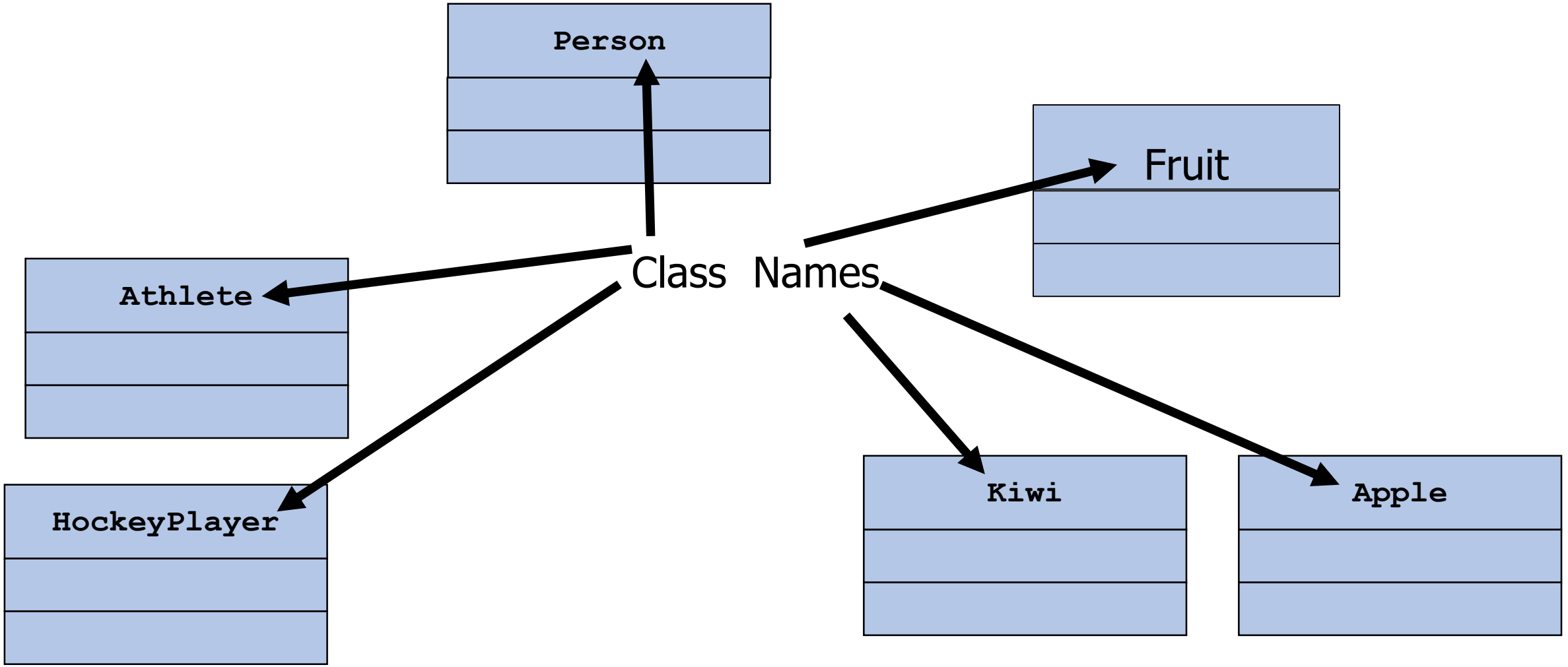
Class Diagramming via UML

Anthony J Souza

UML Class Diagrams

- Class diagrams allow us to express classes in UML
- Including attributes and methods
- Class diagrams allow us to declare our classes (visually)

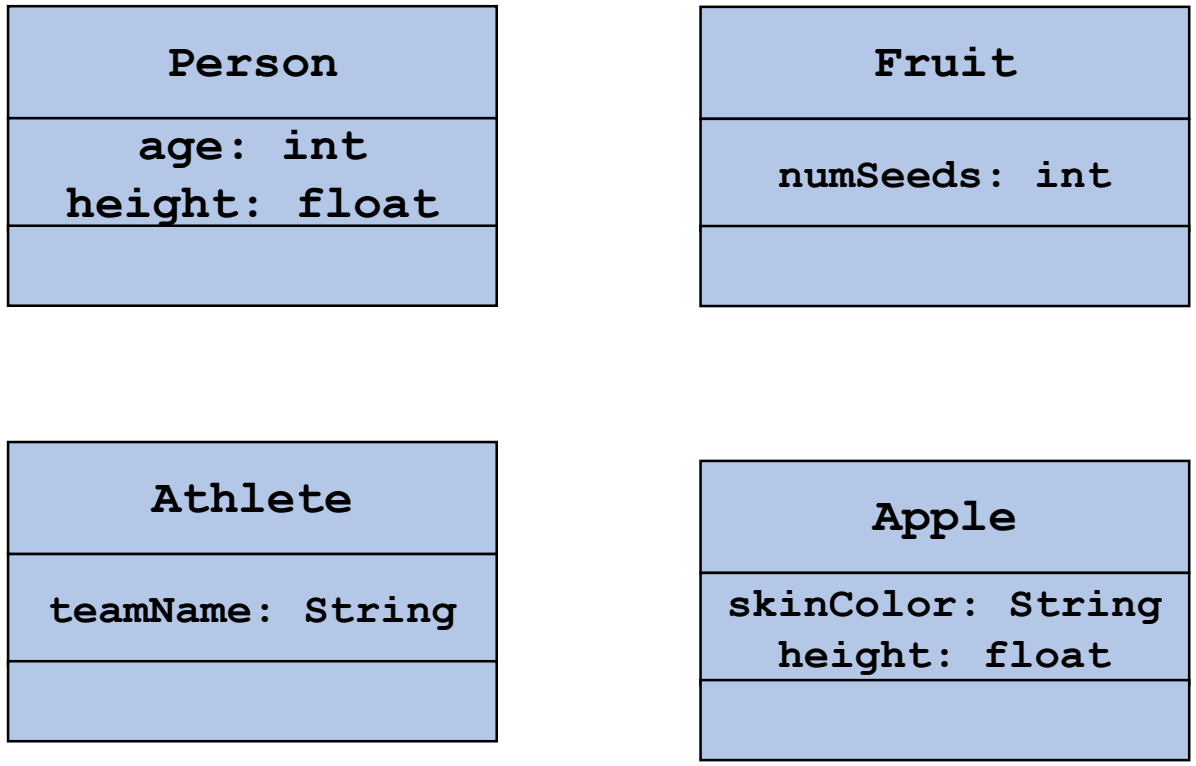
Classes in UML



Classes in UML

- In the previous slide there were two different types of class blocks.
- Both forms are acceptable for classes that have no attributes or methods
- The two empty blocks are used for holding declarations of attributes and methods
- The top block is used to hold the class name

Attributes in Classes



Attributes in Classes

- Attributes are specified using the form:
  - name : type
  - Names are usually alphanumeric
  - Types refer to the type storage used
    - Int, float , long , double, String, etc…
  - Types can also be logical Types that have a specific domain.
    - For example, in the Person block, would have:

      age : Duration
    - Here Duration can be said to have a domain of {int,float,double,String}
    - This means age can have either of these types.

