UML Class Diagram

Unified Modeling Language

❑ A standard notation for describing *software models and code*

❑ Unifies the notation of Booch, OMT (Rumbaugh et al), and OOSE (Jacobson et al)

Many Kinds of UML Diagrams

UML has 20+ different kinds of diagrams.

Each diagram shows a different kind of information (or different *view*) of application.

- Class diagram
- Sequence diagram
- State Machine diagram (*aka State Chart Diagram*)
- Object diagram
- Interaction diagram
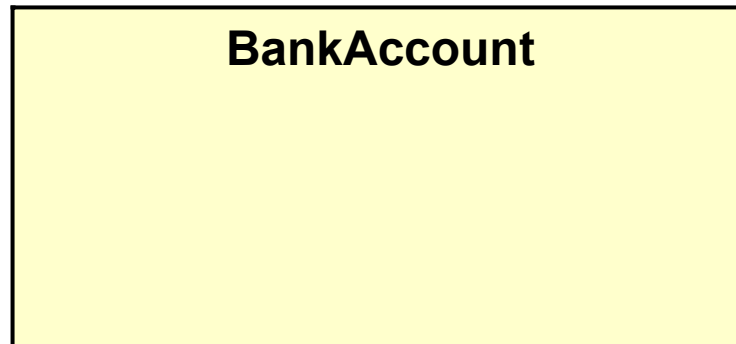- Activity diagram
- Package Diagram
- many others!

These 3 are the most common and most important to know.

Class Diagram

❑ A class diagram shows the structure of a class

❑ It can also show relationships between classes

Here is the *simplest possible class diagram*:

BankAccount

Class Diagrams methods & attributes

BankAccount
deposit(amount   )
withdraw(amount)
getBalance( )

BankAccount
balance
owner
id
deposit(amount)
withdraw(amount)
getBalance( )

Class Diagram with data types

- Class diagram can show data types & visibility

- <u>Not</u> Java notation ("double balance")

BankAccount
balance: double accountId: string
deposit(amount: double): void withdraw(amount: double): boolean getBalance( ): double

Visibility of Members

+ Public. Any code can access

Protected. Only this class and subclasses can access

~ Package. Only classes in same package

– Private. Only this class can access.

BankAccount
-balance: double #accountId: string
+deposit(amount: double): void +withdraw(amount: double): boolean +getBalance( ): double

Visibility Prefixes

+   means **public**

- **Visible everywhere**

–   means **private**

- **Visible only in the class in which it is defined**

\#   means **protected**

- **Visible either within the class in which it is defined or within subclasses of that class**

~   means **package** or **default** visibility

- **visible to other classes in the same package**

Notation for Constructors

BankAccount
-balance: double
<<constructor>> **+BankAccount(owner**) +deposit(amount) . . .

BankAccount
-balance: double
+BankAccount(owner) +deposit(amount) . . .

Static Members

❑ Use <u>underscore</u> to show static (class) attributes or methods.

Example: BankAccount has a static `nextAccountId` attribute.

BankAccount
-**<u>nextAccountId</u>**: long
-balance: double
-id: long
+BankAccount(owner)
+getBalance( ): double
. . .

private static
attribute

Practice: Draw the class diagram

```python
class Student:
    ID_PATTERN = "[1-9]\\d{9}" # regular expression
    def __init__(self, id, name):
        if not re.fullmatch(ID_PATTERN, id):
            raise ValueError("Invalid student id")
        self.id = id
        self.name = name


    def get_name(self):
        return self.name


    @classmethod
    def get_year(cls, id: int):
        # this only works for 10-digit KU ids
        return 2500 + (self.id//100000000)
```
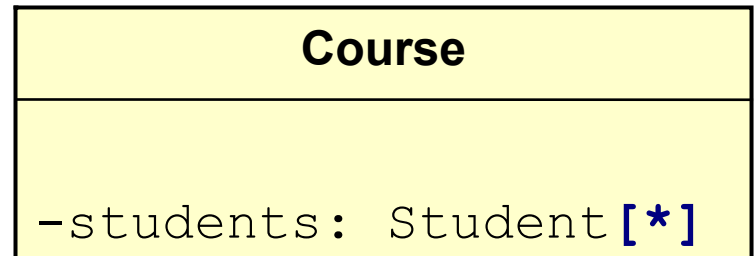
Showing Multiplicity in UML

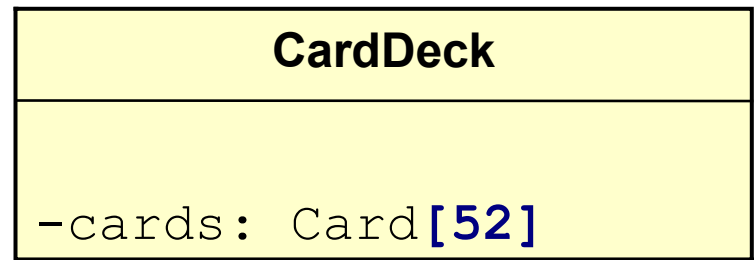`var: Type[*]` means var is a *collection*, including array.

