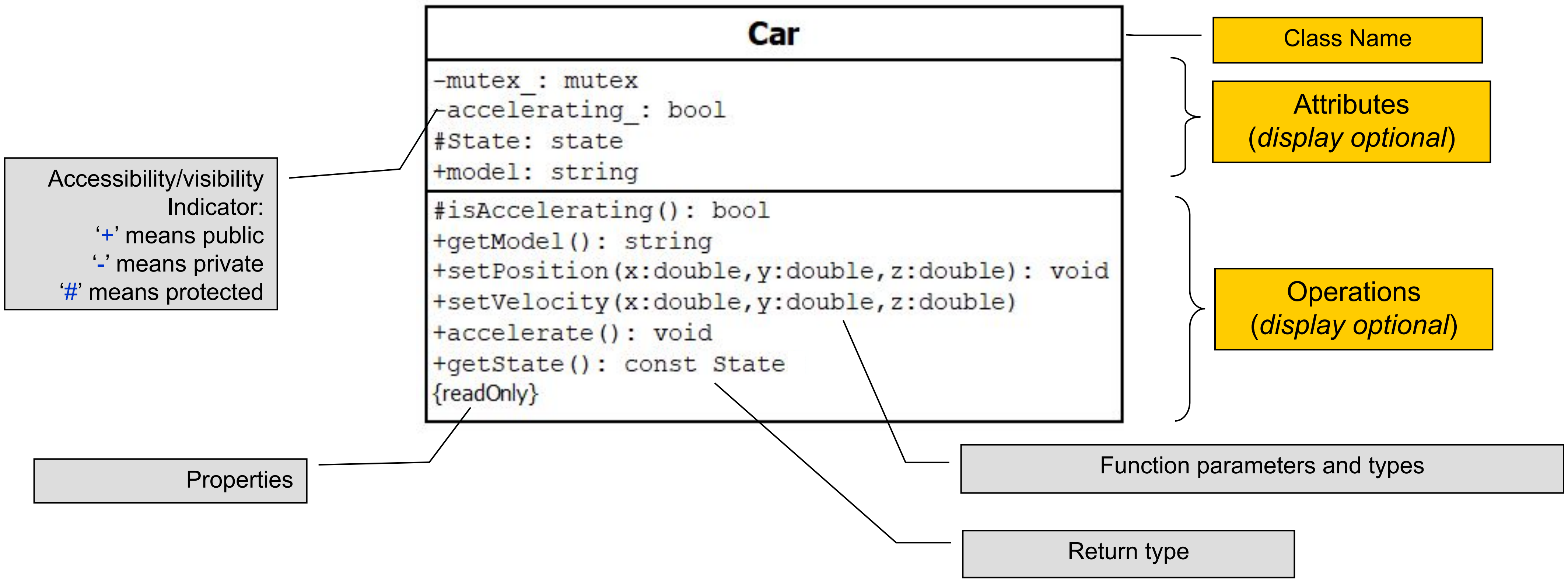
Lecture 10 – Class Diagrams

Learning Goals

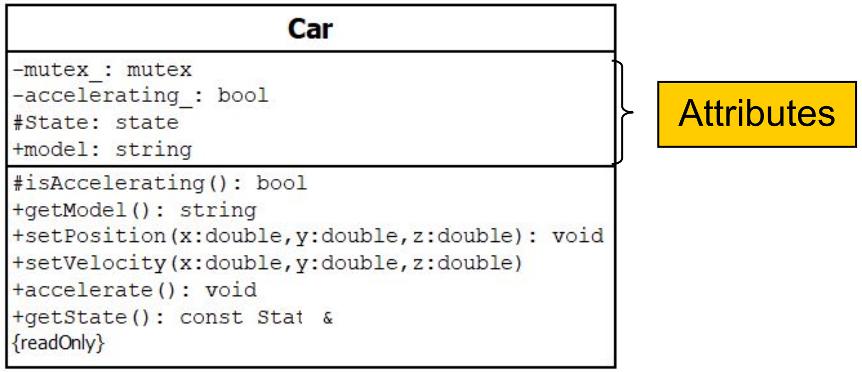
- Create a basic **class diagram** with **attributes** and **operations**
- Convert a class diagram to code
- Describe the three types of class **relationships**
- Draw **associations** between classes, complete with **multiplicities** and **roles**
- Describe the difference between **aggregation** and **composition**, and the implications when writing code
- Define an **interface** and contrast with a base class
- Given a system, draw a class diagram to describe all the relationships