Attribute Modifiers

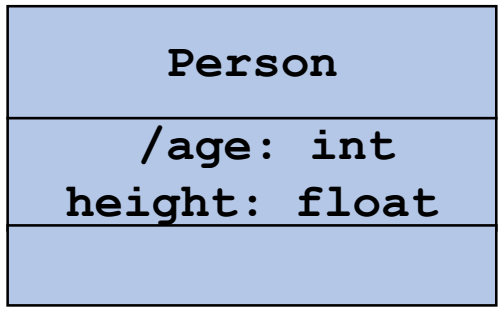
- Attributes can be read/write (default) or read only
  - Attributes that are read only are preceded with the modifier **/**
  - Attributes can be defined on the objects (default) or the class
- Class attributes are preceded with the modifier '**$**'
  - Static variables in Java

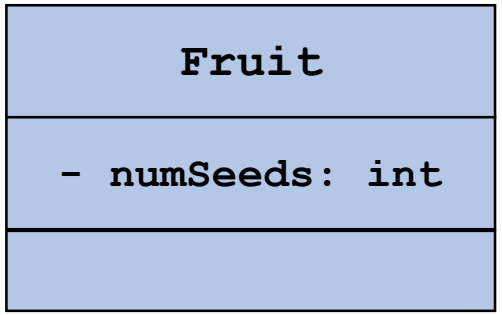
Attribute Modifiers

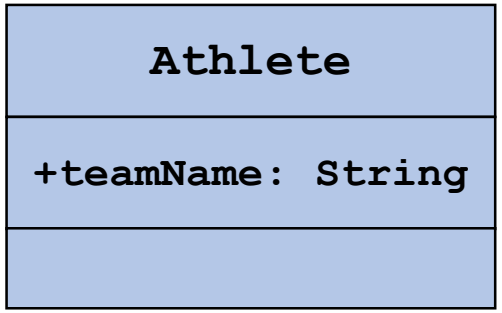
- Attributes can have different visibilities

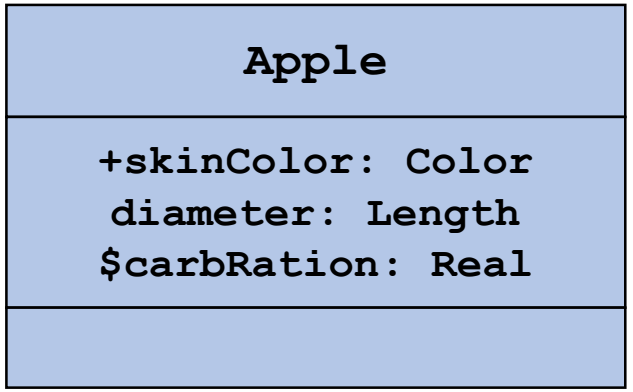
Symbol	Description
+	Public modifier – can be accessed from anywhere.
-	Private Modifier - can only be accessed from within the class
#	Protected Modifier - can be accessed from within the class, or any descendant (e.g. a subclass). Java we also need to state that protected members can only be accessed by classes in the same package

Attributes in Classes

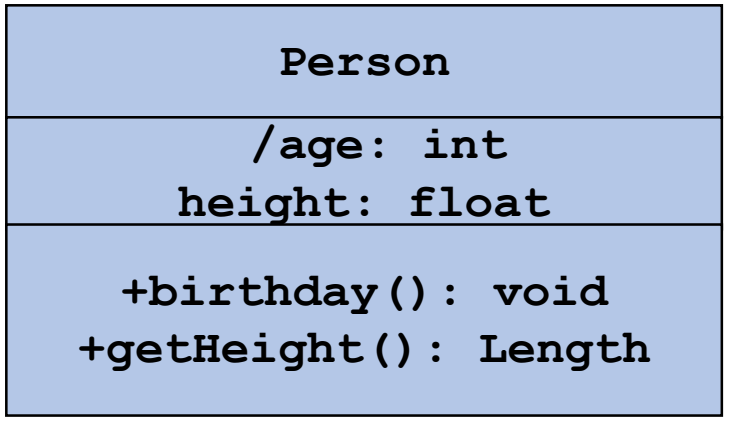








Operations in Classes



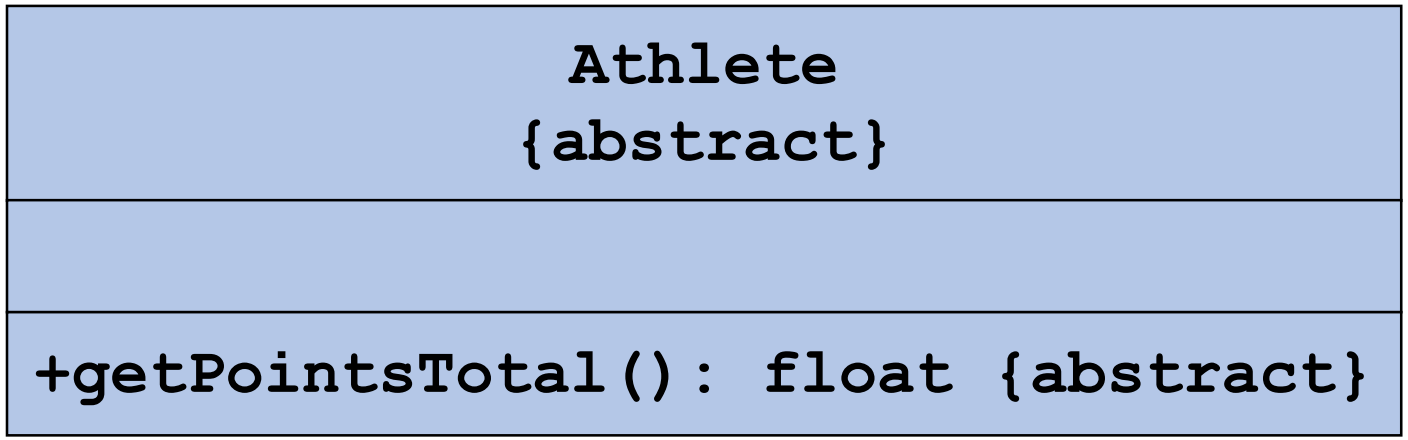
Operations in Classes

- Operations can be defined, using the form:

```
opName([in|out|inout] param1 : type1,…) : returnType
```

- The modifiers in, out, and inout specify which direction(s) the parameters are to travel
  - in: input to the operation
  - out: output from the operation
  - inout: input to the operation, possibly modified, then output from the operation
  - Often, if no modifier is specified, a parameter is assumed to be input only (in)
- Again, type1 and returnType are types from the application domain

