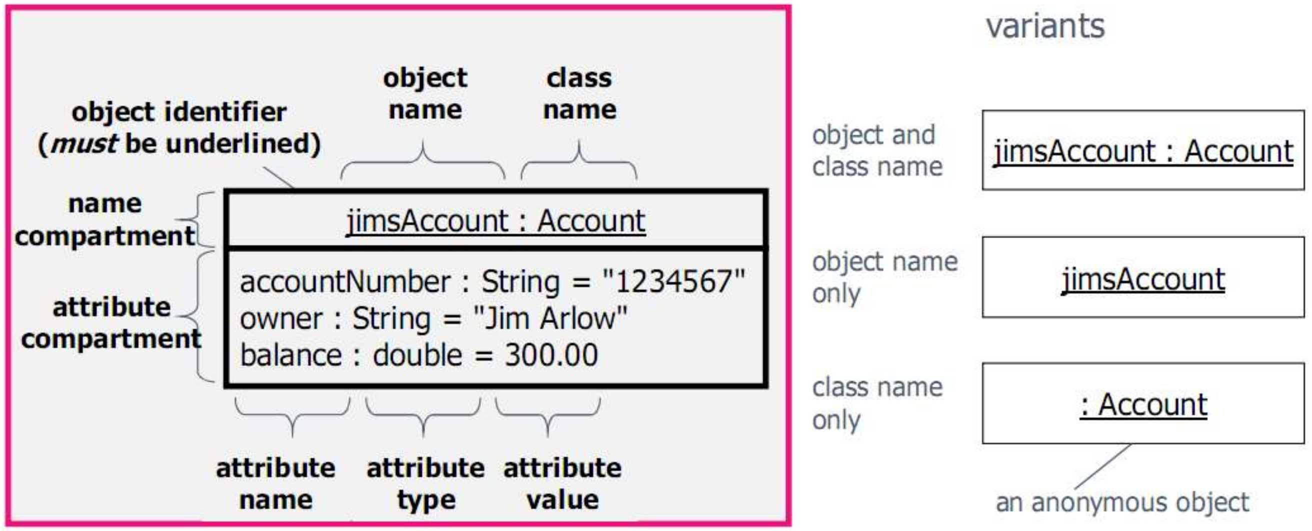
Class Diagram
(Used Object-oriented Design)

- Classes capture the structure of a piece of software
  - Class operations and attributes
  - Class interfaces
  - ⇒ Class Diagrams represent a *static model* of the system

Object-oriented concepts (Modelled by class diagram)

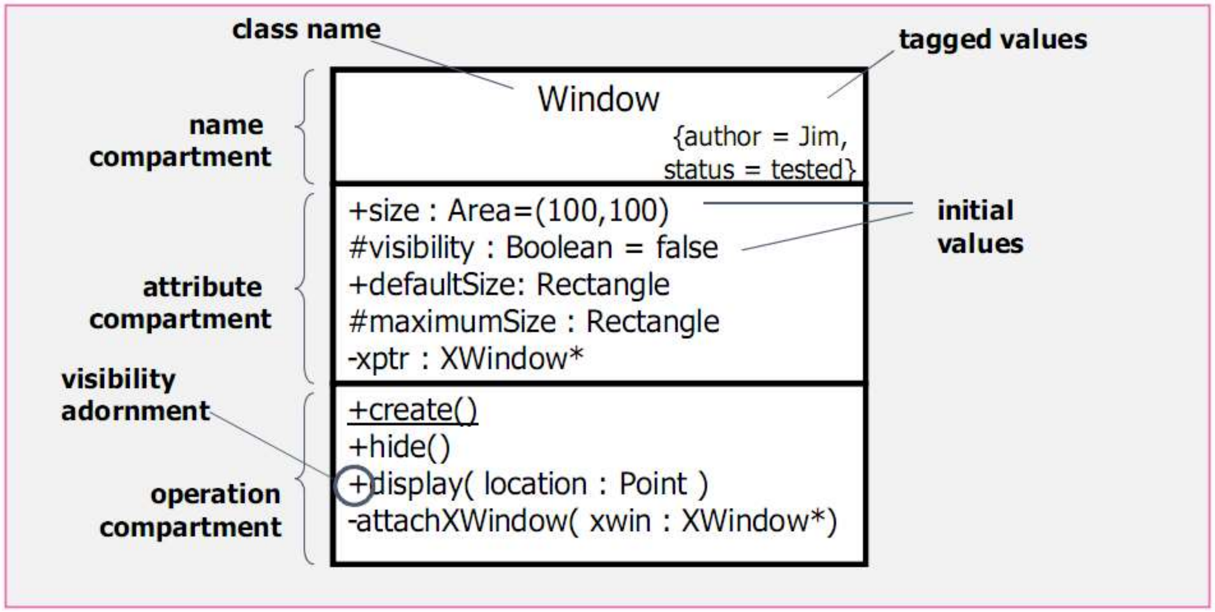
- Classes
  - Encapsulation (information hiding & access methods to hidden data)
- Objects
  - Instances of classes at runtime
- Inheritance and Polymorphism
  - Derived classes
- Aggregation and Composition
  - Objects inside classes
- Interface Realization
  - abstraction
- Domain Model
  - Classes with attributes

UML Object Syntax



- All objects of a particular class have the same set of operations. They are not shown on the object diagram, they are shown on the class diagram (see later)
- Attribute types are often omitted to simplify the diagram
- Naming:
  - object and attribute names in lowerCamelCase
  - class names in UpperCamelCase

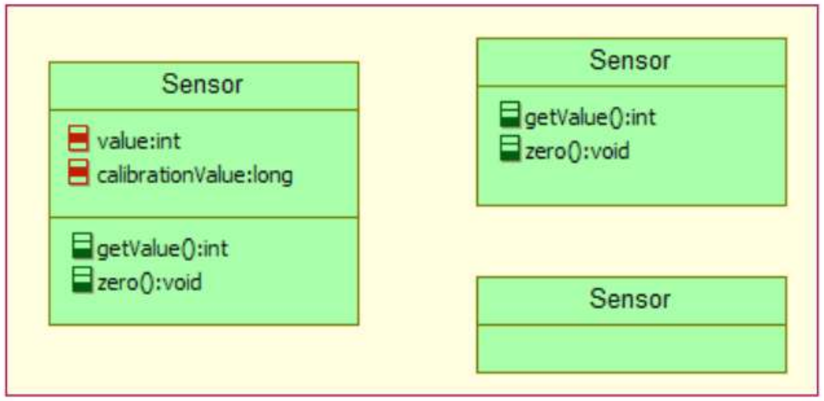
UML Class Notation



- Classes are named in UpperCamelCase
- Use descriptive names that are nouns or noun phrases
- Avoid abbreviations!

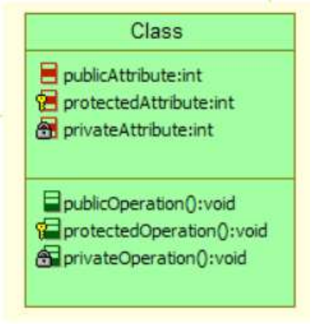
UML Class

- A class can be shown on a Class Diagram in one of many ways:
  - Name
  - Name + operations
  - Name + operations + attributes



Visibility Defined

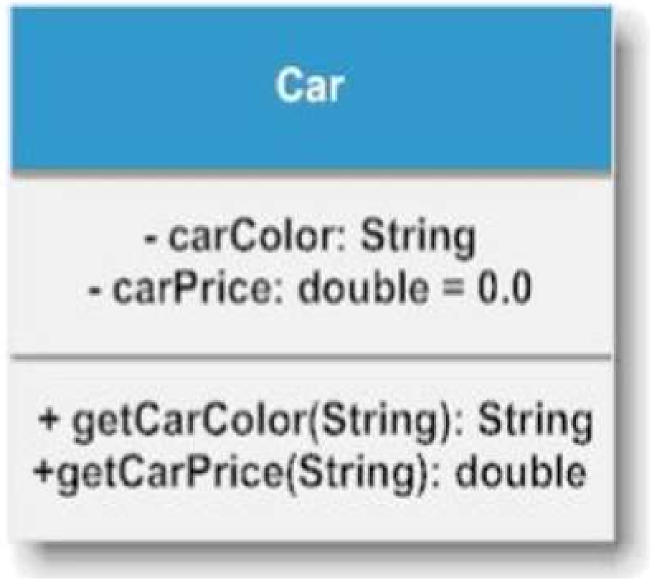
- Attributes and Methods or Operations are *features* of a class
- Features have the visibility adornments
  - + public
    - Accessible by any client of the class
  - # protected
    - Accessible only from within the same class or subclasses
  - - private
    - Accessible only from within the same class



