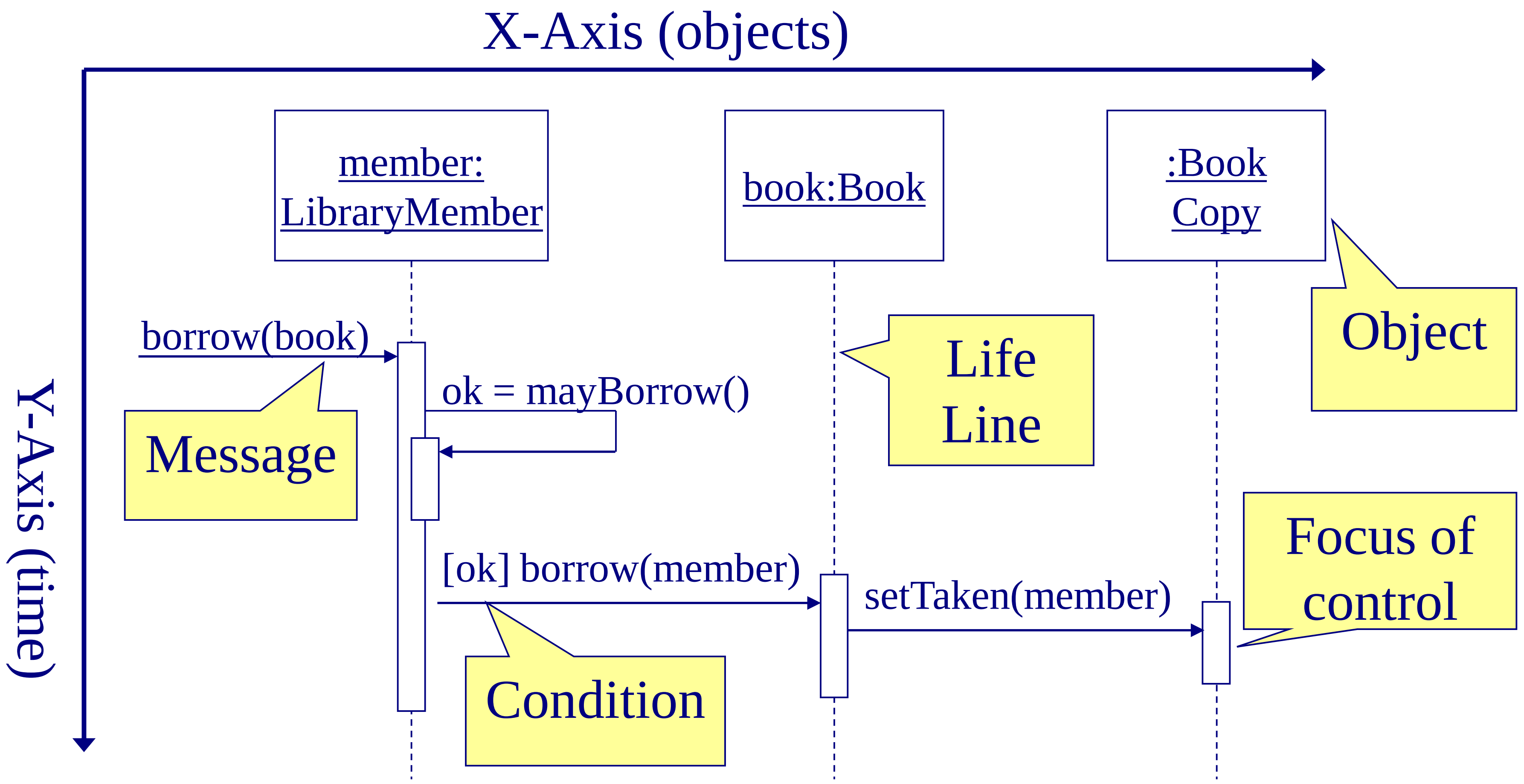
Sequence Diagram

Sequence Diagram

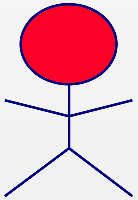
Illustrates the objects that participate in a use case and the messages that pass between them over time for *one* use case

In design, used to distribute use case behavior to classes

Sequence Diagram



Sequence Diagram Syntax

AN ACTOR	
AN OBJECT	**anObject:aClass**
A LIFELINE	
A FOCUS OF CONTROL	
A MESSAGE	aMessage()
OBJECT DESTRUCTION	x

Sequence Diagram

Two major components

- Active objects
- Communications between these active objects
  - Messages sent between the active objects

Sequence Diagram

Active objects

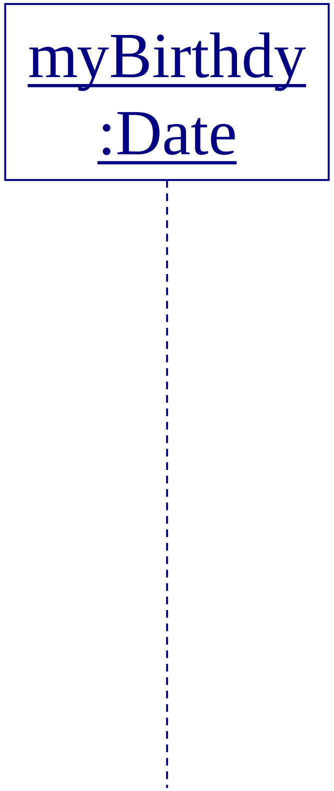
- Any objects that play a role in the system
- Participate by sending and/or receiving messages
- Placed across the top of the diagram
- Can be:
  - An actor (from the use case diagram)
  - Object/class (from the class diagram) within the system

Active Objects

Object

- Can be any object or class that is valid within the system
- Object naming
  - Syntax

    [instanceName][:className]