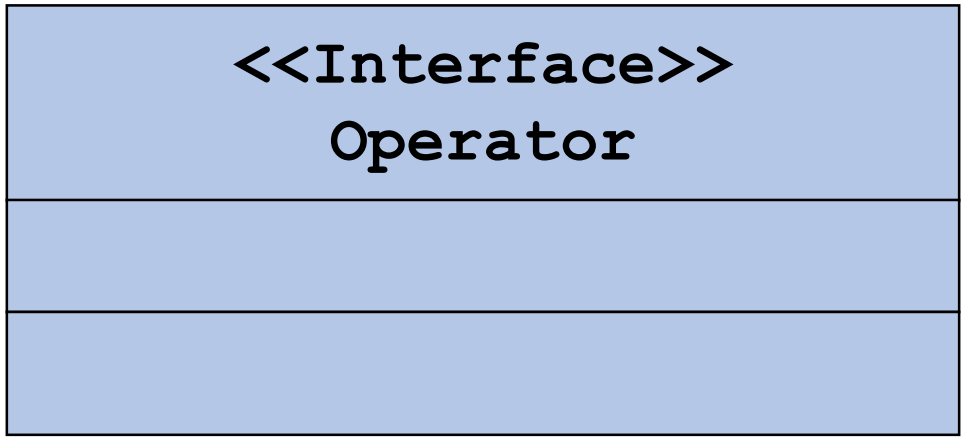
Abstract Classes and Methods



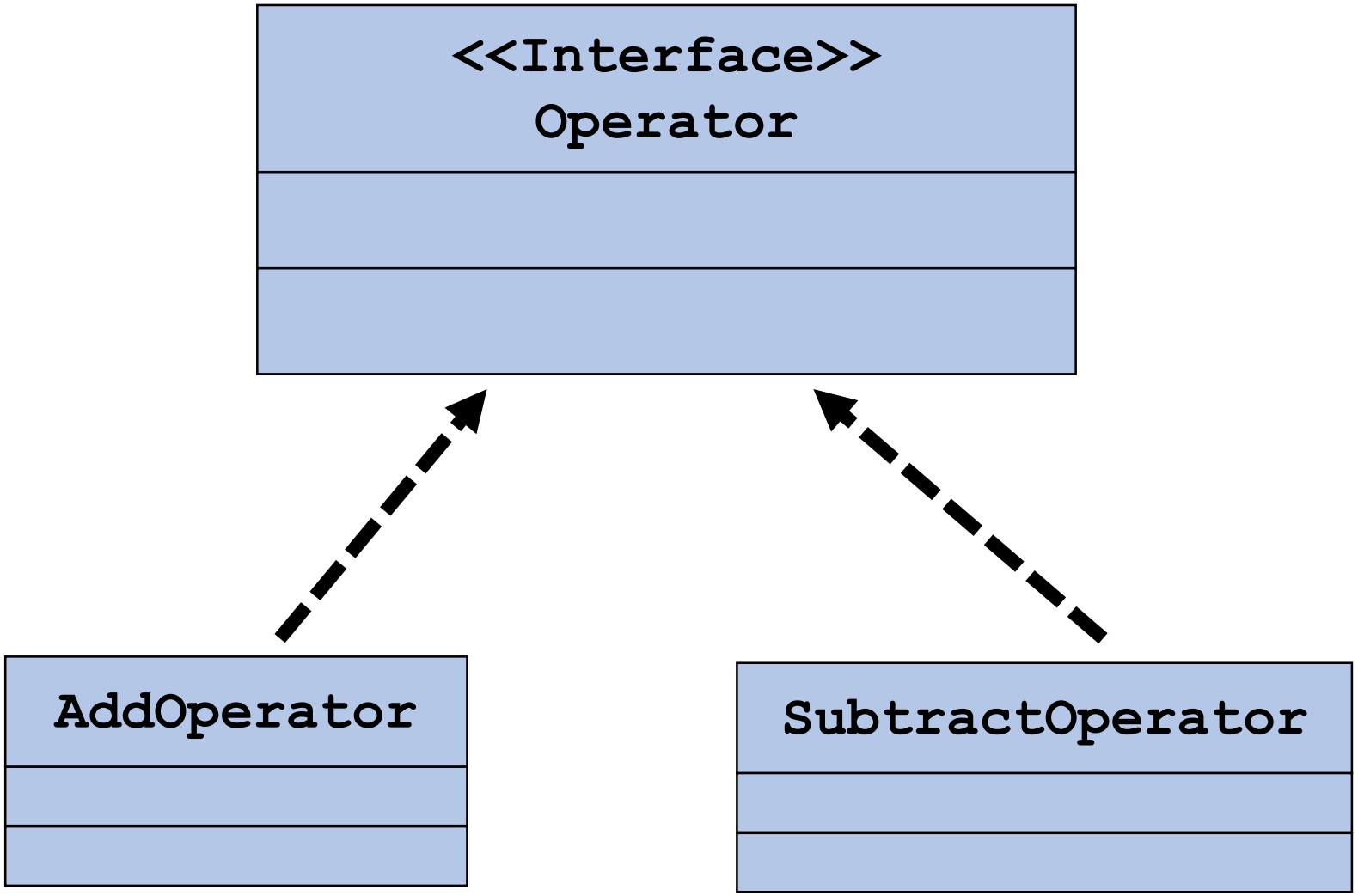
Abstract Classes and Methods

- The class Athlete was declared abstract
- This is analogous to abstract classes in Java
- The getPointTotals() method was declared abstract
- This means the method is not implemented in this class, but left for a (concrete) subclass to implement
- This method being abstract makes it necessary for Athlete to be declared an abstract class
- Note in some UML notations abstract classes are also class names that have been italicized text.
  - *Athlete* instead of Athlete {abstract}
  - Both are fine, one is a little more explicit.

Interfaces



Interfaces



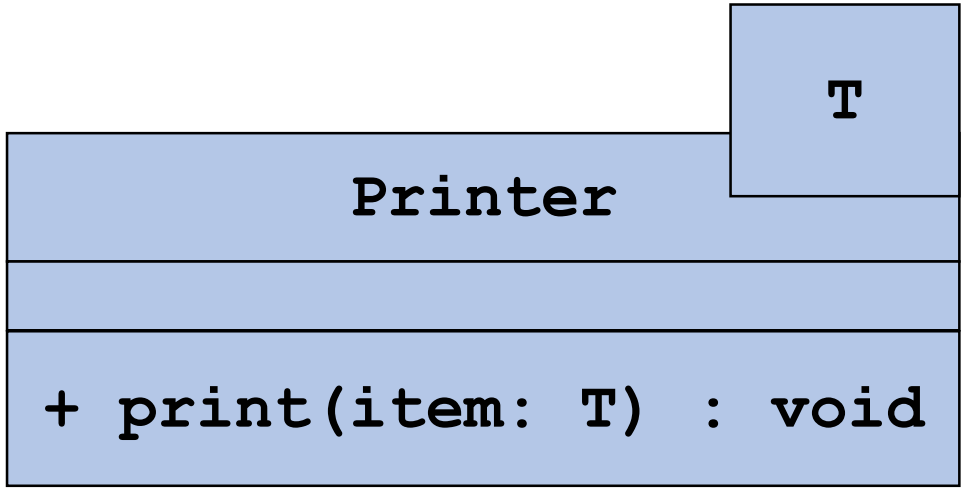
Interfaces

- Interfaces will be defined like class or abstract classes.
- Except name of interface will be enclosed in << name >>.
- Classes that implement a given interface will have their arrows point towards the interface and are dashed.