A Course has zero or more students.

```
class Course:
    students: list
```

Course
-students: Student**[*]**

A deck of cards has *exactly* 52 cards.

```
class CardDeck {
  private Card[] cards =
          new Card[52];
```

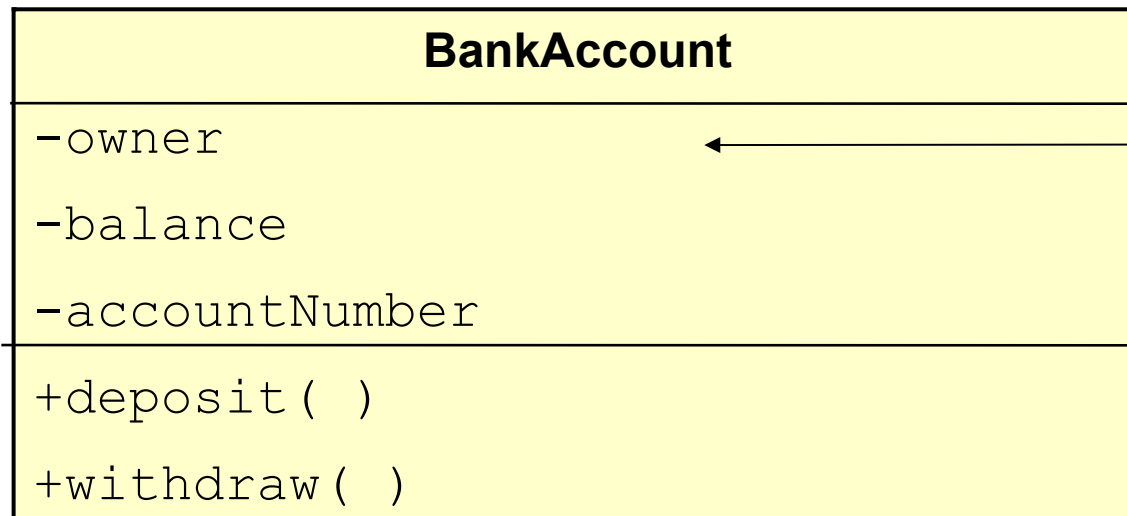| CardDeck |
| --- |
| |
| -cards: Card**[52]** |

How Much Detail to Show?

Purpose of UML is *communication & understanding*.

OK to omit routine, boring methods: __str__, getX( ),

OK to omit "id" attribute used only for persistence.

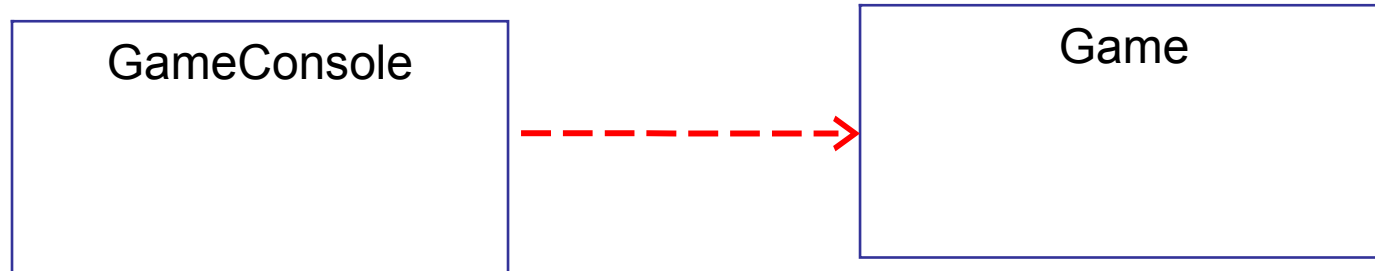In design phase, omit data types and (maybe) params.

BankAccount
-owner
-balance
-accountNumber
+deposit( )
+withdraw( )

id attribute not shown

Showing Relationships in UML

Class Diagram with more than one class

Dependency

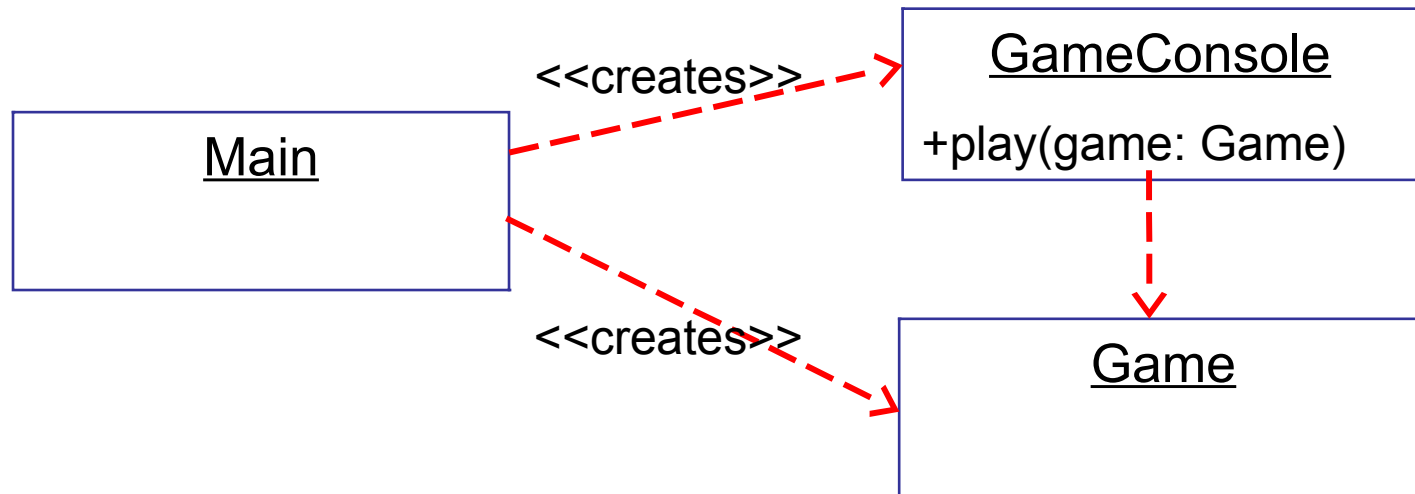- One class <u>uses</u> or depends on another class.

