FIT5SE1 Software Engineering 1

Lectures 1 & 2:
Type hierarchy

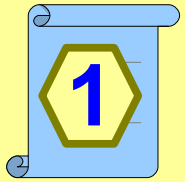
Outline

- Type hierarchy: why & what?
- Type hierarchy features

Lecture 1

- Create a type hierarchy
- Subtypes with attributes
- Abstract class (overview)
- Multiple implementations
- Dispatching

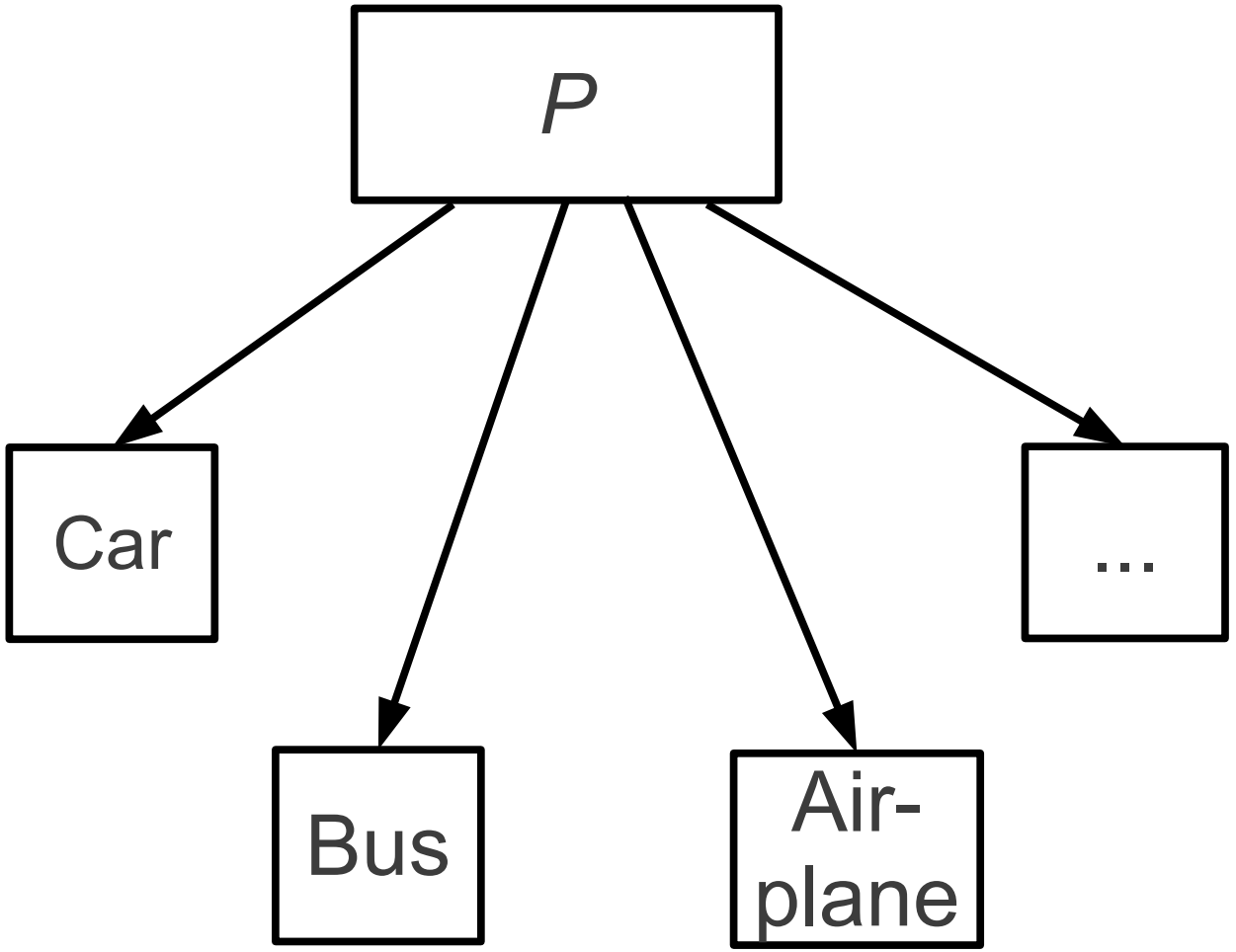
Lecture 2



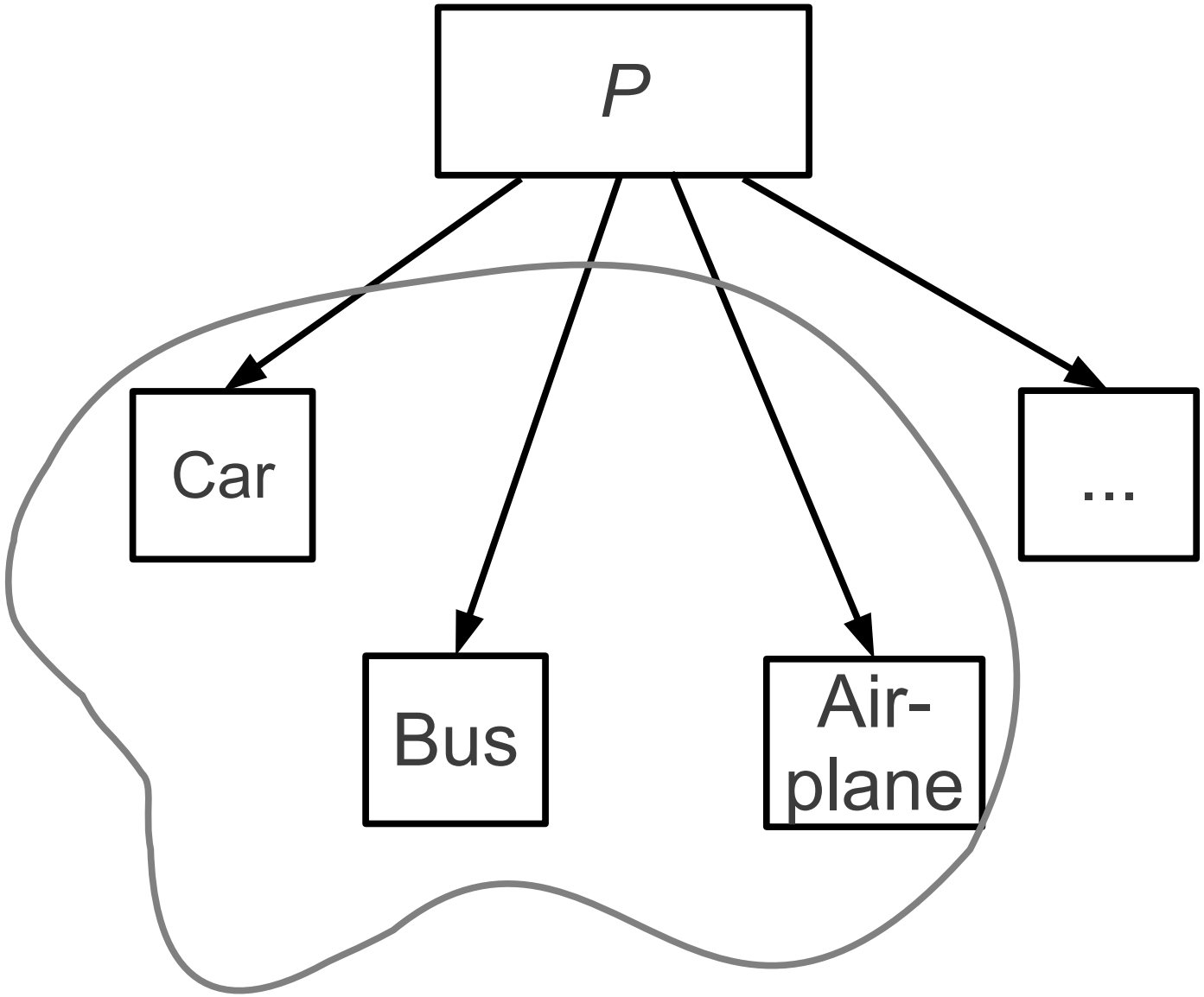
Why type hierarchy?

- Similarities exist among types that require a higher level of abstraction...

