

Software Analysis and Design (CS:3004)

Course Instructor: Nida Munawar

Email Address: nida.munawar@nu.edu.pk

Reference Book: Applying UML and Patterns (An introduction to Object-Oriented Analysis and Design And Iterative Development)

BY Craig Larman

Third Edition

Create a class in Java

- `class ClassName {`
- `// fields`
- `// methods`
- `}`

Create a class in Java

- `class Car {`
- `// state or field`
- `private int gear = 5;`
- `// behavior or method`
- `public void accelrator() {`
- `System.out.println("Working of accelrator");`
- `}`
- `}`

Java Objects

- `className object = new className();`
- `// for Car class`
- `Car suzuki = new Car();`
- `Car toyota = new Car();`

Basic Syntax

- `public class MyFirstJavaProgram {`
- `public static void main(String []args) {`
- `System.out.println("Hello World");`
- `}`
- `}`

how to save the file, compile, and run the program.

- Open notepad and add the code as above.
- Save the file as: MyFirstJavaProgram.java.
- Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.
- Type 'javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
- Now, type ' java MyFirstJavaProgram ' to run your program.
- You will be able to see ' Hello World ' printed on the window.

Basic Syntax

- **Case Sensitivity** – Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.
- **Class Names** – For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.
- **Example:** *class MyFirstJavaClass*
- **Method Names** – All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.
- **Example:** *public void myMethodName()*
- **Program File Name** – Name of the program file should exactly match the class name.
- When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match, your program will not compile).
- **Example:** Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as '*MyFirstJavaProgram.java*'
- **public static void main(String args[])** – Java program processing starts from the main() method which is a mandatory part of every Java program.

Object and Class Example: main within the class

- class Student{
-
- int id;
- String name;
-
- public static void main(String args[]){
-
- Student s1=new Student();//creating an object of Student
- //Printing values of the object
- System.out.println(s1.id);//accessing member through reference variable
- System.out.println(s1.name);
- }
- }

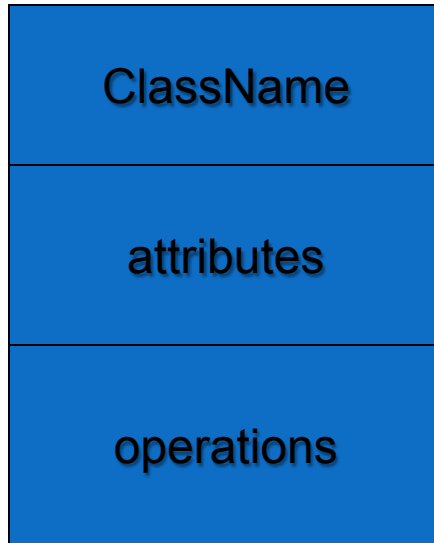
Output

```
E:\jdk\bin>javac Student.java
E:\jdk\bin>java Student
0
null
```

Object and Class Example: main outside the class

- //Creating Student.java class.
- **class** Student{
- **int** id;
- String name;
- }
- //Creating another class TestStudent1.java which contains the main method
- **class** TestStudent1{
- **public static void** main(String args[]){
- Student s1=**new** Student();
- System.out.println(s1.id);
- System.out.println(s1.name);
- }
- }

Classes



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

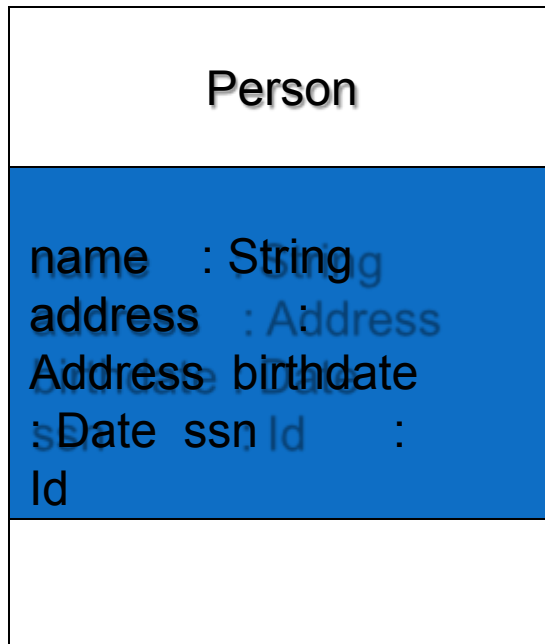
Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

Class Names

ClassName
attributes
operations

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

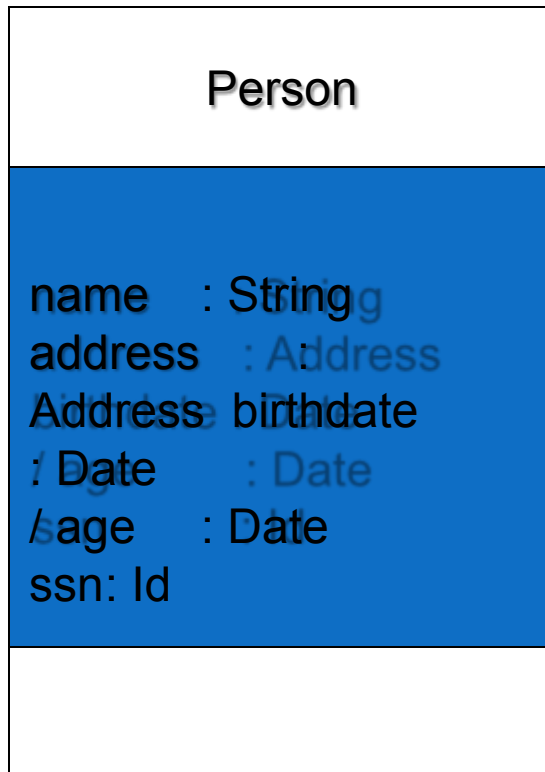


An *attribute* is a named property of a class that describes the object being modeled. In the class diagram, attributes appear in the second compartment just below the name-compartment.

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:

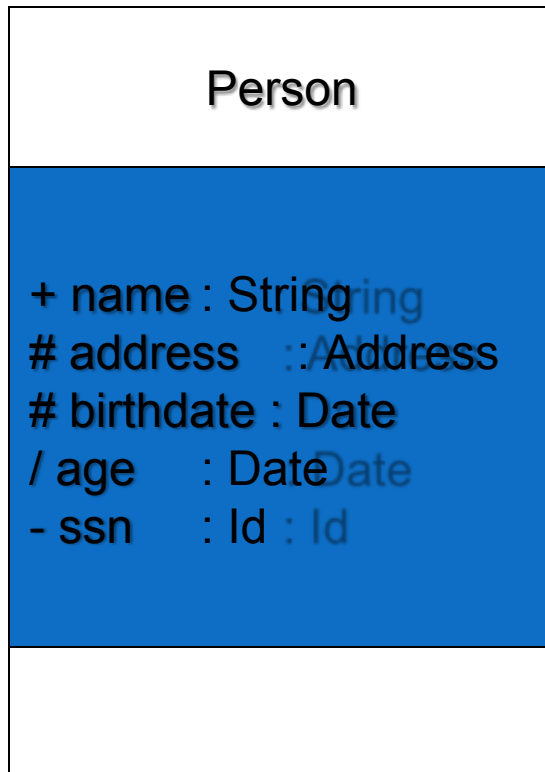
/ age : Date

/ age : Date

Access modifiers

Symbol	Access
+	public
-	private
#	protected
~	default

Class Attributes (Cont'd)



Attributes can be:

- + public
- # protected
- private
- / derived

Describing class and class attributes and constructor

- public class Person {
 - private String name;
 - private int age;
 - public Person(String initialName) {
 - this.name = initialName;
 - this.age = 0;
 - }
 - }

