System Design & Analysis

LECTURE 10

Object Oriented Design

Object Oriented Design

- Uses object oriented decomposition
- Takes input from object oriented analysis
  - Conceptual model
  - Behavior from use cases, activity diagram etc
  - User interface
- High level abstractions from analysis and mapped onto classes
- Association is formed between classes and objects
- Uses notations to express different models of the system

Object Oriented Decomposition

- Locate classes in the problem domain
  - Look for nouns in use cases e.g ATM, Bank, Account , etc
- Find the operations
  - Appears as verbs in use case e.g Withdraw,CheckBalanace etc.
- Determine the responsible classes for the operations
- Requires a few iterations to locate the responsible classes
- Identify and model associations
- Build up the logical structure

Modeling Logical Structure

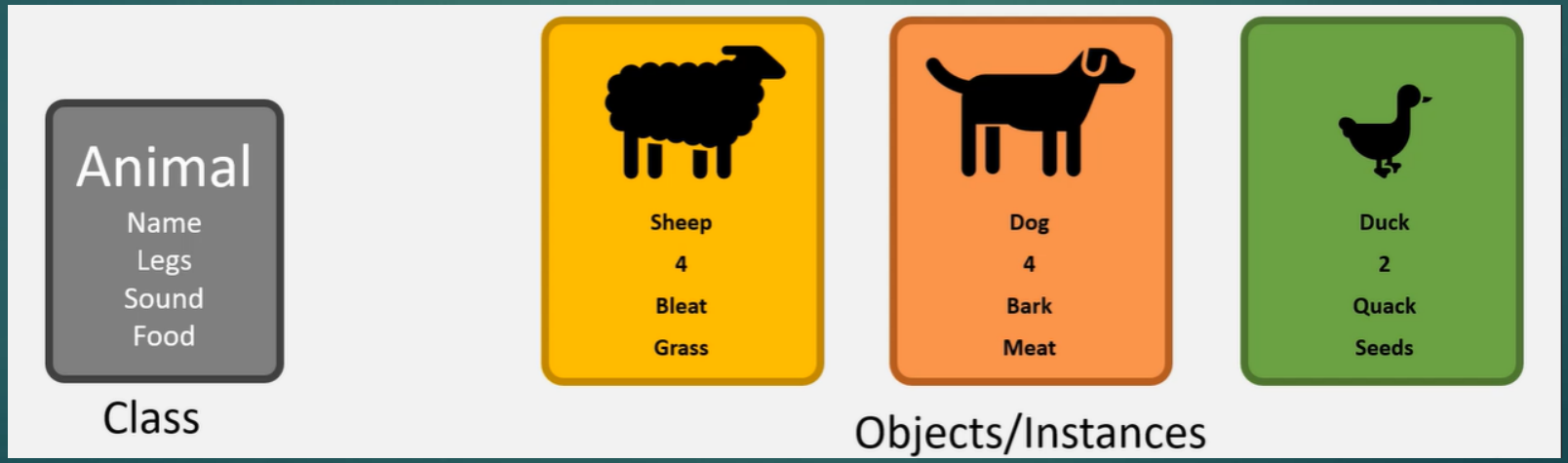
- Use cases describe the behavior of the system
- Activity diagram shows how to accomplish that behavior
- UML provides class diagram
  - Provides information about class and their relationship
  - From part of the model's logical view

Class Diagram

- Describe the structure of the system
- Represents a static view
- Structure is described at the level of classes along with their relationship
- The information is described without any particular implementation or object data
- Helps documents different aspects of a system
- Used to construct executable code for the application

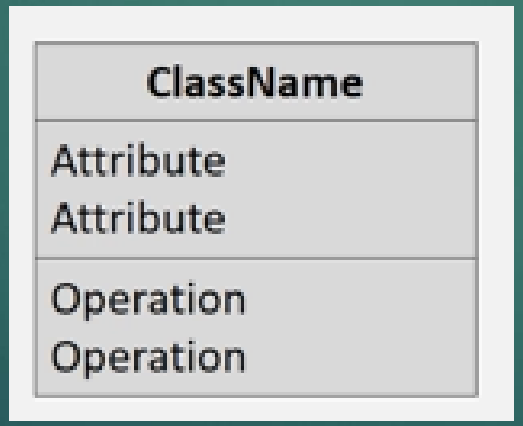
Class

- A class is a type or blue print of an entity
- Its instance is called as objects
- An object of a class will represents a specific type