- Includes "association".

GameConsole		Game

GameConsole - - - - - - -> Game

```
class GameConsole:
    # the play method depends on Game.
    def play(game: Game):
        (width,height) = game.get_size()
```

More Dependency

- Main depends on (uses) Game and GameConsole


```
class Main:

    @classmethod
    def run(cls):
        game = Game(600,800)
        ui = GameConsole()
        ui.play( game )
```

Dependency Example

A Student *uses* the Registrar to enroll in a Course, but he doesn't save a reference (association) to the Registrar.

```
class Student:
    # NO Registrar attribute!

    def add_course(course: Course):
        registrar = Registrar.getInstance()
        registrar.enroll(this, course)
```

Student - - - - - - - - - -> Registrar
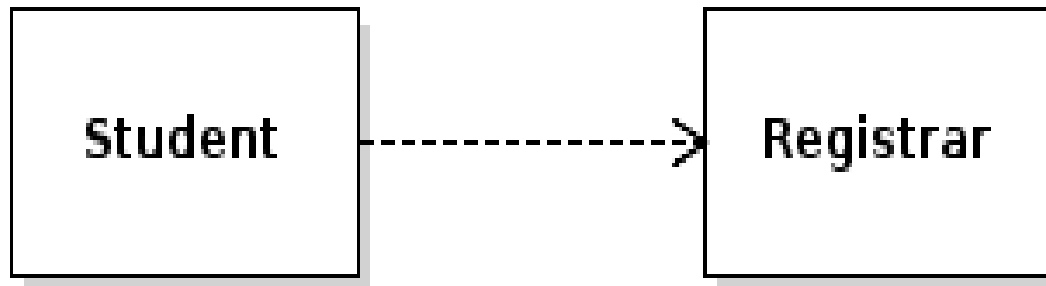
Association

☐ Association means one object <u>has</u> an attribute of another class.

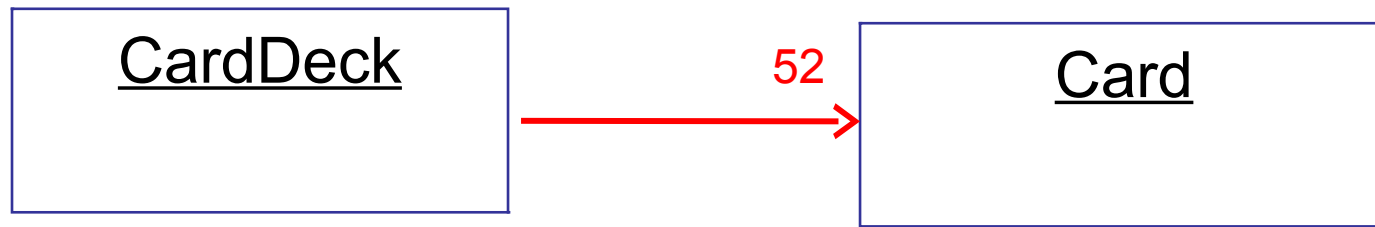| GameConsole	game →	Game

```
class GameConsole:

    public __init__(self, game: Game):
        """console keeps a reference to game"""
        self.game = game
```

Association with Multiplicity

❑ You can indicate *multiplicity* of the association.

A card deck contains exactly 52 cards.

CardDeck	52	Card

```java
// Java
public class CardDeck {
    private Card[] cards;
    public CardDeck() {
        cards = new Card[52];
        ...
    }
```
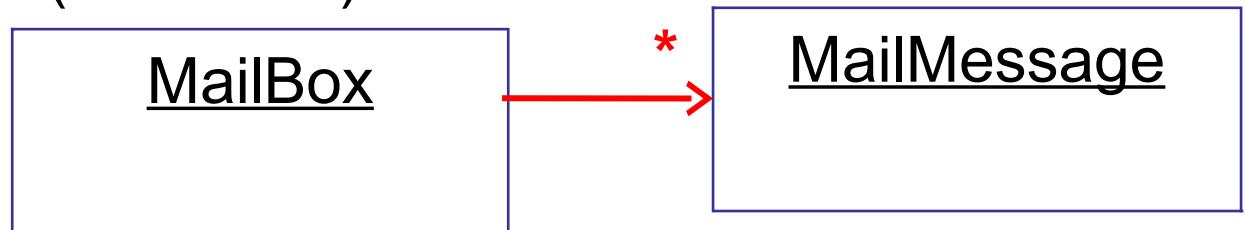
Association with Variable Multiplicity

A MailBox may contain 0 or more mail messages.