Rather than using these symbols, tools such as Rhapsody use more graphical ones which are easier to understand

Java/UML Example

```
public class Car {
   private String carColor;
   private double carPrice = 0.0;
   public String getCarColor(String model) {
      return carColor;
   }
  public double getCarPrice(String model) {
     return carPrice;
  }
}
```



Outline

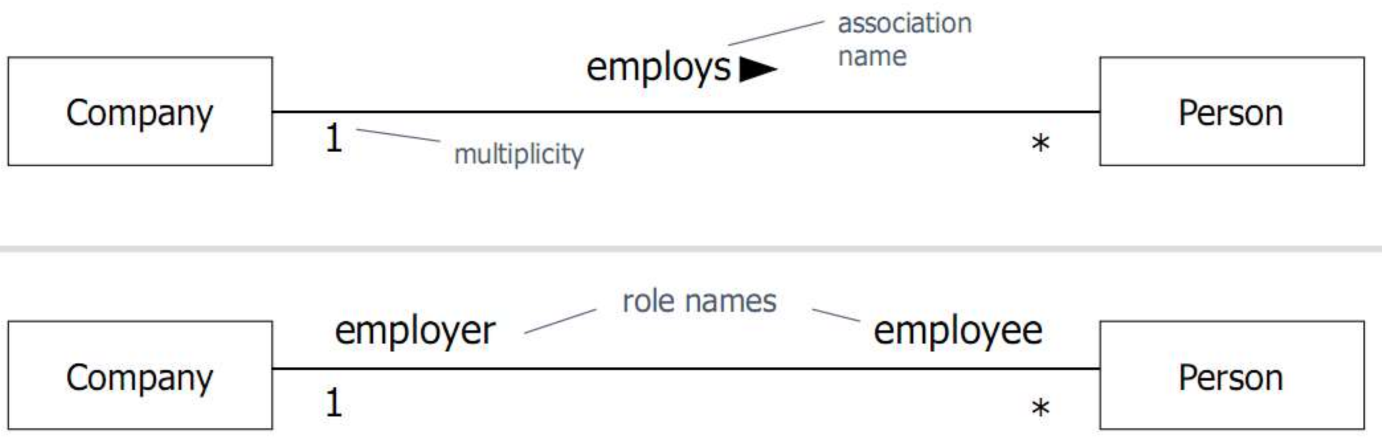
- Objects
- Classes
- Objects and Class Relationships
- Inheritance and Polymorphism
- Aggregation and Composition
- Interface Realization
- Domain Model

What is a Relationship?

- A *relationship* is a connection between modeling elements
- In this section we'll look at:
  - *Associations* between classes
    - aggregation
    - composition
    - association classes
  - *Links* between objects

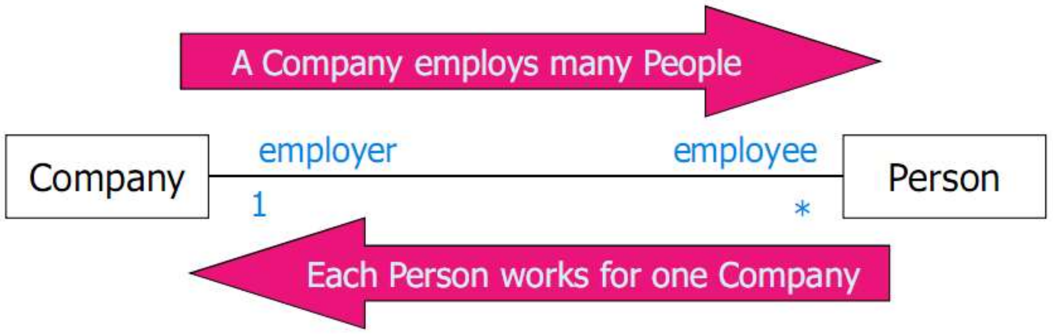
What is an Association?

- *Associations are relationships between classes*



- An association can have an association name *or* *a* role names. It's bad style to have both. The black triangle indicates the direction in which the association name is read: "Company employs many Person(s)"

Multiplicity

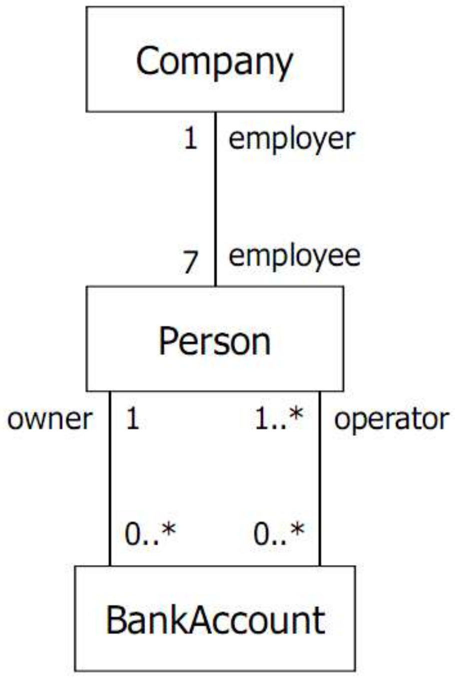

A Company employs many People
Company
employer
1
employee
*
Person
Each Person works for one Company

- Multiplicity is a constraint that specifies the number of objects that can participate in a relationship at *any point in time*
- If multiplicity is not explicitly stated in the model then it is undecided – *there is no default multiplicity*

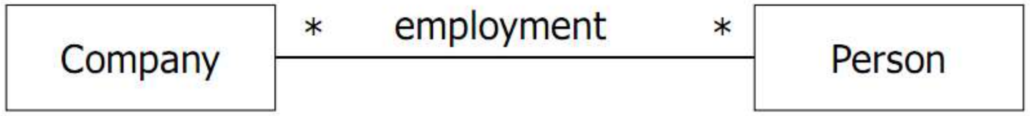
[illegible]	
0..1	zero or 1
1	exactly 1
0..*	zero or more
*	zero or more
1..*	1 or more
1..6	1 to 6

Multiplicity exercise

- How many
  - Employees can a Company have?
  - Employers can a Person have?
  - Owners can a BankAccount have?
  - Operators can a BankAccount have?
  - BankAccounts can a Person have?
  - BankAccounts can a Person operate?



Association Classes

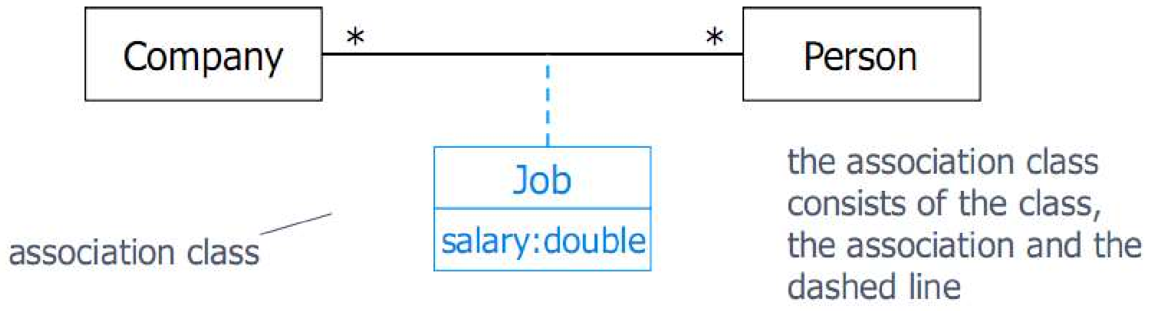


Each Person object can work for many Company objects.
Each Company object can employ many Person objects.
When a Person object is employed by a Company object, the Person has a salary.

Where do we record the Person’s salary?