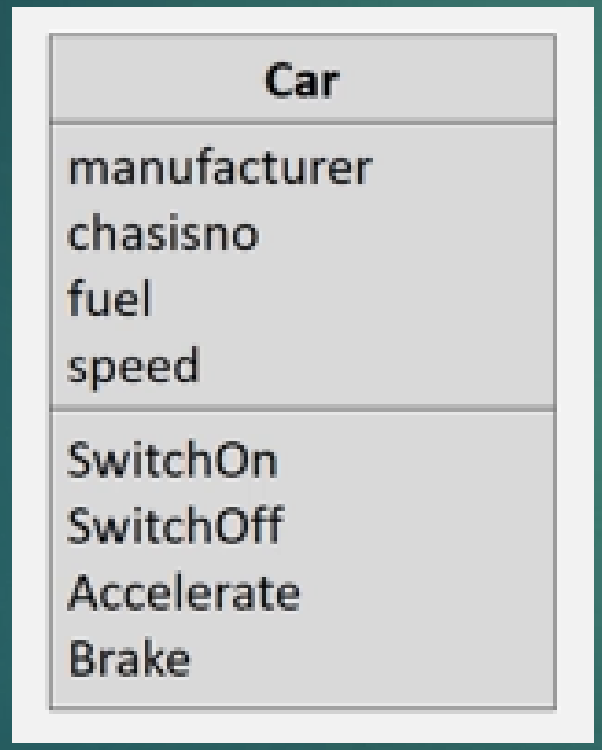


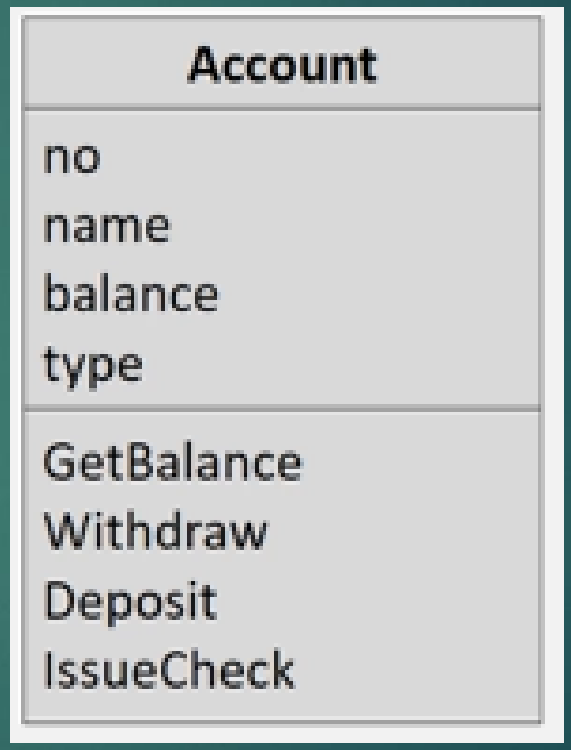
Classes in UML

- Shown as a rectangle split into three sections
  - Top contains the name
  - Middle contains attributes
  - Bottom contains operations (methods)