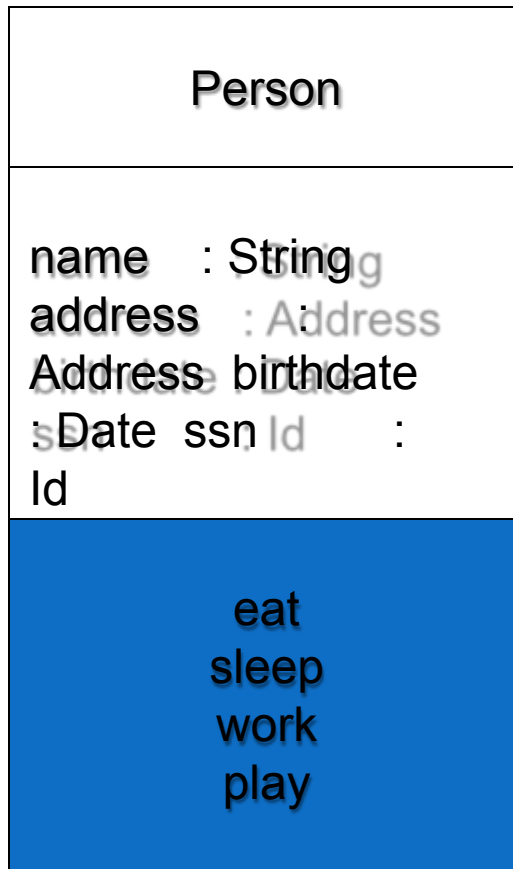
Person
-name: String -age: int
+Person(initialName: String)

Describing class methods

- public class Person {
- private String name;
- private int age;
- public Person(String initialName) {
- this.name = initialName;
- this.age = 0;
- }
- public void printPerson() {
- System.out.println(this.name + ", age " + this.age + " years");
- }
- public String getName() {
- return this.name;
- }
- }

Person
-name: String -age: int
+Person(initialName: String) +printPerson(): void +getName(): String

Class Operations

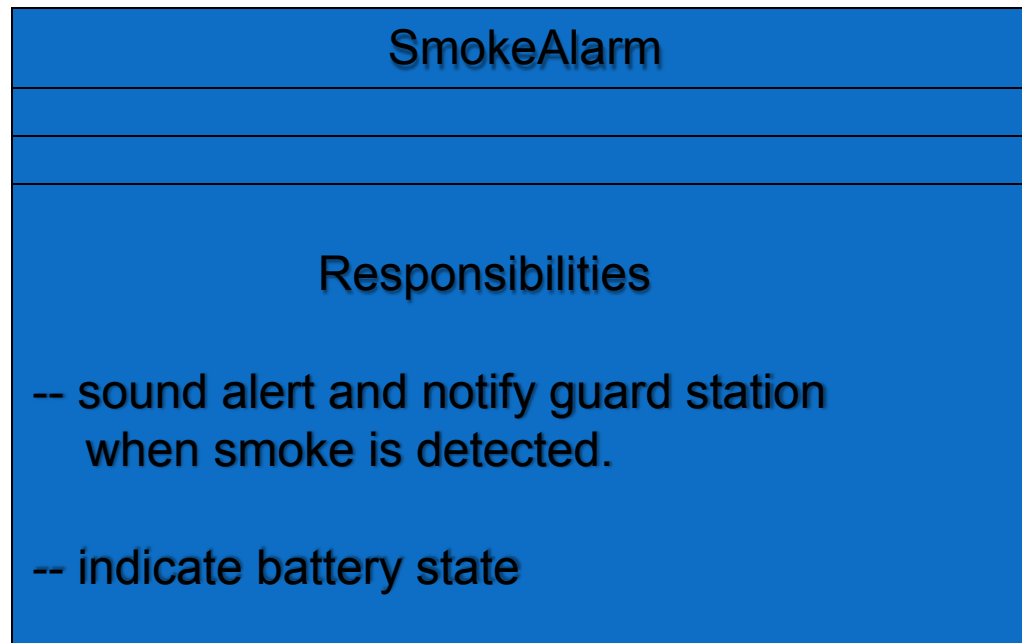


Operations describe the class behavior and appear in the third compartment.

Class Responsibilities

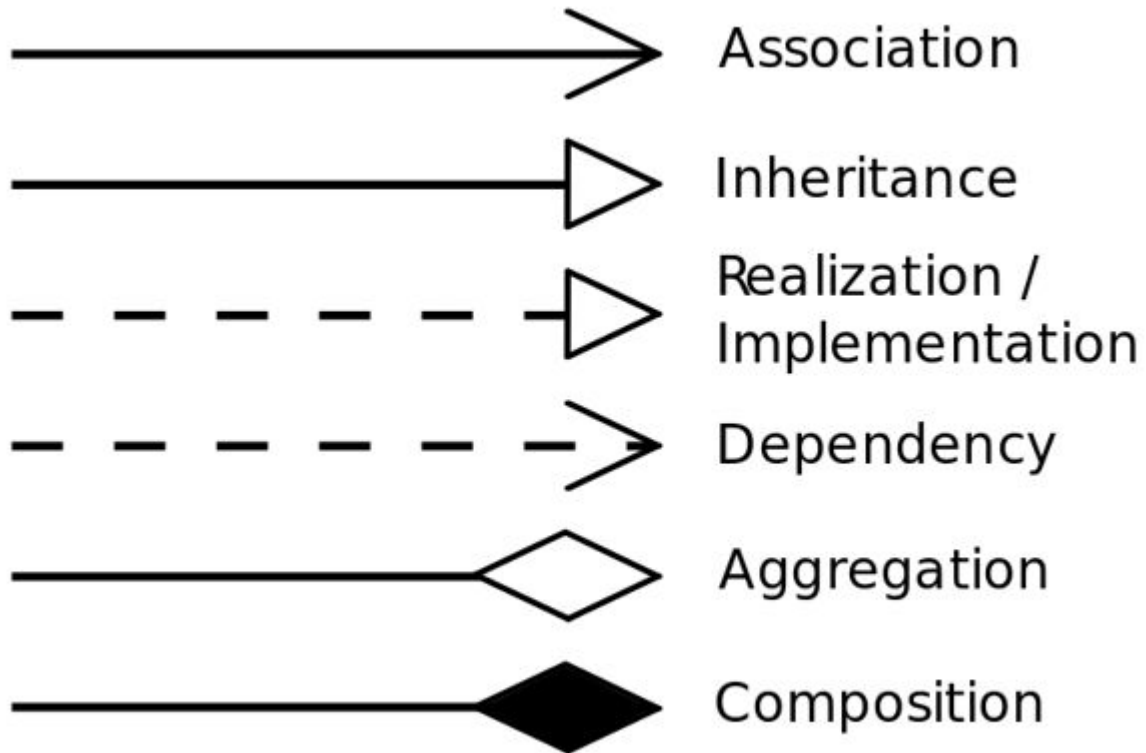
A class may also include its responsibilities in a class diagram.

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



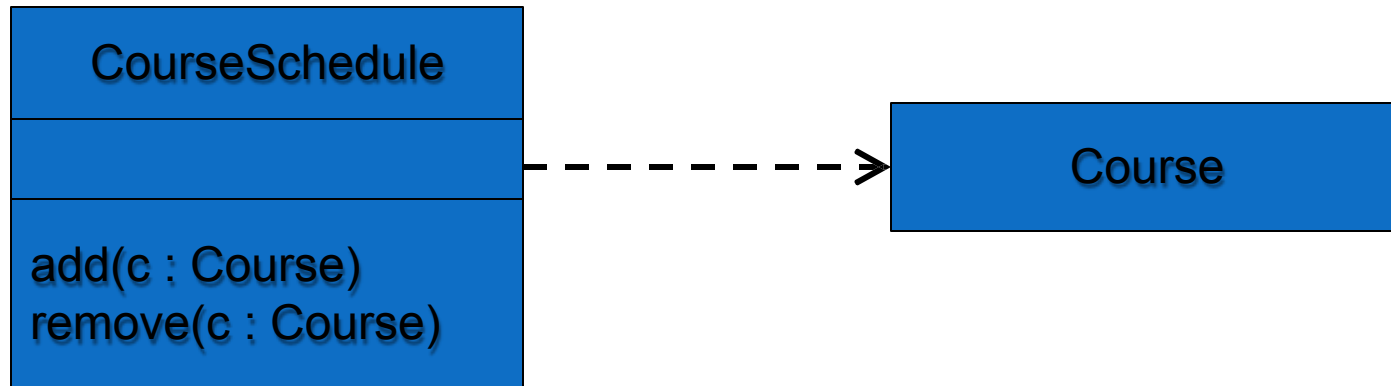
UML Class Diagrams Relationship:

- There are following key relationships between classes in a UML class diagram:
 - 1. Association
 - 2. Generalization/ Inheritance
 - 3. Realization/ Implementation
 - 4. Dependency
 - 5. Aggregation
 - 6. Composition