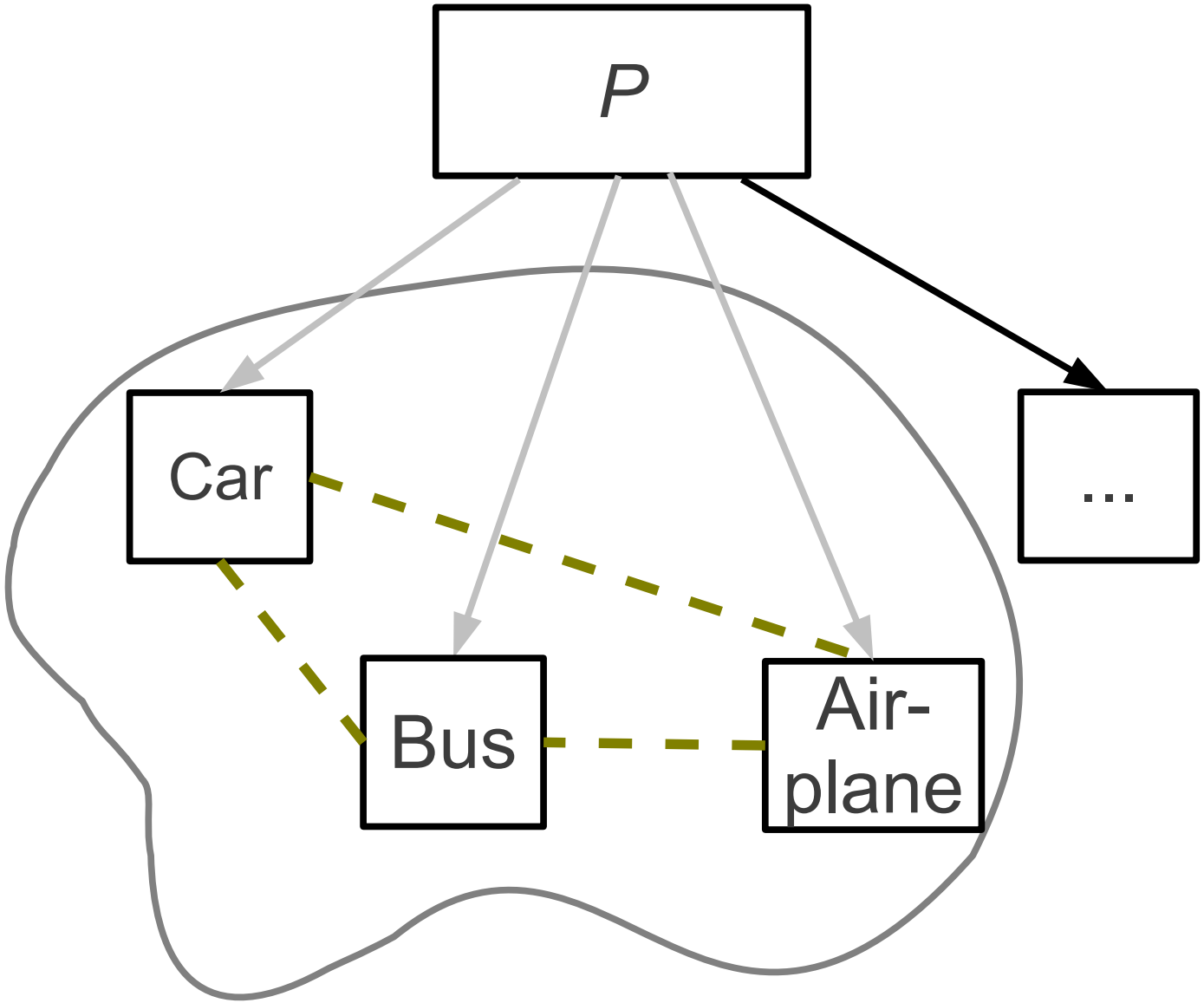
Example: vehicles



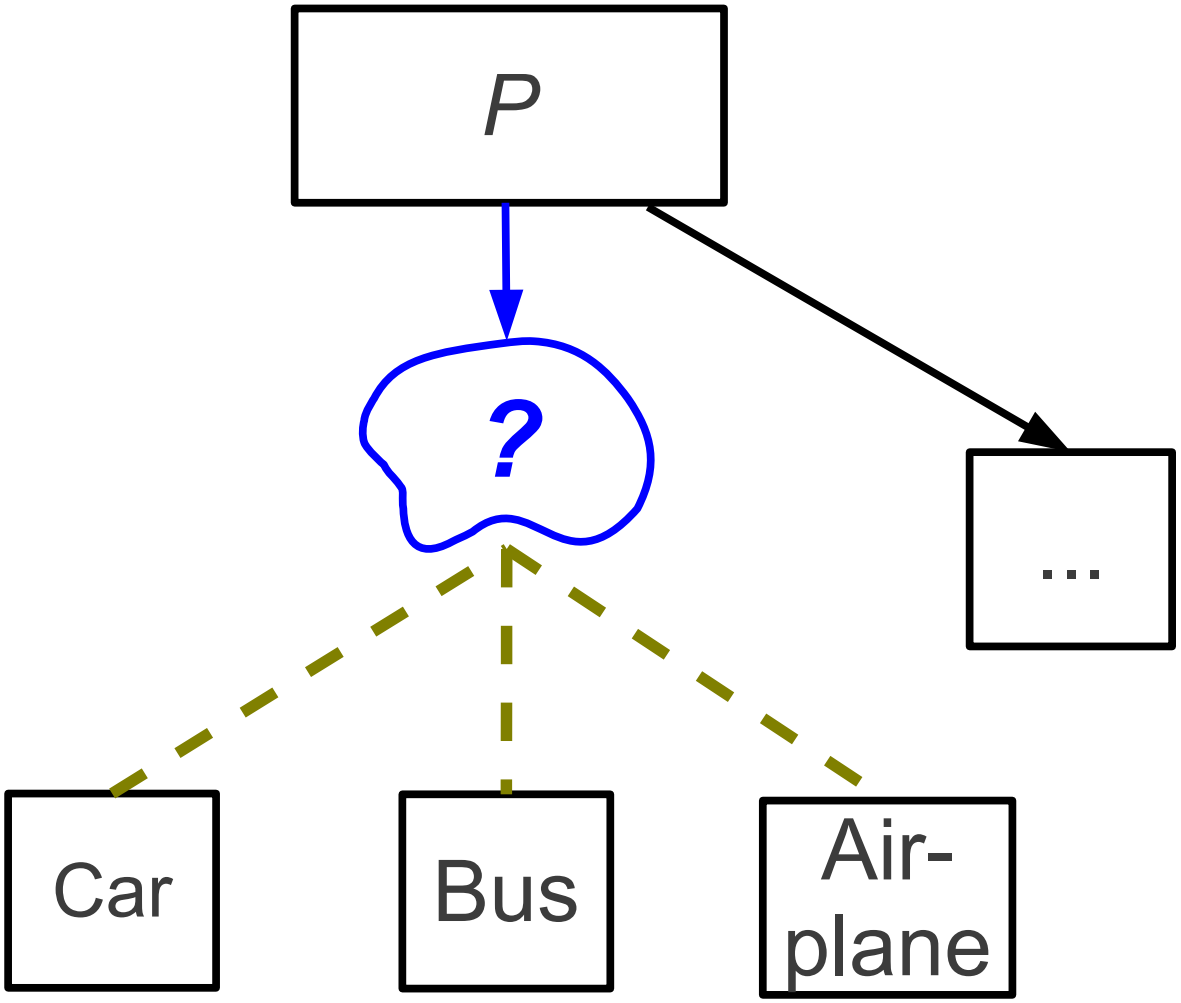
Example: vehicles (2)



Example: vehicles (3)



Example: vehicles (4)



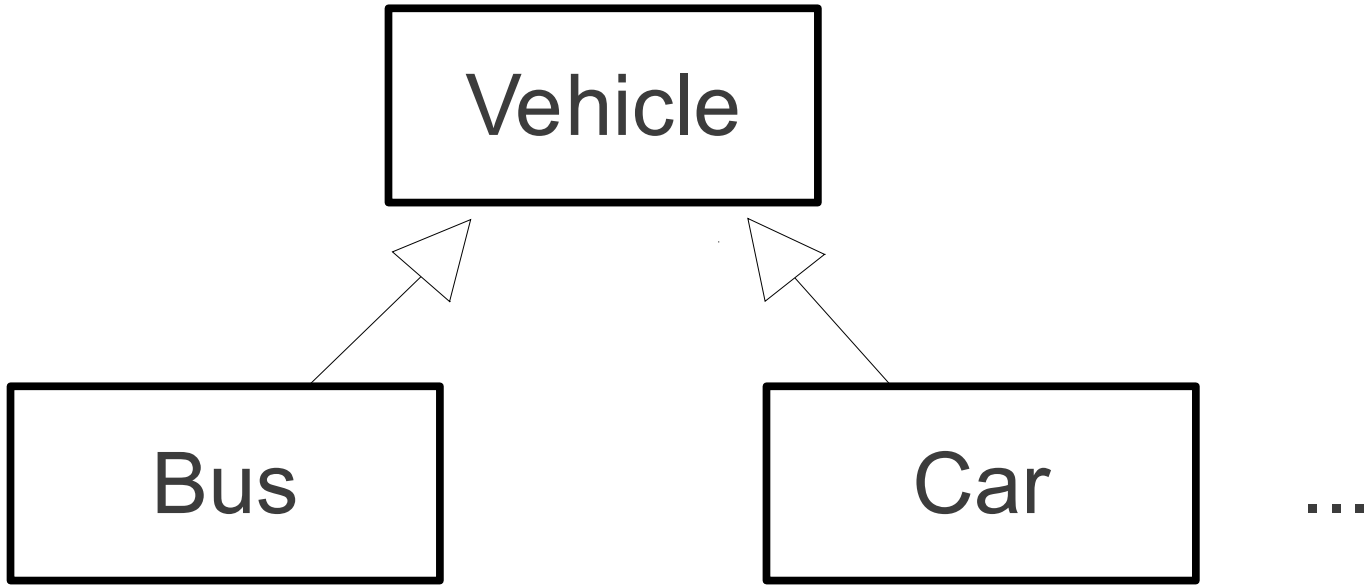
What is a type hierarchy?