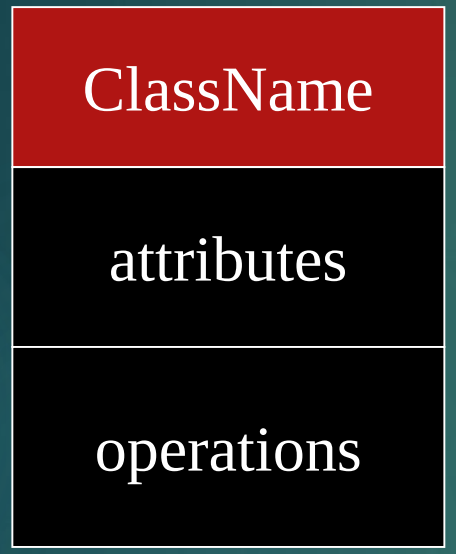


Example: Class



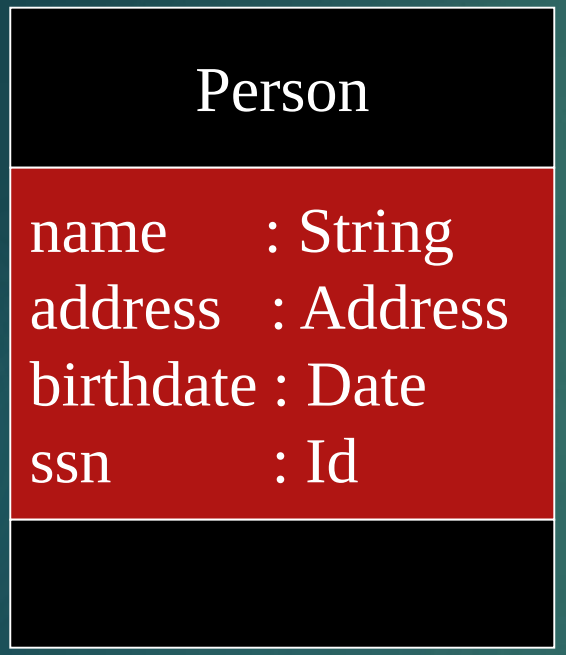


Class Names



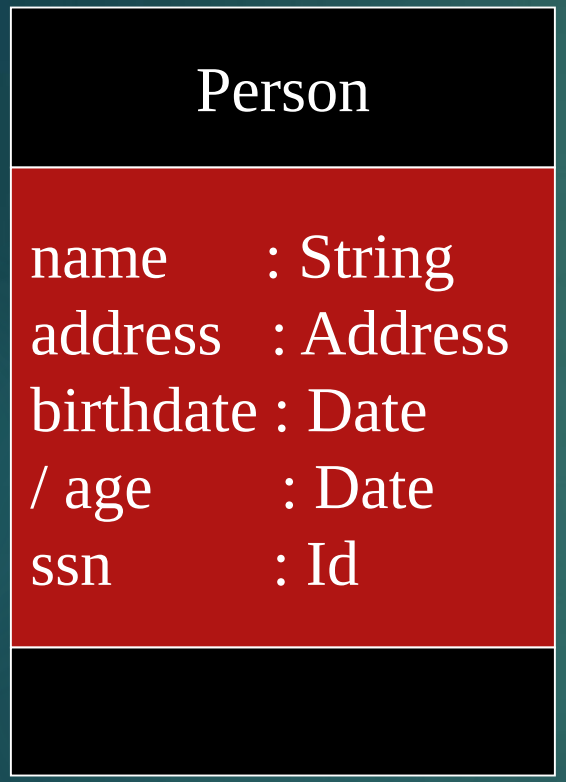
The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.

Class Attributes



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)



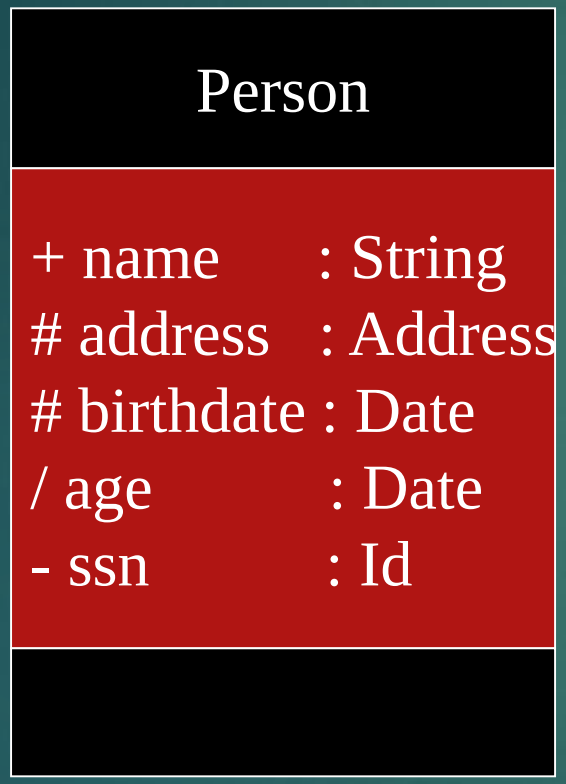
Attributes are usually listed in the form:

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

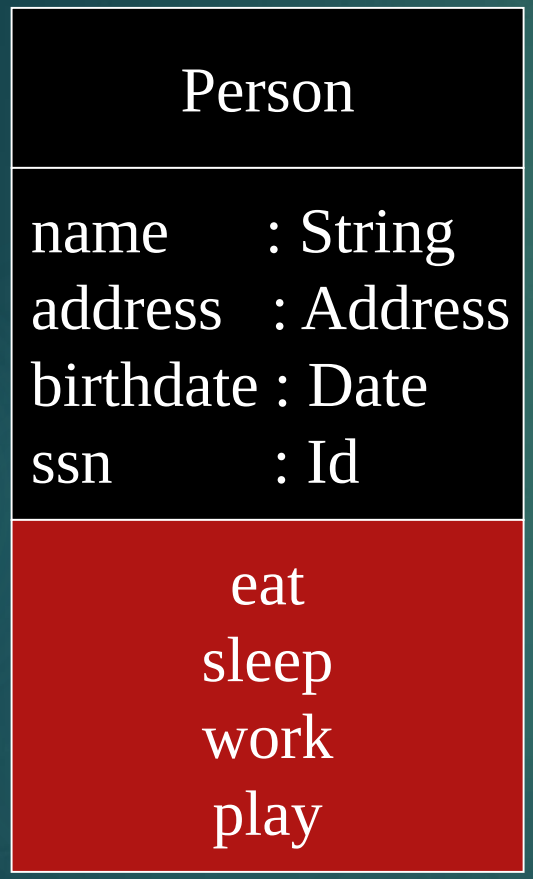
Class Attributes (Cont'd)



Attributes can be:

+ public
# protected
- private
/ derived

Class Operations



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

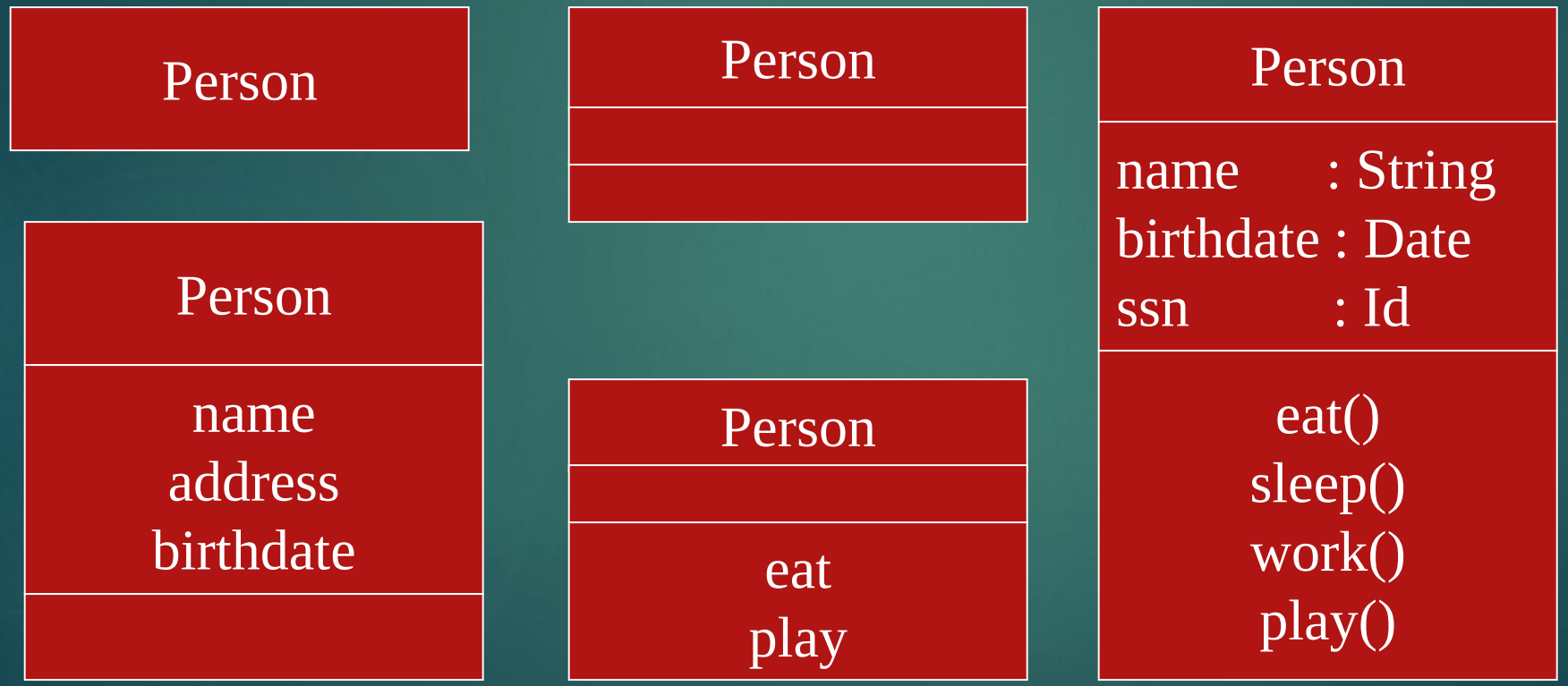
Class Operations (Cont'd)

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.

Depicting Classes

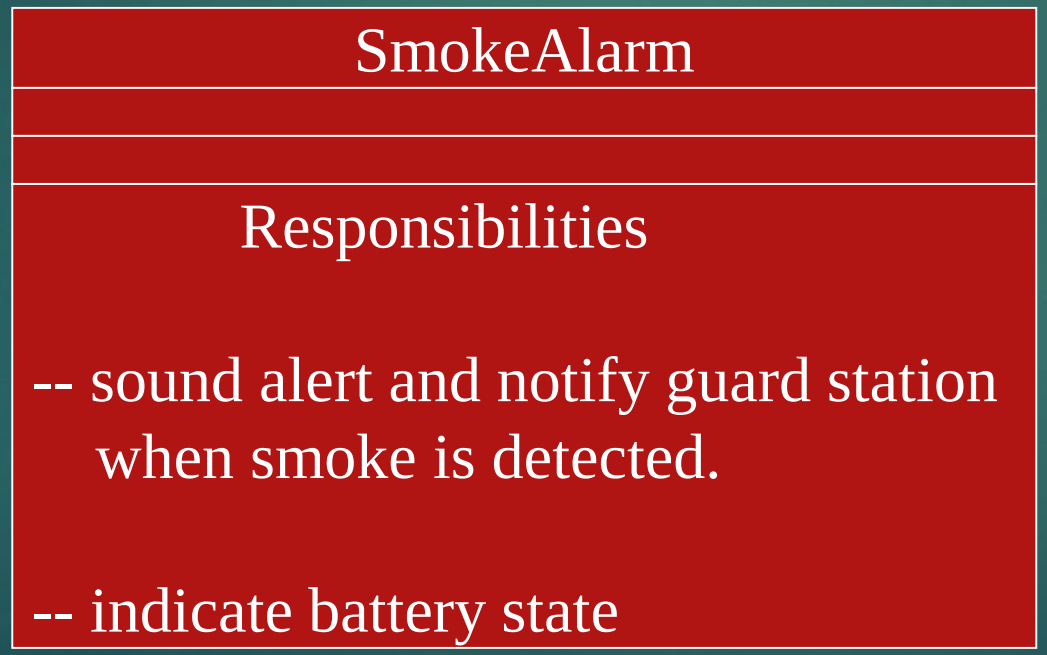
When drawing a class, you needn't show attributes and operation in every diagram.