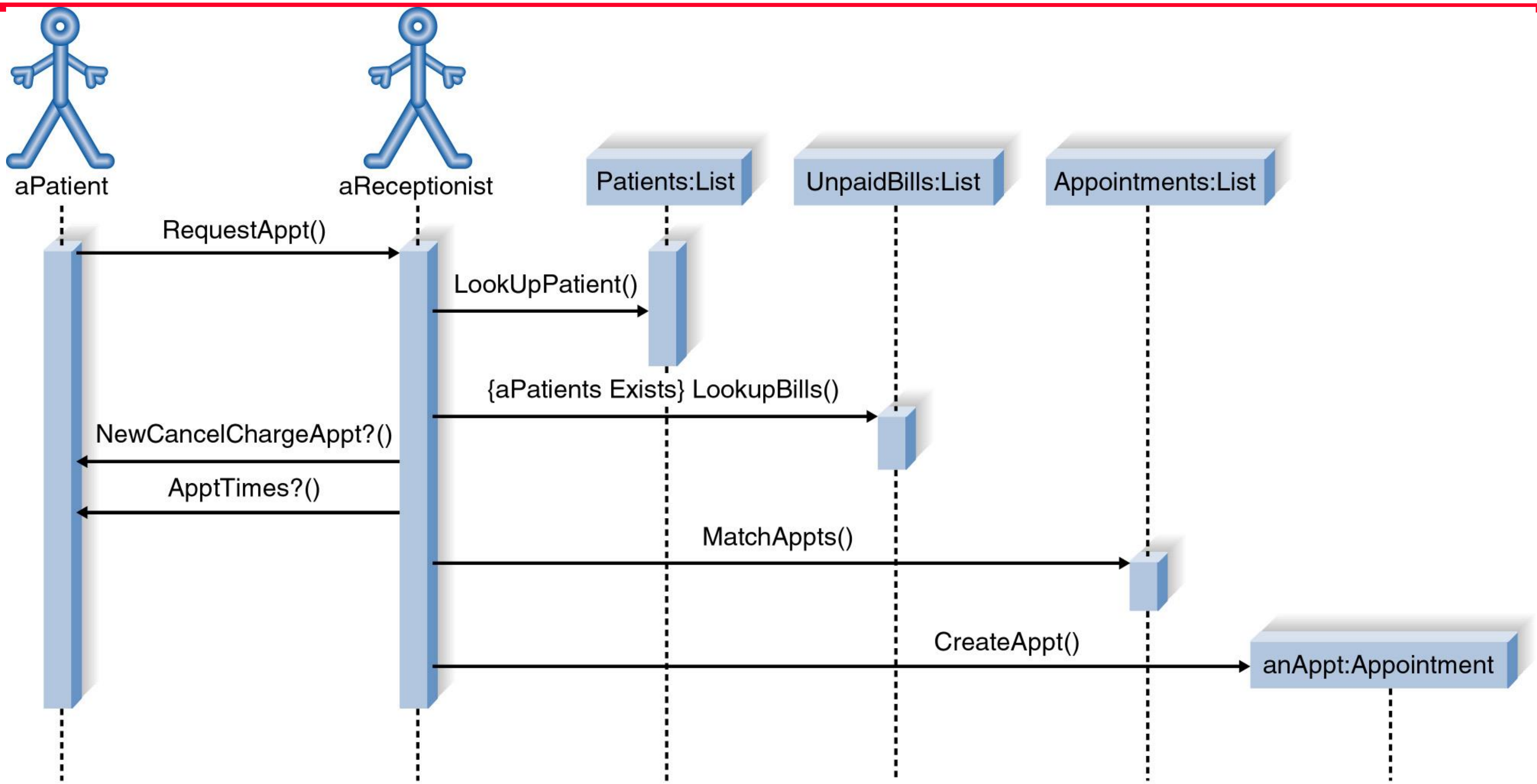
1. Class name only :Classname
2. Instance name only objectName
3. Instance name and class name together object:Class

Active Objects

Actor

- A person or system that derives benefit from and is external to the system
- Participates in a sequence by sending and/or receiving messages

Sequence Diagram



Communications between Active Objects

Messages

- Used to illustrate communication between different active objects of a sequence diagram
- Used when an object needs
  - to activate a process of a different object
  - to give information to another object

Messages

A message is represented by an arrow between the life lines of two objects.

- Self calls are allowed

A message is labeled at minimum with the message name.

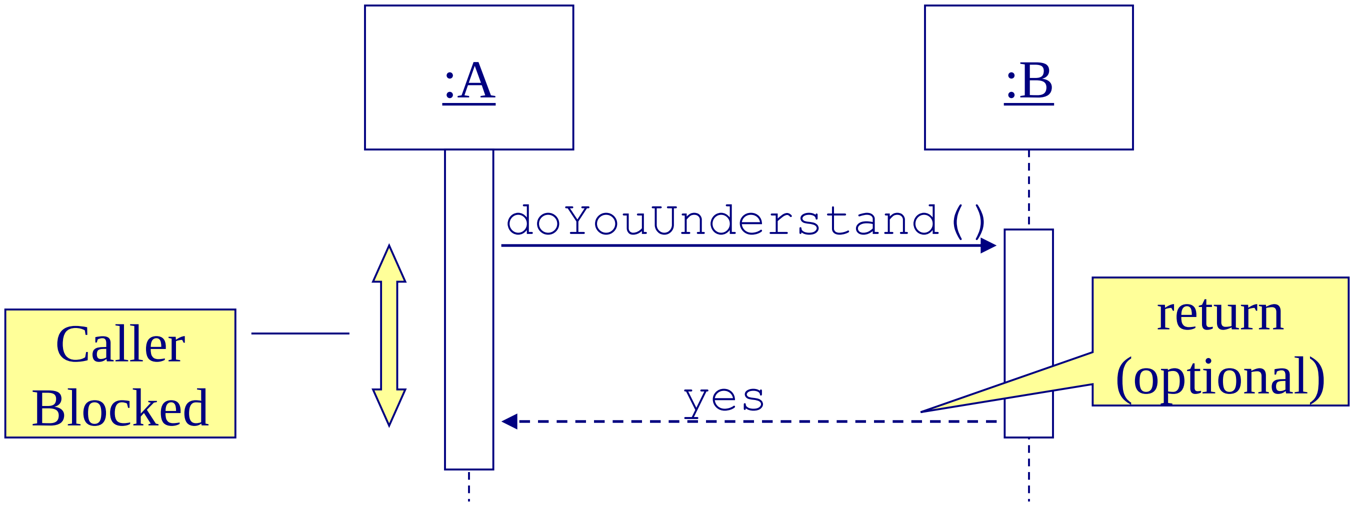
- Arguments and control information (conditions, iteration) may be included.

Types of Messages

- Synchronous (flow interrupt until the message has completed)
- Asynchronous (don't wait for response)
- Return (control flow has returned to the caller)

Synchronous Messages

The routine that handles the message is completed before the calling routine resumes execution.