- Not on the Person class - there is a different salary for each employment
- Not on the Company class - different Person objects have different salaries
- The salary is a property of the *employment relationship*
  - every time a Person object is employed by a Company object, there is a salary

Association Class Syntax

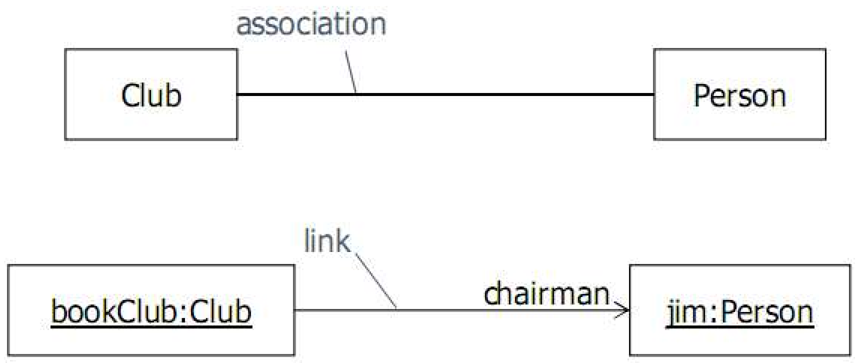

Company
*
*
Person
Job
salary:double
association class
the association class consists of the class, the association and the dashed line

- We model the association itself as an association class. One instance of this class exists for each link between a Person object and a Company object
  - Instances of the association class are links that have attributes and operations
  - Can only use association classes when there is *one unique link* between two specific objects. This is because the identity of links is determined exclusively by the identities of the objects on the ends of the link

What is a Link?

- Links are connections *between objects*
  - Think of a link as a telephone line connecting you and a friend. You can send messages back and forth using this link
- Links are the way that objects communicate
  - Objects send messages to each other via links
  - Messages invoke operations
- OO programming languages implement links as object references or pointers.
  - When an object has a reference to another object, we say that there is a *link* between the objects

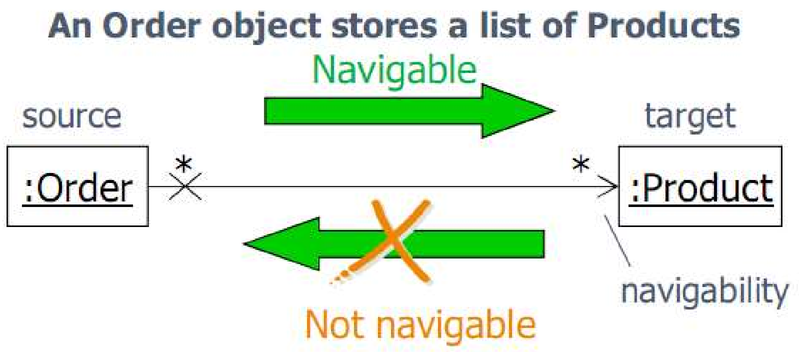
Links and Associations

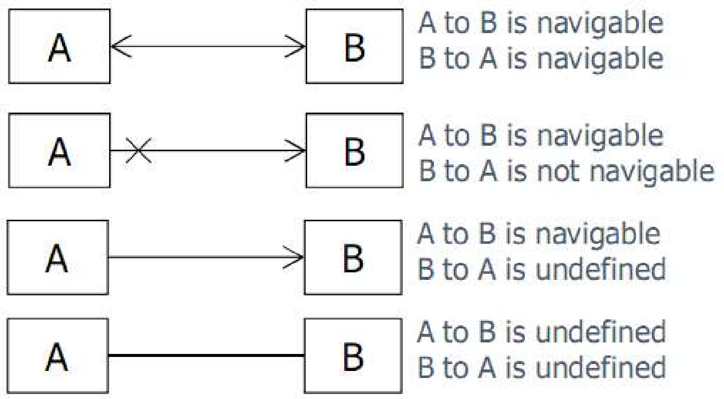


- *Associations are relationships between classes*
- Associations between classes indicate that there are links between objects of those classes
- *A link is an instantiation of an association just as an object is an instantiation of a class*

Navigability

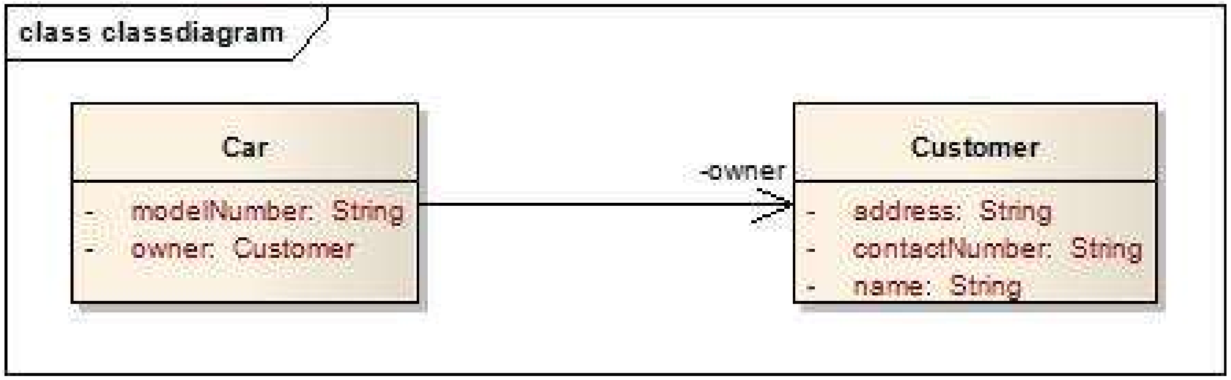
- Navigability indicates that it is possible to traverse from an object of the *source* class to objects of the *target* class
- Even if there is *no* navigability it might still be possible to traverse the relationship via some indirect means. However the computational cost of the traversal might be very high





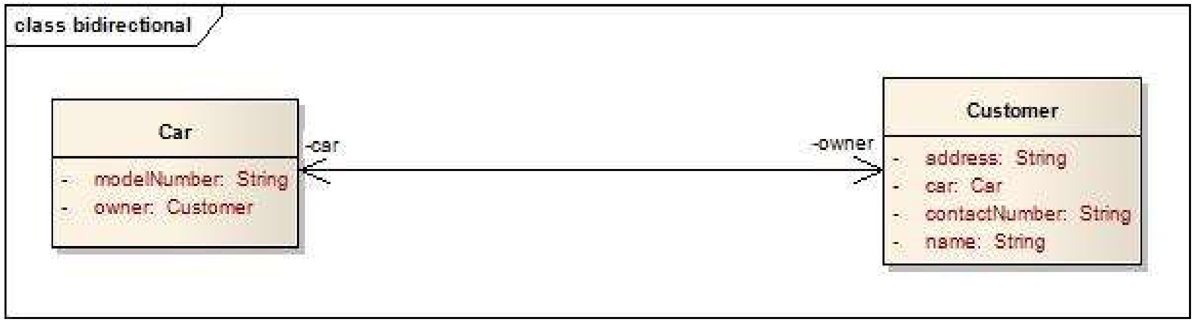
Association Example unidirectional relationship

```
public class Customer {
  private String name;
  private String address;
  private String contactNumber;
}
public class Car {
  private String modelNumber;
  private Customer owner;
}
```



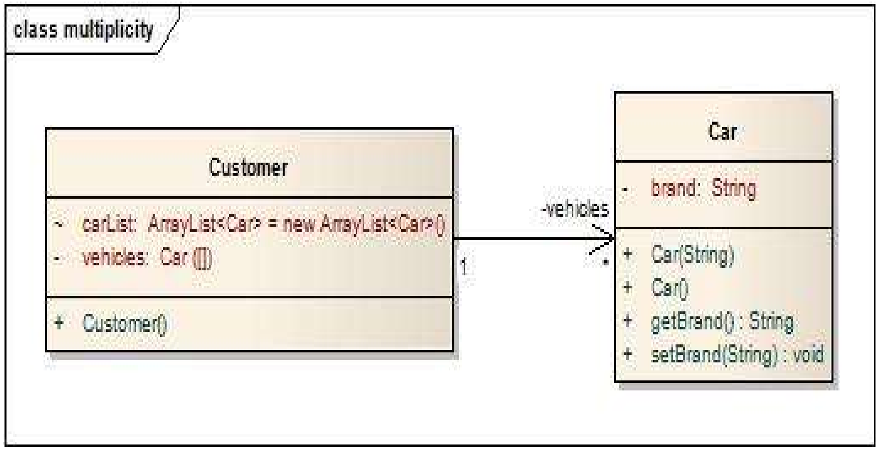
Association Example bidirectional relationship

```
public class Customer {
   private String name;
   private String address;
   private String contactNumber;
   private Car car;
}
public class Car {
   private String modelNumber;
   private Customer owner;
}
```