Class Responsibilities

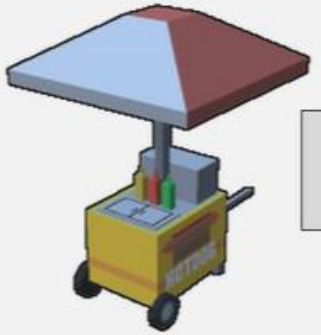
A class may also include its responsibilities in a class diagram.

A responsibility is a contract or obligation of a class to perform a particular service.



Objects in Reckless Driver

TrafficCar



HotdogCart

FireHydrant

Tree

NewspaperBox

PlayerCar

PostBox

Relationships

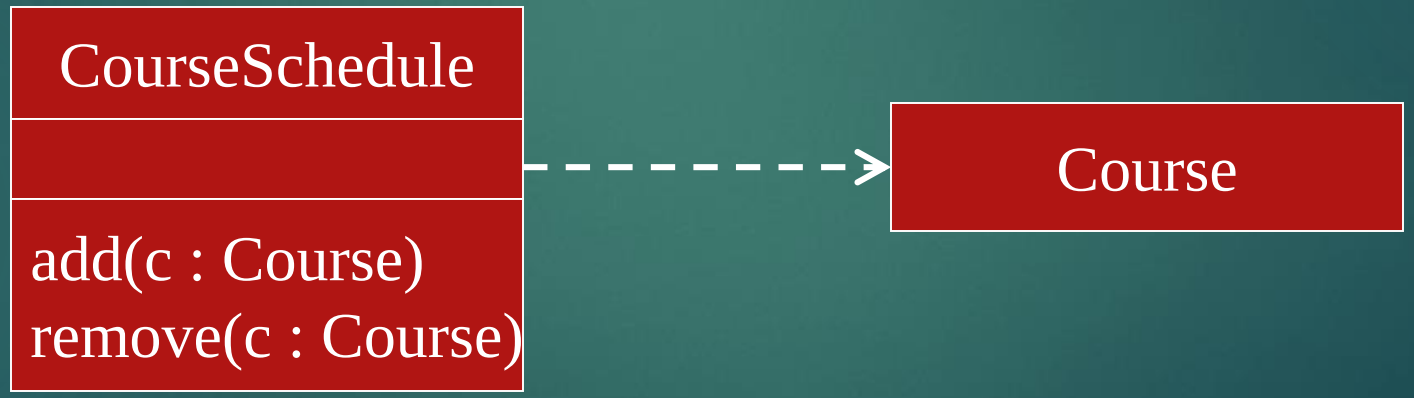
In UML, object interconnections (logical or physical), are modeled as relationships.

There are three kinds of relationships in UML:

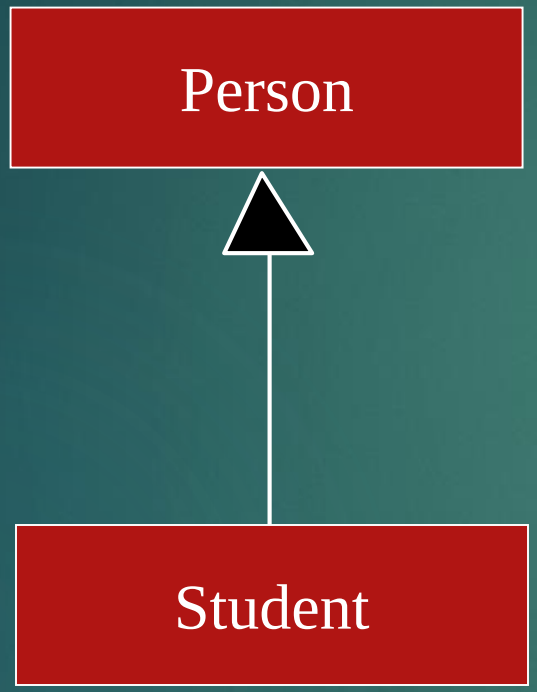
- dependencies
- generalizations
- associations

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*.