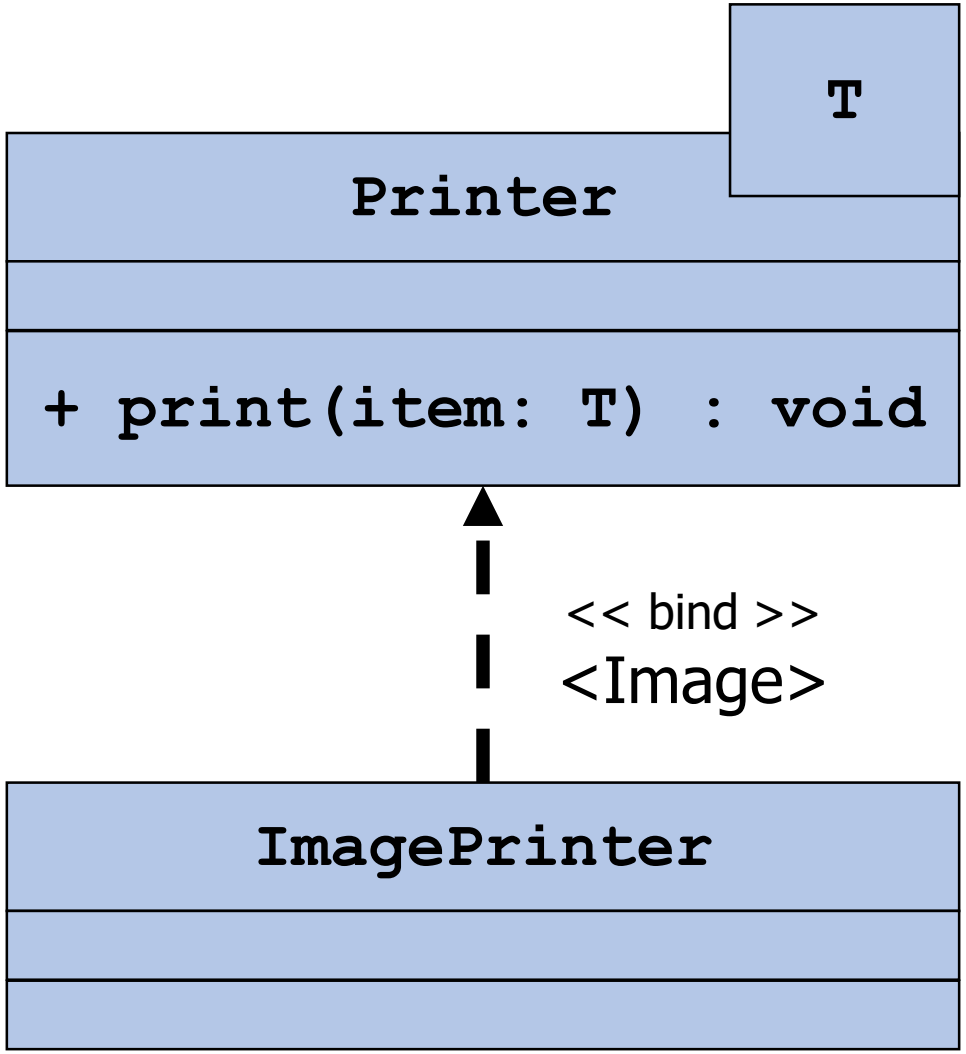
Genericity in Classes

- Genericity, or run-time determined types, in a class can be represented in UML
  - This allows you to design classes to work on a group of types, where the type is ‘generic’ at compile time
- The type, specified at run-time, is an argument or parameter to the class
  - Such classes are called parameterized classes