- A product of *type abstraction*
- A hierarchy of types in which higher-level types are abstractions of lower-level ones
  - a higher-level type is a ***super-type (supertype)***
  - a lower-level type is a ***sub-type (subtype)***

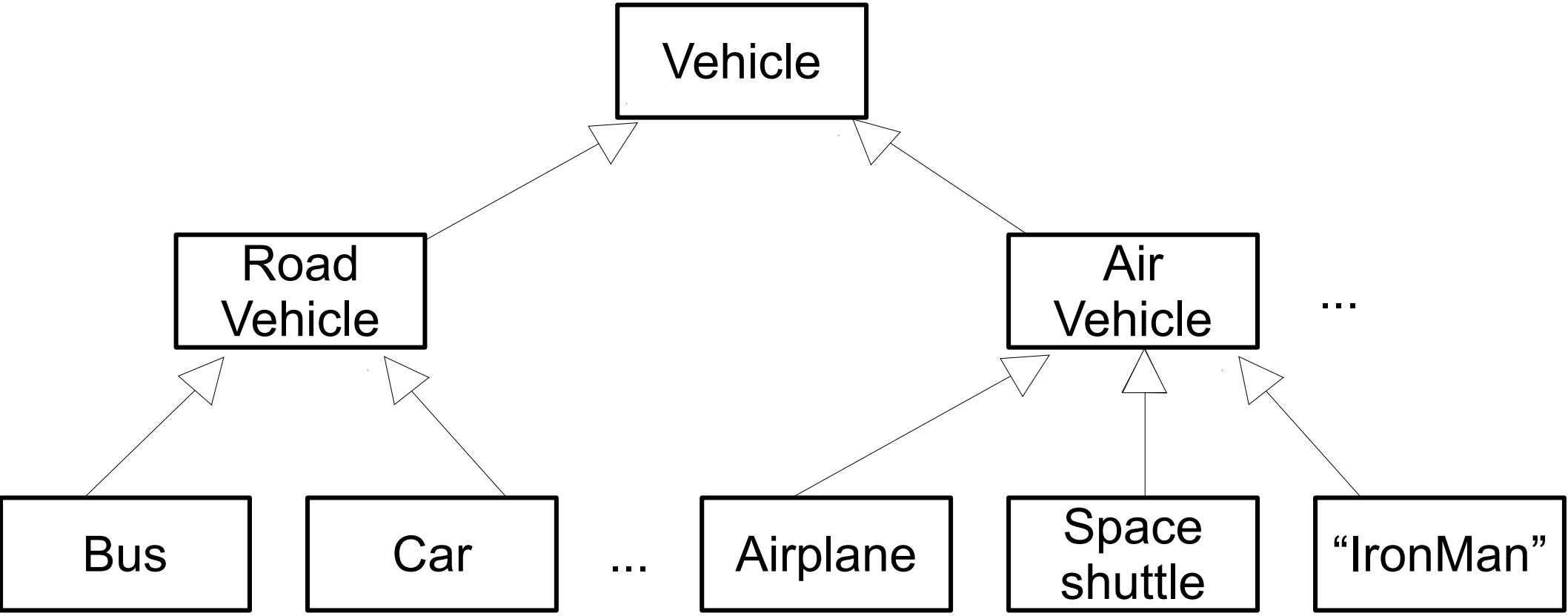
Benefits

- Enhance ability to solve real world problems
- Program modifiability:
  - multiple implementations of a type

