Design Defects and Restructuring

Engr. Abdul-Rahman Mahmood

 abdulrahman@nu.edu.pk

 alphapeeler.sf.net/pubkeys/pkey.htm

 pk.linkedin.com/in/armahmood

 www.twitter.com/alphapeeler

 www.facebook.com/alphapeeler

 abdulmahmood-sss  alphasecure

 armahmood786

 http://alphapeeler.sf.net/me

 alphapeeler#9321

 *reddit.com/user/alphapeeler*

 www.flickr.com/alphapeeler

 http://alphapeeler.tumblr.com

 armahmood786@jabber.org

 alphapeeler@aim.com

 mahmood_cubix  48660186

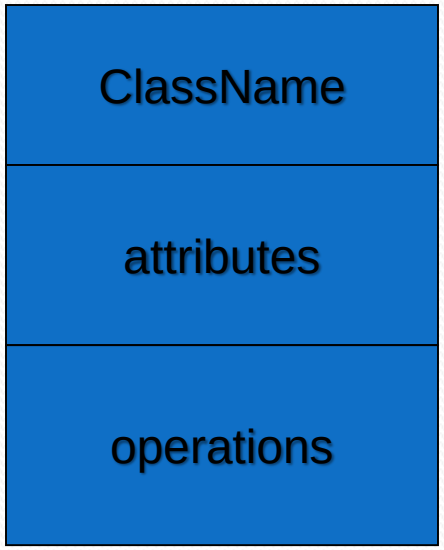
 alphapeeler@icloud.com

 pinterest.com/alphapeeler

 www.youtube.com/user/AlphaPeeler

Class Diagrams

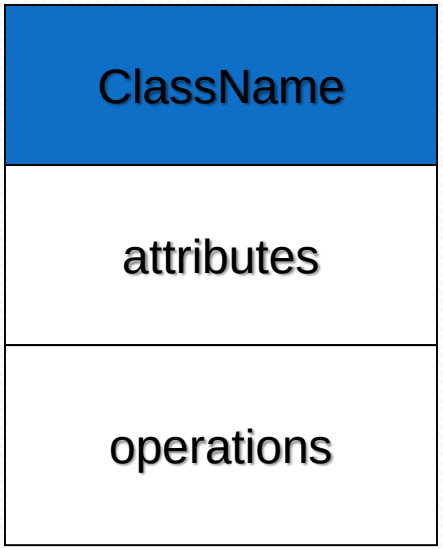
Classes



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

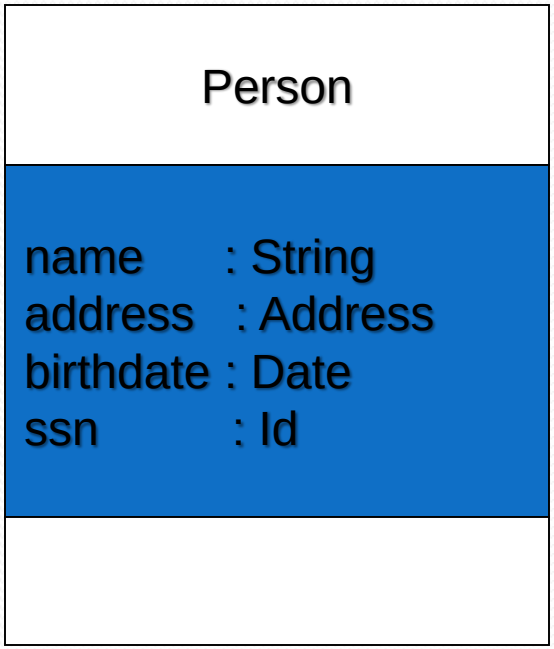
Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

Class Names



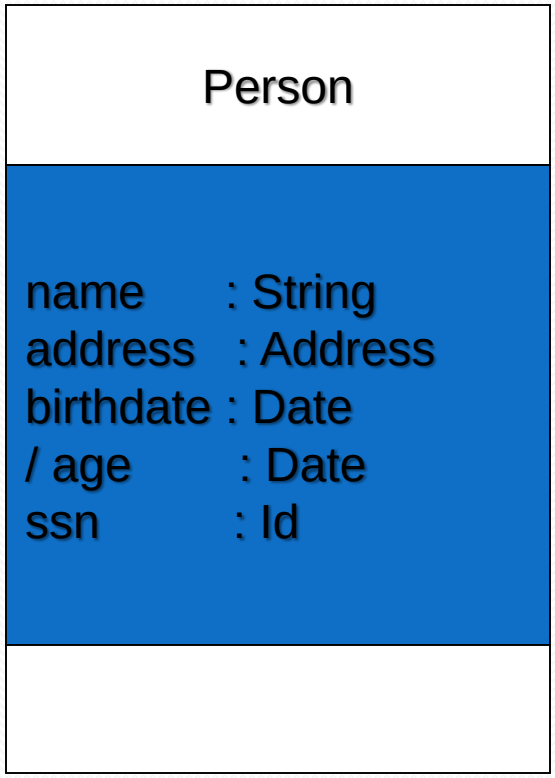
The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.

Class Attributes



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)



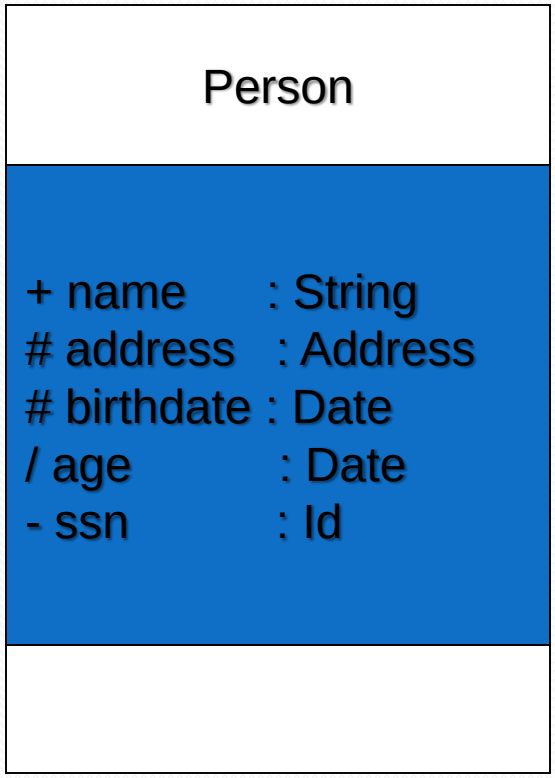
Attributes are usually listed in the form:

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

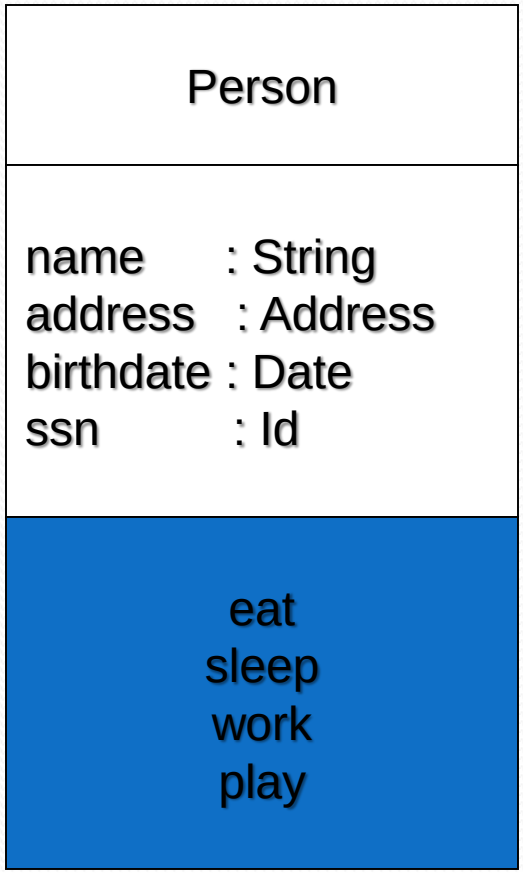
Class Attributes (Cont'd)



Attributes can be:

+ public
# protected
- private
/ derived

Class Operations



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

Class Operations (Cont'd)

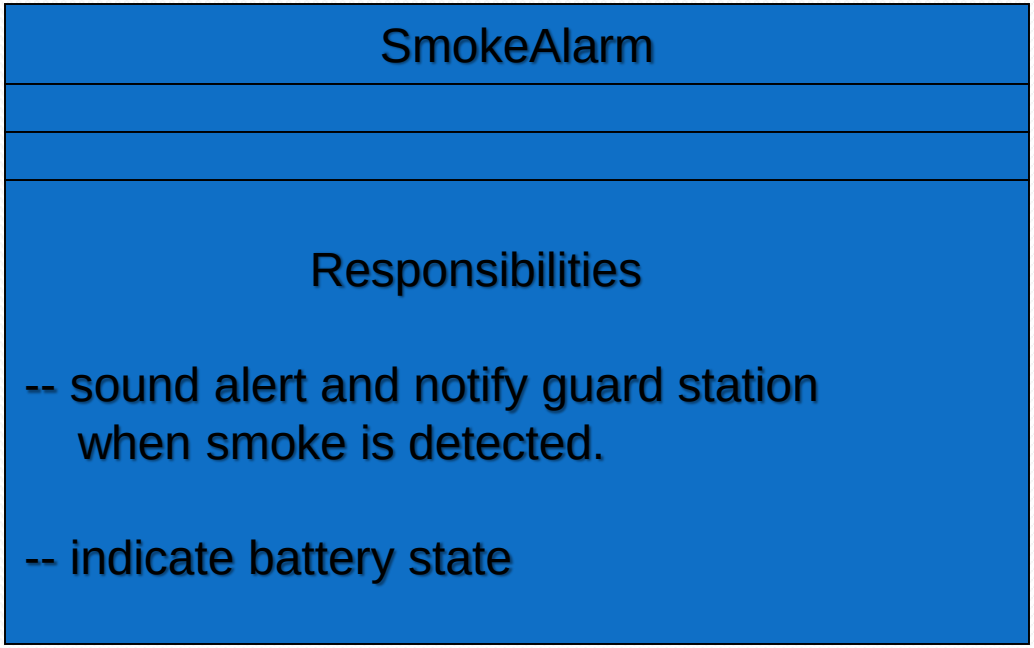
PhoneBook
newEntry (n : Name, a : Address, p : PhoneNumber, d : Description) getPhone (n : Name, a : Address) : PhoneNumber

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.

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.



Relationships

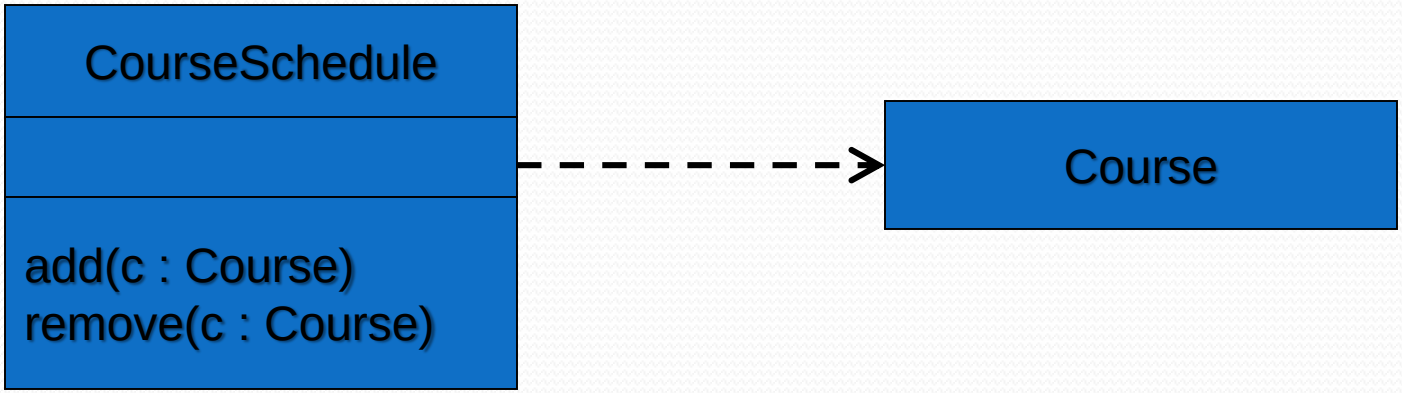
In UML, object interconnections (logical or physical), are modeled as relationships.