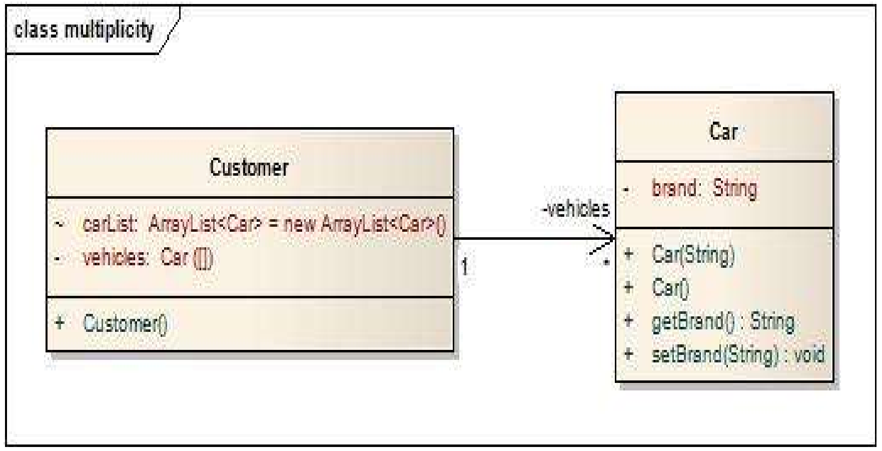
Association Example
Multiplicity in association (1)

```
public class Car {
  private String brand;
  public Car(String brands) {
    this.brand = brands;
  }
 public Car() {
 }
 public String getBrand() {
   return brand;
 }
 public void setBrand(String brand) {
   this.brand = brand;
 }
}
```



Association Example
Multiplicity in association (2)

```
public class Customer {
  private Car[] vehicles;
 ArrayList<Car> carList = new ArrayList<Car>();
 public Customer() {
   vehicles = new Car[2];
   vehicles[0] = new Car("Audi");
   vehicles[1] = new Car("Mercedes");
   carList.add(new Car("BMW"));
   carList.add(new Car("Chevy"));
 }
}
```



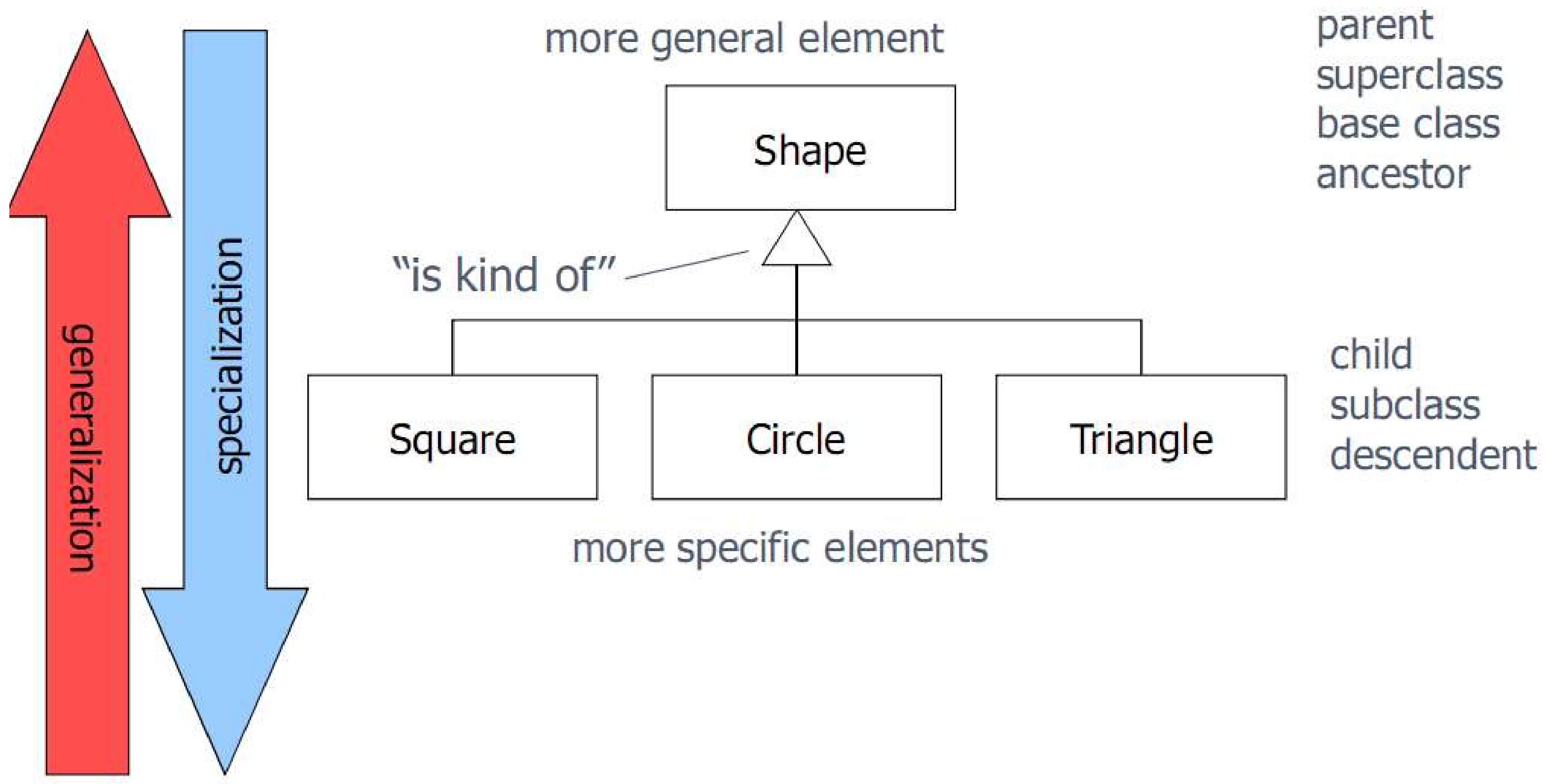
Outline

- Objects
- Classes
- Objects and Class Relationships
- Inheritance and Polymorphism
- Aggregation and Composition
- Interface Realization
- Domain Model

Generalization

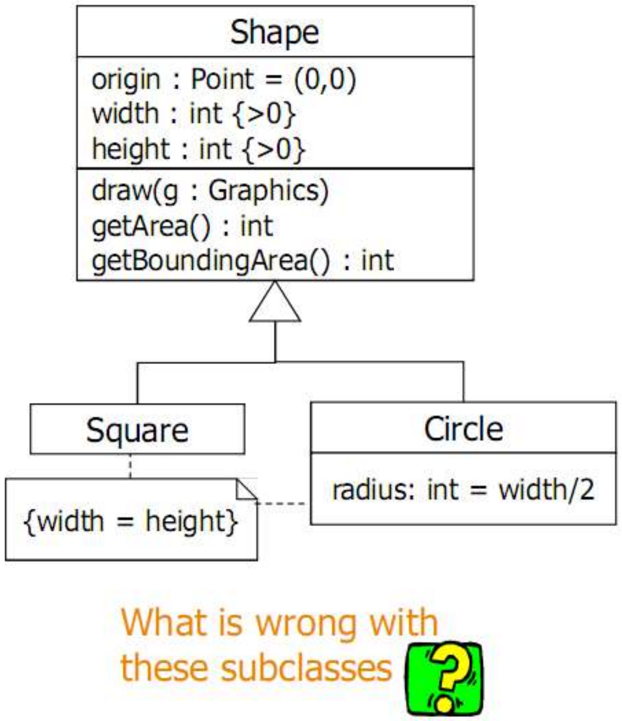
- A relationship between a more general element and a more specific element
- The more specific element is entirely consistent with the more general element but contains more information