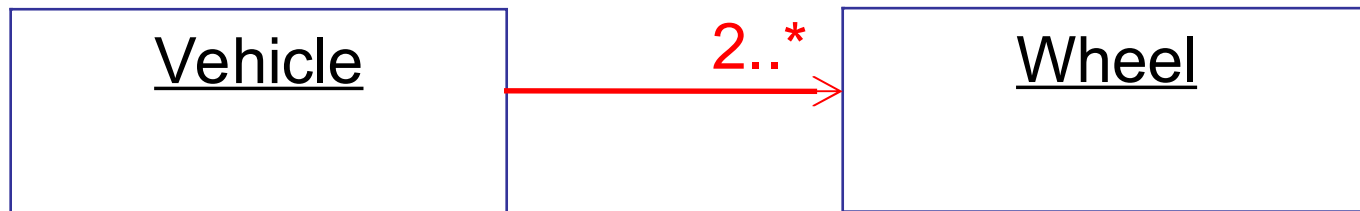* = any number (0 or more)

1..n = 1 to n

n = exactly n

| MailBox | →* | MailMessage |

```python
class MailBox:
    def __init__(self):
        self.messages = []

    def add_message(self, msg: MailMessage):
        self.messages.append( msg )
```
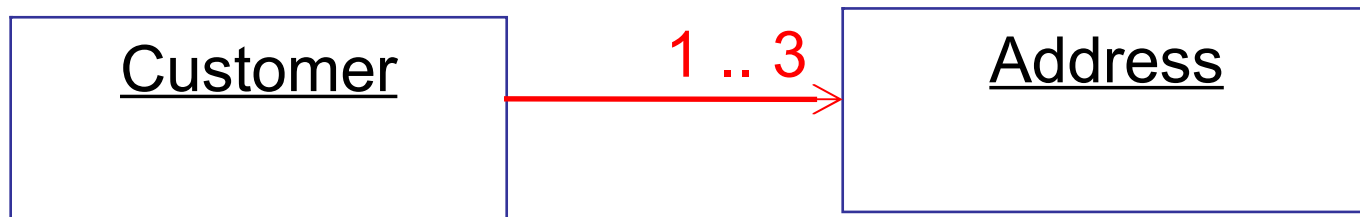
Vehicle has at least 2 Wheels

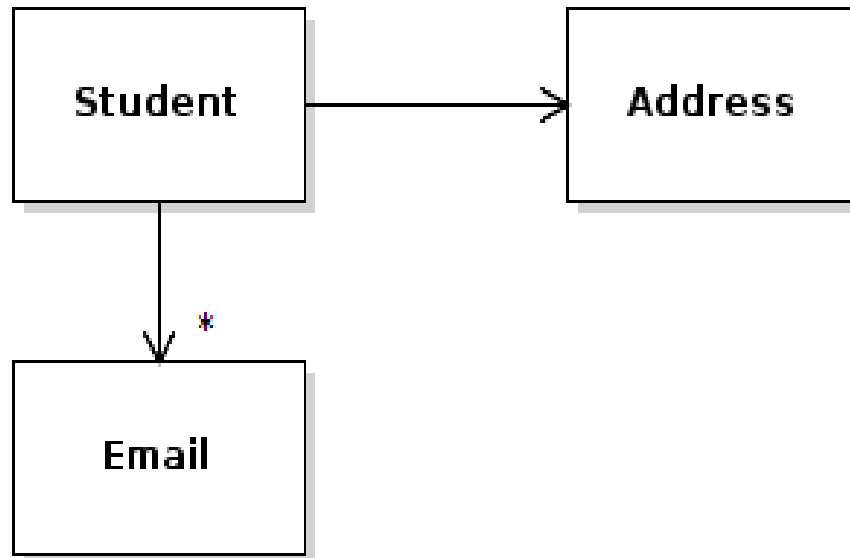A vehicle must have at least 2 wheels.