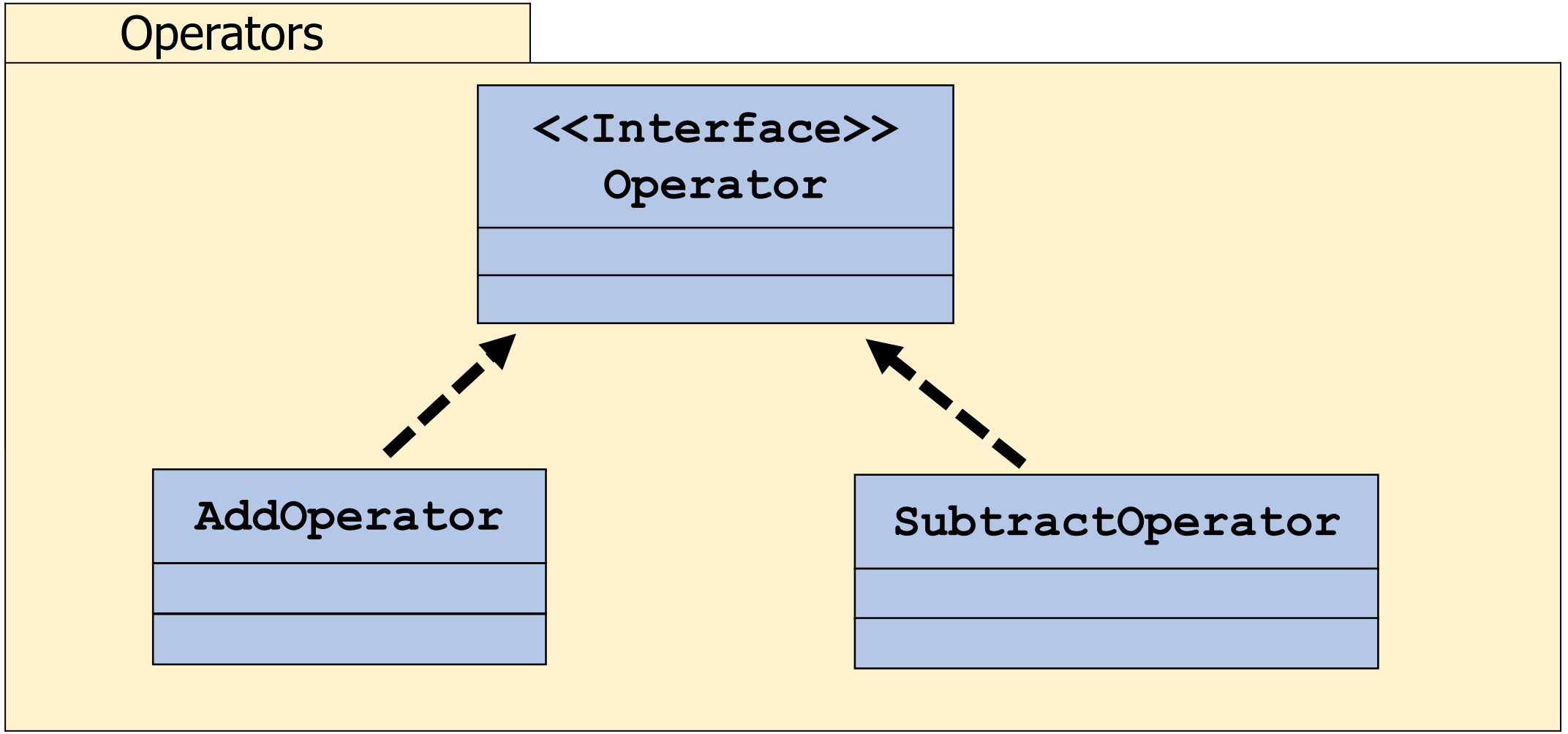
Parameterized Classes



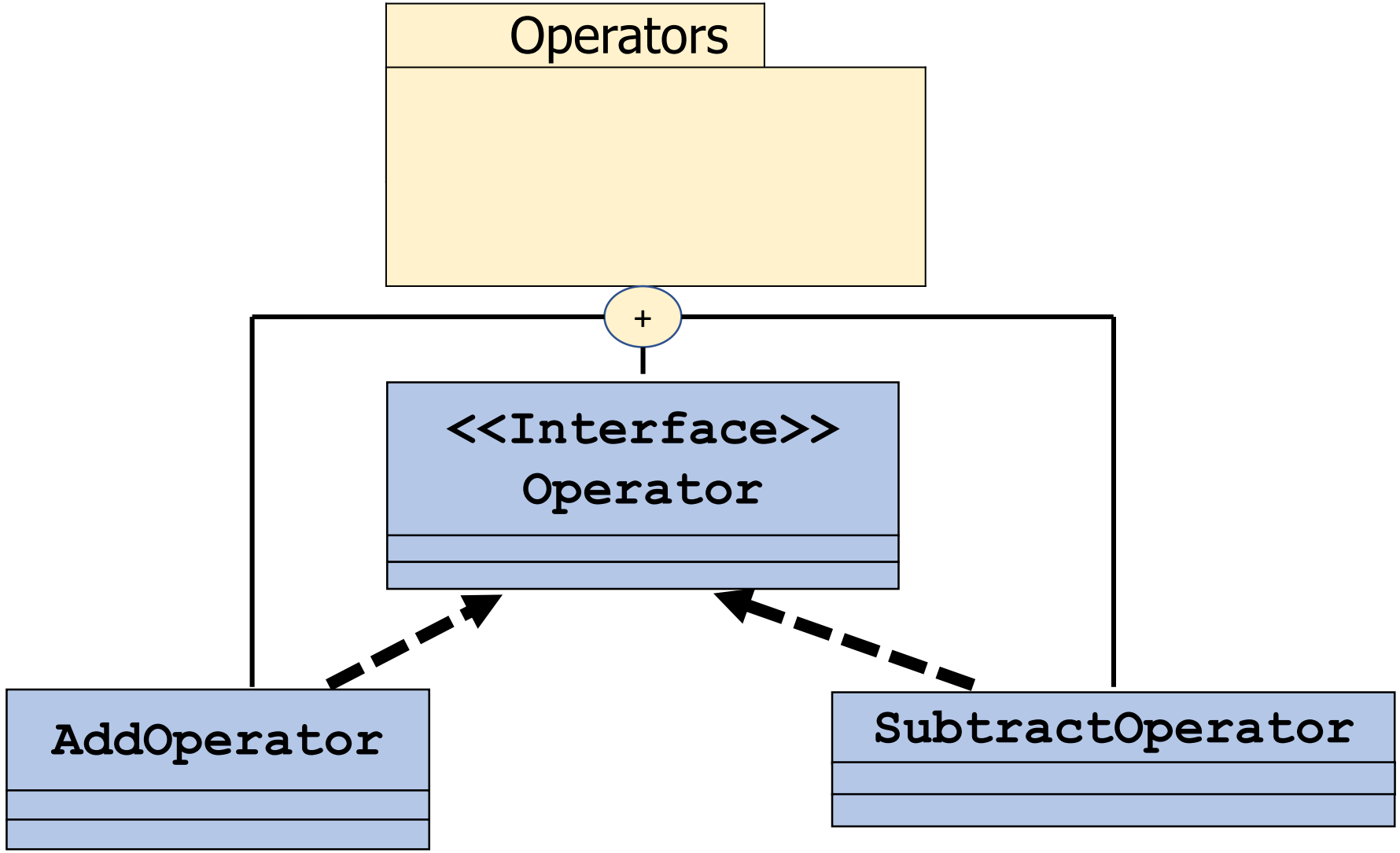
Parameterized Classes



Packages


Operators
<<Interface>>
Operator
AddOperator
SubtractOperator

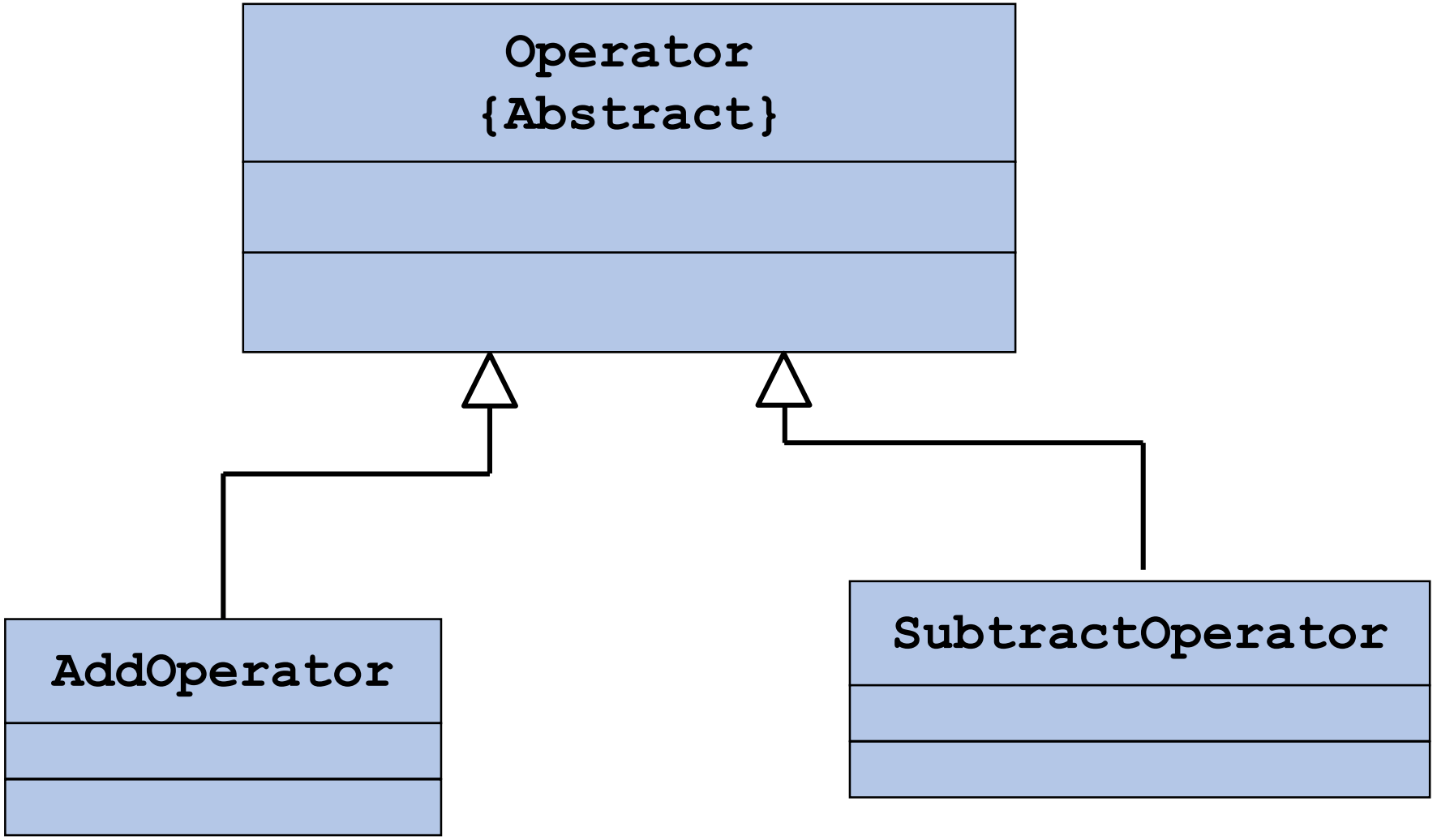
Packages (Alternate Notation)