Example: Class Generalization

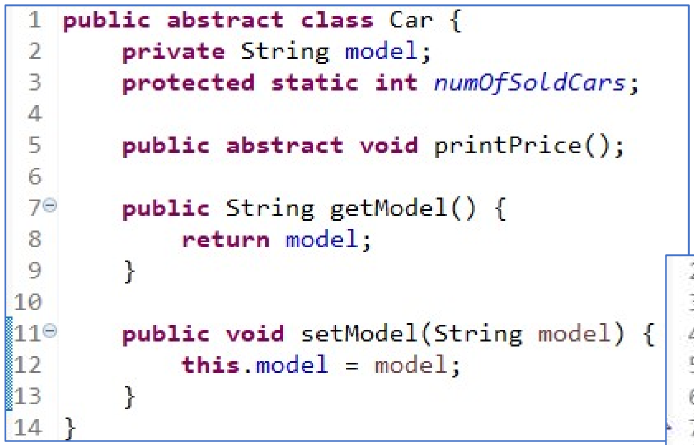


Class Inheritance

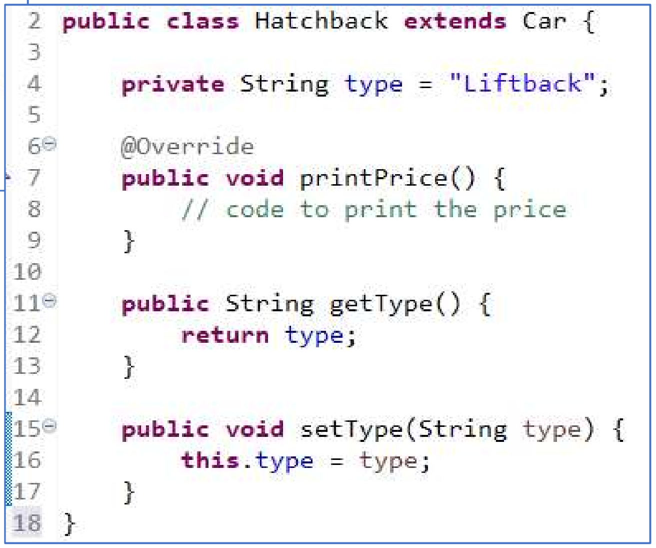
- Subclasses inherit *all* features of their superclasses:
  - attributes
  - operations
  - relationships
  - stereotypes, tags, constraints
- Subclasses can add new features
- Subclasses can override superclass operations
- We can use a subclass instance *anywhere* a superclass instance is expected



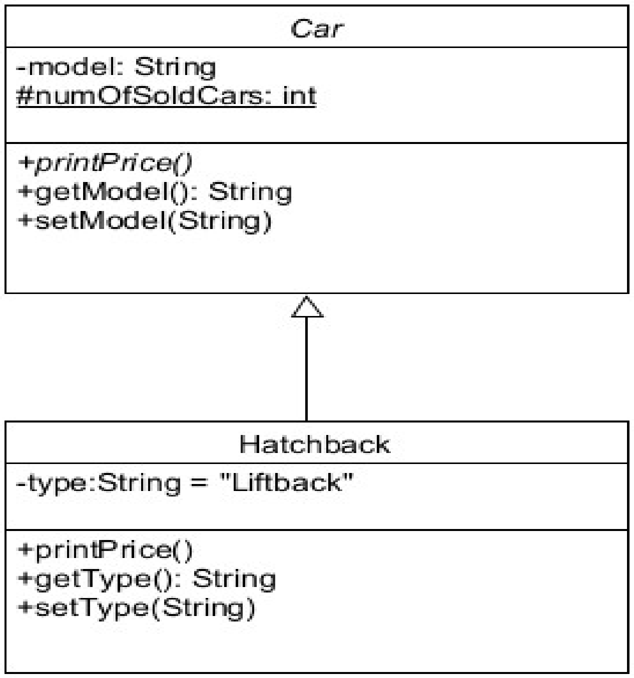
Class Inheritance (UML/Java) Example



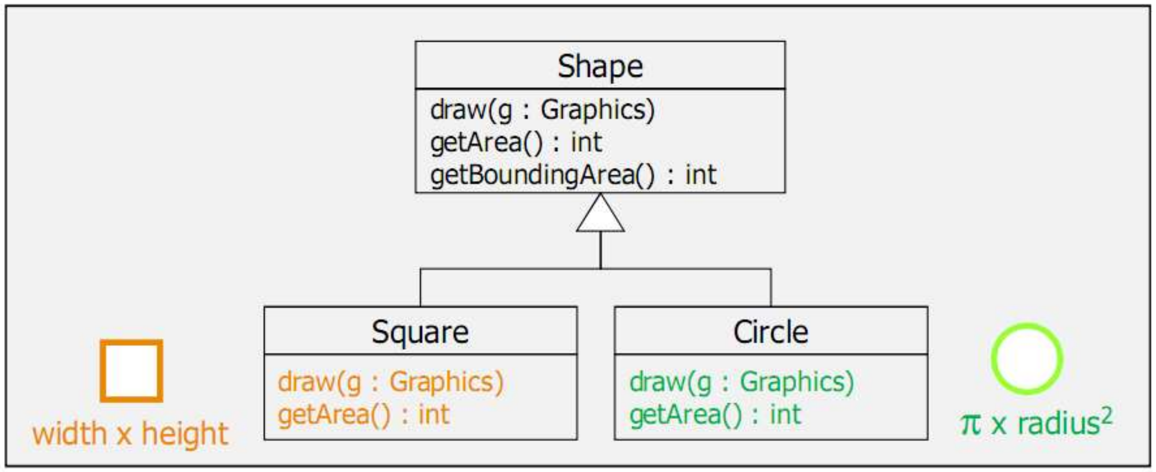
```java
public abstract class Car {
    private String model;
    protected static int numOfSoldCars;

    public abstract void printPrice();

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }
}
```



```java
public class Hatchback extends Car {

    private String type = "Liftback";

    @Override
    public void printPrice() {
        // code to print the price
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}
```

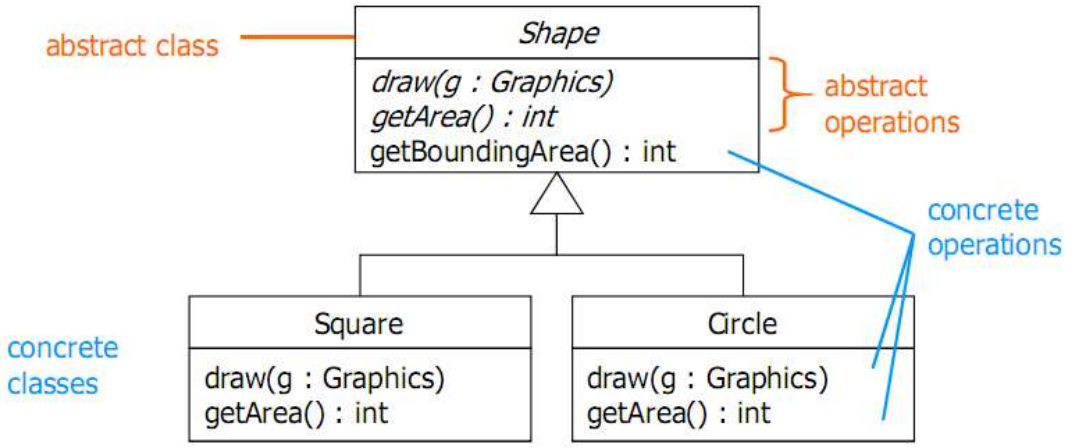


Overriding



- Subclasses often need to *override* superclass behaviour
- To override a superclass operation, a subclass must provide an operation with the same signature
  - The operation signature is the operation name, return type and types of all the parameters
  - The names of the parameters don't count as part of the signature

Abstract Operations & Classes



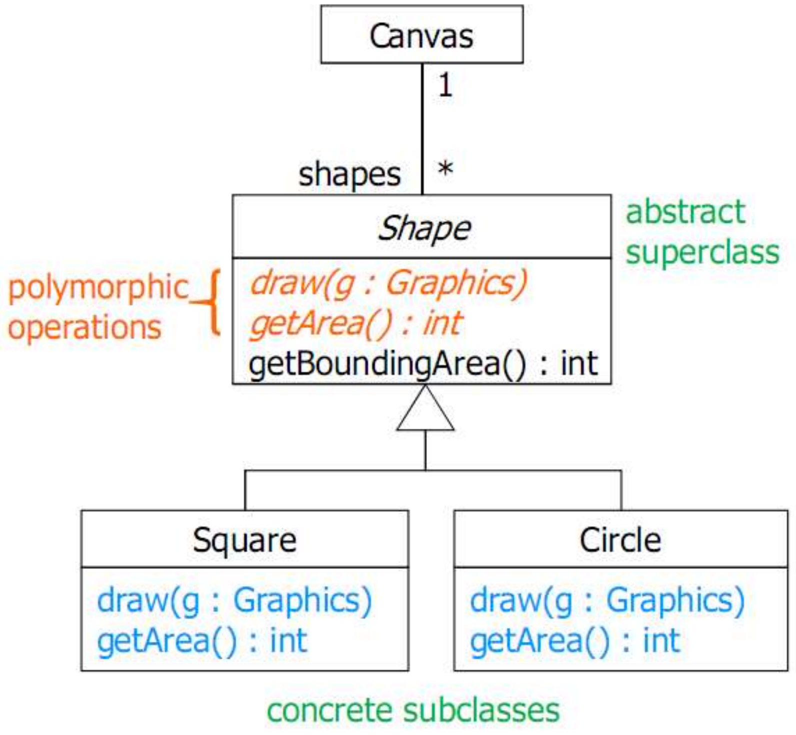
- We can't provide an implementation for *Shape :: draw(g : Graphics)* or for *Shape :: getArea() : int* because we don't know how to draw or calculate the area for a "shape"!
- Operations that lack an implementation are *abstract operations*
- A class with any abstract operations *cannot* be instantiated and is therefore an *abstract class*

