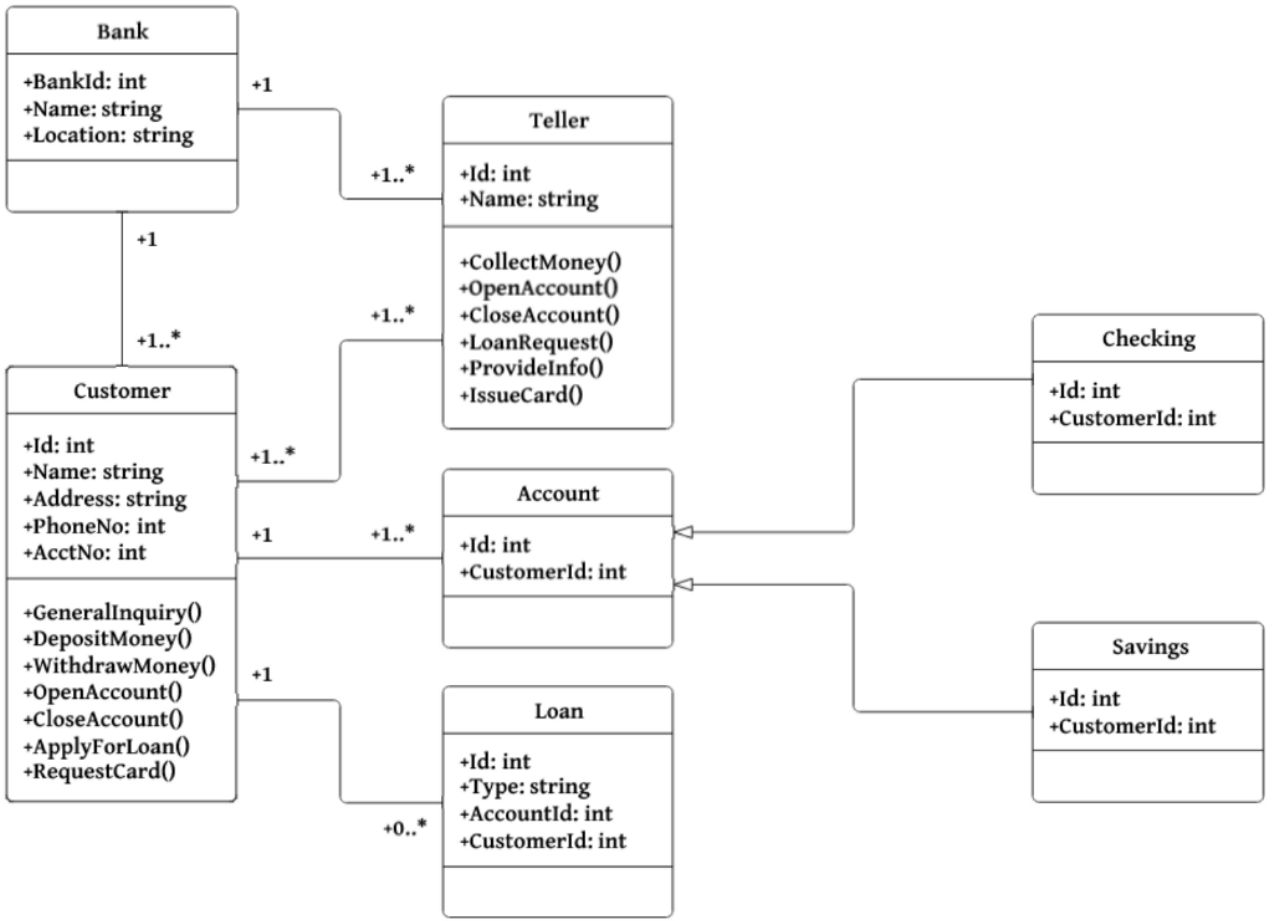
What is a Class Diagram?

- Suppose you have to design a system. Before implementing a bunch of classes, you'll want to have a conceptual understanding of the system
  - What classes do I need?
  - What functionality and information will these classes have?
  - How do they interact with one another?
  - Who can see these classes?
- That's where class diagrams come in. Class diagrams are a neat way of visualizing the classes in your system *before* you actually start coding them up. They're a static representation of your system structure.