Class Diagrams

Give an **overview** of the classes that need to be created, including **important** member **variables** and **functions**, and the relationships between classes.



Class Diagram: Attributes



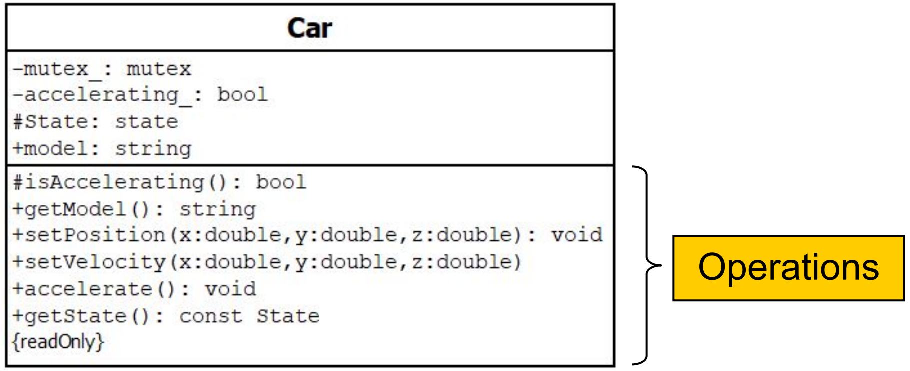
Attribute Syntax:

visibility name: type multiplicity = default {properties}

- **visibility**: public (+), protected (#), or private (-)
- **name**: attribute name
- **type**: type or class of the attribute
- **multiplicity**: number of elements (e.g. size of an array)
- **default**: default value if isn't specified in constructor
- **properties**: any additional properties

- Each attribute will correspond to a **member variable** in code
- Only the **name** is necessary
- Types can be explicit classes (language-specific) or more generic names (i.e. "number") as long as the **intention** is clear
- Include attributes only if they are **important** and you want to explicitly specify their names

Class Diagram: Operations



Operation Syntax:

visibility name (parameters) : return-type {properties}

- **visibility**: public (+), protected (#), or private (-)
- **name**: attribute name
- **parameters**: method arguments

  direction name: type = default
  - **direction:** in, out, or inout
- **return-type**: return type or class of the operation
- **properties**: any additional properties

- Each operation will correspond to a **member function** in code
- Include operations only if they are **important** for interaction with other classes
- No need to specify getters/setters, they are implicit unless you want to clarify them (e.g. the return value is read-only)

Class Relationships

UML defines three class **relationships**: association, encapsulation, and generalization.

Association: describes a *makes-use-of* or *calls-upon* relationship, implies that one class contains a reference to and can send messages to another.

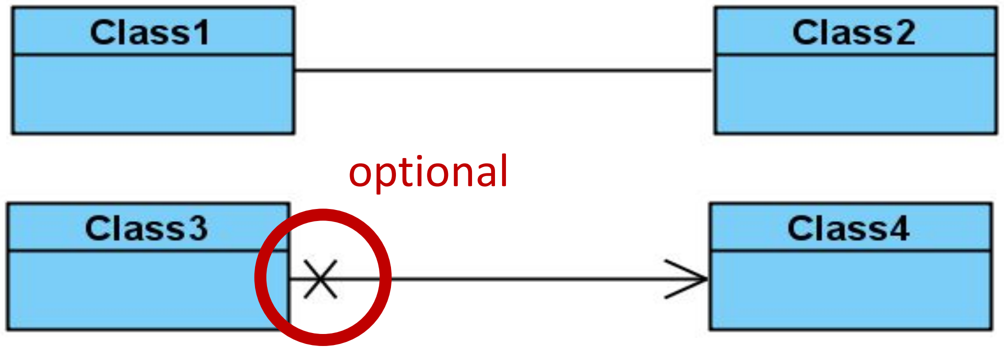
Encapsulation: captures the *part-of* or *belongs-to* relationship. It implies that one class owns or controls another. There are two 'types': **aggregation** and **composition**.