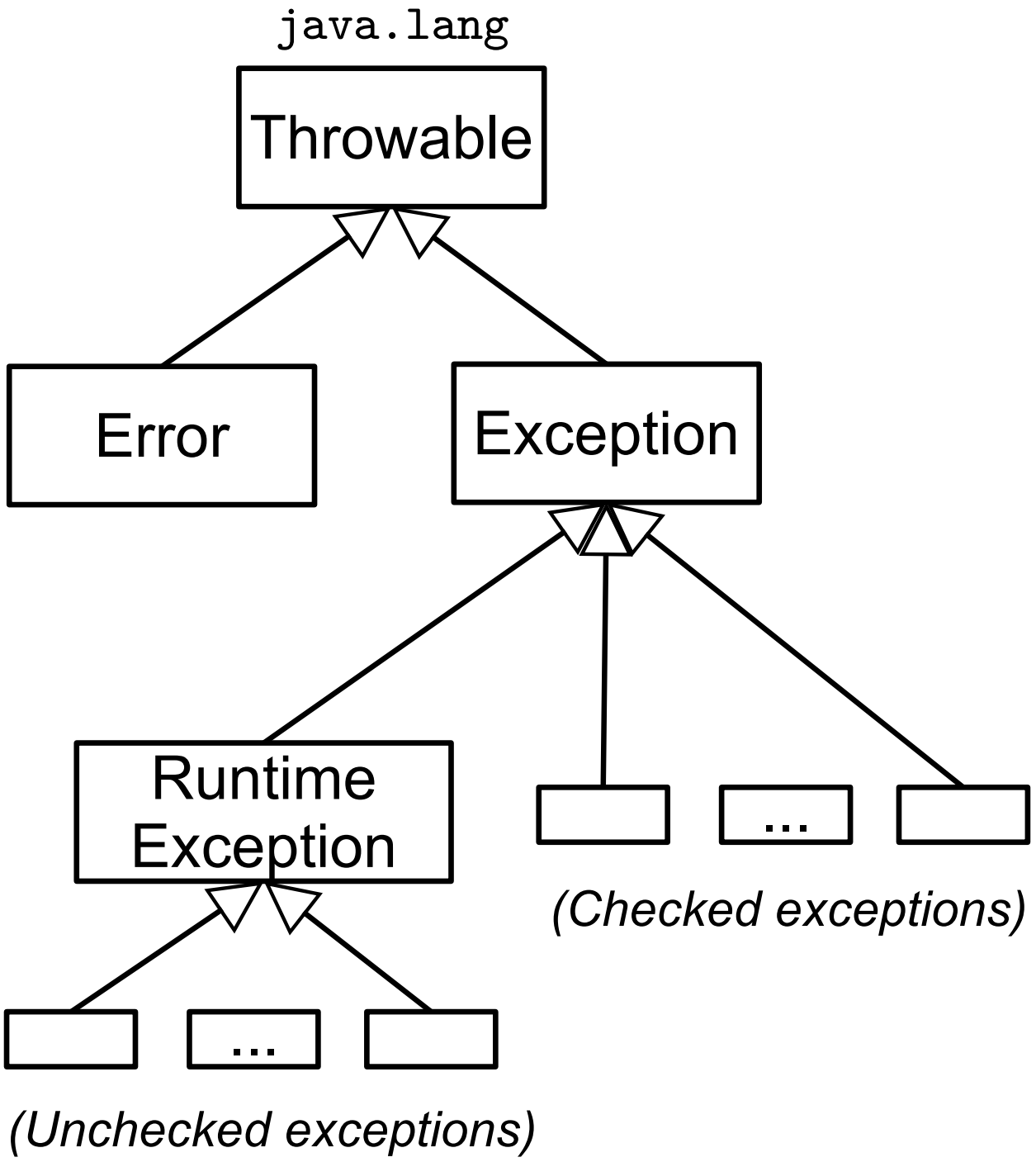
One-level TH example: vehicles



Two-level TH: vehicles



Multi-level TH: exceptions

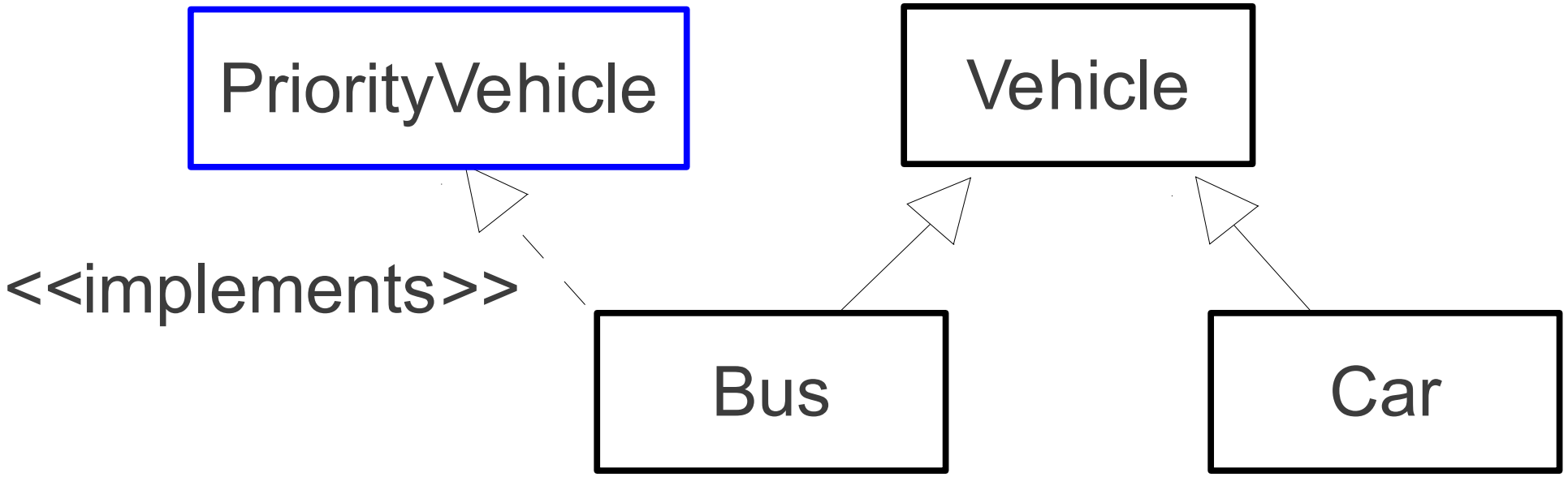


What about multiple super types?

- A subtype can have more than one supertypes
- In Java:
  - only one super type is class, others must be **interfaces**
  - class: specification and code
  - interface: specification only

Example

- Interface `PriorityVehicle` represents vehicles with priorities

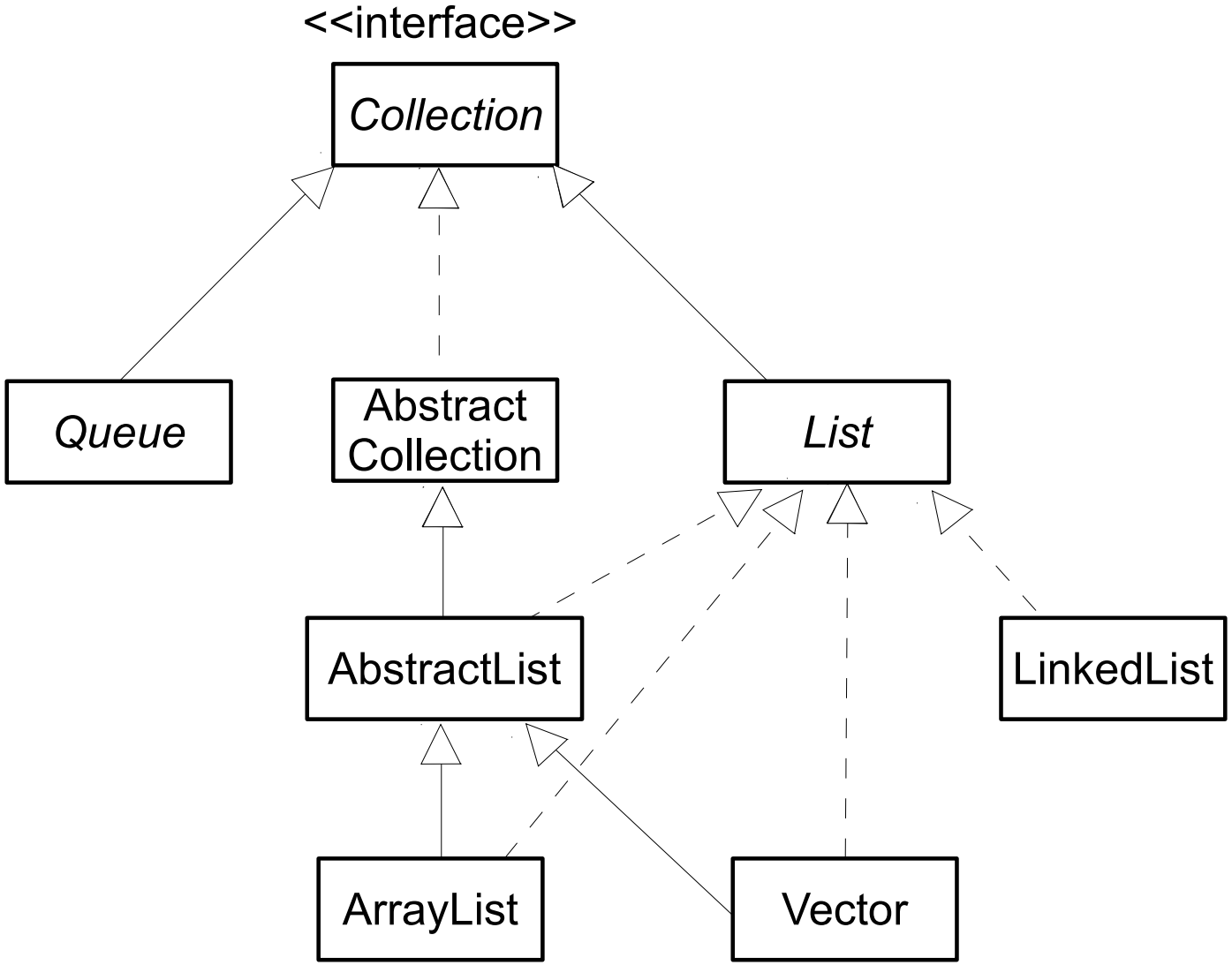


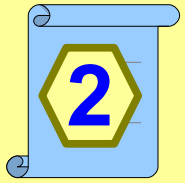
Example: List TH

- List is a sequence of elements
- Two basic orders:
  - insertion
  - sorted: ascending or descending
- Java interface: `java.util.List`
- Two subtypes:
  - `ArrayList`
  - `LinkedList`

List TH

- Includes both classes and interfaces





Type hierarchy features

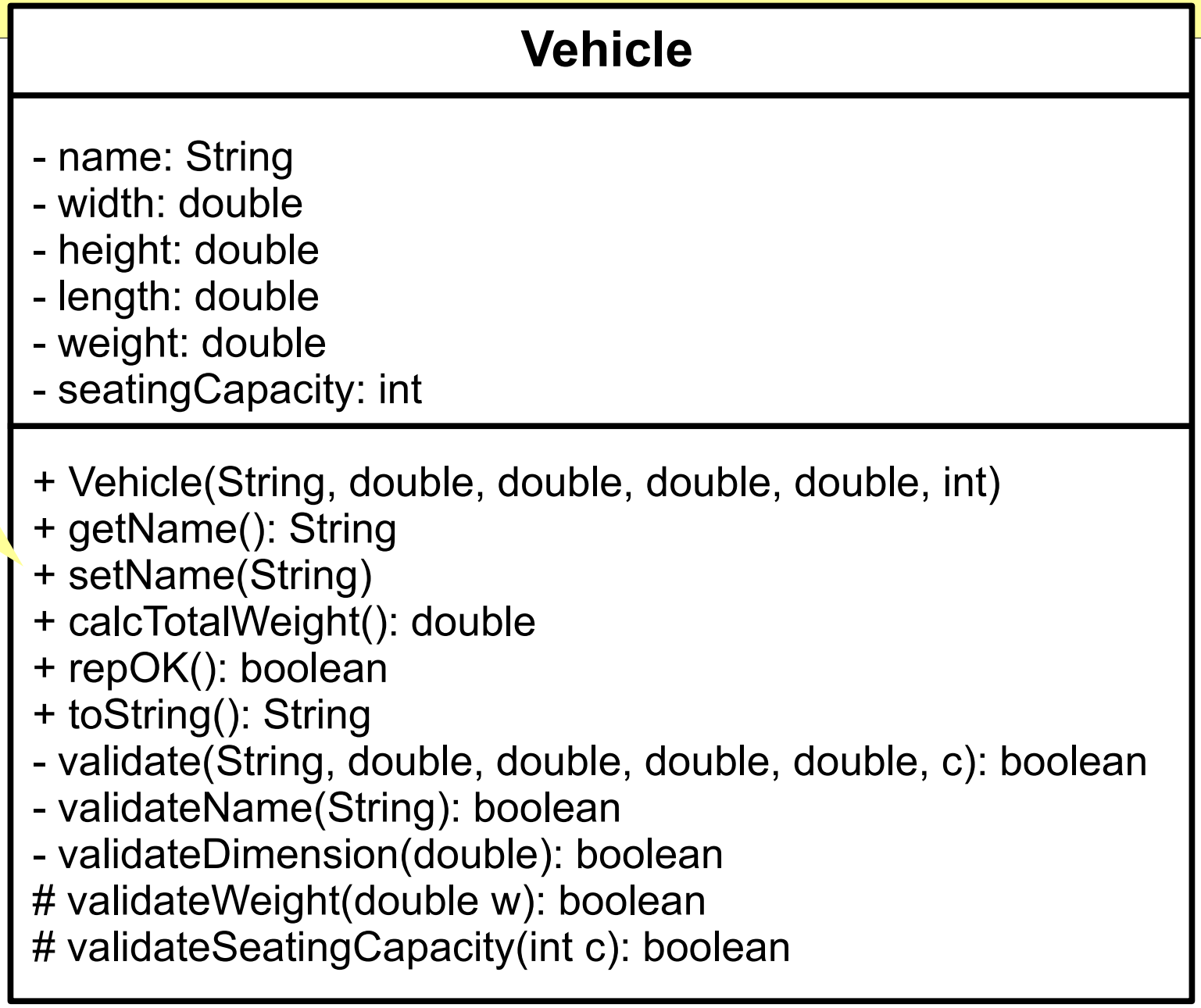
- Inheritance
- Subtypes with more specialised abstract properties
- Subtypes typically *override* certain supertype's behaviour
  - abstraction by specification
- Subtypes can have new attributes
- Subtypes can have new behaviour