Dependency Relationships

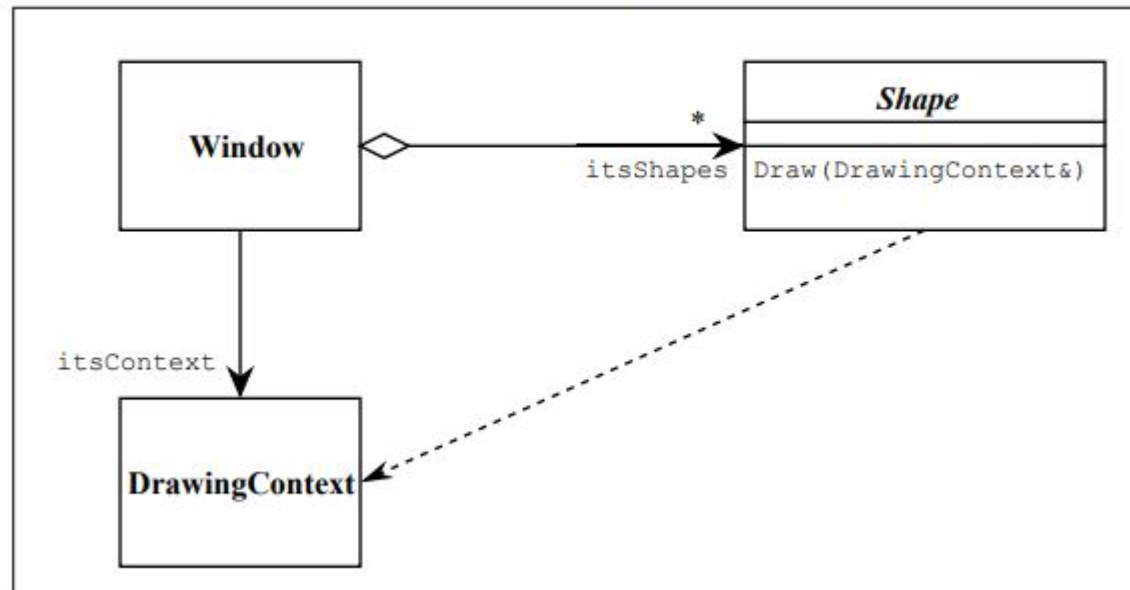
A *dependency* indicates a semantic relationship between two or more elements. The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.



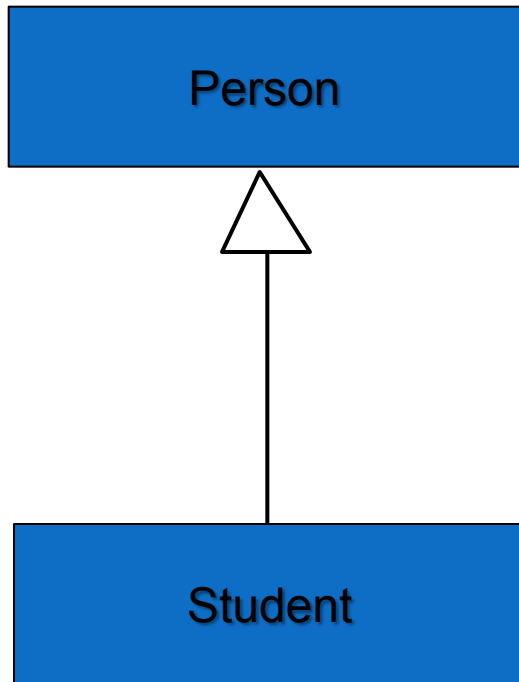
Dependency Relationships('using' relationship)

- Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).

Figure 7: Dependency

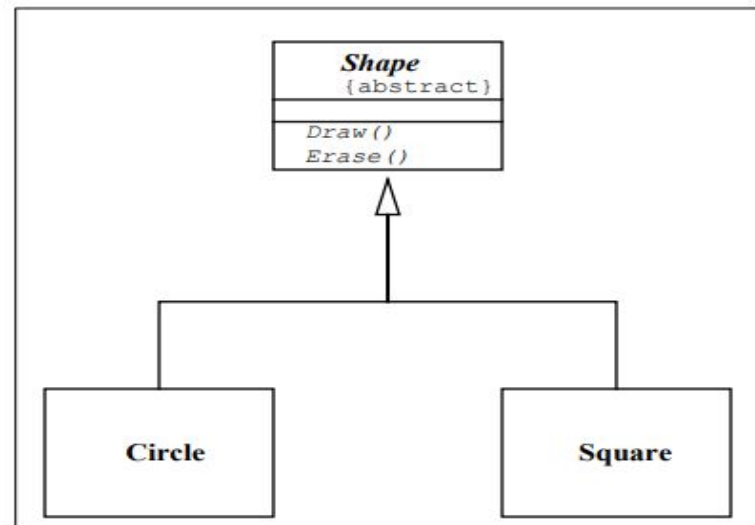


Generalization Relationships



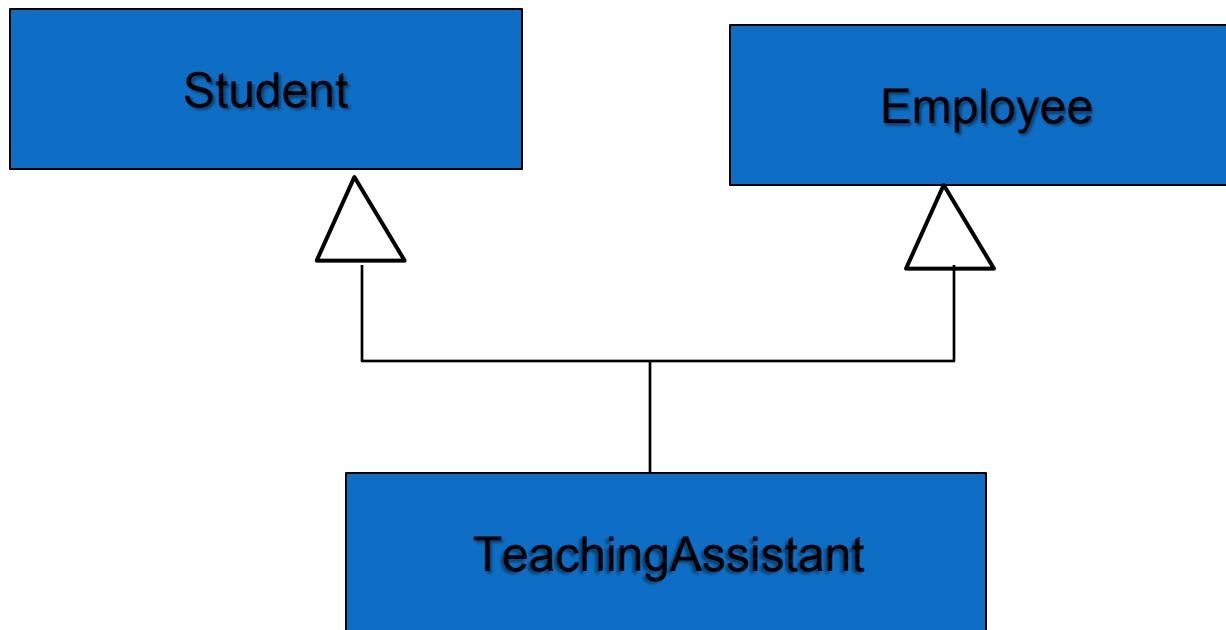
A *generalization* connects a subclass to its superclass. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

Figure 4: Inheritance



Generalization Relationships

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



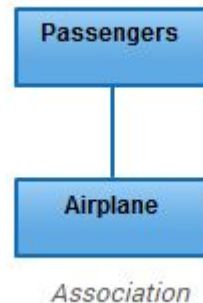
Association Relationships

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

An *association* denotes that link.



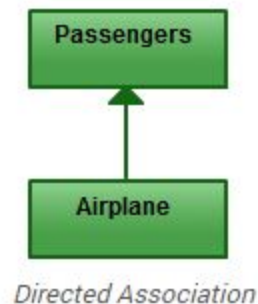
Association is a broad term that encompasses just about any logical connection or relationship between **classes**.



- Some objects are made up of other objects. Association specifies a "has-a" or "whole/part" relationship between two classes. In an association relationship, an object of the whole class has objects of part class as instance data.

Directed Association(**Unidirectional association**)

- In a unidirectional association, two classes are related, but only one class knows that the relationship exists.
- A unidirectional association is drawn as a solid line with an open arrowhead pointing to the known class.