Generalization: describes the *kind-of* relationship, whether a class can be **substituted** for or **inherits** from another.

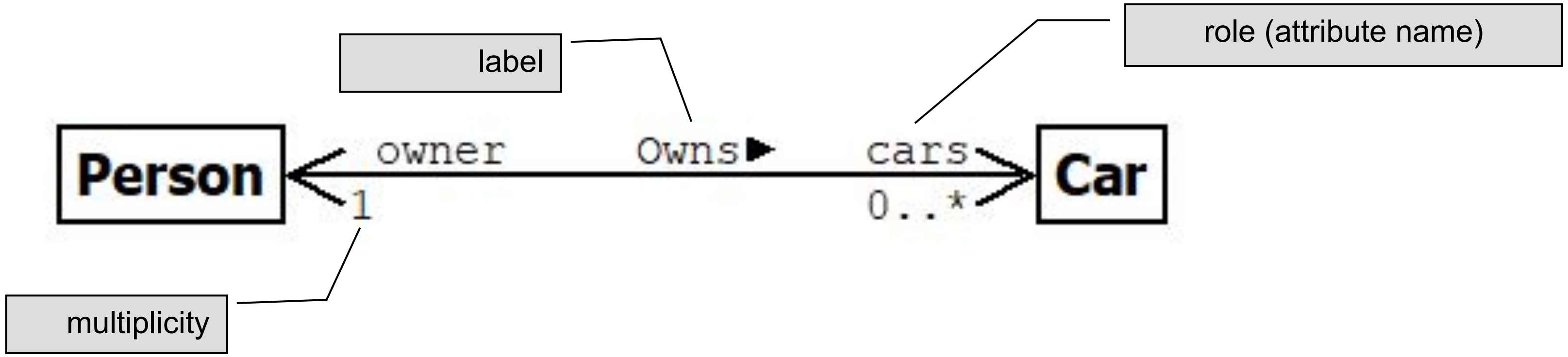
Class Associations

Bidirectional:

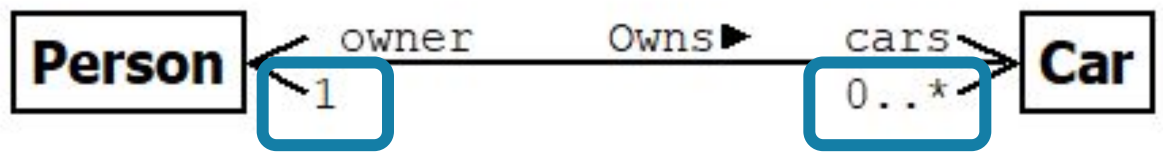
Unidirectional:



Associations imply that one class has the ability to send messages to another. This may be through a member variable, or through a member function.