Example of a Class Diagram for a Banking System



Why Do We Need Class Diagrams?

- This is a fairly simple diagram.
- However, as your system scales and grows, it becomes increasingly difficult to keep track of all these relationships.
- Having a precise, organized, and straight-forward diagram to do that for you is integral to the success of your system.

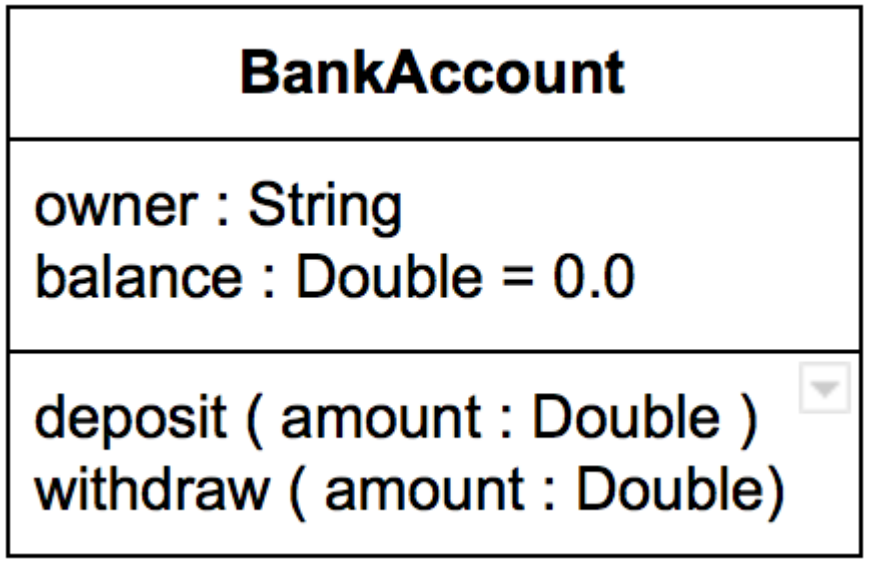
DESIGN

- Planning and modeling ahead of time makes programming much easier.
- Besides that, making changes to class diagrams is easy, whereas coding different functionality after the fact is kind of annoying.
- When someone wants to build a house, they don’t just grab a hammer and get to work. They need to have a blueprint — a design plan — so they can ANALYZE & modify their system.
- You don’t need much technical/language-specific knowledge to understand it.

SOME TECHNICAL STUFF

Class Representation in UML

- A class is represented as a box with 3 compartments.
  - The uppermost one contains the class name.
  - The middle one contains the class attributes
  - The last one contains the class methods.



They Adhere to a Convention:

- attribute name : type
- method name (parameter: type): type
- if you'd like to set a default value to an attribute do as above balance : Dollars = 0
- if a method doesn't take any parameters then leave the parentheses empty. Ex: checkBalance()

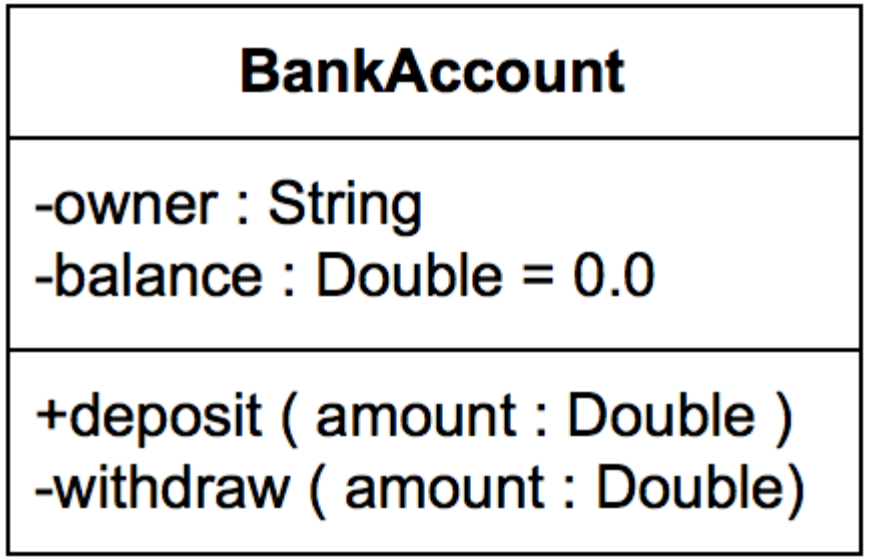
Visibility of Class Members

- Class members (attributes and methods) have a specific visibility assigned to them. See table below for how to represent them in UML.