There are three kinds of relationships in UML:

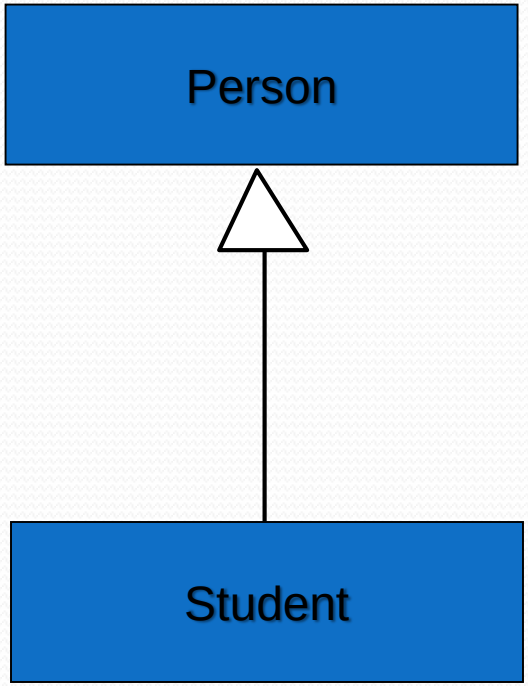
- dependencies
- generalizations
- associations

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*.



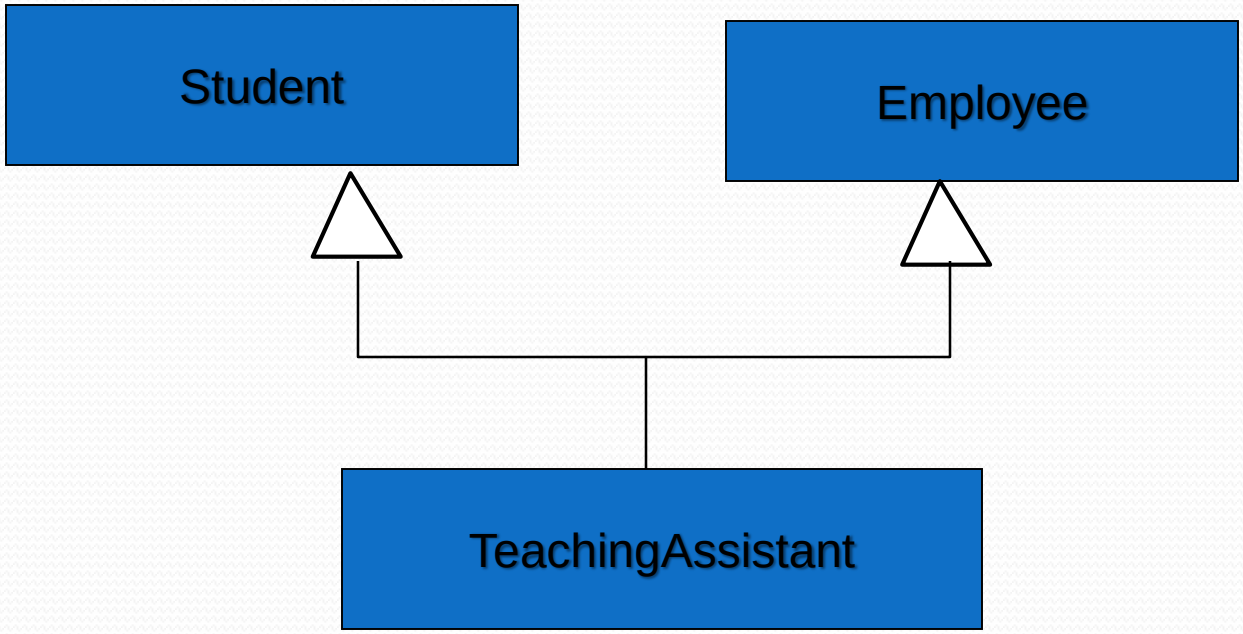
Generalization Relationships



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

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

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

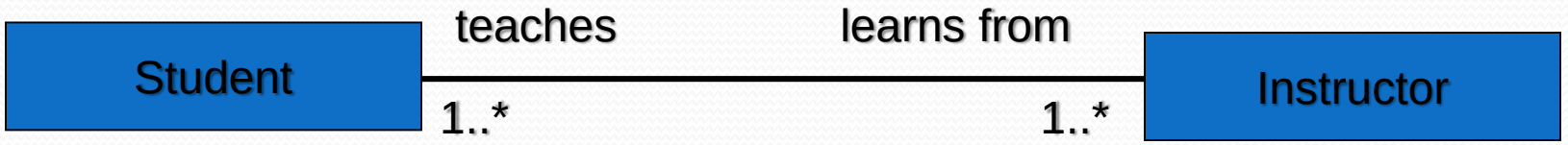
The example indicates that a *Student* has one or more *Instructors*:

Association Relationships

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

Association Relationships

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

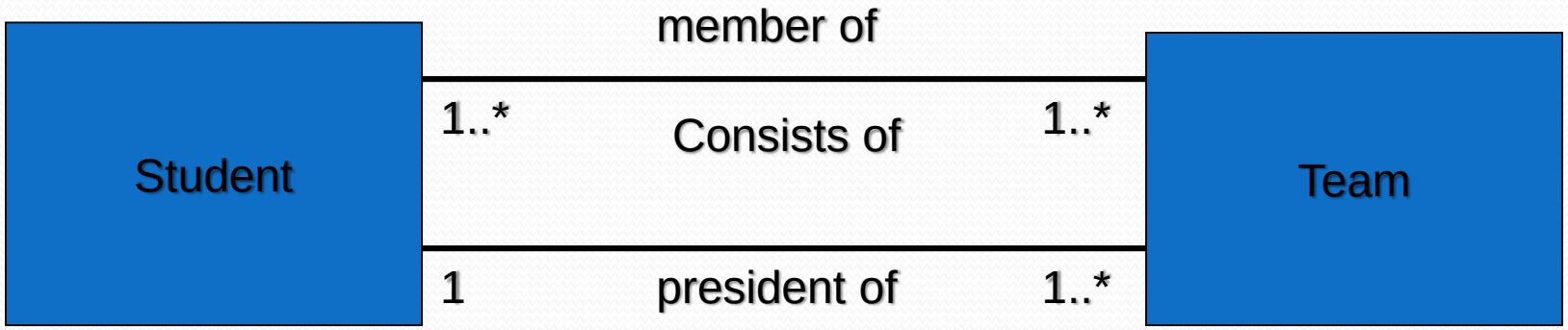


Association Relationships

We can also name the association.

Association Relationships

We can specify dual associations.