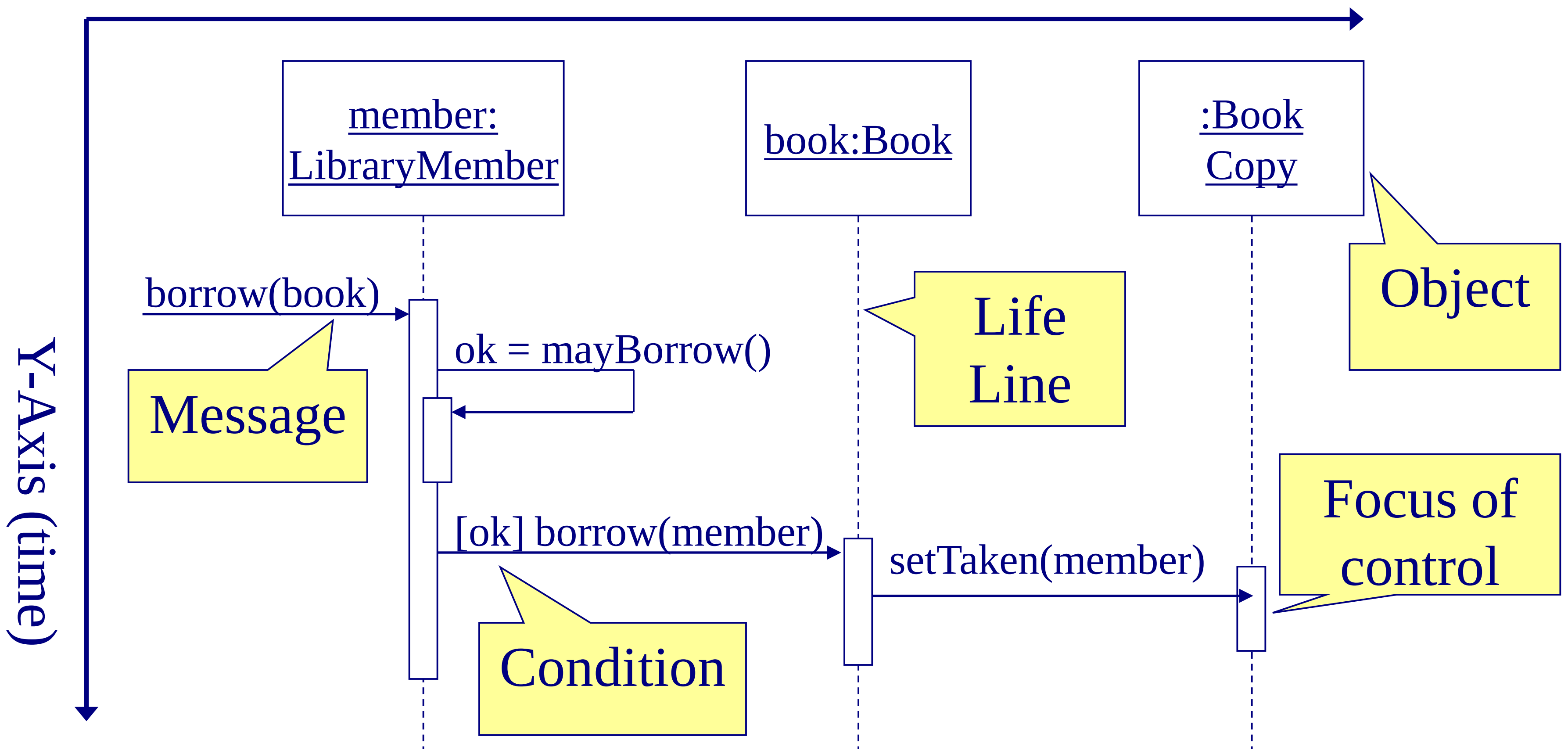
Asynchronous Messages

- Calling routine does not wait for the message to be handled before it continues to execute.
  - As if the call returns immediately
- Examples
  - Notification of somebody or something
  - Messages that post progress information

Return Values

- Optionally indicated using a dashed arrow with a label indicating the return value.
  - Don’t model a return value when it is obvious what is being returned, e.g. getTotal()
  - Model a return value only when you need to refer to it elsewhere (e.g. as a parameter passed in another message)
  -

Sequence Diagram



Other Elements of Sequence Diagram

- Lifeline
- Focus of control (activation box or execution occurrence)
- Control information
  - Condition, repetition