Vehicle	—— 2..* →	Wheel

A Customer must have an Address, and can have at most 3 Addresses (in our e-commerce app).

| Customer | —— 1 .. 3 →| Address |
|----------|-----------|---------|

Class with Many Associations

A Student *has* one Address and 0 or more Email addresses.

```
class Student {
    private Address homeAddress;
    /** he have many (or none) Email addresses. */
    private List<Email> emailAddress;
```

Bidirectional Association

If each object has a *reference* to the other object, then it is *bidirectional.*

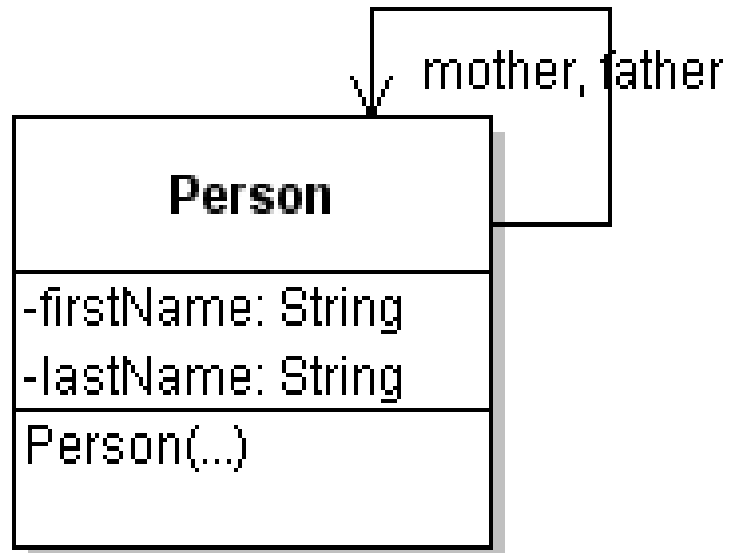| Person		Company
employer: Company	*	employees: Person[*]
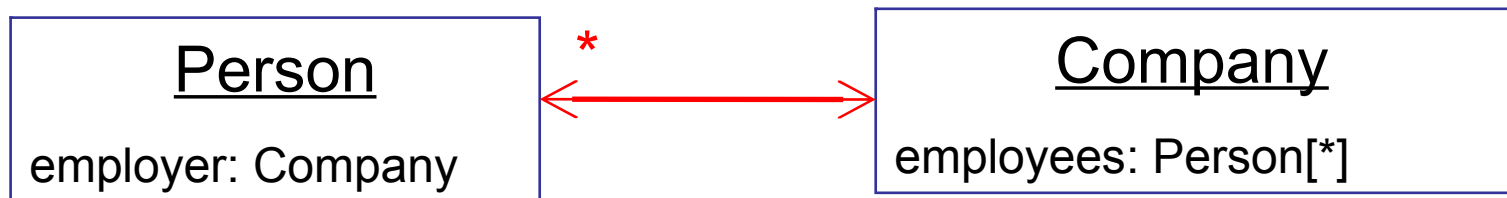
This is rare, in practice.

Try to avoid bidirectional associations (hard to maintain consistency).

Self-Association

A person has a mother and father.

```
class Person:
    father: Person
    mother: Person
    firstName: str
    lastName: str
```

mother, father

Person

-firstName: String
-lastName: String

Person(...)

Exercise: Django Polls Models

Draw a UML class diagram for Question and Choice.
Show attributes and associations with multiplicities.

- Question has a choice_set attribute (added by Django) with zero or more Choices.

- A Choice has only 1 Question.

- Don't show the "id" attribute.

◇ Aggregation: whole-parts relationship

One class "collects" or "contains" objects of another.

A Mail Box stores (collects) Mail Messages

MailBox	◇━━━━━━*	MailMessage

```
public class MailBox {
    private List<MailMessage> messages;
    /* a MailBox consists of MailMessages */
```

Aggregation often shows a whole-parts relationship

The parts *can exist* without the whole. (MailMessage can exist outside of a MailBox.)

When to use Aggregation?

☐ One object "collects" or "aggregates" components.

Advice: **Don't show aggregation.** (*UML Distilled*, Ch. 5.)
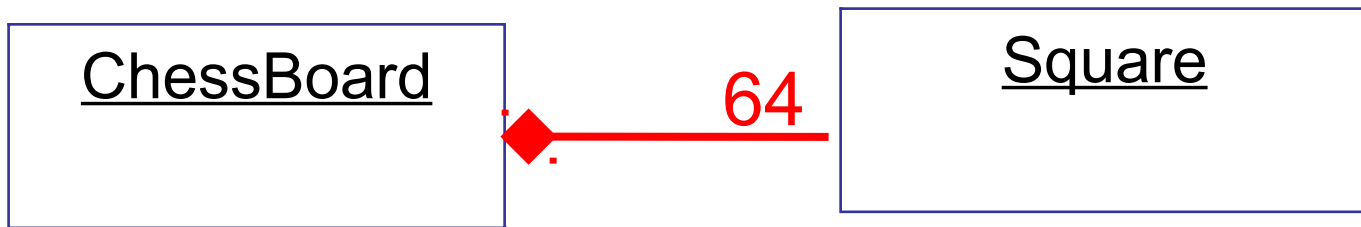
Just show it as **association.**

If it is really "composition" then show composition.

Composition: ownership relation

One class "owns" objects of the other class.

If the "whole" is destroyed, the parts are destroyed, too.