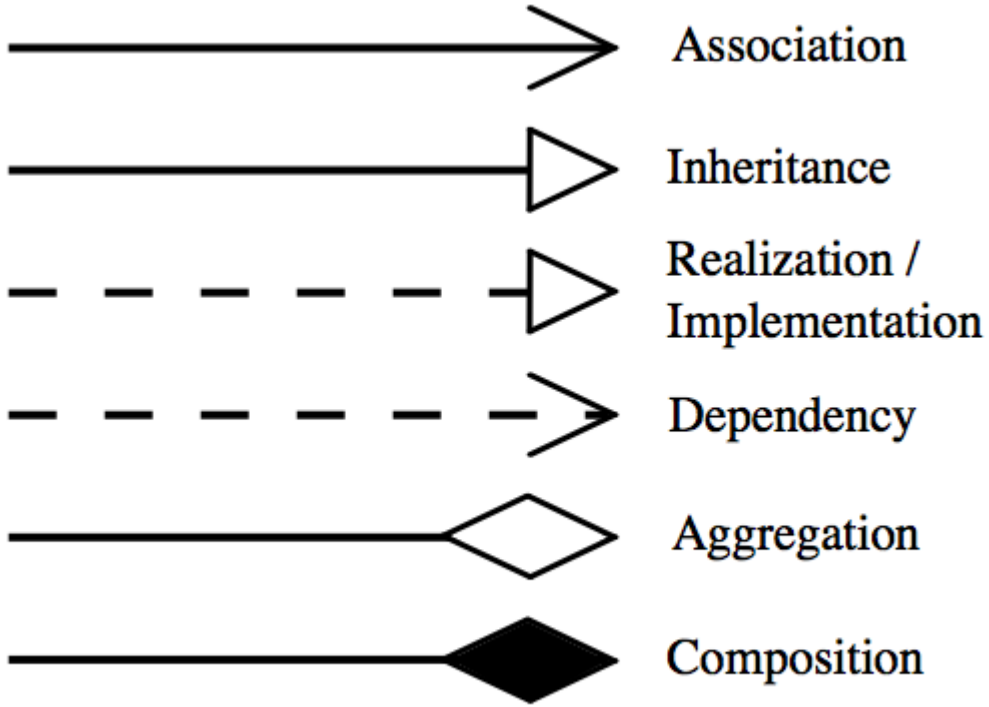
public	+	anywhere in the program and may be called by any object within the system
private	-	the class that defines it
protected	#	(a) the class that defines it or (b) a subclass of that class

Visibility of Members in the BankAccount class

- We made the `owner` and balance private as well as the withdraw method.
- We kept the deposit method public. (Anyone can put money in, but not everyone can take money out. Just as we like it.)