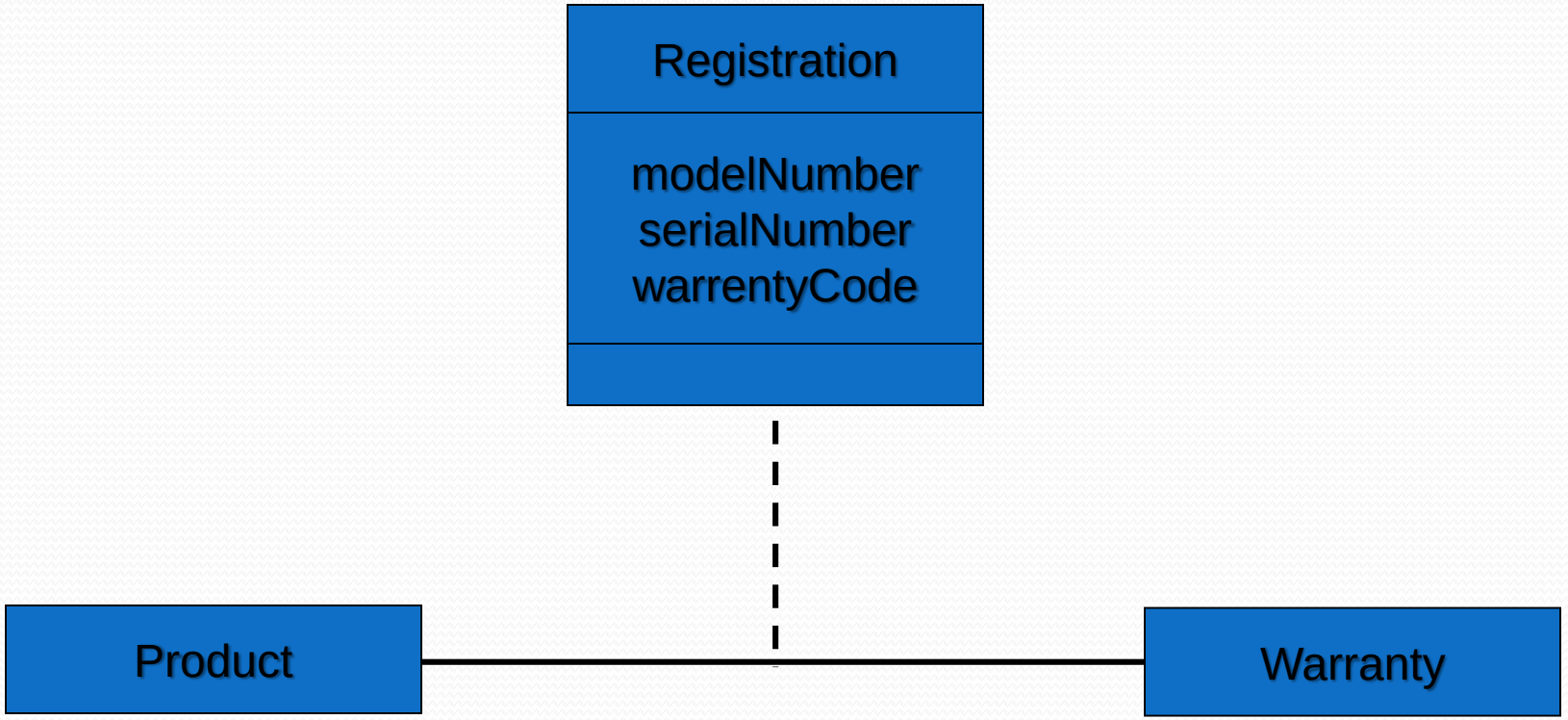


Association Relationships

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*.

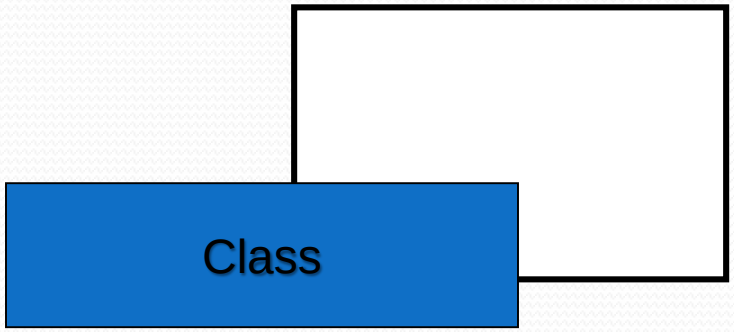
Association Relationships

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

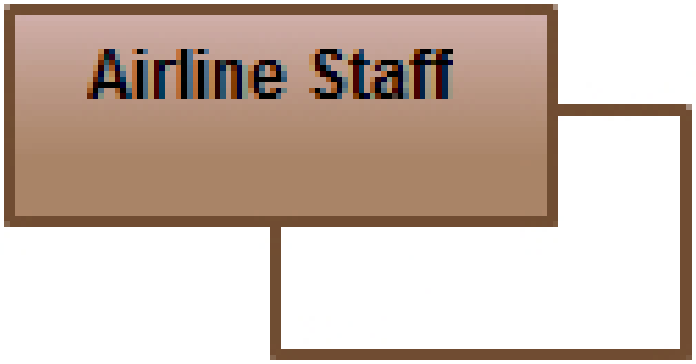


Association Relationships

A class can have a *self association*.



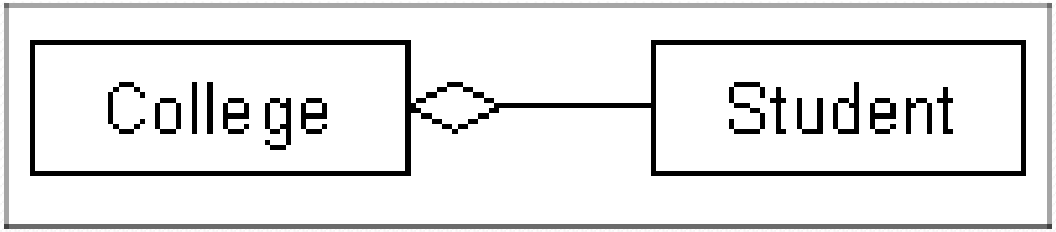
Two instances of the same class:
Pilot
Aviation engineer