Inheritance

- Subtypes inherit attributes and operations of the supertype and all ancestors (except constructors):
  - benefit: code re-use
- Sub-types must define constructors that they wish to use:
  - but must invoke suitable supertype constructor(s) if not the default
- Objects of the subtypes must not violate properties associated to the attributes:
  - see properties rule later

Example: Vehicle

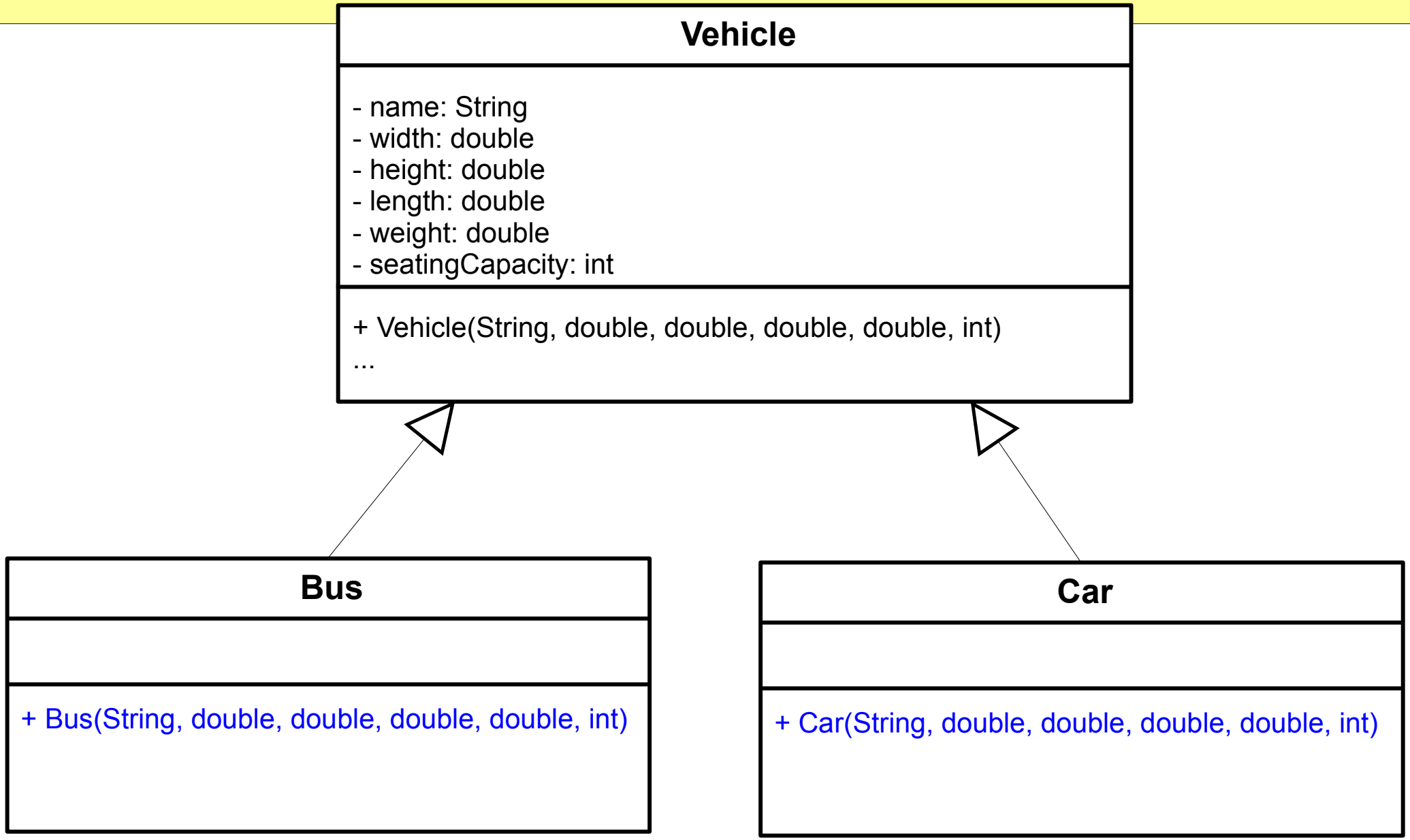
Setters/ getters of other attributes are omitted



`Vehicle`'s abstract properties

Attributes	Formal type	Mutable	Optional	Min	Max	Length
`name`	String	T	F	-	-	100
`width`	Double	T	F	0+	-	-
`height`	Double	T	F	0+	-	-
`length`	Double	T	F	0+	-	-
`weight`	Double	T	F	0+	-	-
`seating Capacity`	Integer	T	F	0+	-	-

Bus and Car inherit Vehicle



Subtypes with specialised abstract properties

- A subtype can have more "restricted" properties concerning one or more attributes that it inherits
- Example:
  - `Bus` and `Car` both have tighter restrictions on attributes `weight` and `seatingCapacity`

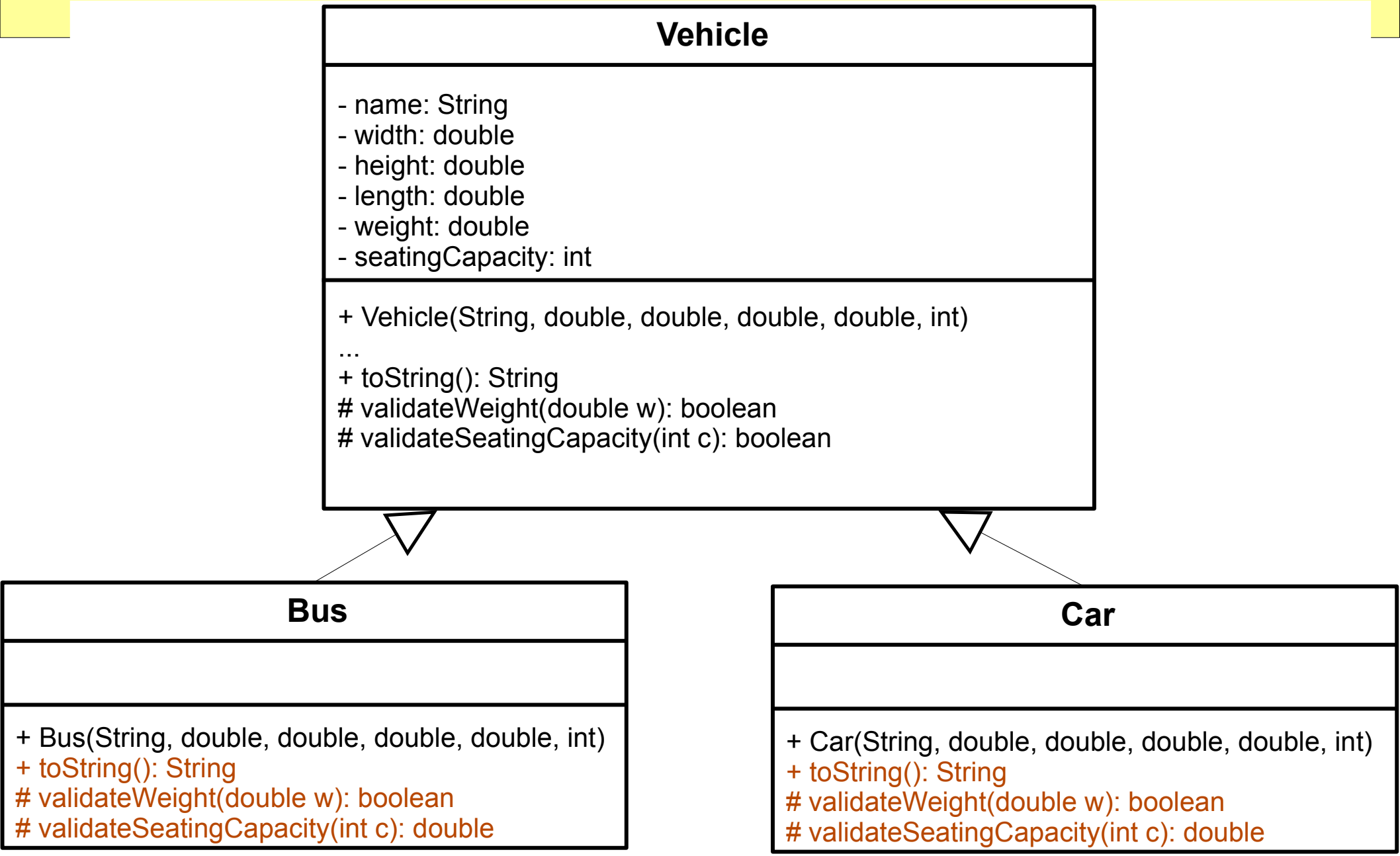
Example: Bus's & Car's restrictions on weight

Attributes	Formal type	Mutable	Optional	Min	Max	Length
name	String	T	F	-	-	100
...	...	...	...	...	...	...
weight	Double	T	F	for Vehicle		
				0+	-	-
				for Bus		
				5000	-	-
				for Car		
				-	2000	-
...	...	...	...	...	...	...