Bidirectional (standard) association

- - An association is a linkage between two classes. Associations are always assumed to be bi-directional; this means that both classes are aware of each other and their relationship, unless you qualify the association as some other type.
- A bi-directional association is indicated by a solid line between the two classes.



Multiplicity

- Place multiplicity notations near the ends of an association. These symbols indicate the number of instances of one class linked to one instance of the other class. For example, one company will have one or more employees, but each employee works for one company only.



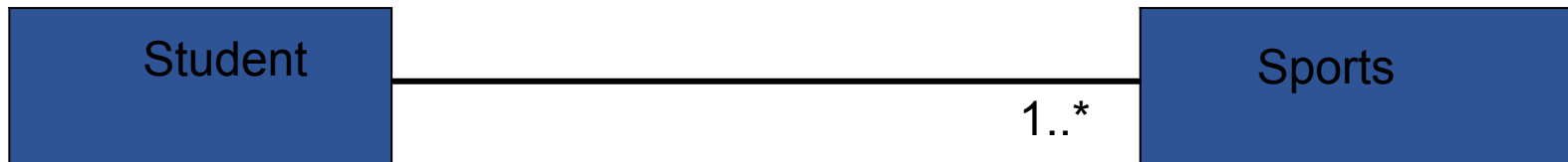
Multiplicities examples:

1	Exactly one, no more and no less
0..1	Zero or one
*	Many
0..*	Zero or many
1..*	One or many

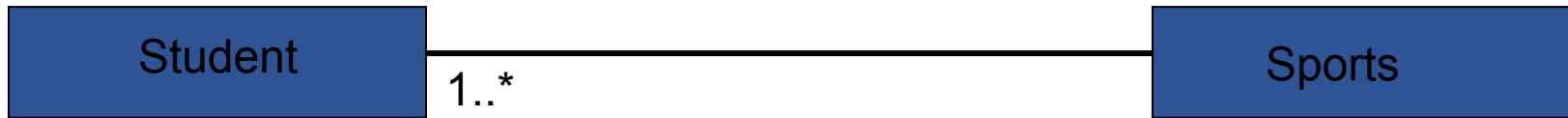
Association Relationships

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

:

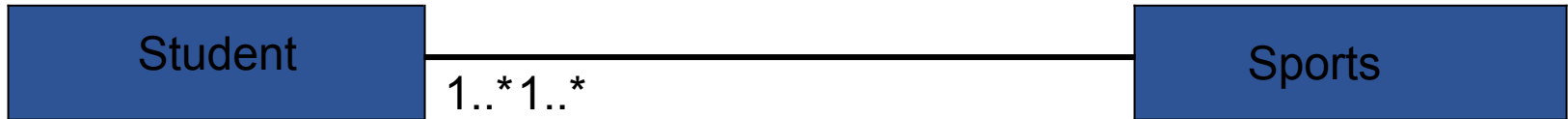


Association Relationships



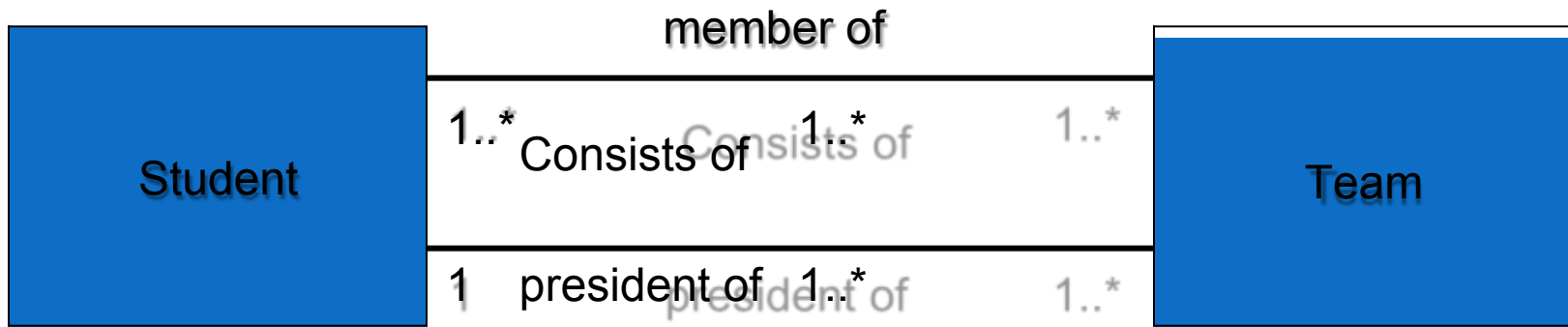
Association Relationships

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



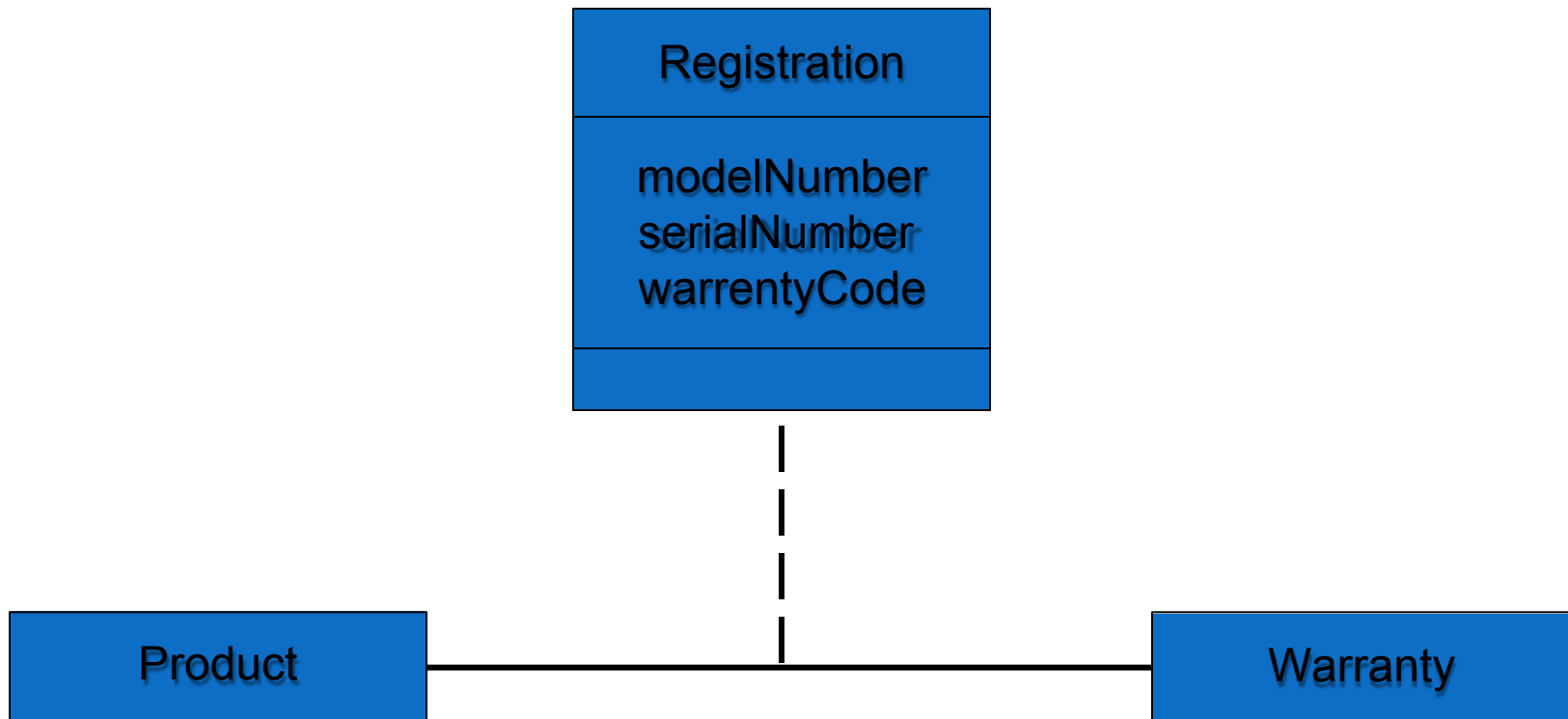
Association Relationships

We can specify dual associations.