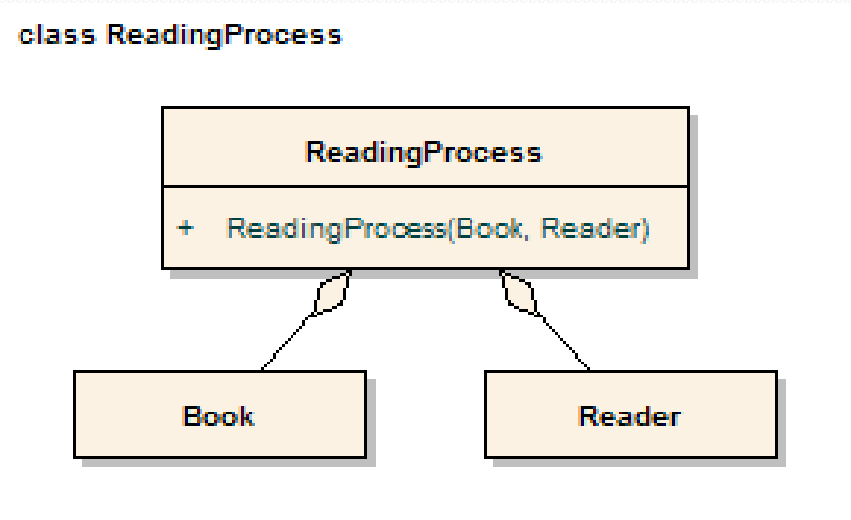


Association Relationships

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

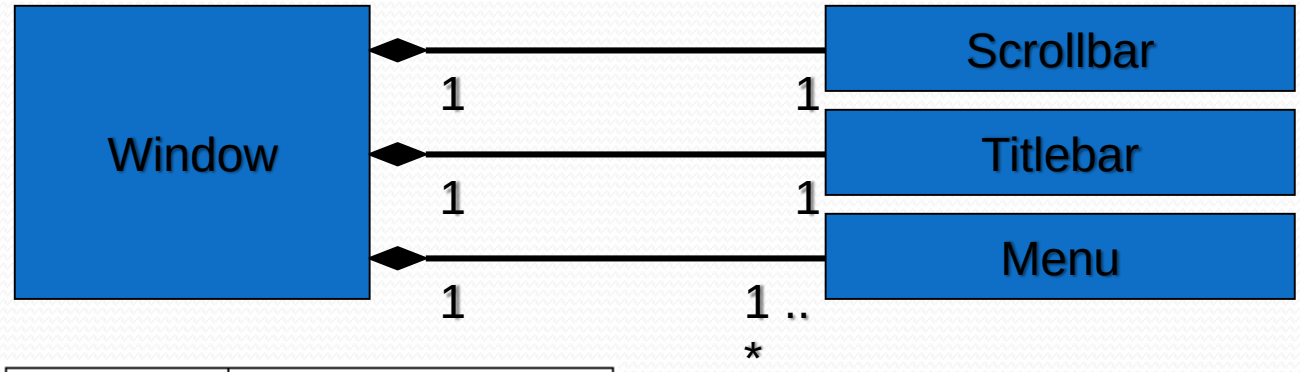
An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.

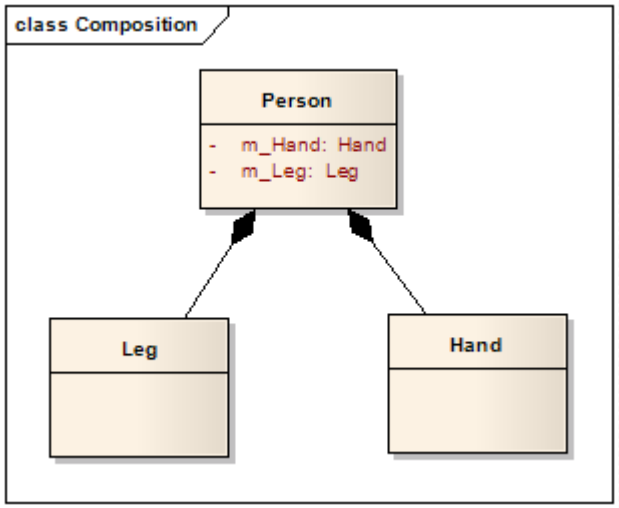


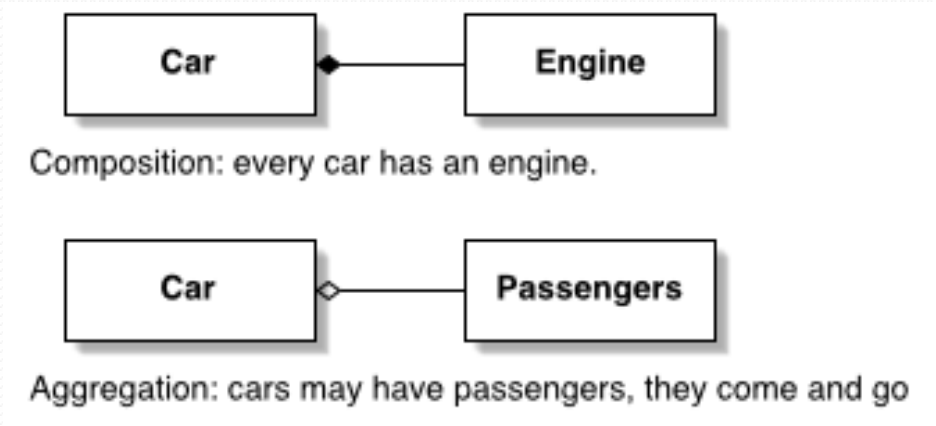


Association Relationships

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.