Operation/Method overriding

- When to override a method in a subtype?
- To take into account:
  - subtype's type information (e.g. type name)
  - subtype's abstract properties
  - subtype's behaviour
- Example:
  - `Bus` and `Car` have specialised properties concerning weight and seating capacity
  - `Bus` and `Car` have different engine-ignition behaviours

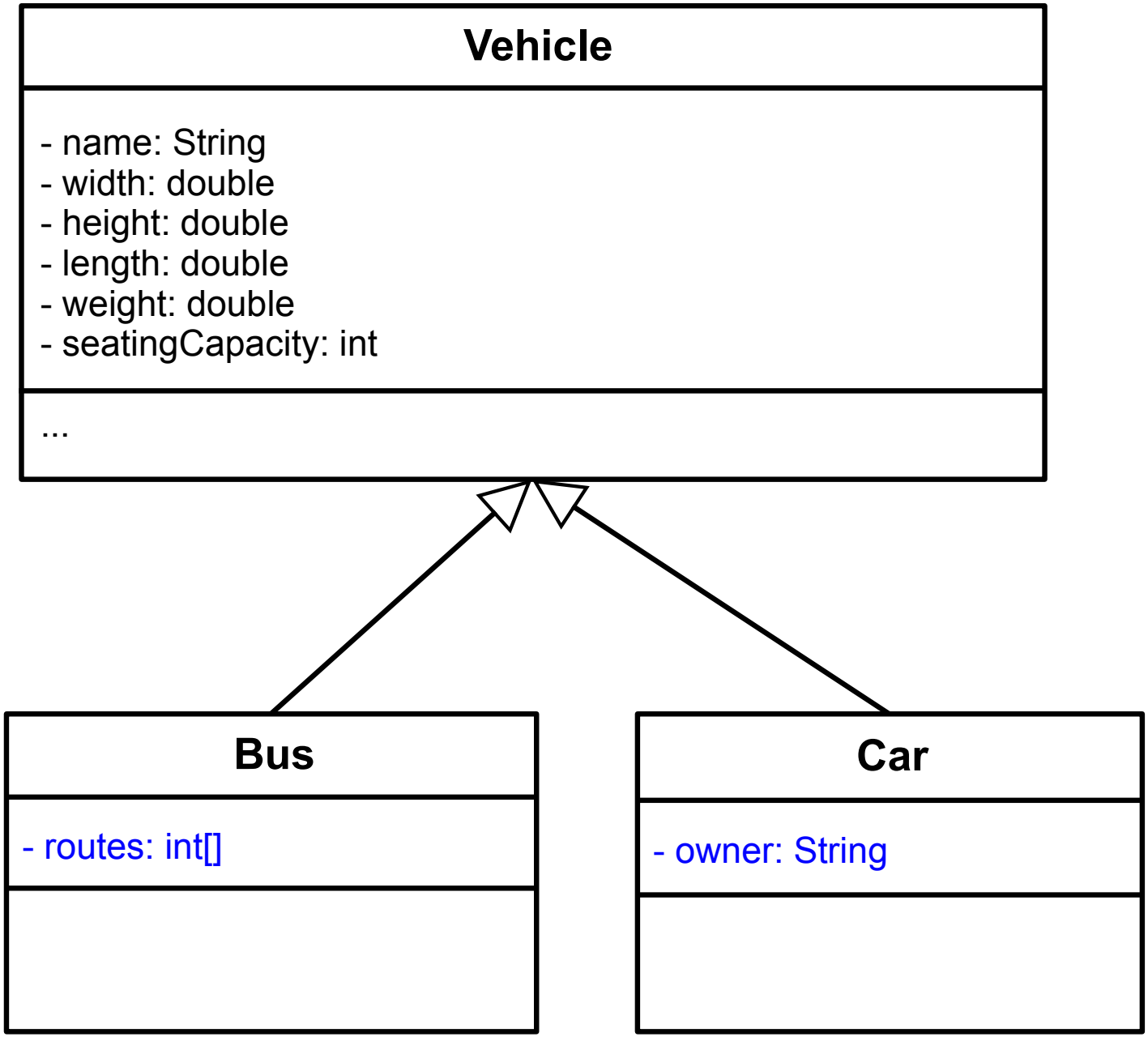
Vehicle TH: overriding methods



Subtype with additional attributes

- A subtype can have additional attributes that are specific to it
- These attributes would require adding new operations
- Example:
  - `Bus`: has routes
  - `Car`: has owner name

Example: Vehicle TH



Subtype with additional behaviour

- Subtype can have additional operations that serve it's specific purpose
- These operations *may* be related to additional attributes that it has
- Example:
  - `Car.openTheTrunk()`:
    - open the cargo trunk at the back of the car
  - `Bus.raiseStopBell()`:
    - (for passenger) to request the bus to stop at the next station

The meaning of subtype: substitution principle

- Substitution principle: "supertype can be used in place of its subtypes"
- That is, objects of a subtype can be assigned to a variable declared with the supertype:
  - supertype is the *apparent* type of the variable
  - subtype is the *actual* or *run-time* type of the variable

Example: Substitution principle

```
// create objects
Vehicle v = new Bus("b1",3.0,3.0,10.0,6000,40);
// use objects
System.out.println("Vehicle " + v.getName() +
 ", weight: " + v.calcTotalWeight());




// some time later...
v = new Car("c1",1.5,1.5,2.5,1500,4);
```

super type variables are assigned to subtype objects

Create a type hierarchy

- Specify
- Implement

Specify supertype & subtypes

- Specify a supertype with common behaviour
- Specify each subtype *relative* to the supertype:
  - (if needed) specialise the abstract properties based on those of supertype
  - use `extends` or `implements` keyword
  - specify new or overriding behaviour
  - (if needed) specify new attributes
- Annotate overriding operations with `@Override`

Class/interface rules

- Supertype/subtype → `class` or `interface`
- `Object` is the (root) supertype of all types
  - need not be specified
- Interface only has specifications
- Interface can only be a subtype of another interface
- Class can be a subtype of:
  - one class and/or
  - multiple interfaces

Specialise the abstract properties

- Given a supertype named *Super* and an attribute *A*, the following is a specialisation of the abstract properties of *A* in a subtype:

$$P\_Super.A \quad \wedge \quad F(A)$$

Super's property on attribute *A* (inherited)

Subtype's further restriction on *A*

Example: Bus's restriction on weight

- `P_Vehicle.weight` $\wedge$ **min**(`weight`) = 5000

`Vehicle`'s property on `weight` (inherited)