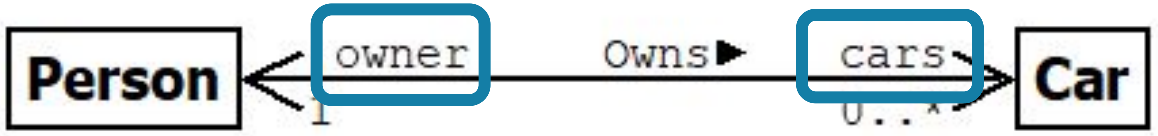
Association: Multiplicity



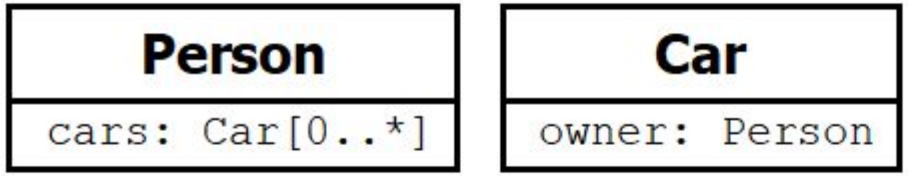
Number of **unique** associates. May need to check for prior existence when adding new association. If variable multiplicity, may need to check existence before usage internally.

0..1	Could be 0 or 1 objects (**caution**: could be 0).
0..*	Could be 0 or more objects (**caution**: could be 0).
*	Unknown many (**caution**: could be 0).
1	Guaranteed **always** to be exactly 1 object.
n	Guaranteed **always** to be exactly ‘n’ objects (e.g. 5).
1..*	Unknown many but **at least** 1 and maybe more objects.

Association: Roles



The **role** identifies the attribute name in the association. They can use full attribute syntax (visibility, type, etc.). The above implies:

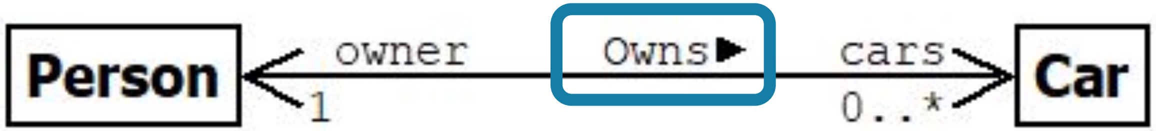


In fact, the two diagrams are **equivalent**.

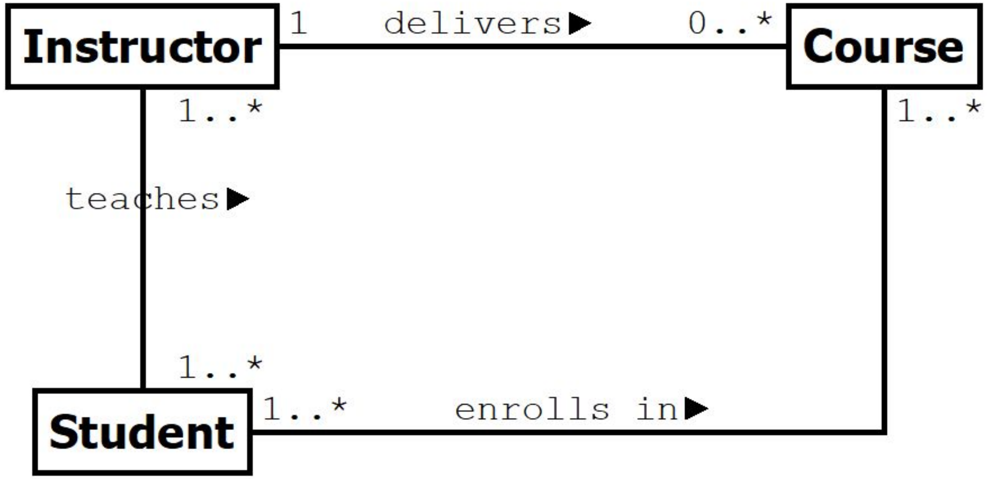
Associations with roles can **emphasize** the interaction.

Internal attributes are often used for primitives or classes not visible in the diagram. Associations are used to visually map out class relationships.

Association: Labels



The **label** or **name** aids in readability. It is usually a verb phrase to describe the relationship. It has no impact on code, only in improving clarity of the model.