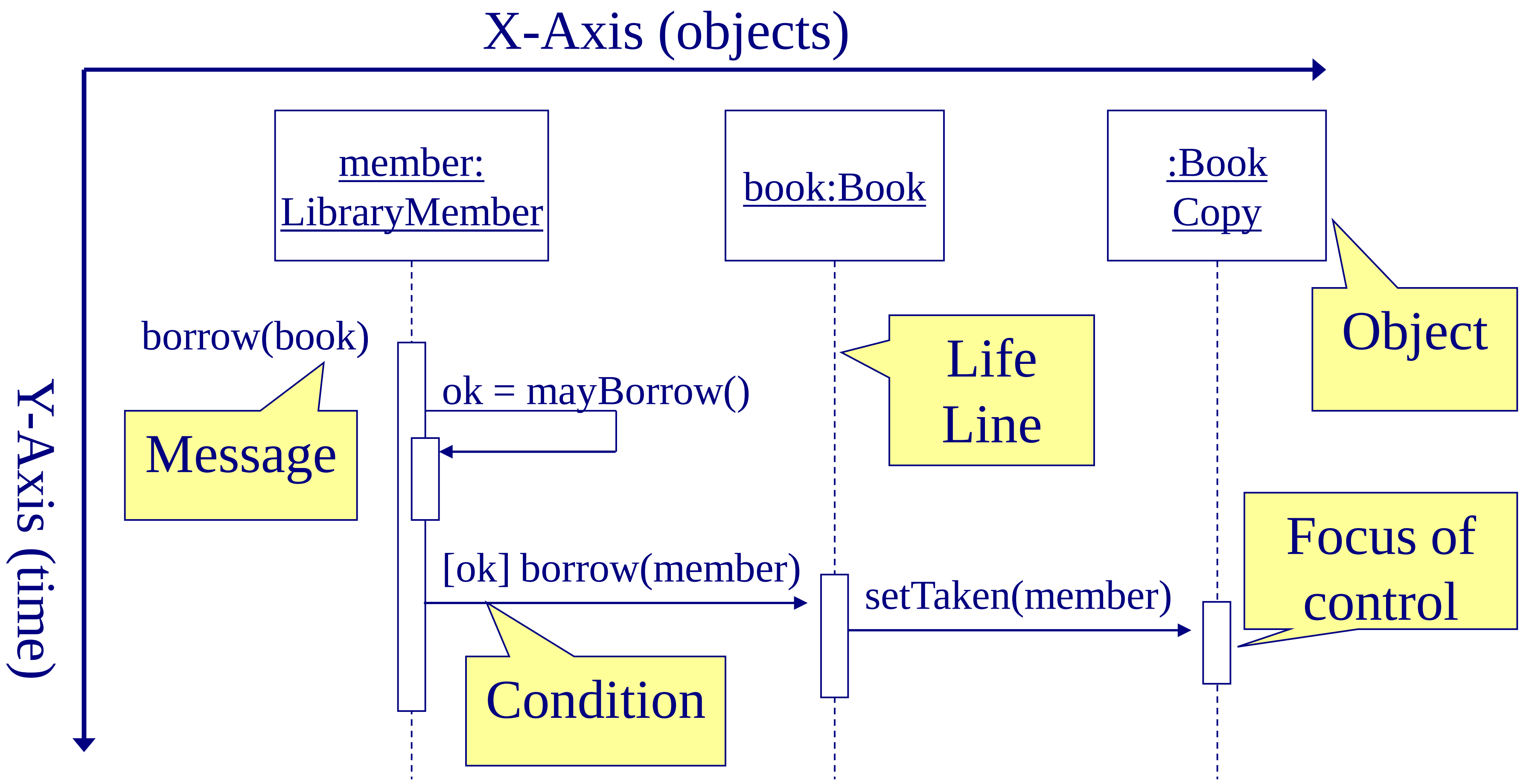
Sequence Diagram

- Lifeline
  - Denotes the life of actors/objects over time during a sequence
  - Represented by a vertical line below each actor and object (normally dashed line)
- For temporary object
  - place X at the end of the lifeline at the point where the object is destroyed

Sequence Diagram

- Focus of control (activation box)
  - Means the object is active and using resources during that time period
  - Denotes when an object is sending or receiving messages
  - Represented by a thin, long rectangular box overlaid onto a lifeline

Sequence Diagram

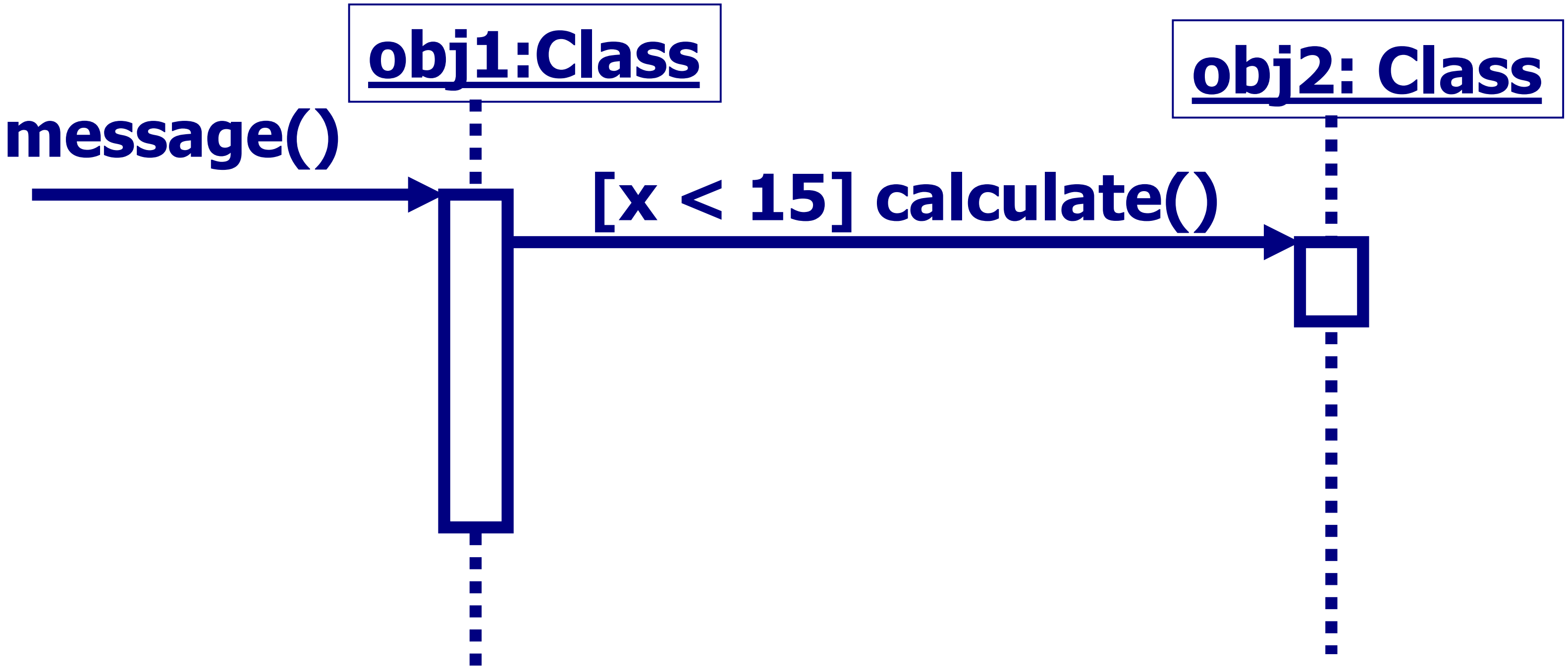


Control Information

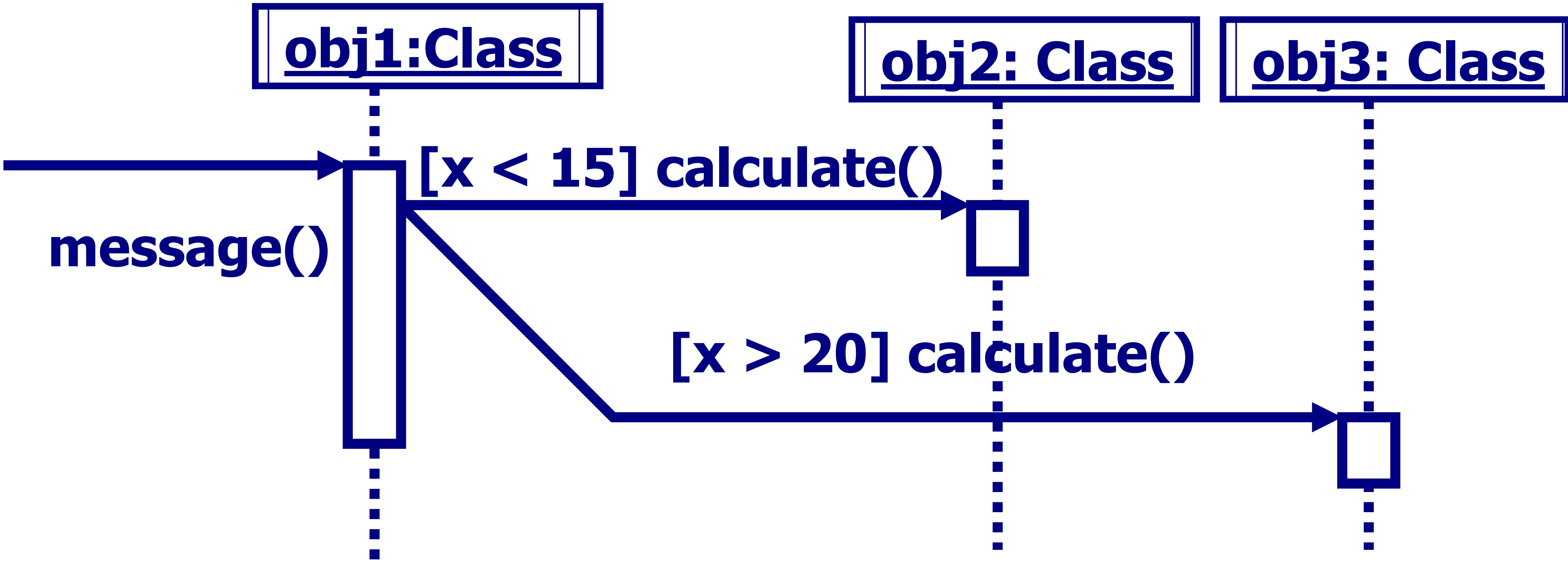
- Condition
  - syntax: ‘[‘ expression ’]’ message-label
  - The message is sent only if the condition is true