ChessBoard	64	Square

```
public class ChessBoard {
  private Square[][] squares = new Square[8][8];
```
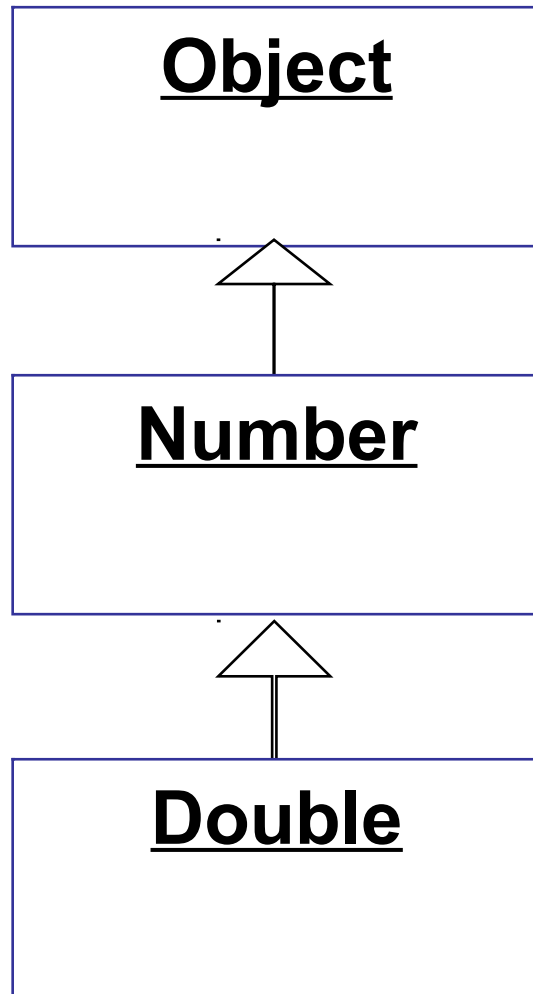
A Student owns his Email Addresses

Composition: A Student *owns* his Email addresses.

1) No one else can have the same email address.

2) When student is deleted, we delete his addresses, too!

```
class Student:
    # student uniquely owns his email addresses
    email_address: list
```

Inheritance
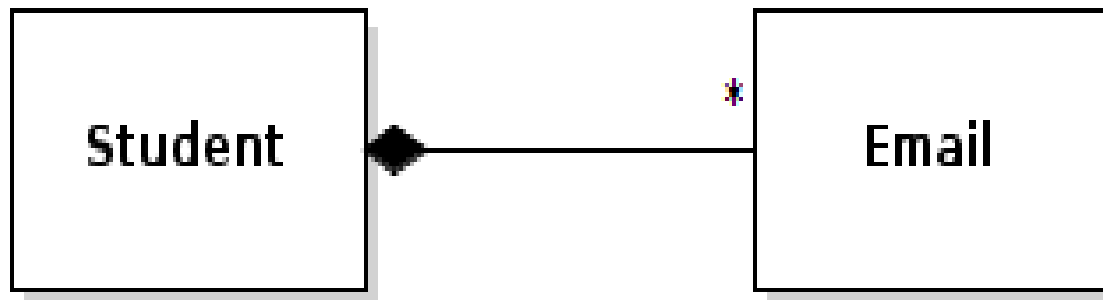
Object

Number

Double

Number is a *subclass* of Object.
Number *inherits* all the methods of Object.
Number *overrides* the definition of some methods, and adds new methods.

Double is a subclass of Number.
Double *inherits* all the methods of Number.
Double *overrides* the definition of some methods, and adds new methods of its own.

Other names for Inheritance

Specialization - a subclass is a *specialization* of the superclass.

Generalization - the superclass *generalizes* behavior of a hierarchy of subclasses.

Python Question

1. What is the (eventual) superclass of all classes? (the "*cosmic superclass*")

2. Name some useful methods that all classes inherit from this *cosmic superclass*?

_____ - *used to print the object*

_____ - *used to print how to recreate object*

_____ - *used to test if two object are equal*

These methods are in the *cosmic superclass to guarantee* that all *classes have them (no exception thrown) even if a subclass doesn't provide them itself.*

Exercise: Django Models & Inheritance

Django models are all subclasses of django.db.models.Model.

> django.db.models.
> <u>Model</u>

Add this to your class diagram for Question and Choice.

Only draw the Model class once. "inheritance" arrows from both Question and Choice connect to it.
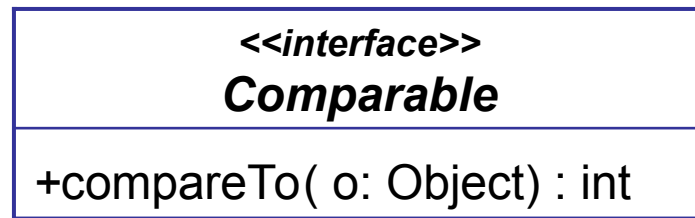
Interface

Interface - a specification of some behavior, without an
implementation.

Example: USB specifies the behavior of USB devices.
Each manufacturer implements it himself.
All USB devices implementing the interfere are
interchangable.