Polymorphism

- Polymorphism = "many forms"
  - A polymorphic operation has many implementations
  - Square and Circle provide implementations for the polymorphic operations *Shape::draw()* and *Shape::getArea()*

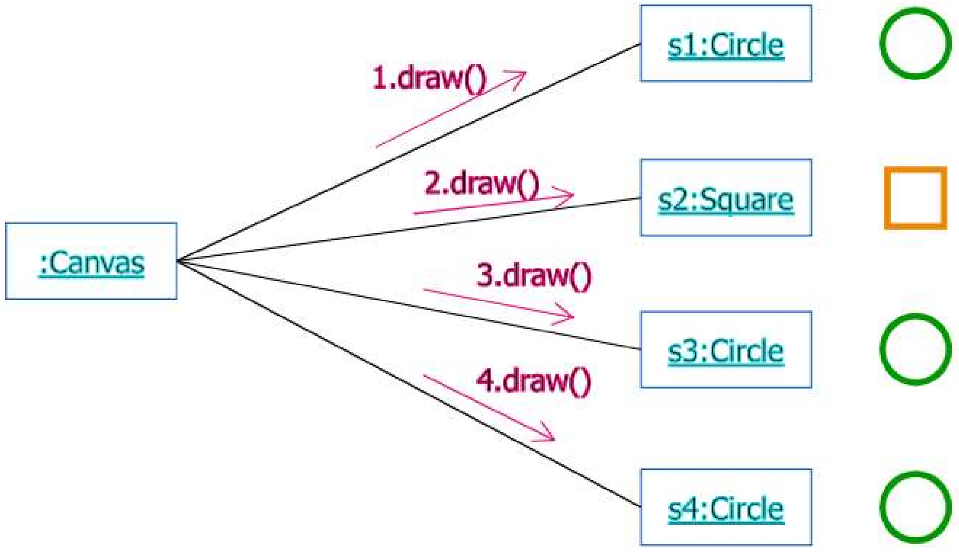
- All concrete subclasses of Shape *must* provide concrete draw() and getArea() operations because they are abstract in the superclass

A Canvas object has a collection of *Shape* objects where each *Shape* may be a Square or a Circle



What happens?

- Each class of object has its own implementation of the draw() operation
- On receipt of the draw() message, each object invokes the draw() operation specified by its class
- We can say that each object "decides" how to interpret the draw() message based on its class



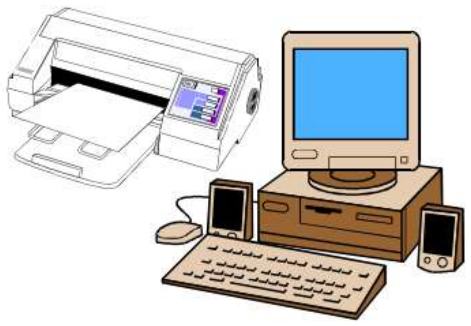
Outline

- Objects
- Classes
- Objects and Class Relationships
- Inheritance and Polymorphism
- Aggregation and Composition
- Interface Realization
- Domain Model

Aggregation and Composition

UML defines two types of associations:

Aggregation



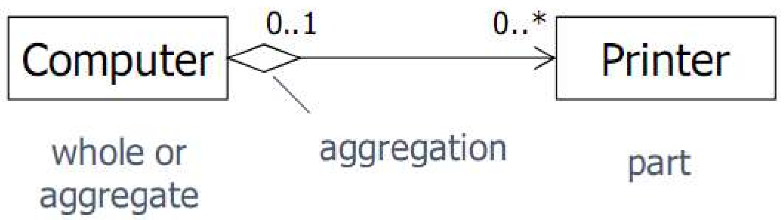
Some objects are weakly related like a computer and its peripherals

Composition

Some objects are strongly related like a tree and its leaves

Aggregation semantics

aggregation is a *whole–part* relationship



A Computer may be attached to 0 or more Printers

At any one point in time a Printer is connected to 0 or 1 Computer

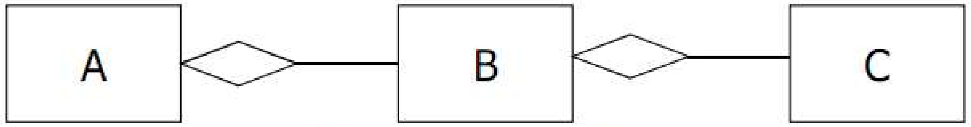
Over time, many Computers may use a given Printer

The Printer exists even if there are no Computers

The Printer is independent of the Computer

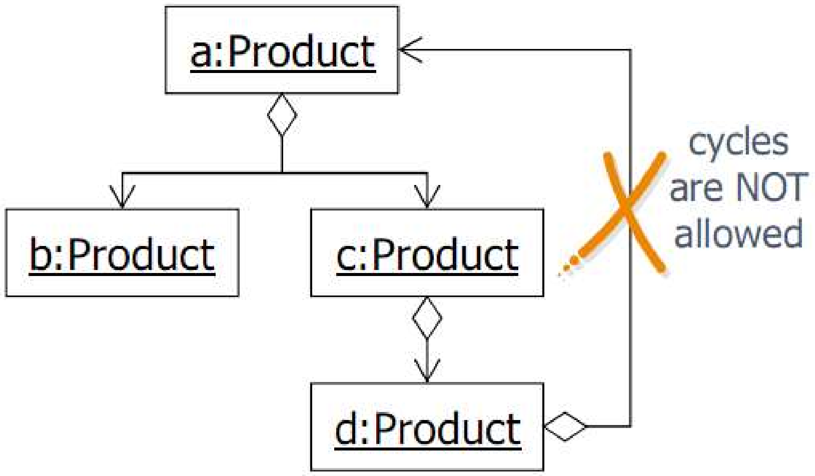
- The aggregate can sometimes exist independently of the parts, sometimes not
- The parts can exist independently of the aggregate
- The aggregate is in some way incomplete if some of the parts are missing

Transitive and asymmetric

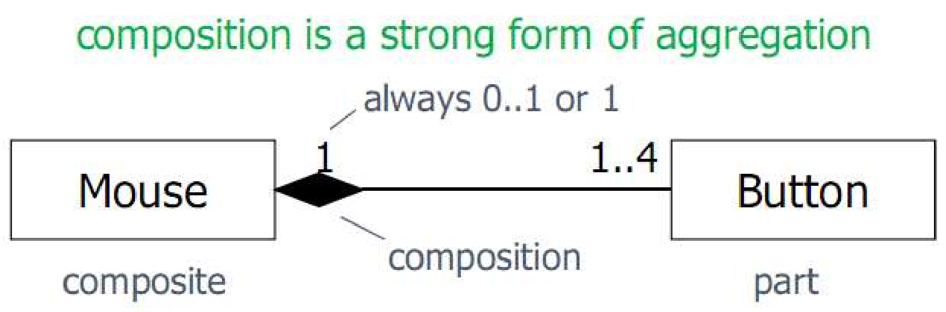


Aggregation (and composition) are *transitive*
If C is a part of B and B is a part of A, then C is a part of A

Aggregation (and composition) are *asymmetric*
An object can *never* be part of itself!