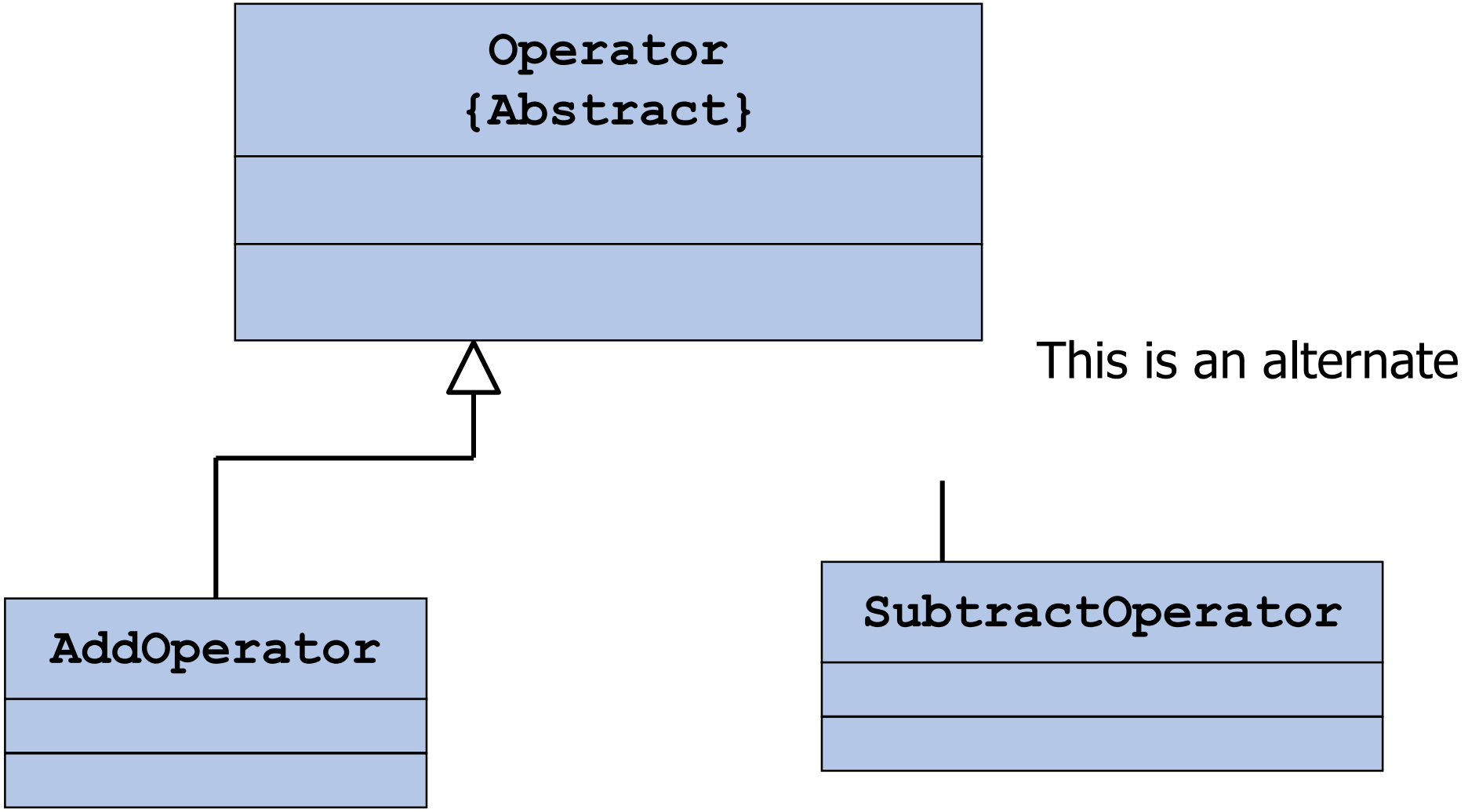
Class Diagrams

- We now need to know how to define relationships between these classes
- There are three types of relationships:
  - A ‘is a type of’ B (inheritance)
  - A ‘is associated with’ B (association)
  - A ‘is a part of’ B (composition)

Inheritance



Inheritance



This is an alternate

Association

- Associations between classes represent logical relationships objects of those classes might hold
- For example, a ‘Person’ class might have an association ‘HomeAddress’ with another class ‘Location’
  - This is because an instance of Person might have an instance of Location as its home address

Association

- Keep in mind, associates deal with objects, rather than classes
  - Thus they declare what associations are possible between their instances
  - There may be several instances of the association at any time
  - The number of instances changes over time
  - One instance may hold several instances of the same association

Association

- An association contains:
  - The name of the association
  - Multiplicity at each end of the association
    - This describes the number of instances that may occur at that end of the association
  - The role of the class at each end (optional)
    - This is usually a name used to describe the instances with respect to that association
- The last two will be described individually

Association

- Let’s illustrate these concepts with an example
- Say we have two classes, ‘Person’ and ‘MusicalInstrument’ , and an association called ‘Plays’
- The association Plays represents all instances where a Person plays a Musical Instrument

Association

- We might have the following instances of ‘Plays’:
  - Jose plays Piano
  - Anthony plays Saxophone
  - Rico plays Trumpet
  - Tony plays Drums

Association

- The multiplicity of Person means:
  - How many people can play a given instrument?

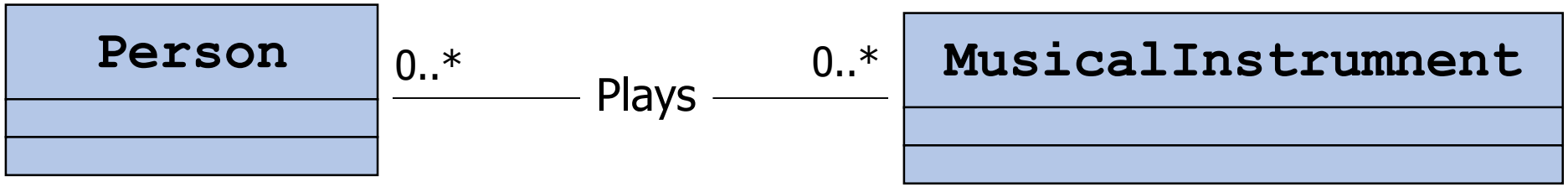
Association

- The multiplicity of Person means:
  - How many people can play a given instrument?
  - Any number of people can play a given instrument, including zero
  - This is represented using 0..*

Association

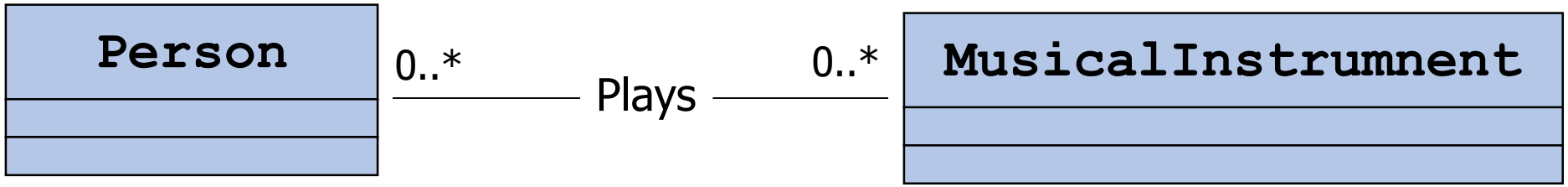
- The multiplicity of MusicalInstrument means:
  - How many instruments can a person play?

Association



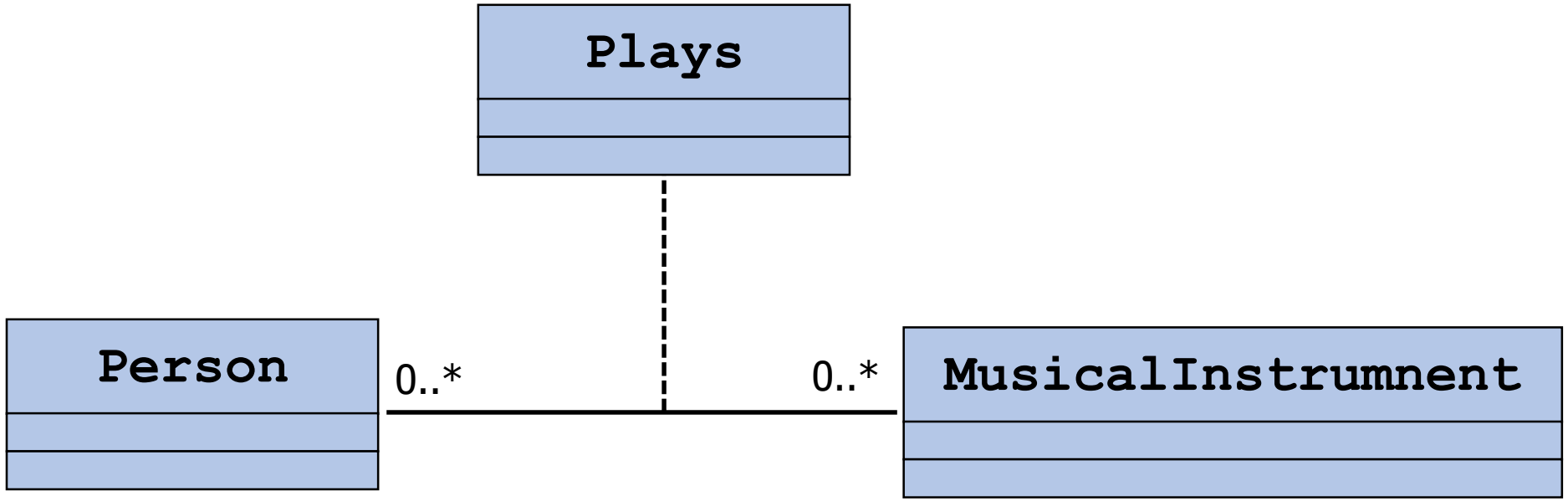
- The multiplicity of MusicalInstrument means:
  - How many instruments can a person play?
  - A person may play zero instruments, or any number of them
  - Again, this is 0..*