Class Encapsulation

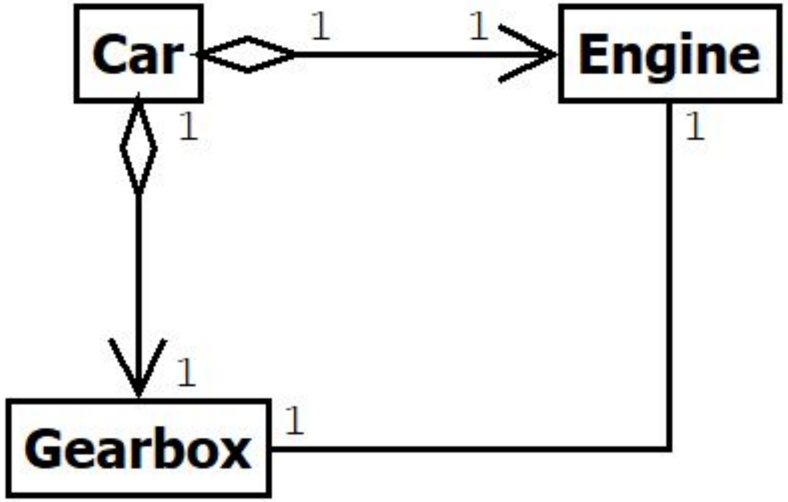
Used when an association is **so strong** that an association does not capture the the **essense** of the relationship.

- A *CPU* is **part of** a *Computer*
- The *Gearbox* and *Engine* are **part of** a *Car*
- A *Room* is **part of** a *Building*
- A *Student ID* **belongs to** a *Student*

Encapsulation often has implications when it comes to memory management. The “owner” of an object is often tasked with creating and destroying it.

Encapsulation: Aggregation

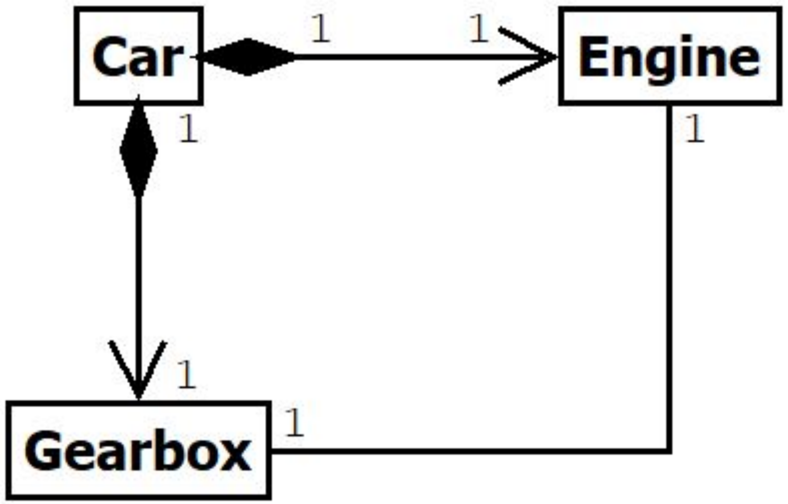
Aggregation implies that ownership is **transitory** – i.e. the object is being "**borrowed**". The object can be detached from the owner and given to another one at run-time.



The lifetime of the owner and object are not necessarily tied together. Usually means class has add/remove methods for borrowed objects.

Encapsulation: Composition

Composition implies a stronger **ownership** than aggregation: the owner and parts have the same **lifetime**, implying the owner is responsible for creation/deletion of resource.



Implies parts cannot be detached from owner, and ownership cannot be shared (i.e. multiplicity at owner size is 1).