[ok] borrow(member)

Elements of Sequence Diagram

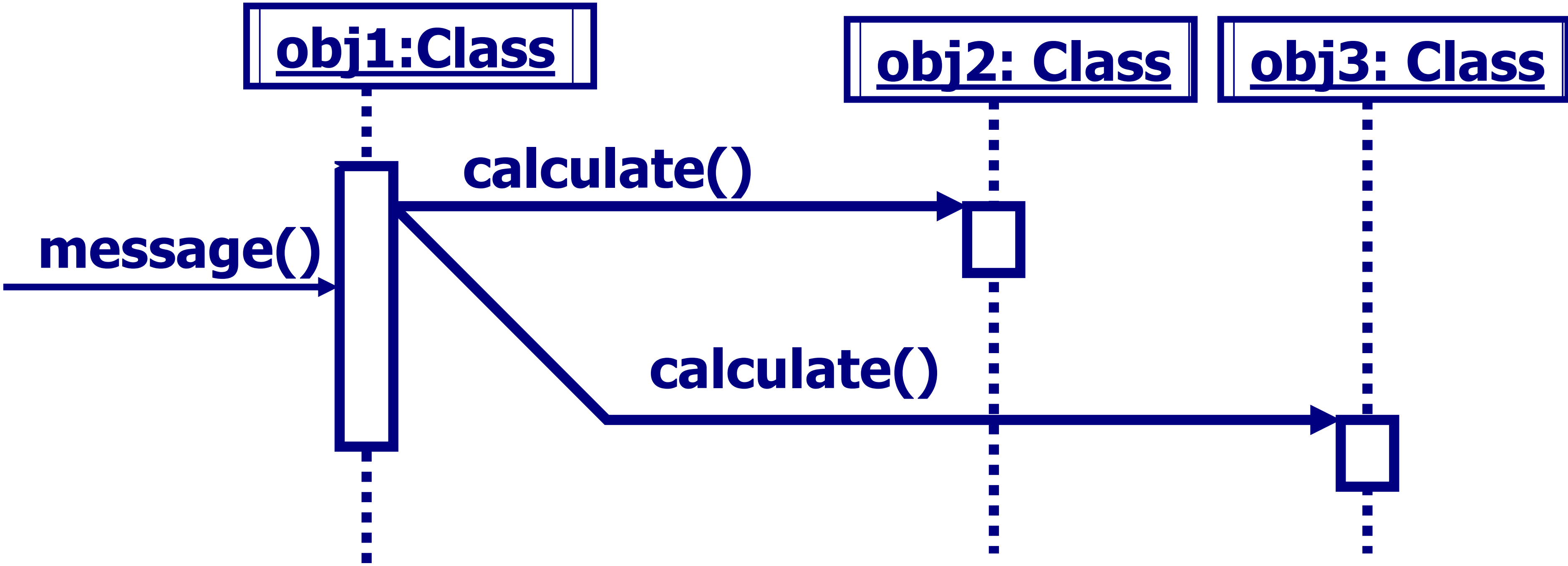


Sequence Diagrams



Sequence Diagrams

Concurrency



Elements of Sequence Diagram

- Control information
  - Iteration
    - may have square brackets containing a continuation condition (until) specifying the condition that must be satisfied in order to exit the iteration and continue with the sequence
    - may have an asterisk followed by square brackets containing an iteration (while or for) expression specifying the number of iterations

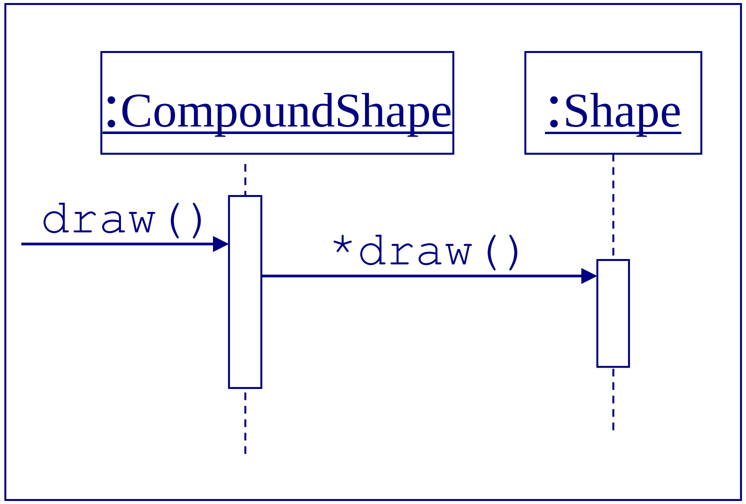
Control Information

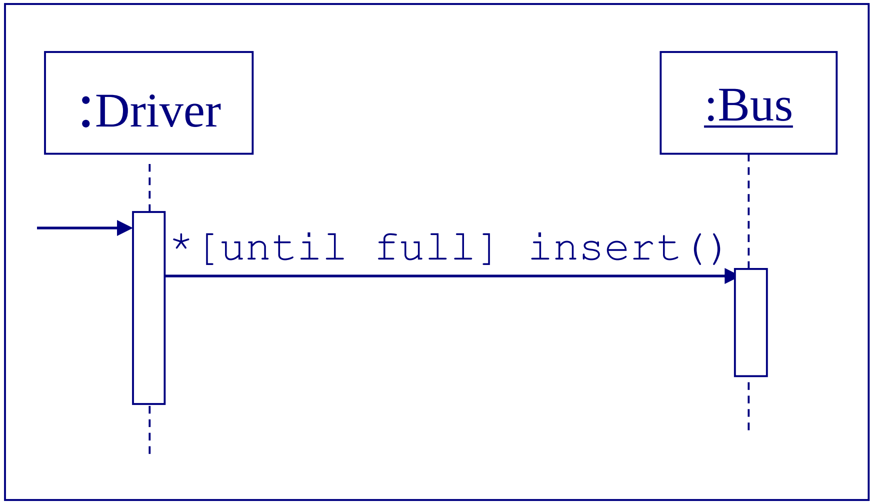
- Iteration
  - syntax: * [‘[‘ expression ‘]’] message-label
  - The message is sent many times to possibly multiple receiver objects.

```
*draw()
```