Association



- Essentially, an association could be considered a class itself
- Every instrument any person plays is an instance of the Plays association

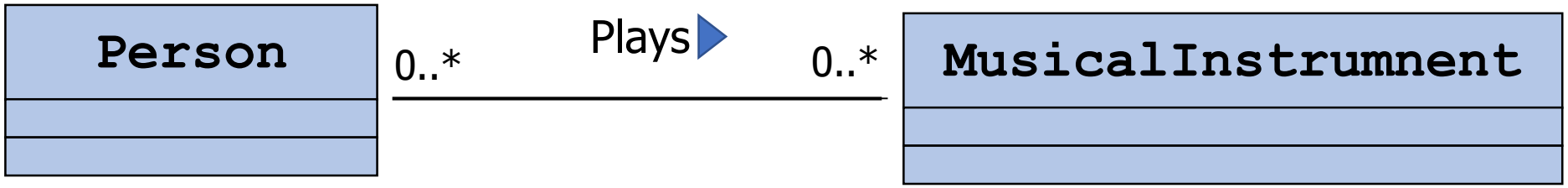
Association



Higher-Order Associations

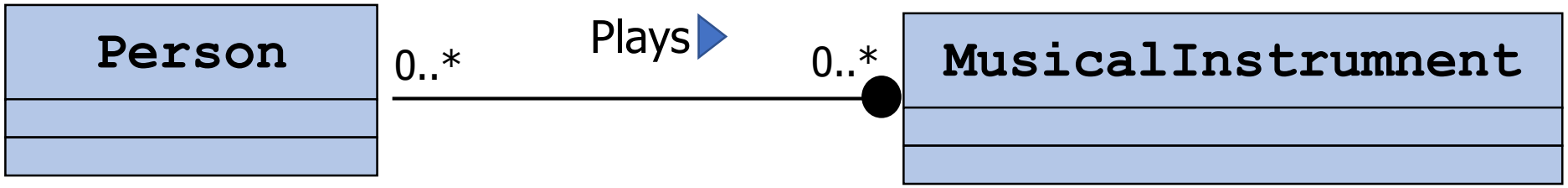
- Associations need not always be between 2 classes
  - Associations can exist between 3 or more classes as well

Association Reading Order



- We can add a reading order by adding an arrow pointing in the direct of how the relationship should read.
- Here we are saying a Person plays a MusicalInstrument.

Association Ownership



- We can show ownership by using a filled circle on one end of the association.
- The class on the other end of the dot owns the class that the dot is attached to.
- Person owns a MusicalInstrument.

Aggregation

- Aggregations is also a type of association, but again, a special one that is given its own notation
  - This is because, like composition, it is very common
- An aggregation is another relationship between classes which says one class ‘is a part of’ another class
- it is binary association
- it is asymmetric - only one end of association can be an aggregation,
- it is transitive - aggregation links should form a directed, acyclic graph, so that no composite instance could be indirect part of itself,

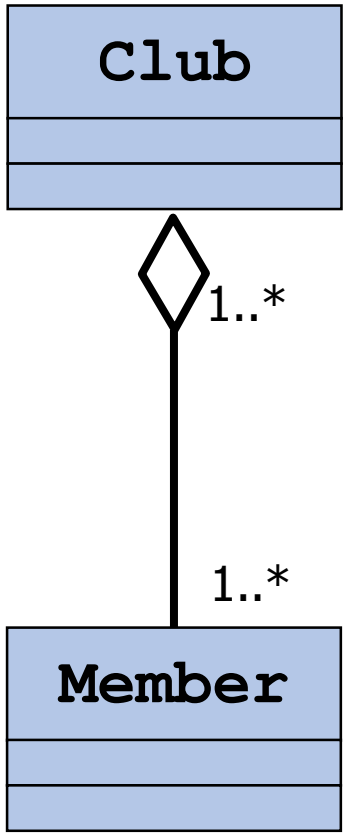
Aggregation

- To illustrate a constituent, again, let’s use a few examples:
  - A Forest is an aggregate of Trees
  - A Program is an aggregate of Statements
  - A Fleet is an aggregate of Ships
  - A Deck is an aggregate of Cards
- Parts of an aggregate are called constituents:
  - A Tree is a constituent of Forest
  - A Statement is a constituent of Program
  - A Ship is a constituent of Fleet
  - A Card is a constituent of Deck

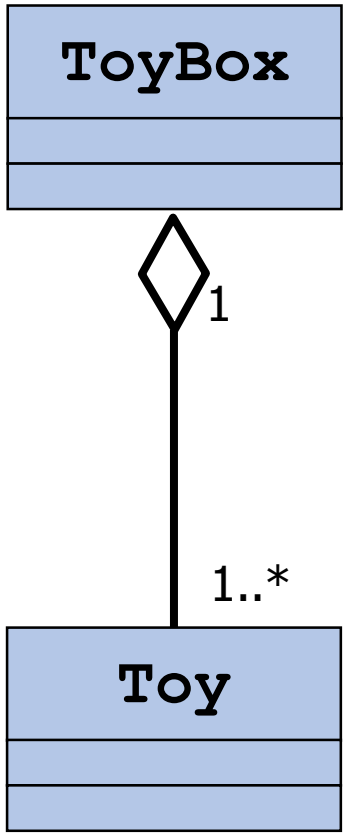
Aggregation

- These constituents share one common property:
  - The constituents of an aggregate are usually the same type
- Two other properties are also defined for aggregation:
  - An object may be a constituent of more than one aggregate simultaneously
  - It is possible to have an aggregate without any constituents.
  - This is identified by the multiplicity of the constituent end of the association