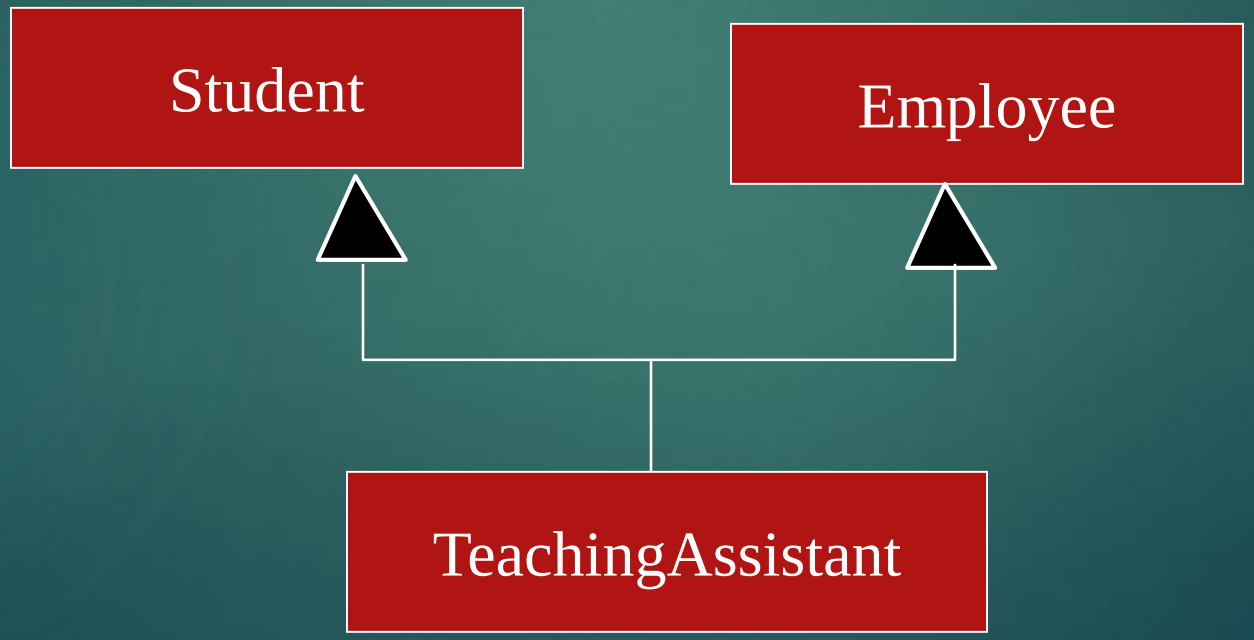
Generalization Relationships



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)

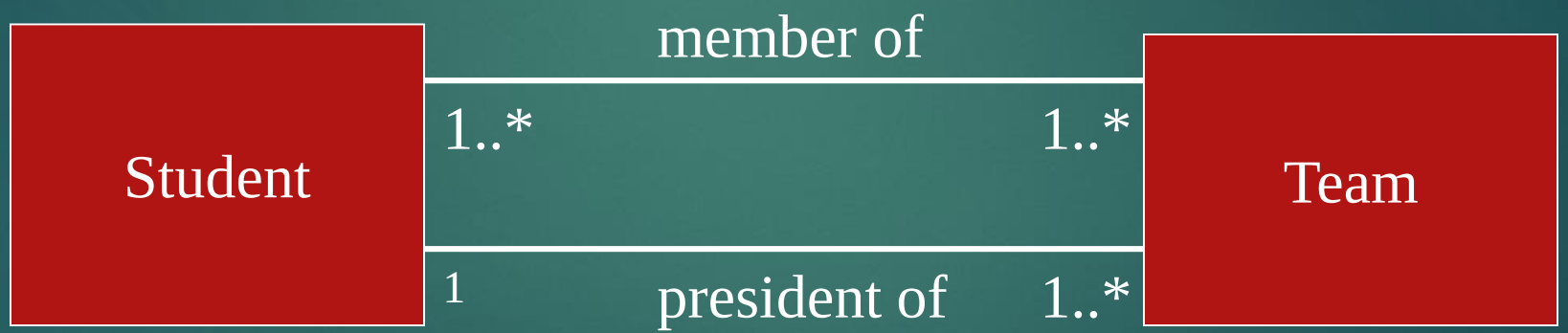
We can also indicate the behavior of an object in an association (*i.e.*, the *role* of an object) using *rolenames*.

Association Relationships (Cont'd)

We can also name the association.

Association Relationships (Cont'd)

We can specify dual associations.