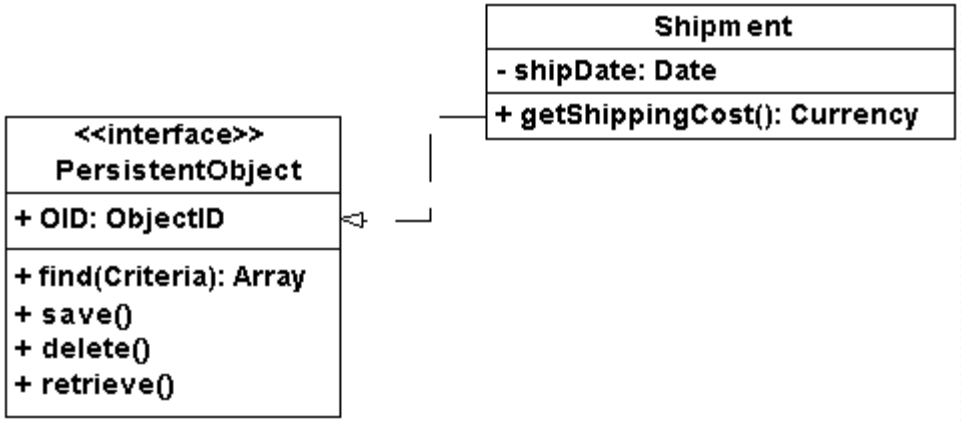




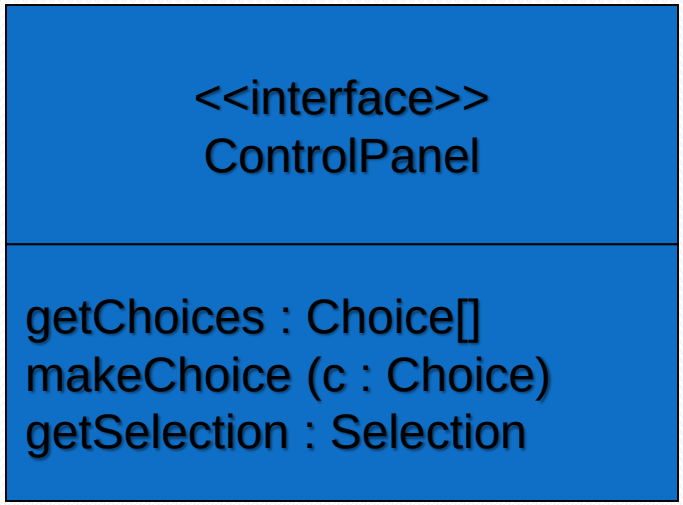
Interfaces

<<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.

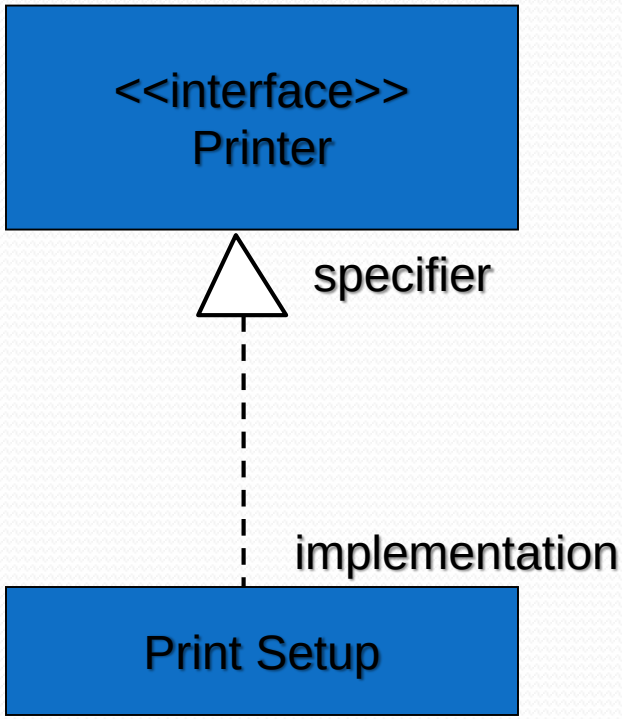


Interface Services

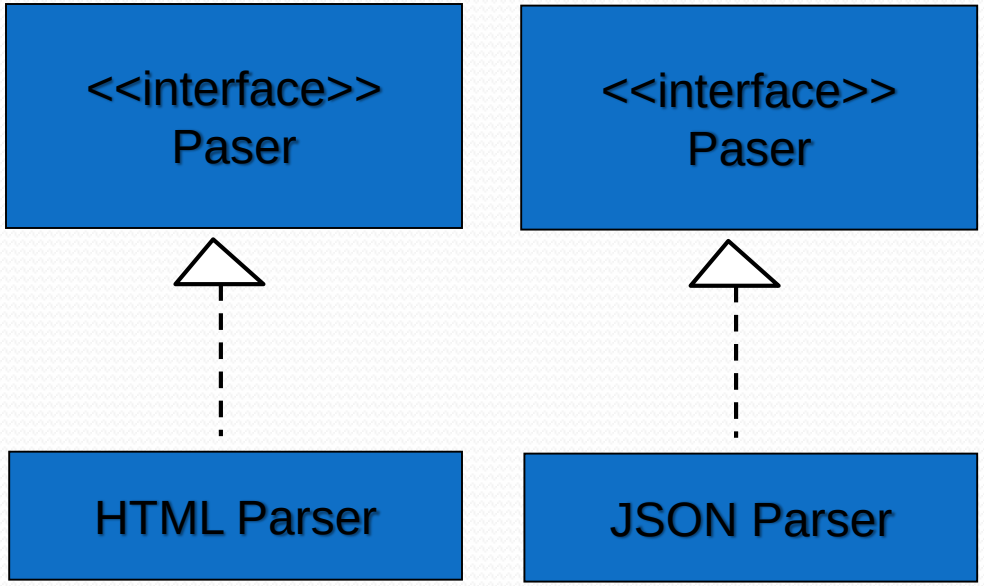


Interfaces do not get instantiated. They have no attributes or state. Rather, they specify the services offered by a related class.