Aggregation



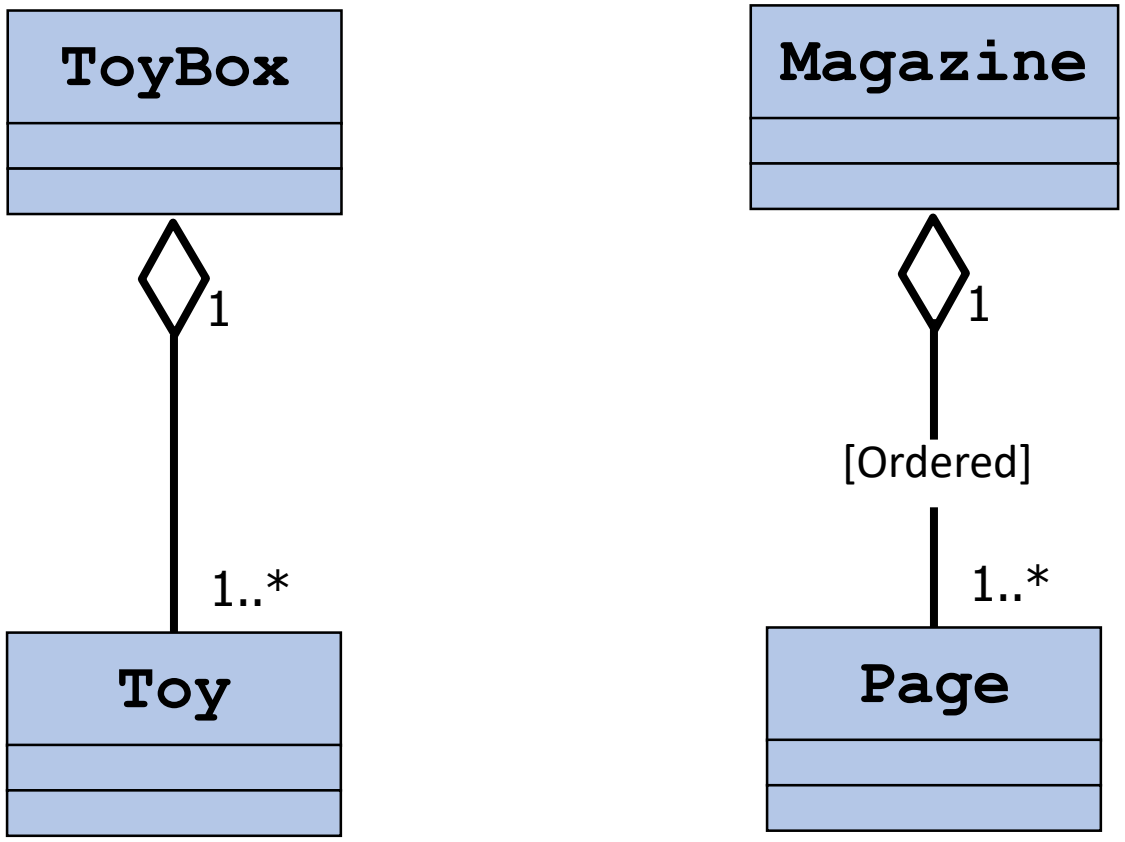
Aggregation



Aggregate Order

- Aggregates can be ordered or unordered
  - In ordered aggregates, the order of the constituents within the aggregate is important
  - In unordered aggregates, the order is unimportant
- Ordered aggregates are indicated using the symbol
- [ordered]
  - Unordered is the default for aggregates

Aggregate Order



Composition

- Composition is a relationship between two classes where one class is a part of another (an ‘is a part of’ relationship)
- If a class A is composed of classes B, C, and D
  - we may say that A has components B, C, and D
- Compositions typically consist of a name, as well as the multiplicity of the component

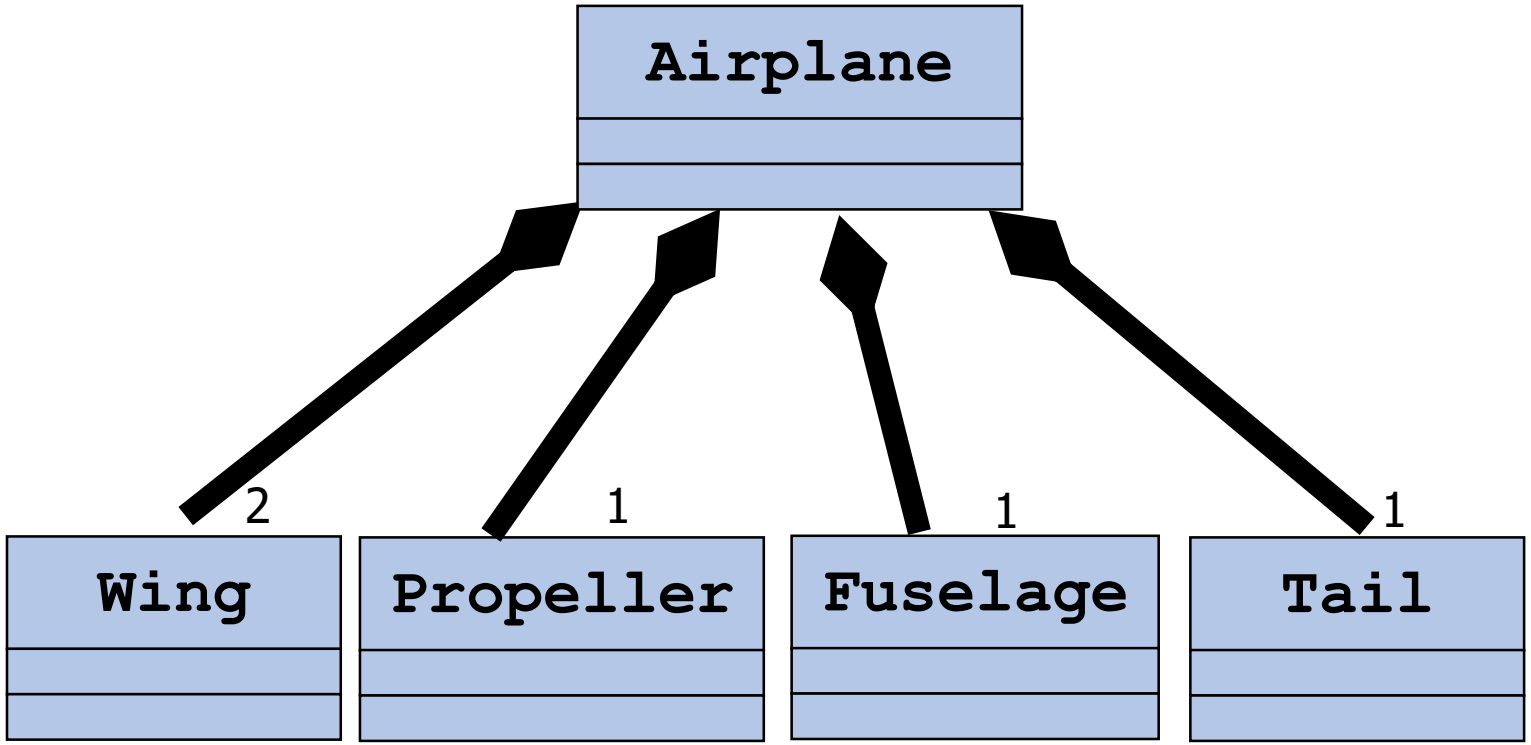
Composition

- it is binary association,
- it is a whole/part relationship,
- a part could be included in at most one composite (whole) at a time
- if a composite (whole) is deleted, all its composite parts are "normally" deleted with it

Composition

- If a class ‘Car’ has a composition relationship with another class ‘Wheel’, we should agree that normally the multiplicity of Wheel is 4 (exactly)
- ‘Vehicle’, on the other hand, could have any number of wheels (0..*)
- Composition is a special form of association
  - It is important enough to have its own notation

Composition



Aggregation and Composition

- In programming, aggregation and composition are usually represented by class attributes
- However, for clarity, in class diagrams, aggregation and composition should always be used instead of the attribute form
  - This is because it allows the definition of the class being used to be shown

Aggregation vs. Composition

- Both aggregation and composition represent the ‘is a part of’ relationship
- The main difference is what the part represents
  - In aggregation, the part (constituent) is meaningful without the whole (aggregate)
  - In composition, the part (component) is not meaningful without the whole (container)