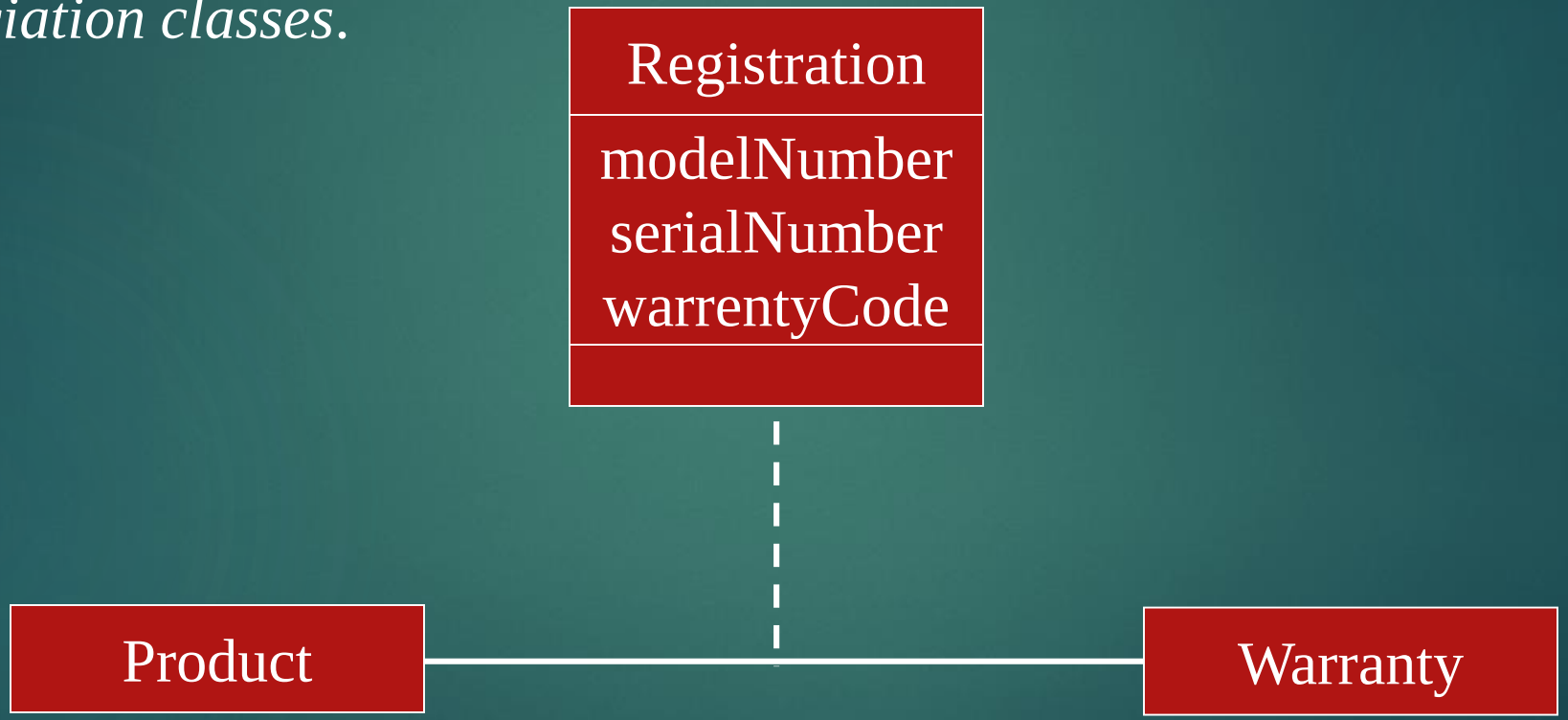


Association Relationships (Cont'd)

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*.

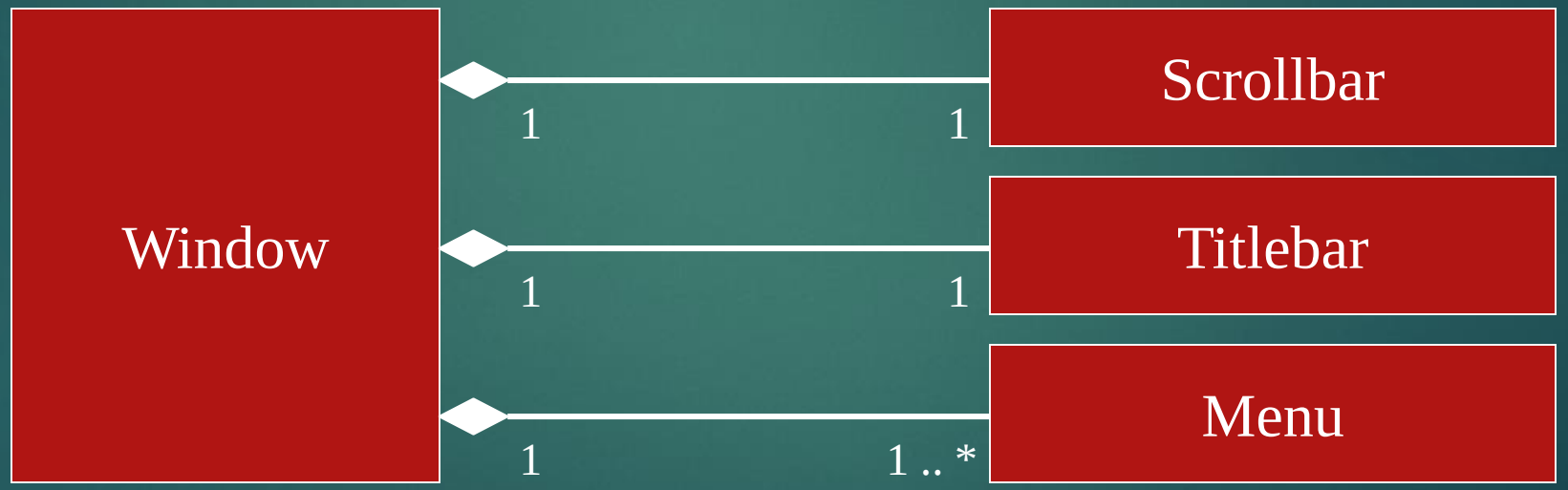
Association Relationships (Cont'd)

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)