Interface Realization Relationship

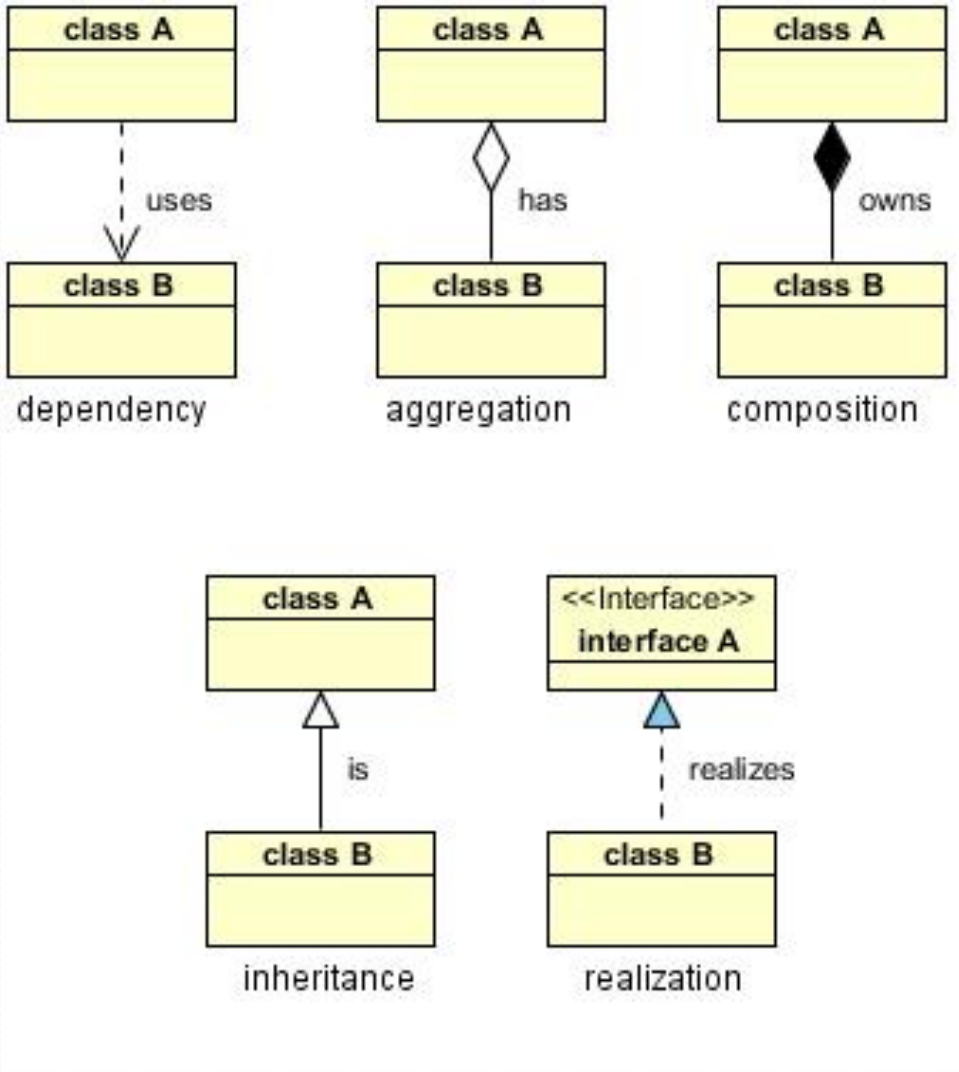


A *realization* relationship connects a class with an interface that supplies its behavioral specification. It is rendered by a dashed line with a hollow triangle towards the specifier.



Relationships in Nutshell

- **Dependency** : class A uses class B
- **Aggregation** : class A has a class B
- **Composition** : class A owns a class B
- **Inheritance** : class B is a Class A (or class A is extended by class B)
- **Realization** : class B realizes Class A (or class A is realized by class B)



Dependency

- **Dependency** is represented when a reference to one class is passed in as a method parameter to another class. For example, an instance of class B is passed in to a method of class A:

```
1public class A {
2
3    public void doSomething(B b) {
```

Aggregation

- Now, if class A stored the reference to class B for later use we would have a different relationship called **Aggregation**. A more common and more obvious example of Aggregation would be via setter injection:

```
1public class A {
2
3    private B _b;
4
5    public void setB(B b) { _b = b; }
```

Composition

- Aggregation is the weaker form of object containment (one object contains other objects). The stronger form is called **Composition**. In Composition the containing object is responsible for the creation and life cycle of the contained object (either directly or indirectly). Following are a few examples of Composition. First, via member initialization:

```
1public class A {
2
3    private B _b = new B();
```

```
1public class A {
2
3    private B _b;
4
5    public A() {
6        _b = new B();
7    } // default constructor
```

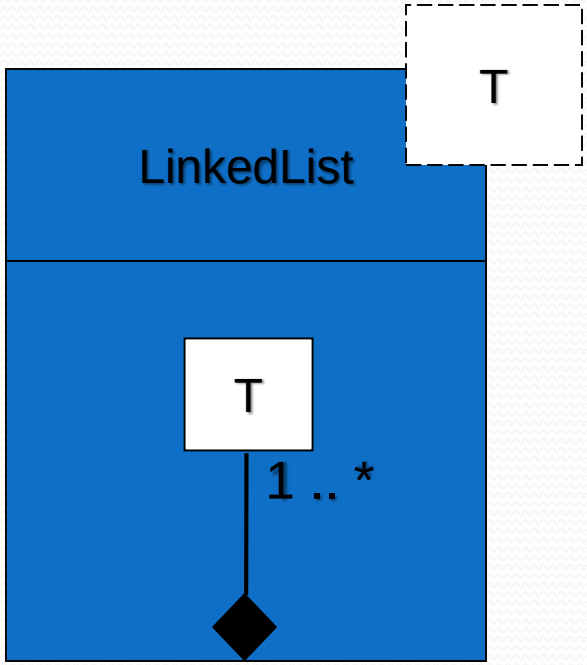
Inheritance

```
 1public class A {
 2
 3    ...
 4
 5} // class A
 6
 7public class B extends A {
 8
 9    ....
10
11} // class B
```

Realization

```
1public interface A {
2
3    ...
4
5} // interface A
6
7public class B implements A {
8
9    ...
10
11} // class B
```

Parameterized Class

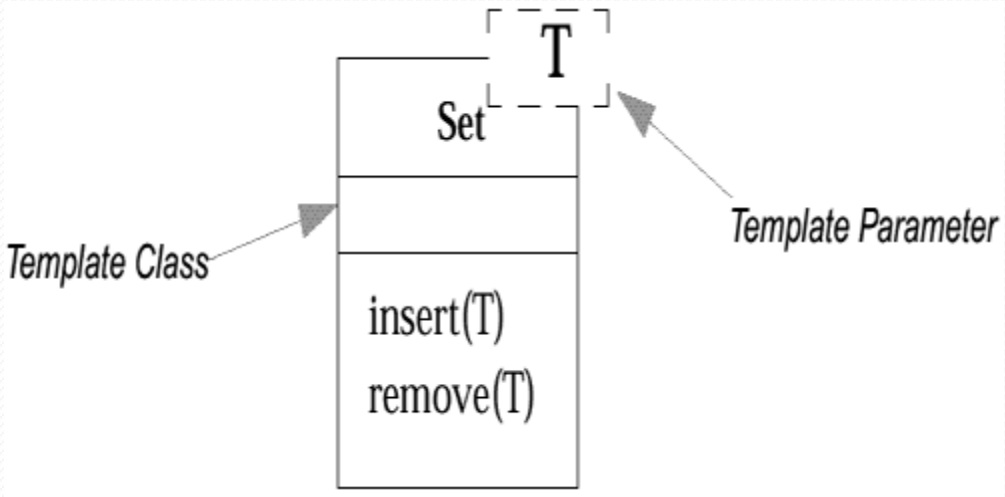


A *parameterized class* or *template* defines a family of potential elements.

To use it, the parameter must be bound.

A *template* is rendered by a small dashed rectangle superimposed on the upper-right corner of the class rectangle. The dashed rectangle contains a list of formal parameters for the class.