`Bus`'s further restriction on `weight`

Car's restriction on weight

- `P_Vehicle.weight` $\wedge$ **max**(`weight`) = 2000

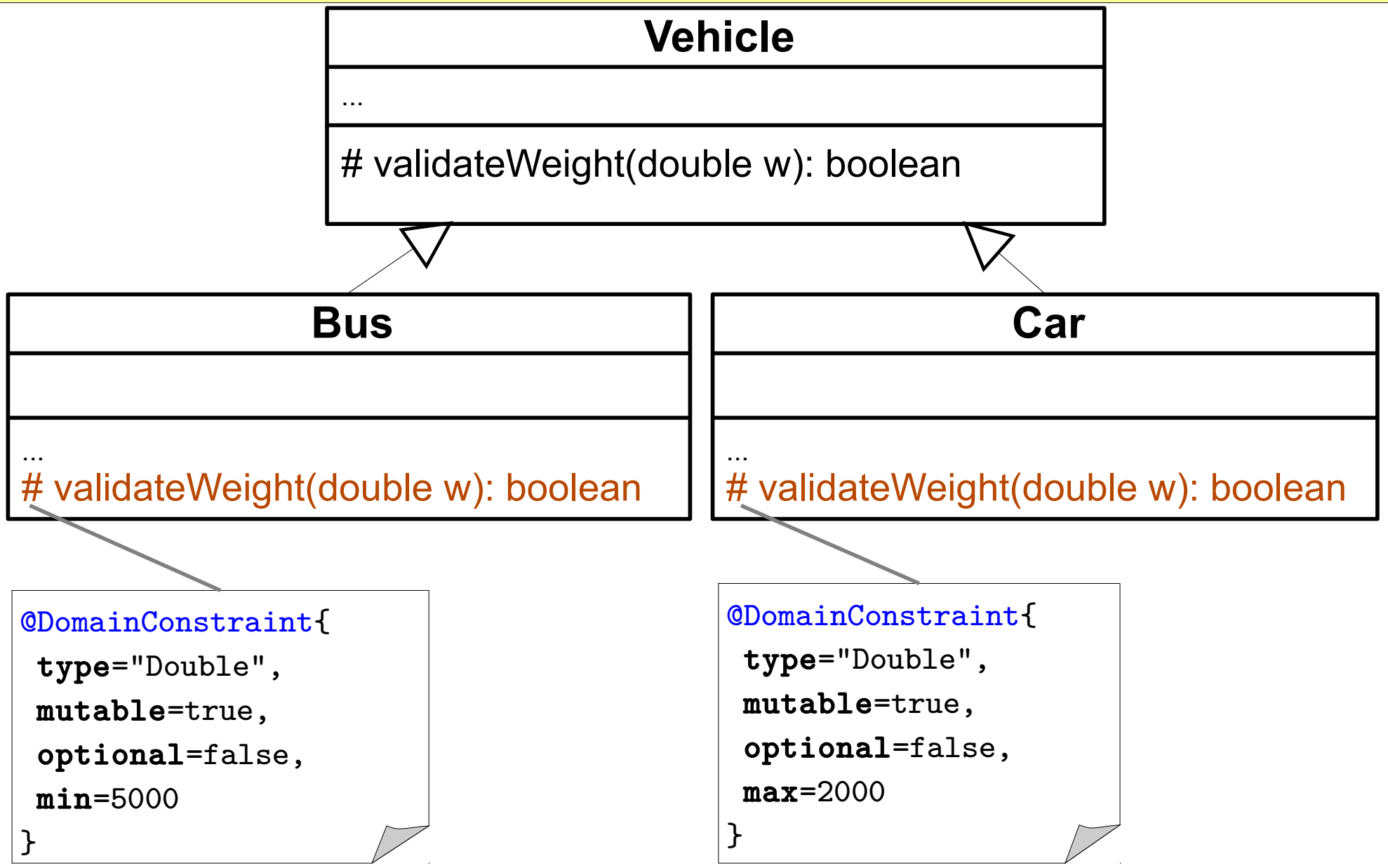
`Vehicle`'s property on `weight` (inherited)

`Car`'s further restriction on `weight`

Using `DomainConstraint` to realise property specialisation

- We can specify in a subtype a `DomainConstraint` for a property specialisation
- But NOT in the usual way (that is to attach it to an attribute):
  - Why? because the attribute is not available in the subtype!
- The solution involves two parts:
  - define an overriding method in the subtype that overrides a supertype's method concerning the attribute (e.g. data validation or observer method)
  - attach a `DomainConstraint` to this overriding method

Example: `validateWeight`



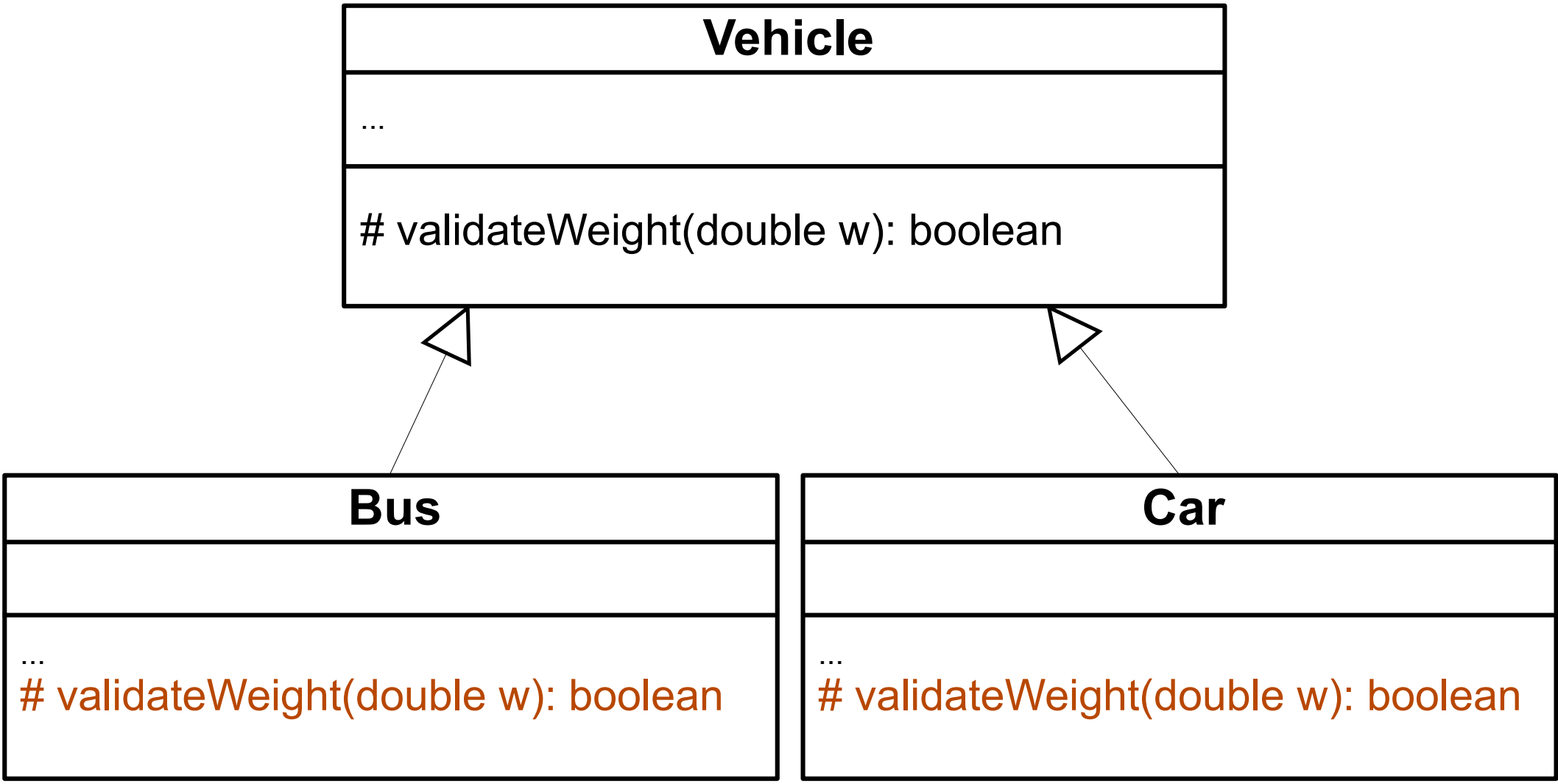
Specify the overriding methods

- An *overriding* method in the subtype must satisfy two rules w.r.t *overriden* method:
  - header rule
  - methods rule

Header rule

- Overriding method must be *header compatible* with the overriden method
- Method header includes:
  - signature: method name, number and types of parameters (also means their order)
  - return type
  - thrown exceptions: (details next lecture)
- Compatibility means:
  - same signature
  - return type: same (Jdk < 1.4) or subtype (>= 1.5)
  - exceptions: (details next lecture)

Example: validateWeight



What about these methods?

```
+ validateWeight(float w): boolean

+ validateWeight(double w): int

+ validateW(double w): boolean

+ validateWeight(double w)

+ validateWeight(): boolean
```