Relationships

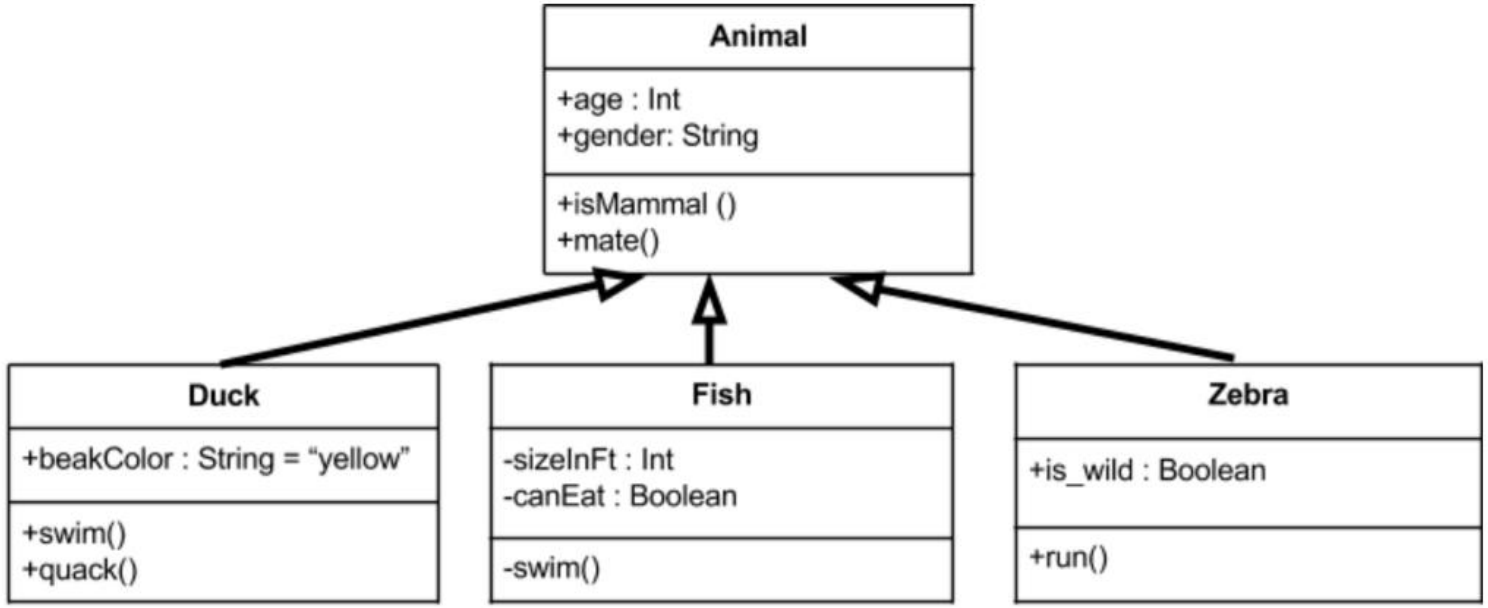


Association

- An association is a relationship between two separate classes. It joins two entirely separate entities.
- There are four different types of association:
  - Bi-directional
  - Uni-directional
  - Aggregation (includes composition aggregation)
  - Reflexive.
- Bi-directional and uni-directional associations are the most common ones.
- This can be specified using multiplicity (one to one, one to many, many to many, etc.).
- A typical implementation in Java is through the use of an instance field. The relationship can be bi-directional with each class holding a reference to the other.