Control Information

- Iteration example



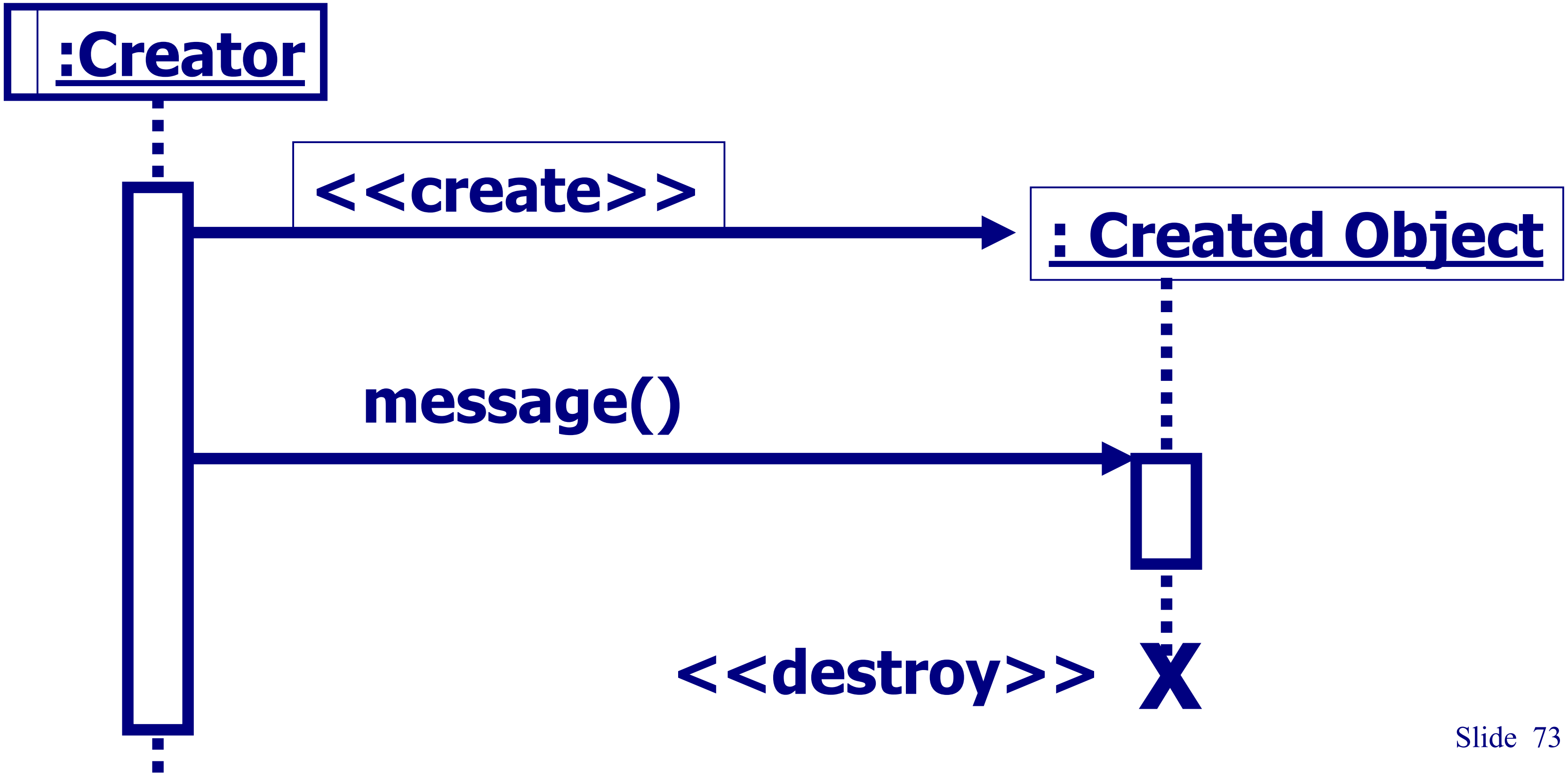


Control Information

- The control mechanisms of sequence diagrams suffice only for modeling simple alternatives.
  - Consider drawing several diagrams for modeling complex scenarios.
  - Don’t use sequence diagrams for detailed modeling of algorithms (this is better done using *activity diagrams, pseudo-code* or *state-charts*).

Sequence Diagrams

- Creation and destruction of an object in sequence diagrams are denoted by the stereotypes <<create>> and <<destroy>>