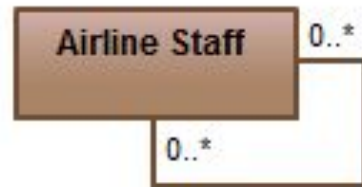
Association Relationships

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



Reflexive Association

- This occurs when a class may have multiple functions or responsibilities. For example, a staff member working in an airport may be a pilot, aviation engineer, a ticket dispatcher, a guard, or a maintenance crew member.

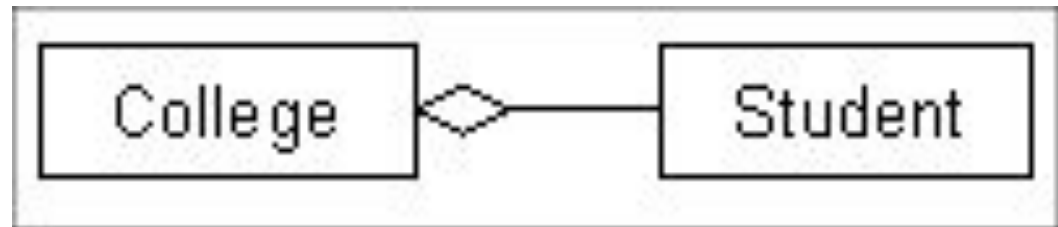


Reflexive Association

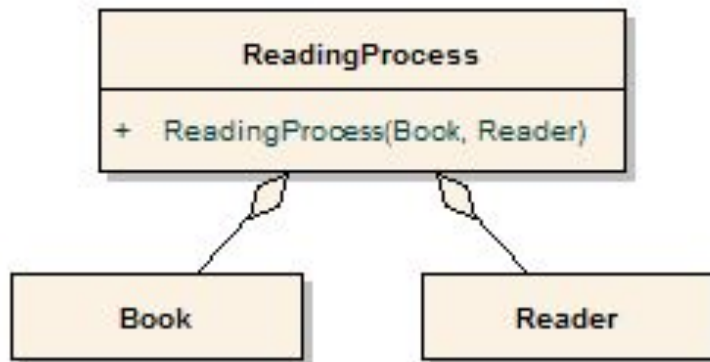
Association Relationships

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.

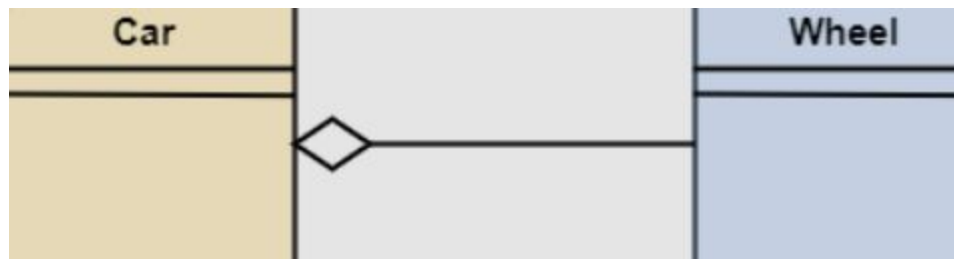


class ReadingProcess



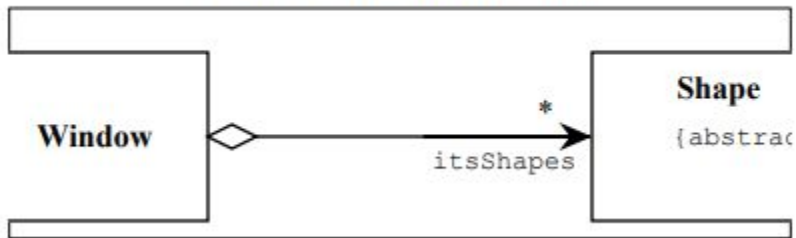
Aggregation

- In aggregation, the contained classes are not strongly dependent on the lifecycle of the container.
- Let us consider an example of a car and a wheel. A car needs a wheel to function correctly, but a wheel doesn't always need a car. It can also be used with the bike, bicycle, or any other vehicles but not a particular car. Here, the wheel object is meaningful even without the car object. Such type of relationship is called an aggregation relation.



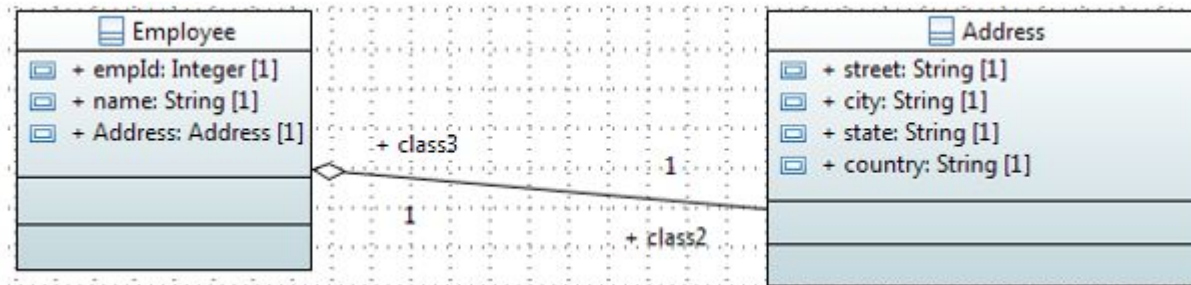
Aggregation in c++

Figure 5: Aggregation



```
class Window {  
public:  
//...  
private:  
vector itsShapes;  
};
```

Aggregation in java



Address.java

```
• public class Address {  
    String city,state,country;  
  
    public Address(String city, String state, String country) {  
        this.city = city;  
        this.state = state;  
        this.country = country;  
    }  
}
```

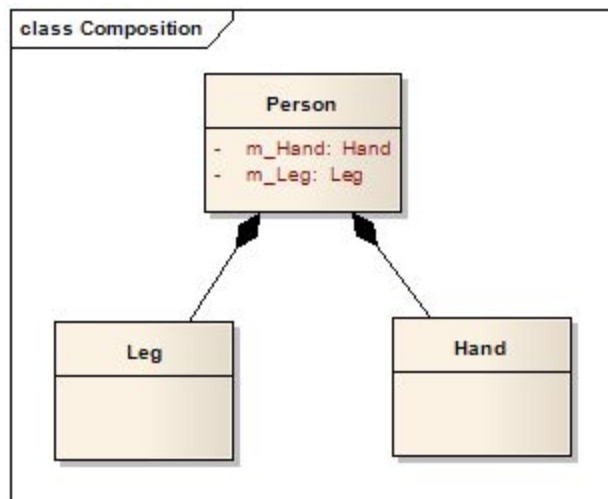
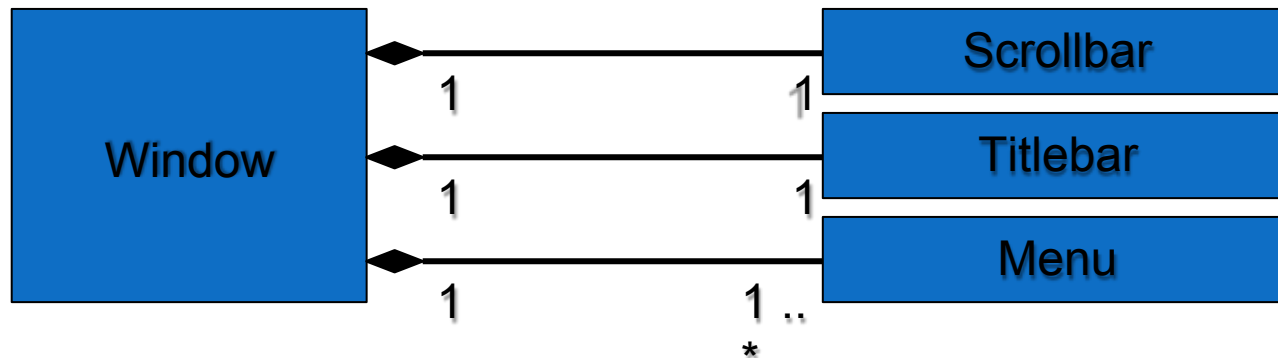
Employee.java

```
• public class Employee {  
    int id;  
    String name;  
    Address address;  
  
    public Employee(int id, String name, Address address) {  
        this.id = id;  
        this.name = name;  
        this.address=address;  
    }  
}
```

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.