Inheritance

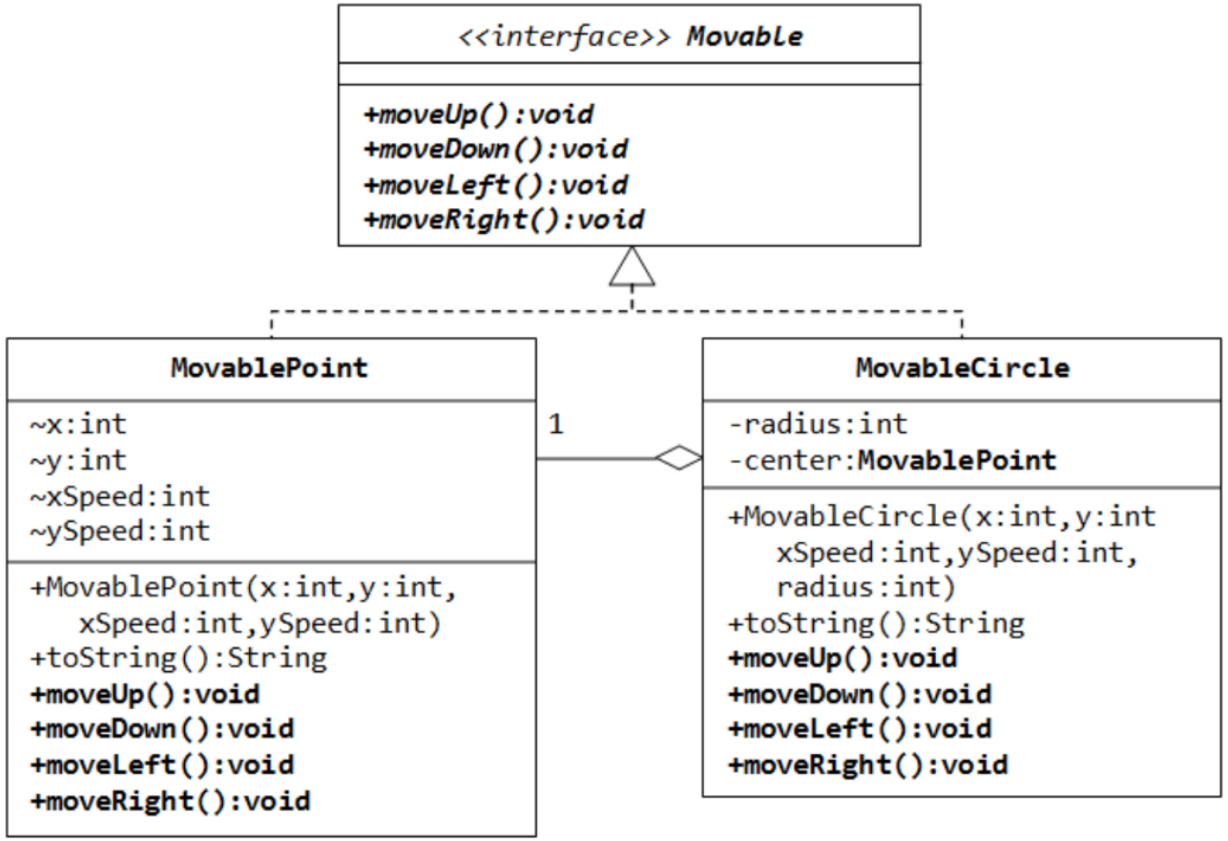
- Indicates that child (subclass) is considered to be a specialized form of the parent (super class).
- For example consider the following:



- Above we have an animal parent class with all public member fields.
- You can see the arrows originating from the duck, fish, and zebra child classes which indicate they inherit all the members from the animal class.
- Not only that, but they also implement their own unique member fields.
- You can see that the duck class has a swim() method as well as a quack() method.

Interface

- A relationship between two model elements, in which one model element implements/executes the behavior that the other model element specifies.

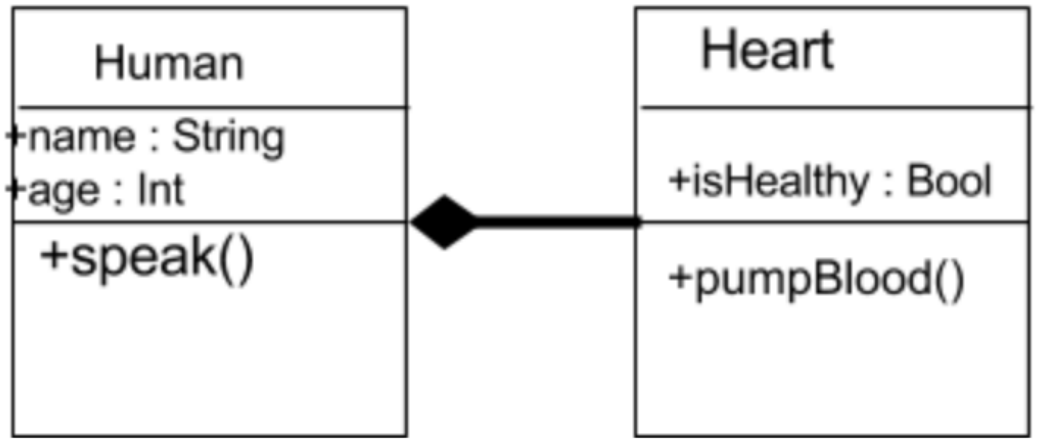


Dependency - Aggregation

- A special form of association which is a unidirectional (a.k.a one way) relationship between classes.
- The best way to understand this relationship is to call it a “has a” or “is part of” relationship.
- For example, consider the two classes: Wallet and Money.
- A wallet “has” money. But money doesn’t necessarily need to have a wallet so it’s a one directional relationship.