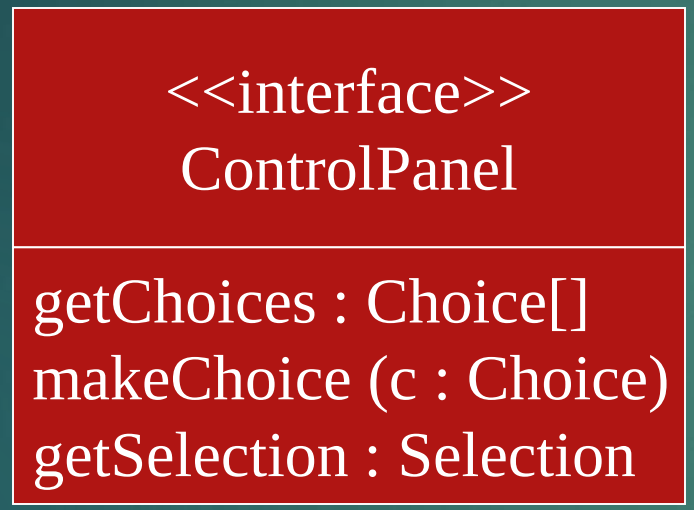
A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.*, they live and die as a whole). Compositions are denoted by a f**illed-diamond** adornment on the association.



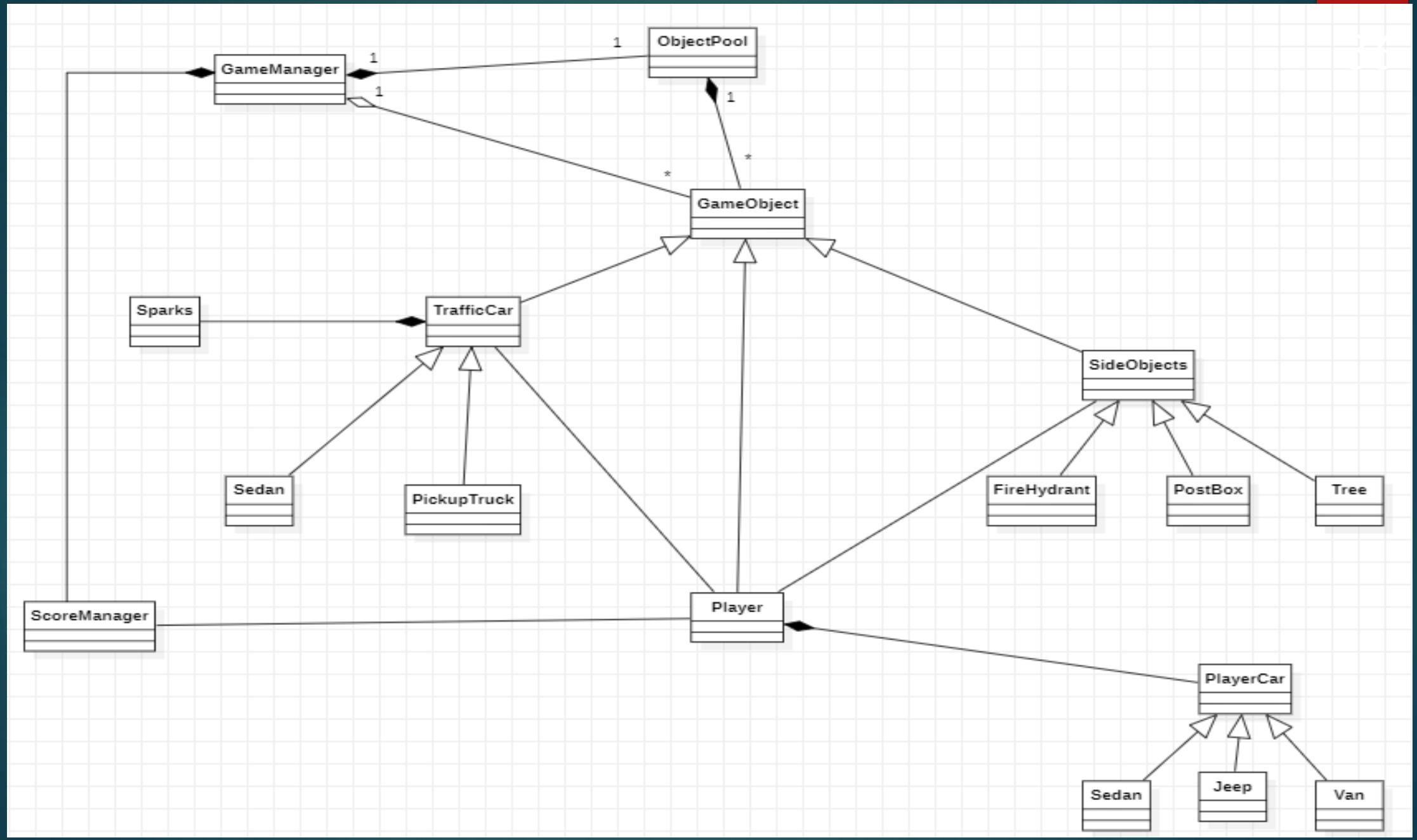
Interfaces

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 Services



Interfaces do not get instantiated. They have no attributes or state. Rather, they specify the services offered by a related class.


GameManager
ObjectPool
1
1
1
1
*
*
GameObject
Sparks
TrafficCar
SideObjects
Sedan
PickupTruck
FireHydrant
PostBox
Tree
ScoreManager
Player
PlayerCar
Sedan
Jeep
Van