Composition semantics



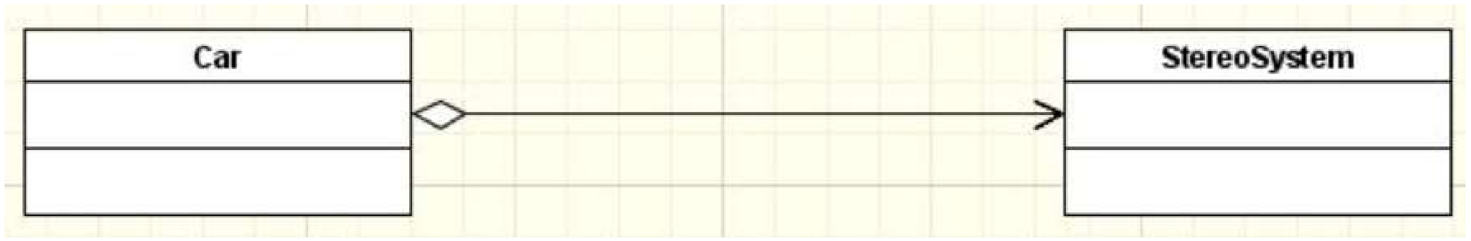
The buttons have no independent existence. If we destroy the mouse, we destroy the buttons. They are an integral part of the mouse

Each button can belong to exactly 1 mouse

- The parts belong to exactly 0 or 1 whole at a time
- The composite has sole responsibility for the disposition of all its parts. This means responsibility for their creation and destruction
- The composite may also release parts provided responsibility for them is assumed by another object
- If the composite is destroyed, it must either destroy all its parts, OR give responsibility for them over to some other object

Aggregation (Java/UML) Example

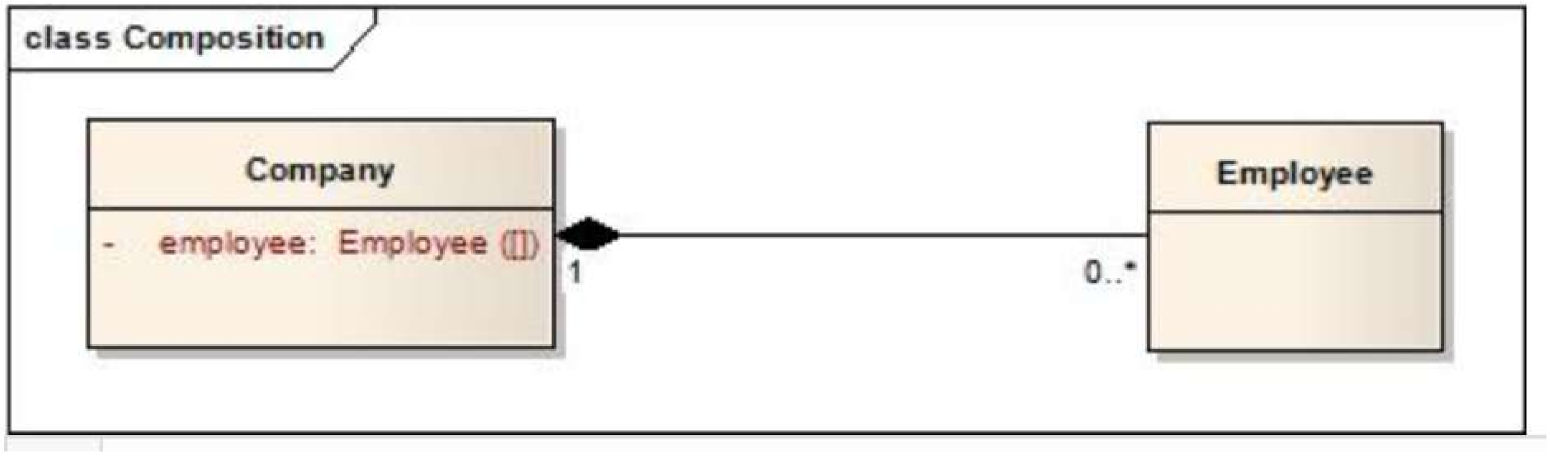
```
class StereoSystem {
    ....
}
class Car {
    private StereoSystem s ;
   ....
}
```



Composition (UML/Java) Example

the objects' lifecycles are tied. It means that if we destroy the owner object, its members also will be destroyed with it.

```
public class Employee {
}
public class Company {
    private Employee[] employee;
}
```

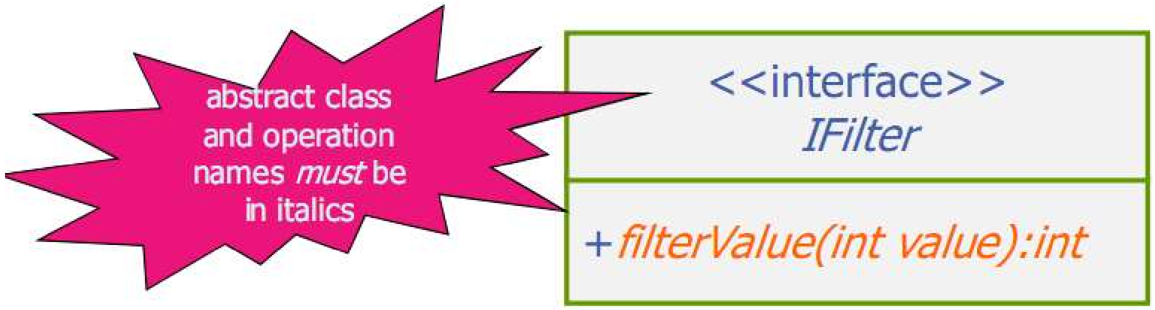


Outline

- Objects
- Classes
- Objects and Class Relationships
- Inheritance and Polymorphism
- Aggregation and Composition
- Interface Realization
- Domain Model

Abstract/Interface Classes

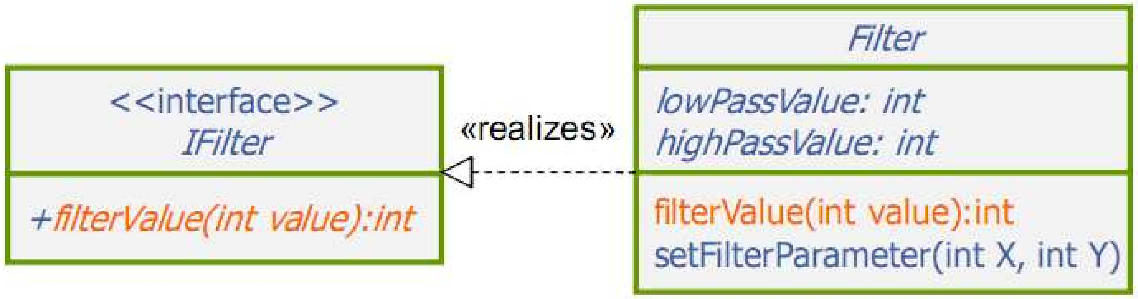
- An Interface is a specification of a named contract offered by a class.
- It primarily consists of a collection of operations
- All interface operations are *public* and *abstract*


abstract class and operation names *must* be in italics
<<interface>>
IFilter
+*filterValue(int value):int*

- Interfaces are not required, but allow to separate the specification of a set of services from their implementation

Interfaces

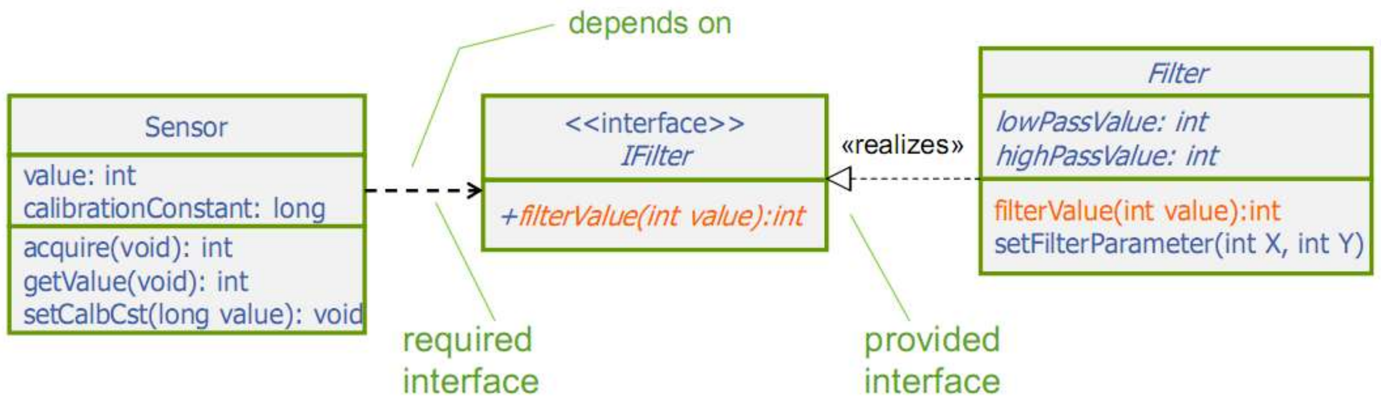
- A class is said to *realize* an interface if it provides a method for every operation specified in the interface
  - Classes may realize any number of interfaces
  - Interfaces may be realized by any number of classes