Example : Parameterized Class



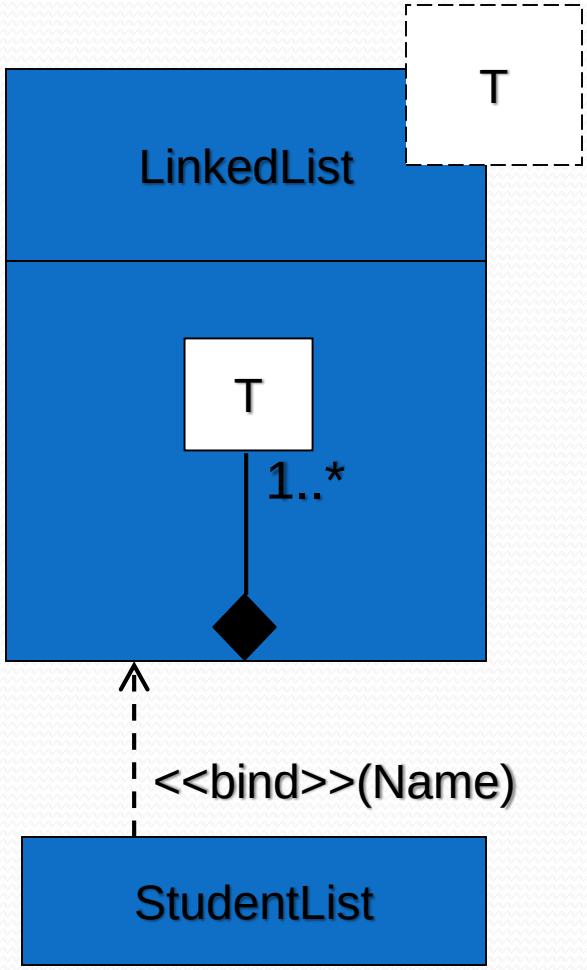

```
class Set <T> {
        void insert (T newElement);
        void remove (T anElement);
```

Parameterized Class (Cont'd)

- Some object-oriented languages such as C++ and Ada support the concept of parametrized classes.
- They are most commonly used for the element type of collection classes, such as the elements of lists.
- For example, in Java, suppose that a *Board* software object holds a *List* of many *Squares*. And, the concrete class that implements the *List* interface is an *ArrayList*:

```
public class Board
{
private List<Square> squares = new ArrayList<Square>();
// …
}
```

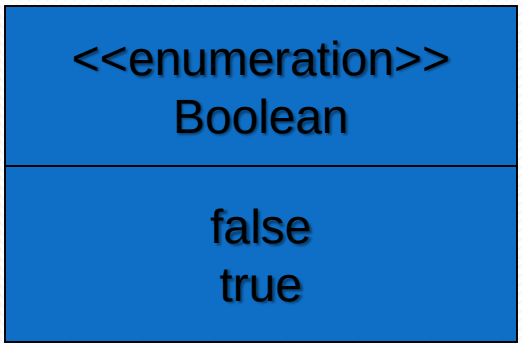
Parameterized Class (Cont'd)



Binding is done with the <<bind>> stereotype and a parameter to supply to the template. These are adornments to the dashed arrow denoting the realization relationship.

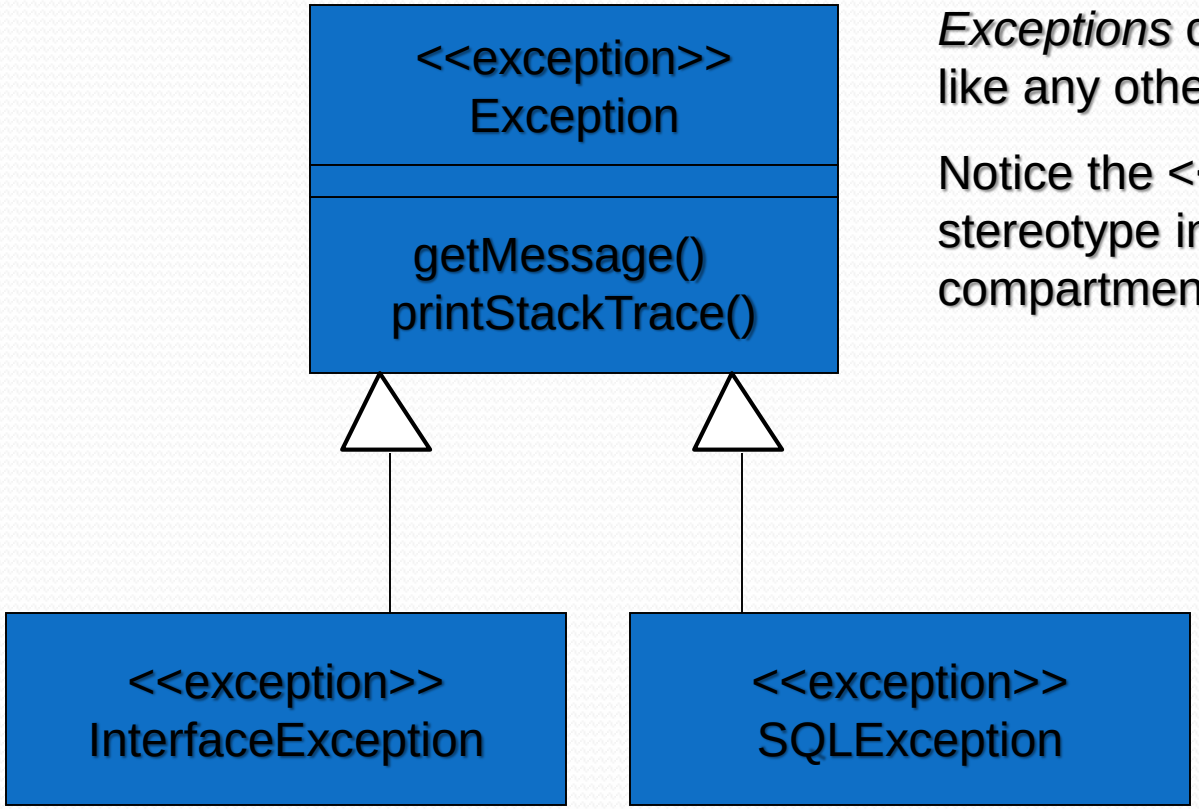
Here we create a linked-list of names for the Students List.

Enumeration



An *enumeration* is a user-defined data type that consists of a name and an ordered list of enumeration literals.

Exceptions



Exceptions can be modeled just like any other class.

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