It is not a standard UML relationship,



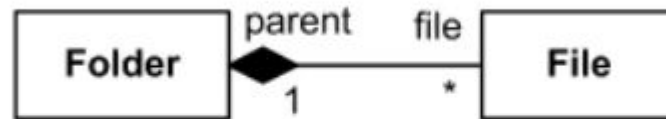
Composition: every car has an engine.



Aggregation: cars may have passengers, they come and go

Composition

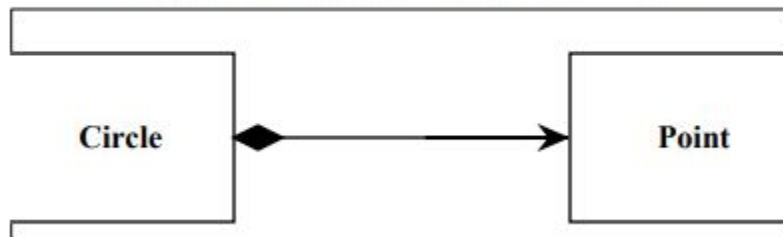
When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.



Composition in UML

Composition in c++

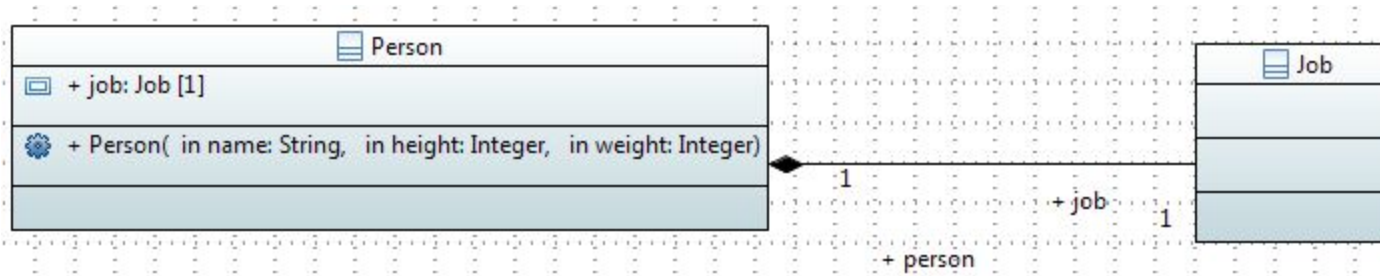
Figure 3: Circle contains Point



In this case we have represented the composition relationship as a member variable. We could also have used a pointer so long as the destructor of **Circle** deleted the pointer

```
class Circle {
public:
void SetCenter(const Point&);
void SetRadius(double);
double Area() const;
double Circumference() const;
private:
double itsRadius;
Point itsCenter;
};
```


Composition in java



Job.java

```
• public class Job {  
    private long salary;  
  
    public long getSalary() {  
        return salary;  
    }  
    public void setSalary(long salary) {  
        this.salary = salary;  
    }  
}
```

Person.java

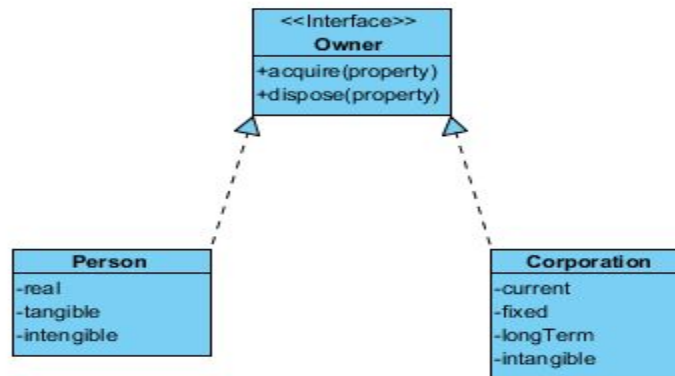
```
• public class Person {  
    //composition has-a relationship  
    private Job job;  
  
    public Person(){  
        this.job=new Job();  
        job.setSalary(1000L);  
    }  
    public long getSalary() {  
        return job.getSalary();  
    }  
}
```

Main .java

```
• public class TestPerson {  
  
    public static void main(String[] args) {  
        Person person = new Person();  
        long salary = person.getSalary();  
    }  
  
}
```

Realization relation(interface)

- A realization is a relationship between two things where one thing (an interface) specifies a contract that another thing (a class) guarantees to carry out by implementing the operations specified in that contract.
- In a class diagram, realization relationship is rendered as a dashed directed line with an open arrowhead pointing to the interface.
- For example, the Owner interface might specify methods for acquiring property and disposing of property. The Person and Corporation classes need to implement these methods, possibly in very different ways.



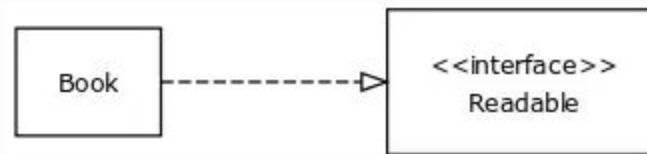
Interfaces



```
graph LR; subgraph ControlPanel; direction TB; S1[<<interface>>]; S2[ControlPanel]; end;
```

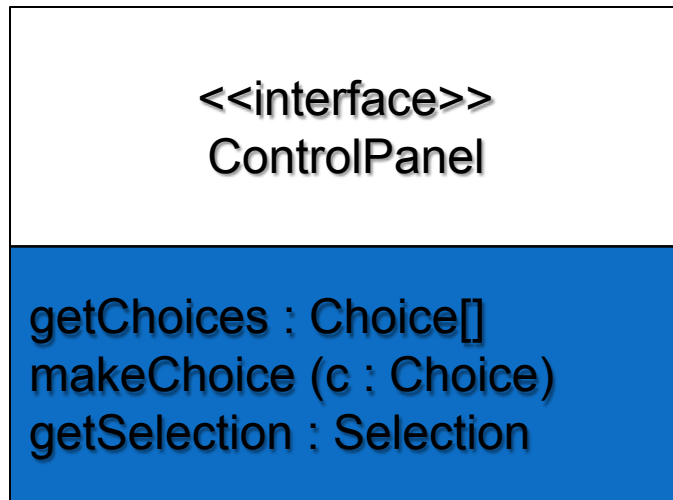
<<interface>>
ControlPanel

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.



Book implements interface Readable

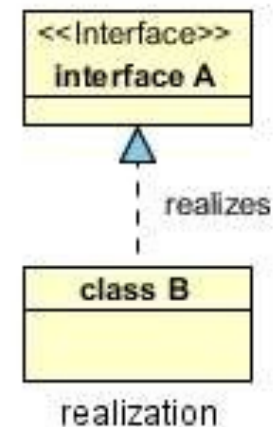
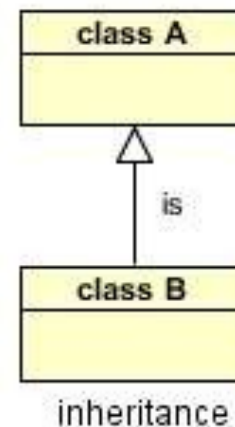
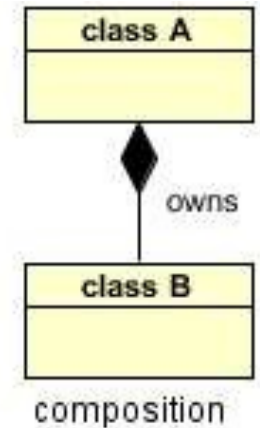
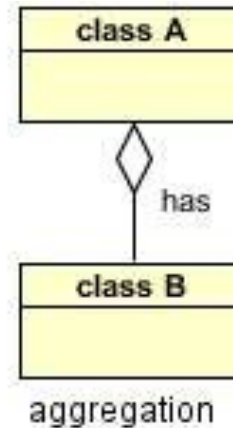
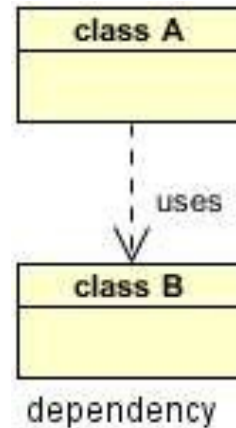
Interface Services