<<interface>>
IFilter
+filterValue(int value):int
«realizes»
Filter
lowPassValue: int
highPassValue: int
filterValue(int value):int
setFilterParameter(int X, int Y)

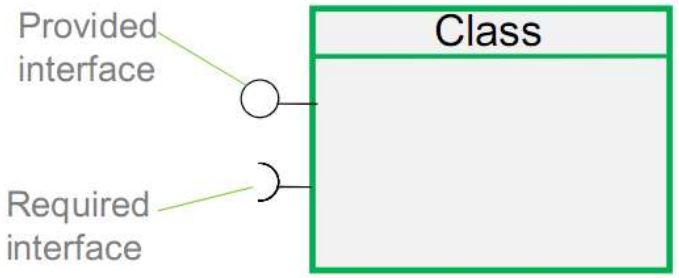
Interfaces

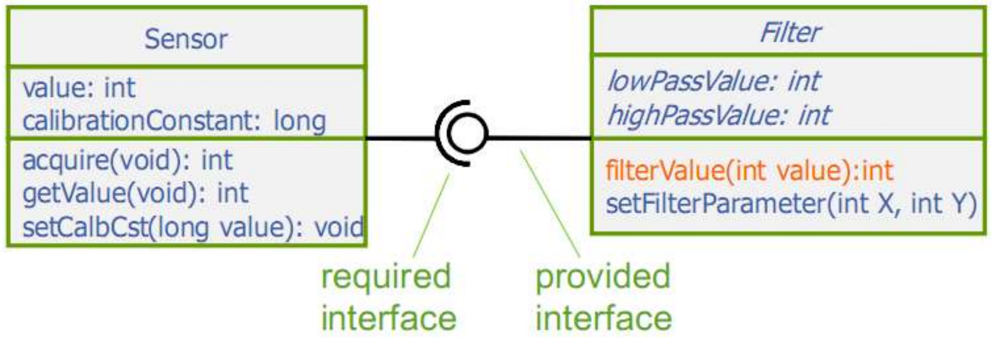
- An interface can be defined as either
  - Provided: the containing classifier supplies the operations defined by the named interface element
  - Required: the classifier is able to communicate with some other classifier which provides operations defined by the named interface element



Composite Structures - Interfaces

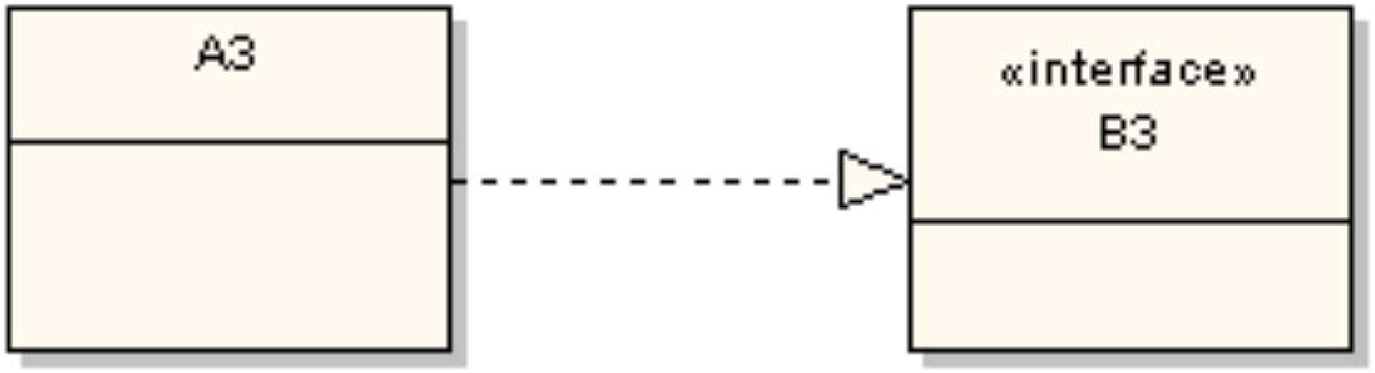
Ball and Socket Notation (lollipop notation)





Realization (UML/Java) Example

```
public class A3 implements B3 {
    // . . .
}
```



Outline

- Objects
- Classes
- Objects and Class Relationships
- Inheritance and Polymorphism
- Aggregation and Composition
- Interface Realization
- Domain Model

Domain Model

- A domain model is a visual representation of conceptual classes or real-situation objects in a domain, *not* of software components (e.g., Java or C++ classes).
- A domain model is illustrated with a set of class diagrams in which no operations are defined.
- It may show
  - Domain objects or conceptual classes
  - Associations between conceptual classes
  - Attributes of conceptual classes

How to Create a Domain Model?

Guideline:

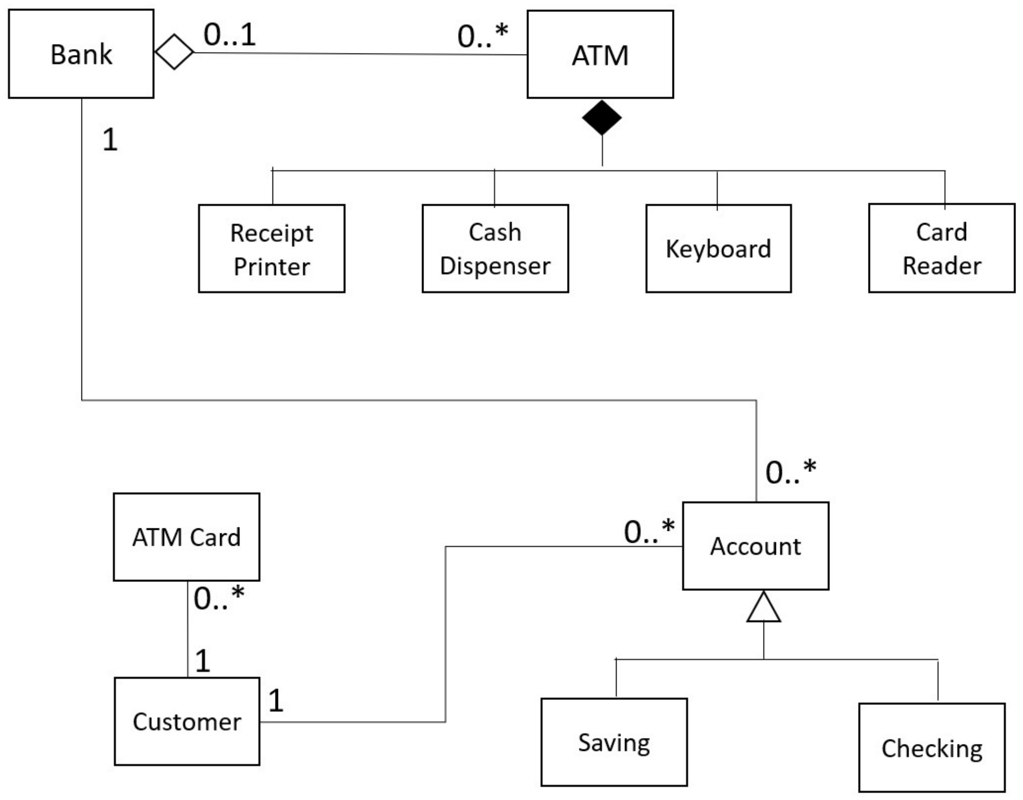
1. Find the conceptual classes
2. Draw them as classes in a UML class diagram
3. Add associations and attributes

Exercise - ATM

A bank has several ATMs which are geographically distributed and connected via a wide area network to a central server.

An ATM machine has a card reader, a cash dispenser, a keyboard/display, and a receipt printer. By using the ATM machine, a customer can withdraw cash from either a checking or savings account, query the balance of an account, or transfer funds from one account to another. A transaction is initiated when a customer inserts an ATM card into the card reader . Encoded on the magnetic strip on the back of the ATM card are the card number, the start date, and the expiration date. Assuming the card is recognized, the system validates the ATM card to determine that the expiration date has not passed, that the user-entered PIN matches the PIN maintained by the system, and that the card is not lost or stolen. The customer is allowed three attempts to enter the correct PIN; the card is confiscated if the third attempt fails. Cards that have been reported lost or stolen are also confiscated.

Exercise - ATM



Bank
0..1
0..*
ATM
1
Receipt Printer
Cash Dispenser
Keyboard
Card Reader
0..*
ATM Card
0..*
0..*
Account
1
1
Customer
Saving
Checking