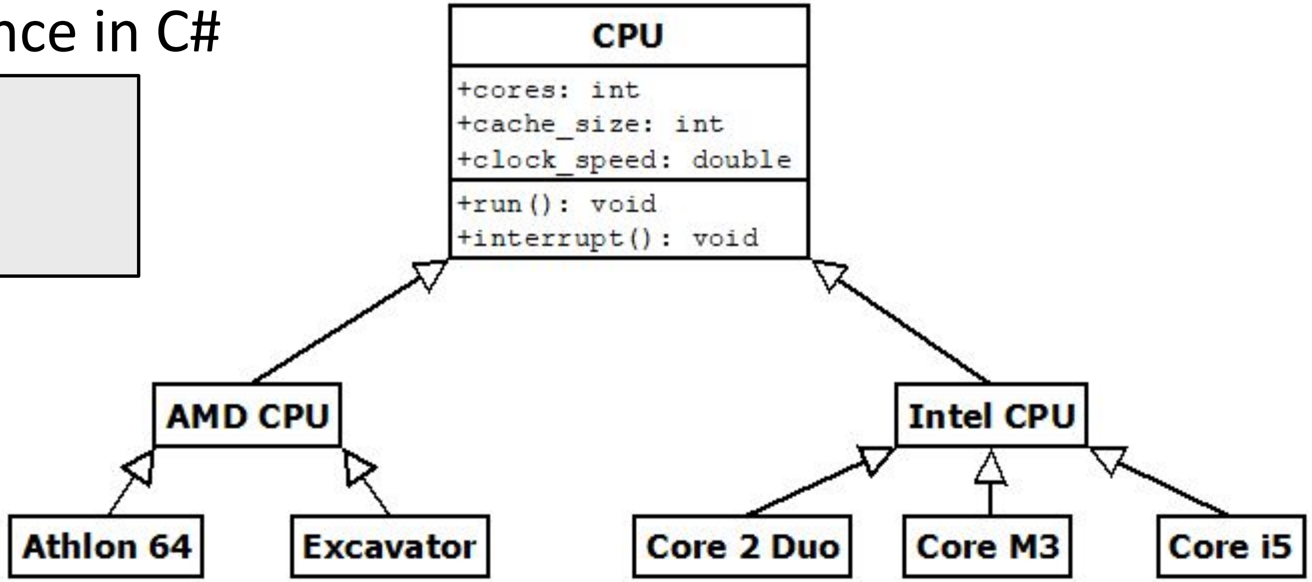
Class Generalization

Describe the **kind-of** or **substitutes-for** relationship. This implies two things:

- **Inheritance**: derived class inherits all attributes/operations of the base
- **Substitutability**: anywhere the base is used, the derived class can also be used

Usually implies class inheritance in C#

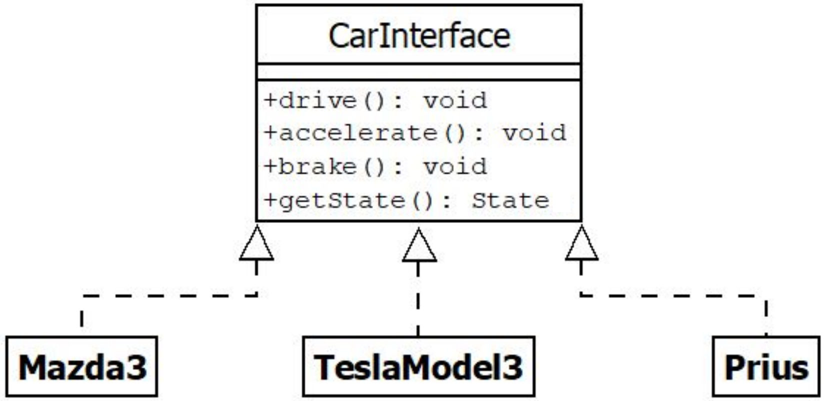
```
// Inheritance
class IntelCPU : CPU {
   // ...
};
```



Class Generalization: Interfaces

An **interface** describes the **contract** a class must abide by, but has no explicit inheritance of attributes or operations. An **implementation** class *implements* the interface.

```
interface ICar{
  void drive();
  void accelerate();
  void brake();
  State getState();
}
// Only pure virtual member functions
public class Car: ICar {
  private on {get;set;}
  void drive();
  void accelerate() = 0;
  void brake() = 0;
  public State getState(Car car){
      State state = new State();
      state.on = car.on;
      return state;
  }
};
```



Class Diagrams Summary