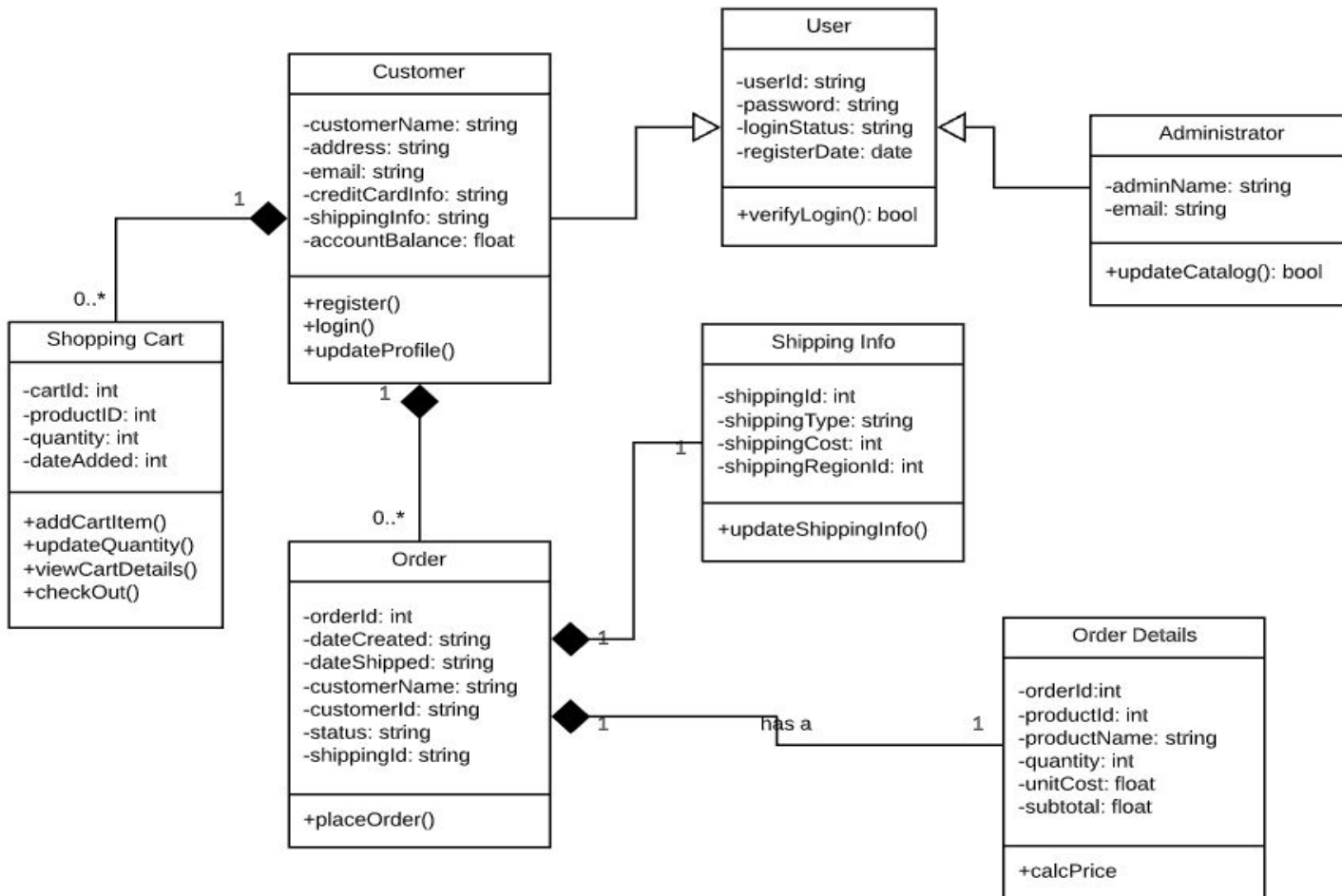
Interfaces do not get instantiated. They have no attributes or state. Rather, they specify the services offered by a related class.

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)



Identify the relationship

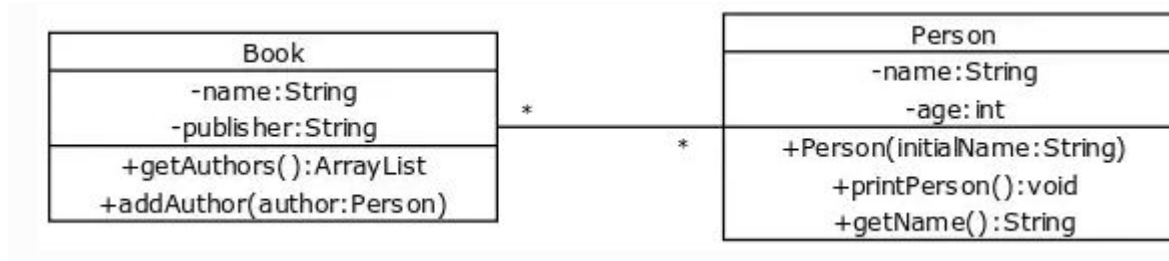


Dependency

 **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:

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

Association



```
public class Book {  
    private String name;  
    private String publisher;  
    private ArrayList<Person> authors;  
  
    // ..  
}
```

```
import java.util.ArrayList;  
  
public class Person {  
    private String name;  
    private int age;  
    private ArrayList<Book> books;  
  
    // ...  
}
```

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:

```
1 public 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:

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

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

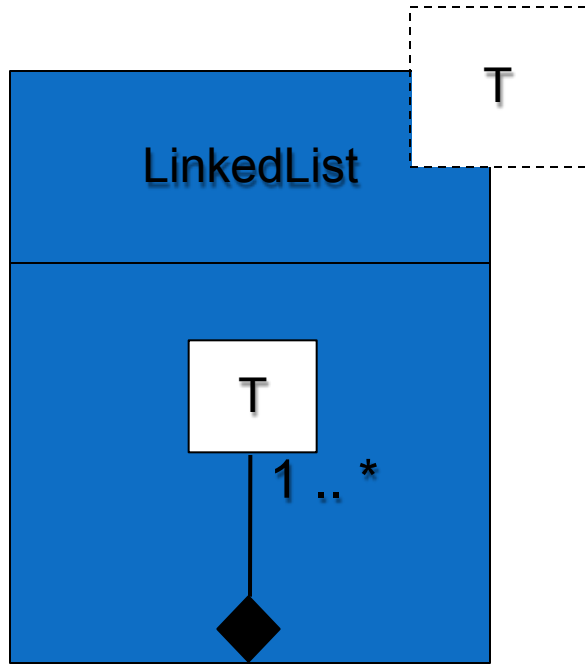
Inheritance

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

Realization

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

Parameterized Class

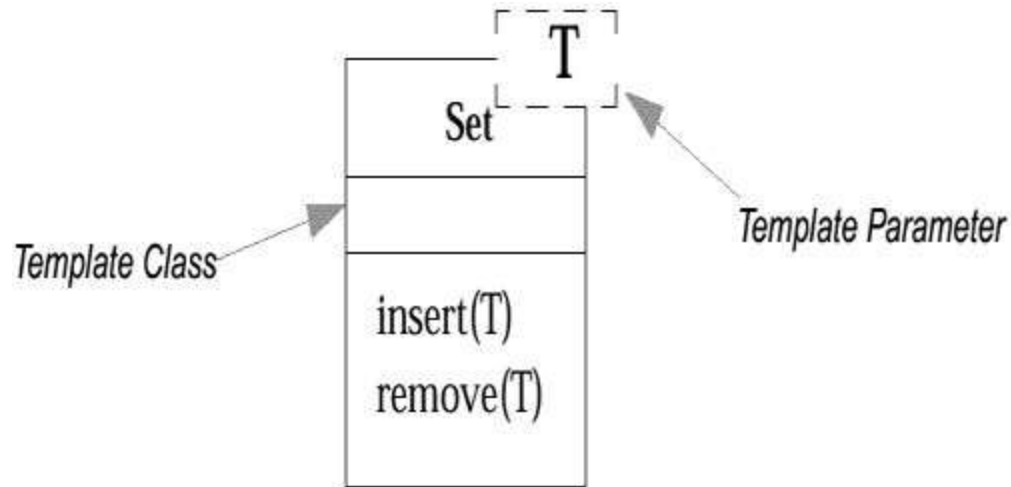


A parameterized class or template defines a family of potential elements.

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.*

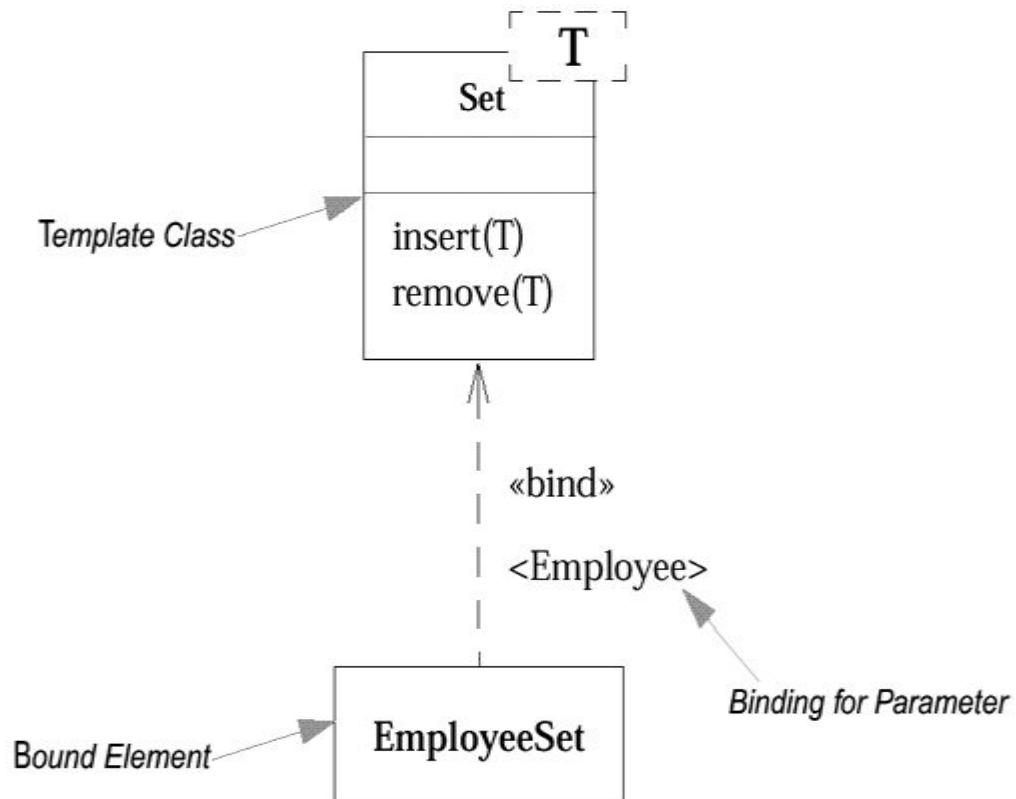
Example : Parameterized Class



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