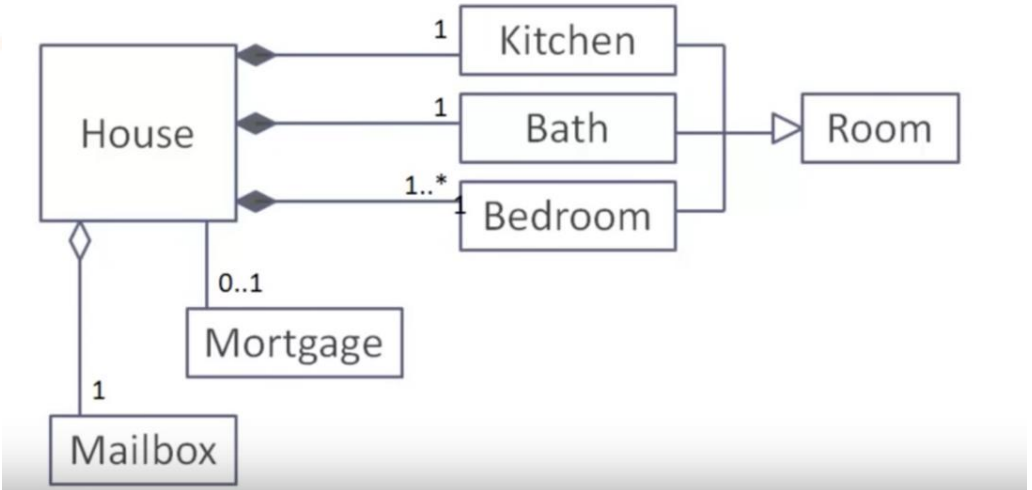
Composition

- A restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other.
- A human needs a heart to live and a heart needs a human body to function on. In other words when the classes (entities) are dependent on each other and their life span are same (if one dies then another one does too) then it’s a composition.