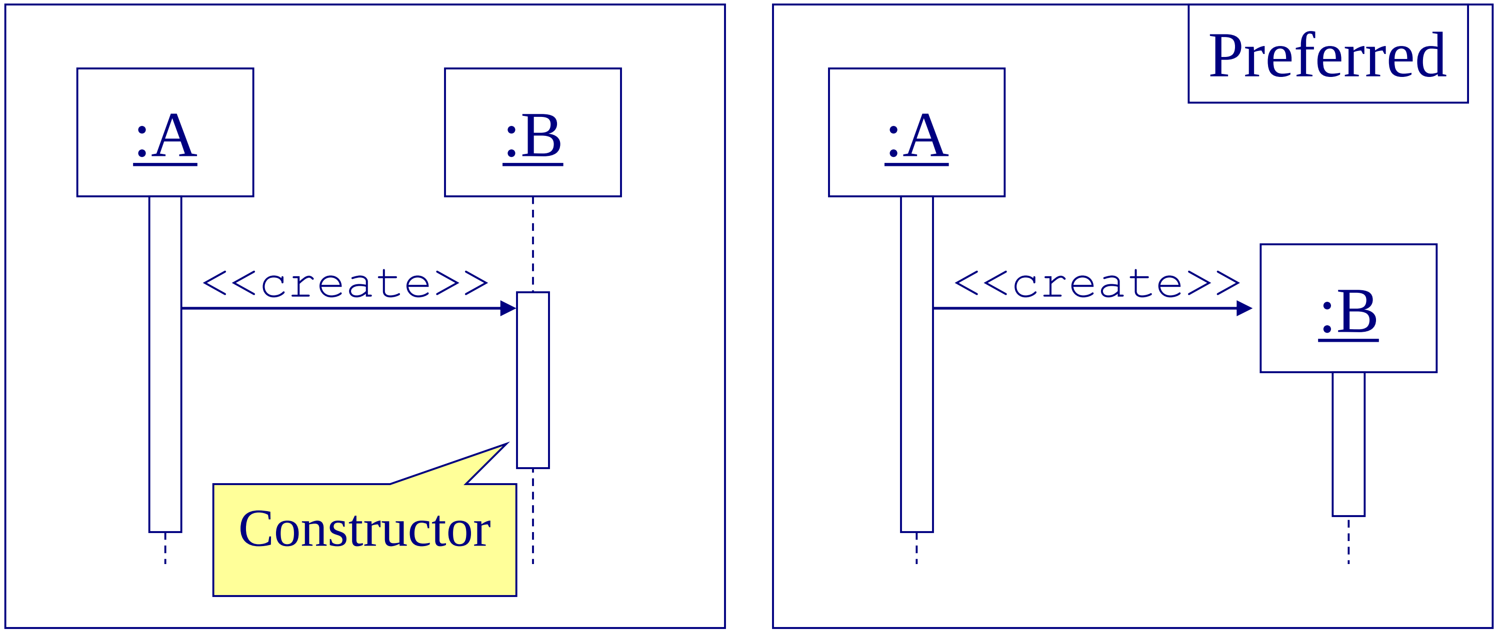


Creating Objects

- Notation for creating an object on-the-fly
  - Send the <<create>> message to the body of the object instance
  - Once the object is created, it is given a lifeline.
    - Now you can send and receive messages with this object as you can any other object in the sequence diagram.

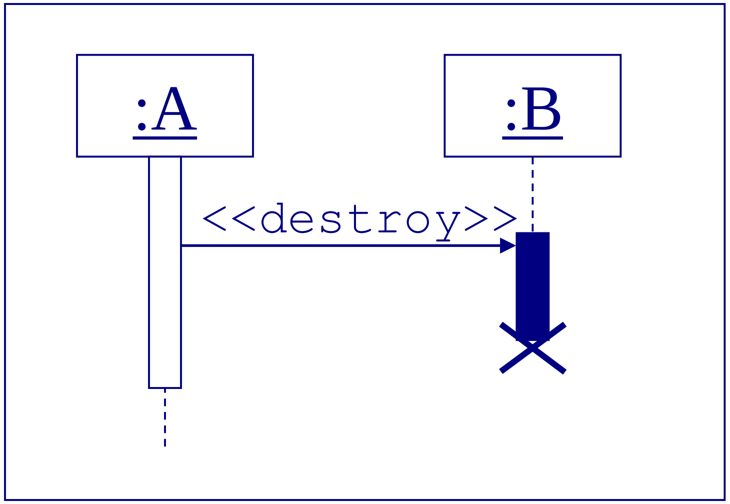
Object Creation

- An object may create another object via a `<<create>>` message.

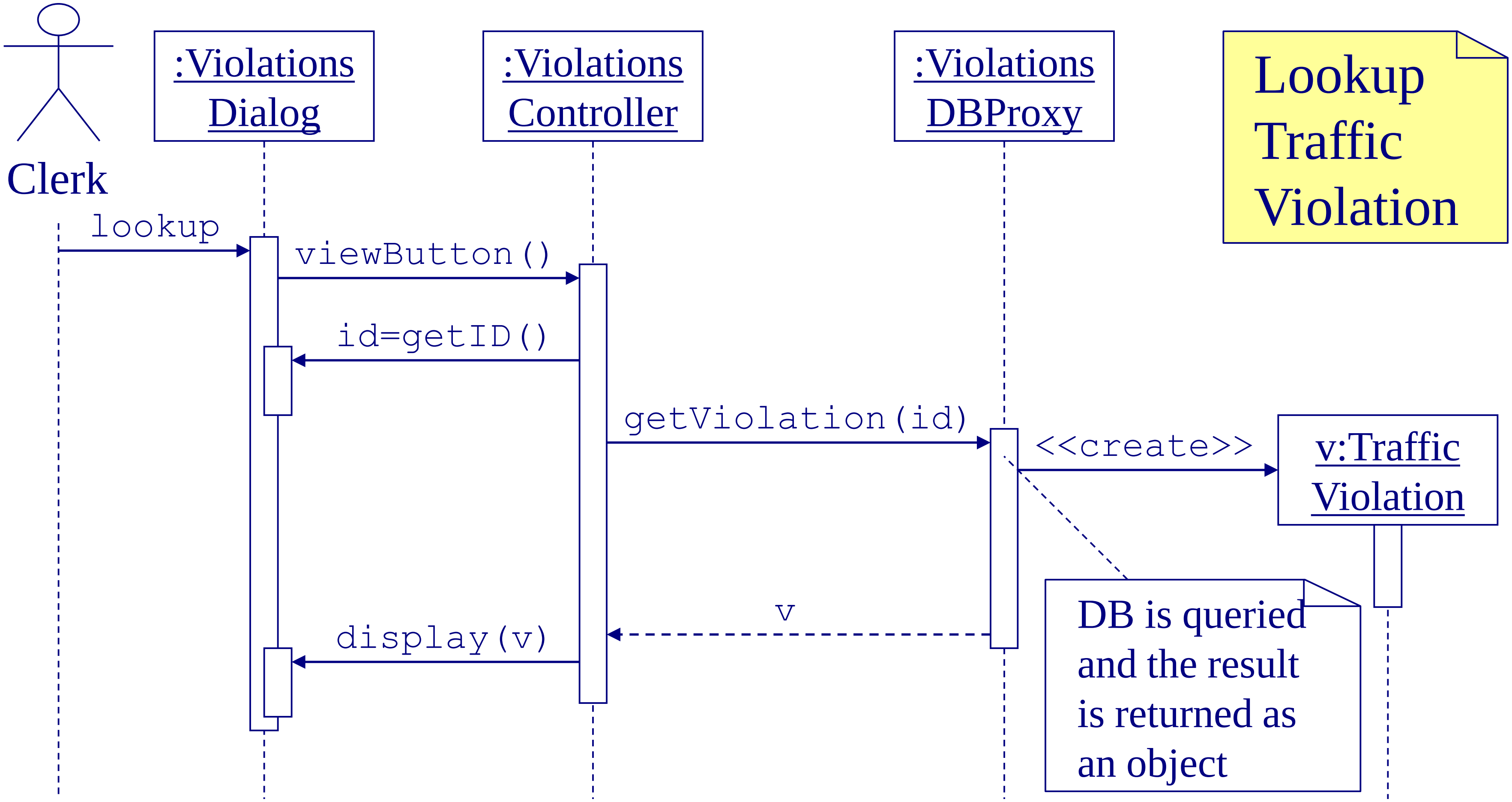


Object Destruction

- An object may destroy another object via a `<<destroy>>` message.
  - An object may destroy itself.
  - Avoid modeling object destruction unless memory management is critical.