```
Set <Employee> employeeSet;
```

- A use of a parameterized class, such as *Set<Employee>*, is called a **bound element**.

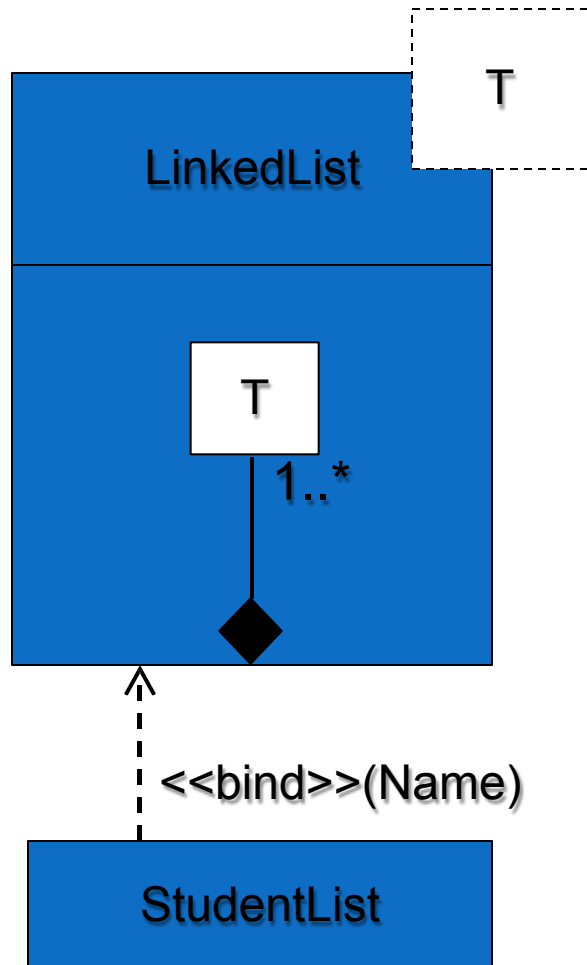


Parameterized Class (Cont'd)

- 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*:

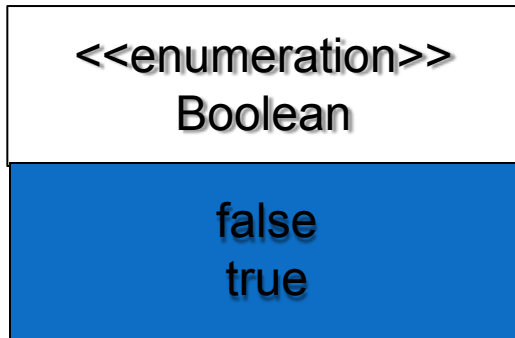
```
public class Board
{
    private List<Square> squares = new ArrayList<Square>();
    // ...
}
```

Parameterized Class (Cont'd)



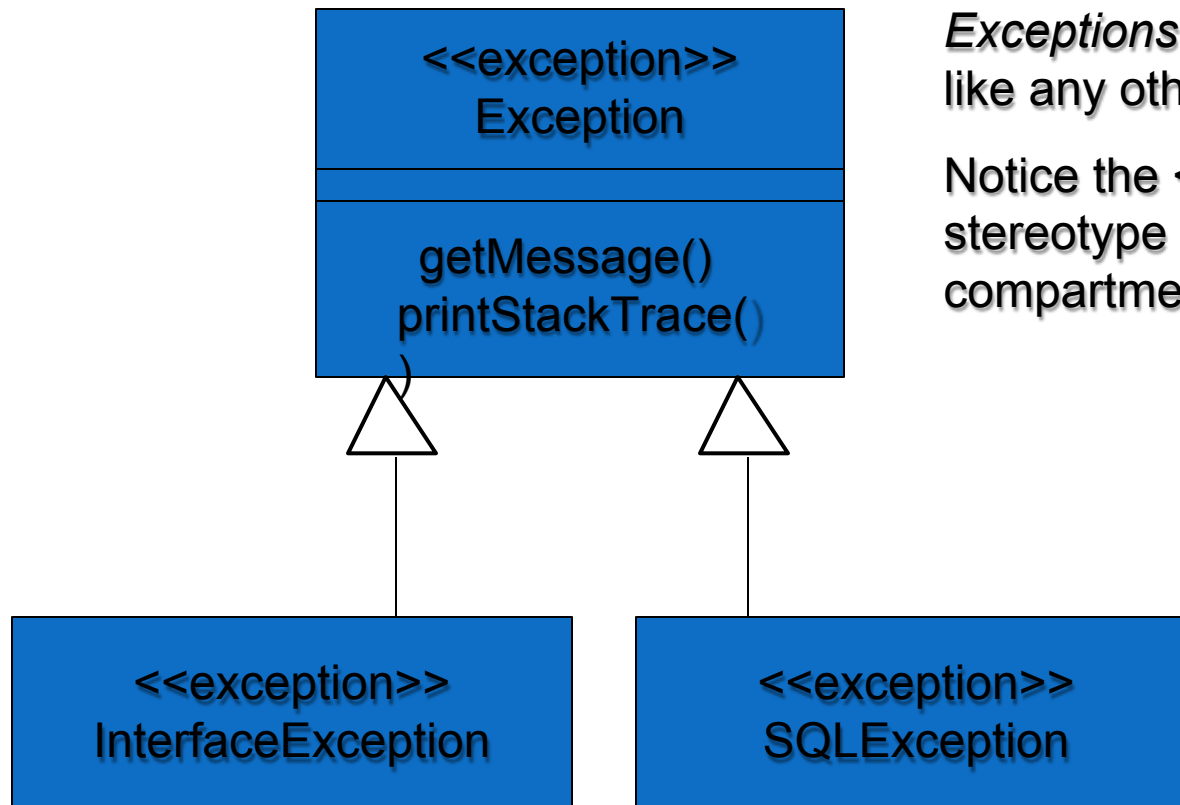
- *Binding* is done with the `<<bind>>` stereotype and a parameter to supply to the template. These are adornments to the dashed arrow denoting the realization relationship.
- Here we create a linked-list of names for
- the Students List.

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.