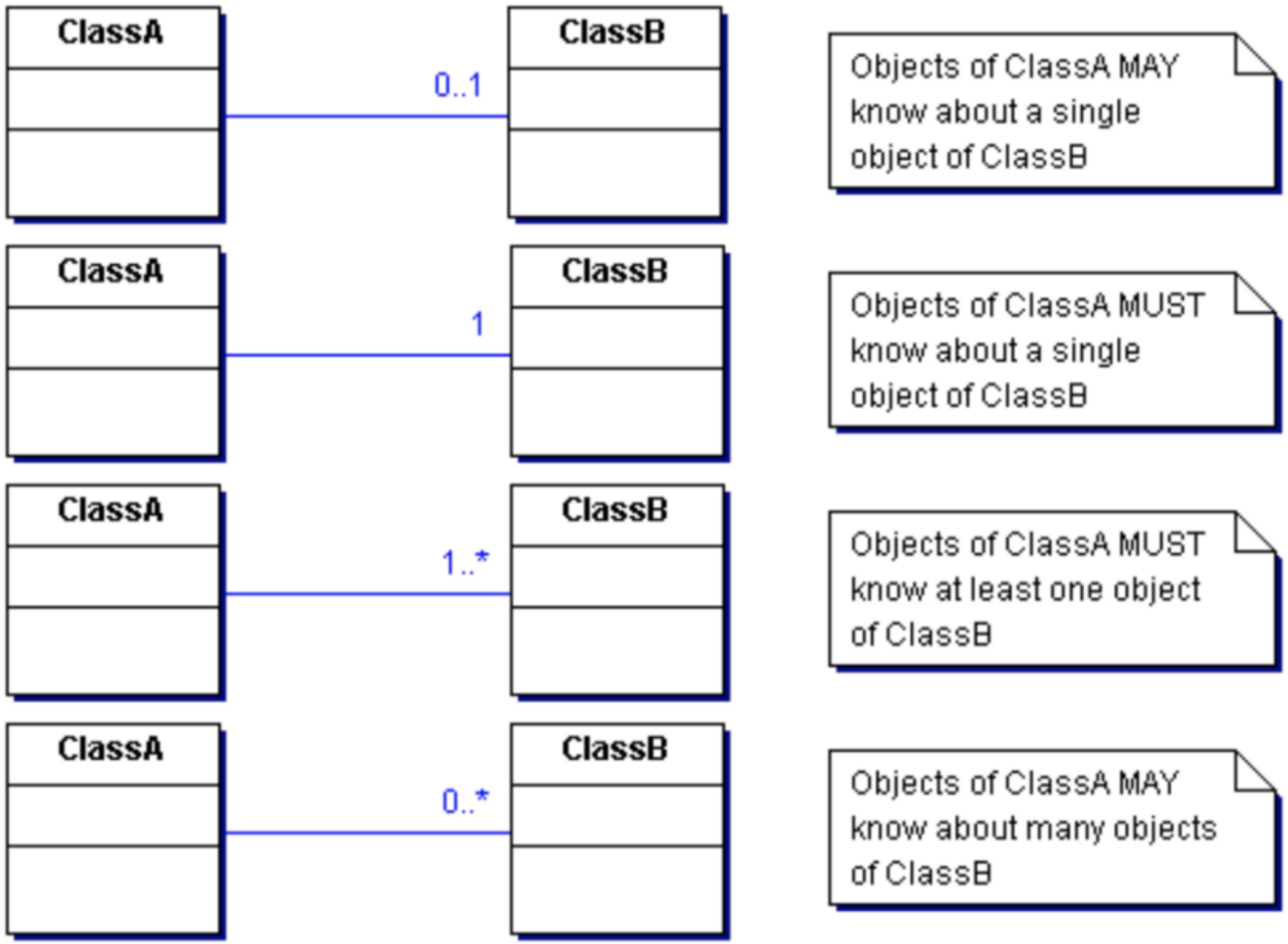


Multiplicity

- After specifying the type of association relationship by connecting the classes, you can also declare the cardinality between the associated entities. For example:



- The above UML diagram shows that a house has exactly one kitchen, exactly one bath, at least one bedroom (can have many), exactly one mailbox, and at most one mortgage (zero or one).


ClassA
ClassB
0..1
Objects of ClassA MAY know about a single object of ClassB
ClassA
ClassB
1
Objects of ClassA MUST know about a single object of ClassB
ClassA
ClassB
1..*
Objects of ClassA MUST know at least one object of ClassB
ClassA
ClassB
0..*
Objects of ClassA MAY know about many objects of ClassB

UML Class Diagram Example

- Now, let's take what we've learned in the previous tutorial and apply it.
- In this example we are asked to create a class diagram for a banking system. It must have the following classes:
  - Bank
  - ATM
  - Customer
  - Account
  - Transaction
  - Checking Account
  - Savings Account

Determine Possible Class Members Bank

- The bank class represents a physical bank.
- It has a location and a unique id.
- This bank also manages several accounts. **There's an association!**
  - What type of association is this?
  - Is a bank entirely composed of accounts (composition)?
  - Or are accounts 'part of' a bank (aggregation)?
  - It looks like aggregation.
  - It can't be composition because that would mean that both classes live and die together.
  - That's not quite right because you can have a bank without accounts and you can have accounts without a bank.
- We'll add a method called getAccounts().

ATM

- The ATM class represents a physical ATM.
- Right off the bat, we can come up with three methods for the ATM:
  - withdraw()
  - deposit()
  - checkBalance()
- Each of these methods takes the card number as input.
- In terms of attributes, an ATM has a location and is managed by a specific bank.

Customer

- The customer class represents a real customer.
- This customer has a
  - name
  - address
  - date of birth (dob)
  - card number
  - pin
- For this person to be considered a customer, they must have an account. **There's another association!**
  - This isn't aggregation or composition, it's just a bi-directional association (drawn using a blank line no arrows).

Account

- The account class represents a bank account.
- Common attributes of bank accounts include
  - account number
  - balance
  - And more
- You can deposit() withdraw() money from the account.
- In addition, banks might offer two types of accounts:
  - A checking account
  - A savings account.
  - These two can thus be considered child classes of the account class and can inherit from it too.
  - We'll denote this by using a solid black line with an unfilled arrow going into the account class.

Completed Diagram