Sequence Diagram



Steps for Building a Sequence Diagram

1) Set the context
2) Identify which objects and actors will participate
3) Set the lifeline for each object/actor
4) Lay out the messages from the top to the bottom of the diagram based on the order in which they are sent
5) Add the focus of control for each object's or actor's lifeline
6) Validate the sequence diagram

Steps for Building a Sequence Diagram

1) Set the context.
   a) Select a use case.
   b) Decide the initiating actor.

Steps for Building a Sequence Diagram

2) Identify the objects that may participate in the implementation of this use case by completing the supplied message table.
   a) List candidate objects.
      1) Use case controller class
      2) Domain classes
      3) Database table classes
      4) Display screens or reports

Steps for Building a Sequence Diagram

2) Identify the objects (cont.)
   b) List candidate messages. (in message analysis table)
      1) **Examine each step in the normal scenario of the use case description to determine the messages needed to implement that step.**
      2) **For each step:**
         1) Identify step number.
         2) Determine messages needed to complete this step.
         3) For each message, decide which class holds the data for this action or performs this action
      3) **Make sure that the messages within the table are in the same order as the normal scenario**

Steps for Building a Sequence Diagram

2) Identify the objects (cont.)
   c) Begin sequence diagram construction.
      1) Draw and label each of the identified actors and objects across the top of the sequence diagram.
      2) The typical order from left to right across the top is the actor, primary display screen class, primary use case controller class, domain classes (in order of access), and other display screen classes (in order of access)
3) Set the lifeline for each object/actor

Steps for Building a Sequence Diagram

4) Lay out the messages from the top to the bottom of the diagram based on the order in which they are sent.

   Working in sequential order of the message table, make a message arrow with the message name pointing to the owner class.

   Decide which object or actor initiates the message and complete the arrow to its lifeline.

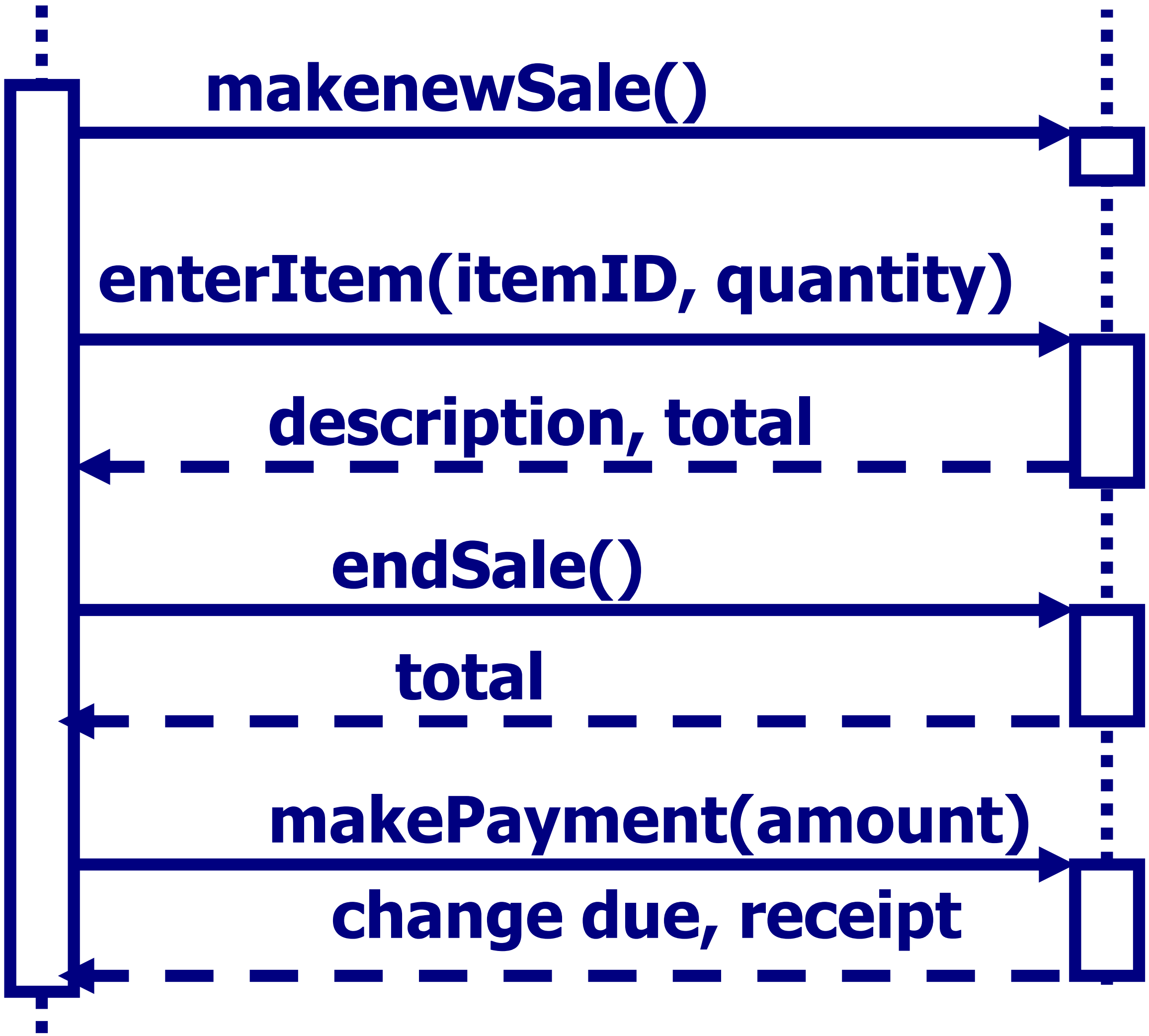
   Add needed return messages.

   Add needed parameters and control information.

Steps for Building a Sequence Diagram

5) Add the focus of control (activation box) for each object's or actor's lifeline.
6) Validate the sequence diagram.

Sequence Diagrams



Sequence Diagram