*Example: you can connect an Acer USB mouse to a
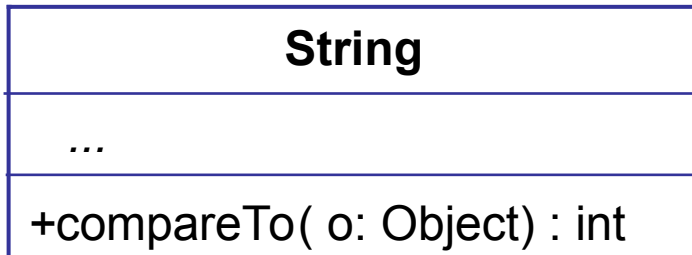Dell laptop and it works.*

Implements an Interface

The **String** class *implements* **Comparable** interface

```
+----------------------------------+
|        <<interface>>             |  <----  write <<interface>> above the name
|        Comparable                |
+----------------------------------+
| +compareTo( o: Object) : int     |
+----------------------------------+
              △
              ┊
              ┊  implements
              ┊
+----------------------------------+
|            String                |
+----------------------------------+
|  ...                             |
+----------------------------------+
| +compareTo( o: Object) : int     |
+----------------------------------+
```

You don't need to write "implements" on the diagram,

but you MUST use a dashed arrow and open triangle arrowhead as shown here.

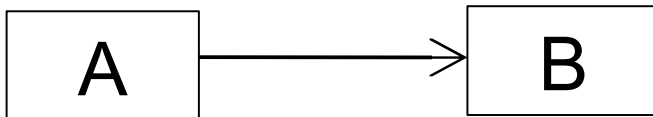# Inheritance & Implements

You can have both in one class.

```
public class Student extends Person
        implements Comparable<Student> {
```
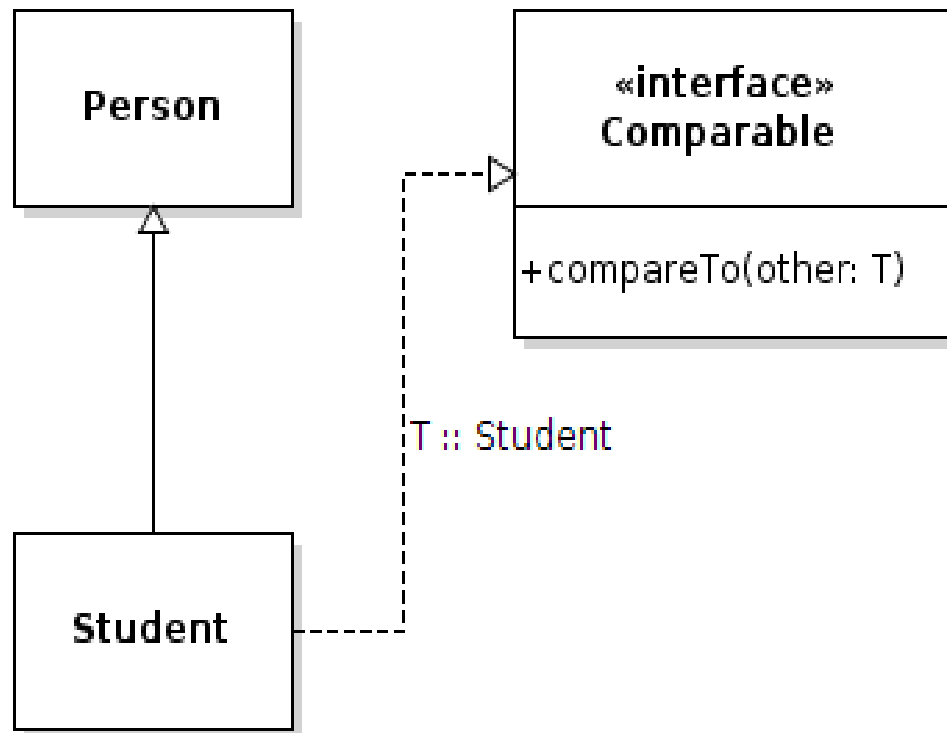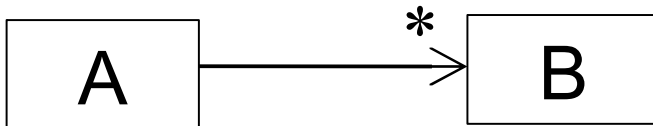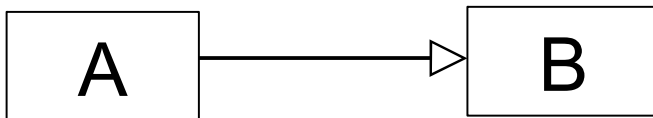
Summary of relationships

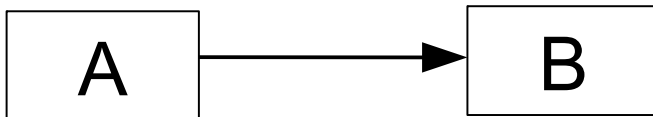|		
A ┈┈┈▶ B		A depends on or uses B
A ───▶ B		A has an association to a B
A ──*─▶ B		A has association with zero or more B objects
A ───▷ B		A is a subclass of B
A ┄┄┄▷ B		A implements interface B
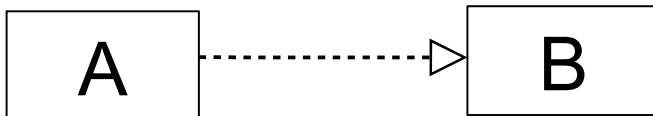
Bogus relationships

A ┈┈┈┈▶ B   Incorrect. (OK in casual drawing, not in class).

A ────▶ B   Incorrect.

A ┈┈┈┈> B   *
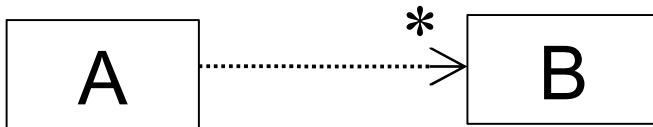
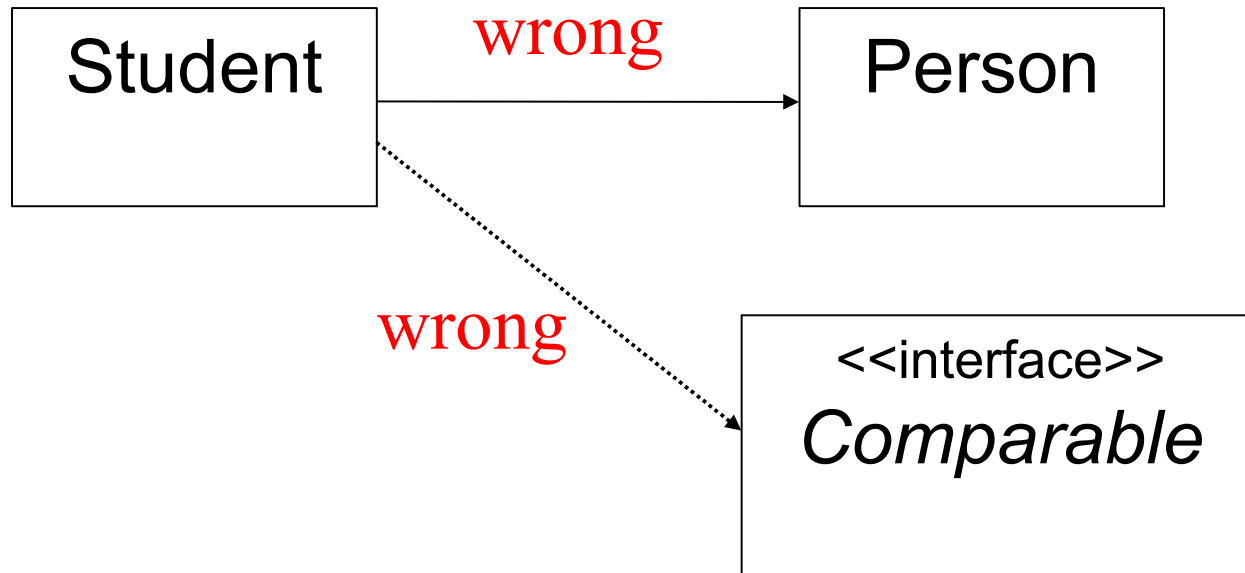Nonsense (only association has multiplicity)

A ────> B   1

Nonsense

A ◇┈┈┈┈> B
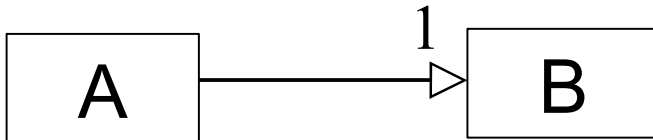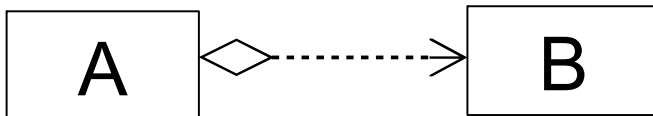
Nonsense (aggregation is a form of association)

UML is for Communication

To communicate <u>clearly</u>, use the correct notation.

No partial credit for wrong relationships or bad notation.

Exercise: *design* with UML

Draw a UML diagram showing Sale, LineItem, Product and their relationships. Try to show what is described here:

A Sale contains one or more Line Items that a Customer is buying.

Each Line Item is something the Customer is buying; it has a quantity and reference to a Product being bought (e.g. 3 units of Nescafe Ice Coffee).

Line Item can compute its own total price (e.g. 3x20).

A Product is a *kind* of item the store sells.
It has a description, a unit price, and unit type.
For example, "Nescafe Ice Coffee", 20 Bt, "can"

References

UML Distilled, 3rd Edition. Chapters 3 & 5 cover Class Diagrams.

- Chapters are short with many examples.
- Chapter 2 *Development Process* is good, too.

UML-diagrams.org.

Detailed, concise explanation of UML syntax.

https://www.uml-diagrams.org/class-diagrams-overview.html

https://www.uml-diagrams.org/property.html

Video

UML Class Diagram - https://youtu.be/UI6IqHOVHic. Video by LucidChart, online diagramming software.

- In the "Animal" example, age should be a method, not an attribute. Why?

- What should they use instead of age as attribute?