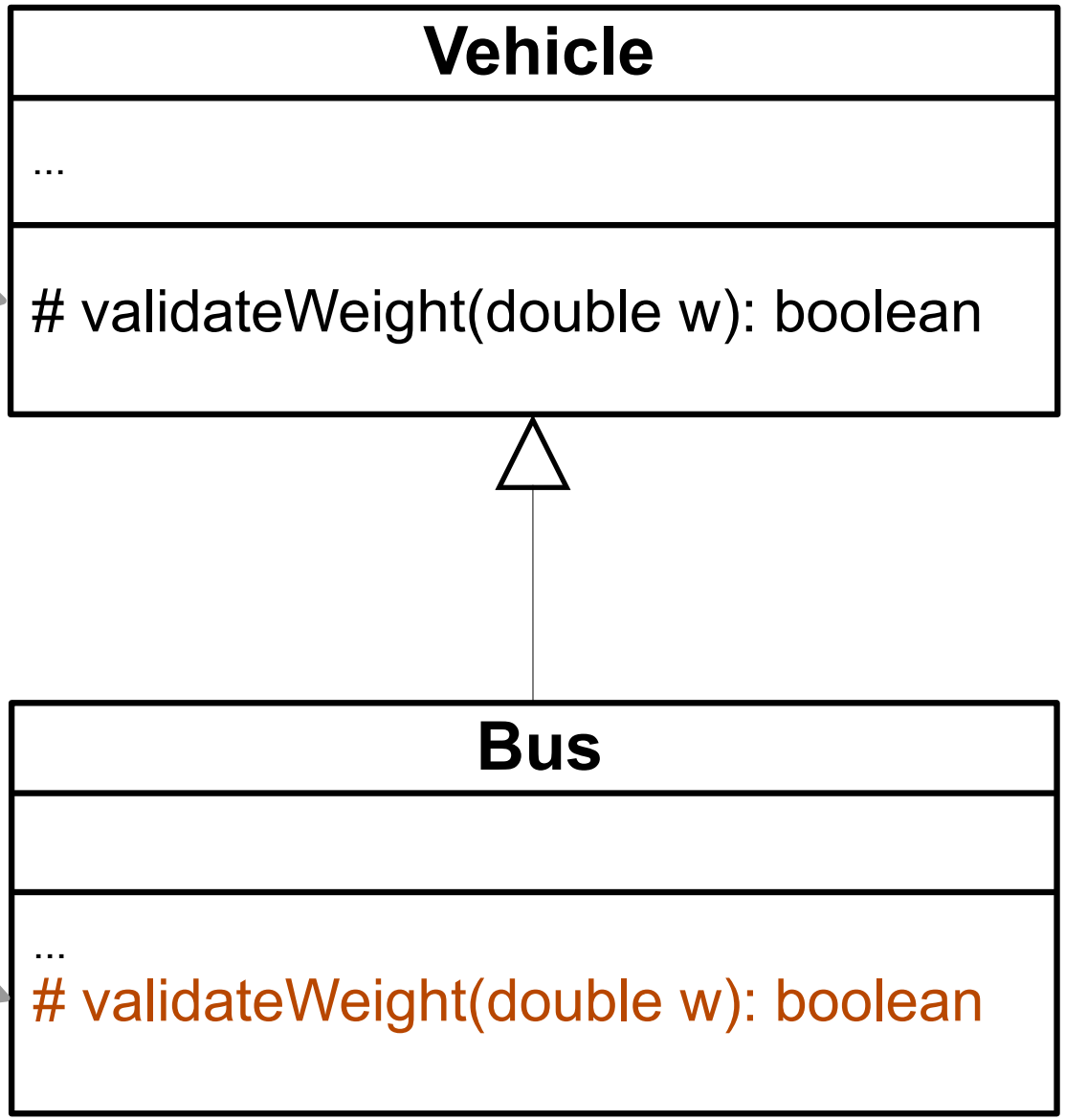
Are these correct overriding methods 

Methods rule

- Pre-condition (@requires) is the same or *weaken*:
  - $Pre_{super} \rightarrow Pre_{sub}$
- Post-condition (@effects) is the same or *strengthen*:
  - $(Pre_{super} \wedge Post_{sub}) \rightarrow Post_{super}$

Example: Bus.validateWeight (1)

```
/**
 * @effects
 *    if w is valid
 *      return true
 *    else
 *      return false
 */
```




```
/**
 * @effects
 *    if w is valid
 *      return true
 *    else
 *      return false
 */
```

Vehicle and Bus properties w.r.t weight

- Vehicle properties w.r.t weight (**P_Vehicle.weight**):

mutable(weight)=true ⋀

optional(weight)=false ⋀

min(weight)=0+

- Bus properties w.r.t weight:

P_Vehicle.weight ⋀ **min**(weight) = 5000

Example: `Bus.validateWeight` (2)

- $Pre_{Vehicle} \rightarrow Pre_{Bus}$:

true because both are empty □

- $(Pre_{Vehicle} \wedge Post_{Bus}) \rightarrow Post_{Vehicle}$:

$Post_{Vehicle}$ = P_Vehicle.weight.

$Pre_{Vehicle}$ = true.

$Post_{Bus}$ = P_Vehicle.weight $\wedge$ min(weight)=5000

$\rightarrow$ P_Vehicle.weight. □

Specification Example: `Vehicle`

`ch7.vehicles.`**`Vehicle`**

- Note:
  - property statements are easy to code directly
  - constant `DomainConstraint.ZERO_PLUS`
  - two validation methods are declared `protected`:
    - `validateWeight`
    - `validateSeatingCapacity`

Bus

ch7.vehicles.**Bus**

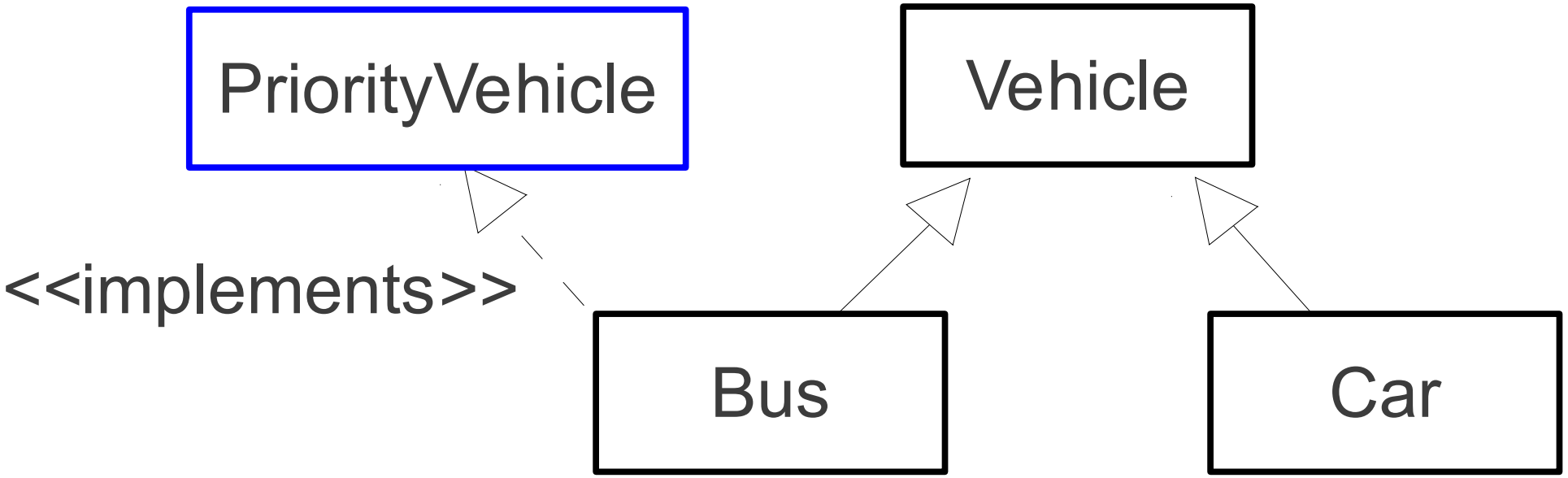
- Note:
  - `P_Vehicle`: abstract properties of `Vehicle`
  - abstract properties = `Vehicle`'s + two new constraints on `weight` and `seatingCapacity`
  - constructor is redefined (not inherited)
  - override two protected validation methods:
    - `validateWeight`
    - `validateSeatingCapacity`

Car

ch7.vehicles.**Car**

- Note:
  - `Car` is specified in a similar manner, except for the constraints on `weight` and `seatingCapacity`

Bus is a PriorityVehicle



PriorityVehicle

ch7.vehiclesintf

.**PriorityVehicle**

.**Bus**

- Note:
  - `Bus` uses the `implements` keyword

Qualities of subtype specification

- To conform to the substitution principle, a subtype specification must satisfy three substitution rules:
  - *header rule*
  - *methods rule*
  - ***properties*** *rule*
- Properties rule: subtypes must not violate the supertype's properties

Implement a TH in Java

- Keyword `super` refers to supertype's members
  - can access `protected` members of `super`
- Implementation can be full or partial
  - *abstract class* is partial (later)
- Overriding `repOK` must invoke `super.repOK`

Vehicle

`ch7.vehicles.`**`Vehicle`**

- Note:
  - `repOK` invokes `validate`
  - `toString` uses `Vehicle` prefix

Bus

ch7.vehicles.**Bus**

- Note:
  - constructor invokes `super` constructor
  - `toString` uses `Bus` prefix
  - validation methods check against the `min` values

Car

ch7.vehicles.**Car**

- Note:
  - constructor invokes `super` constructor
  - `toString` uses `Car` prefix
  - validation methods:
    - invoke `super`'s validation methods and
    - check against the `max` values

PriorityVehicle

ch7.vehiclesintf

.**PriorityVehicle**

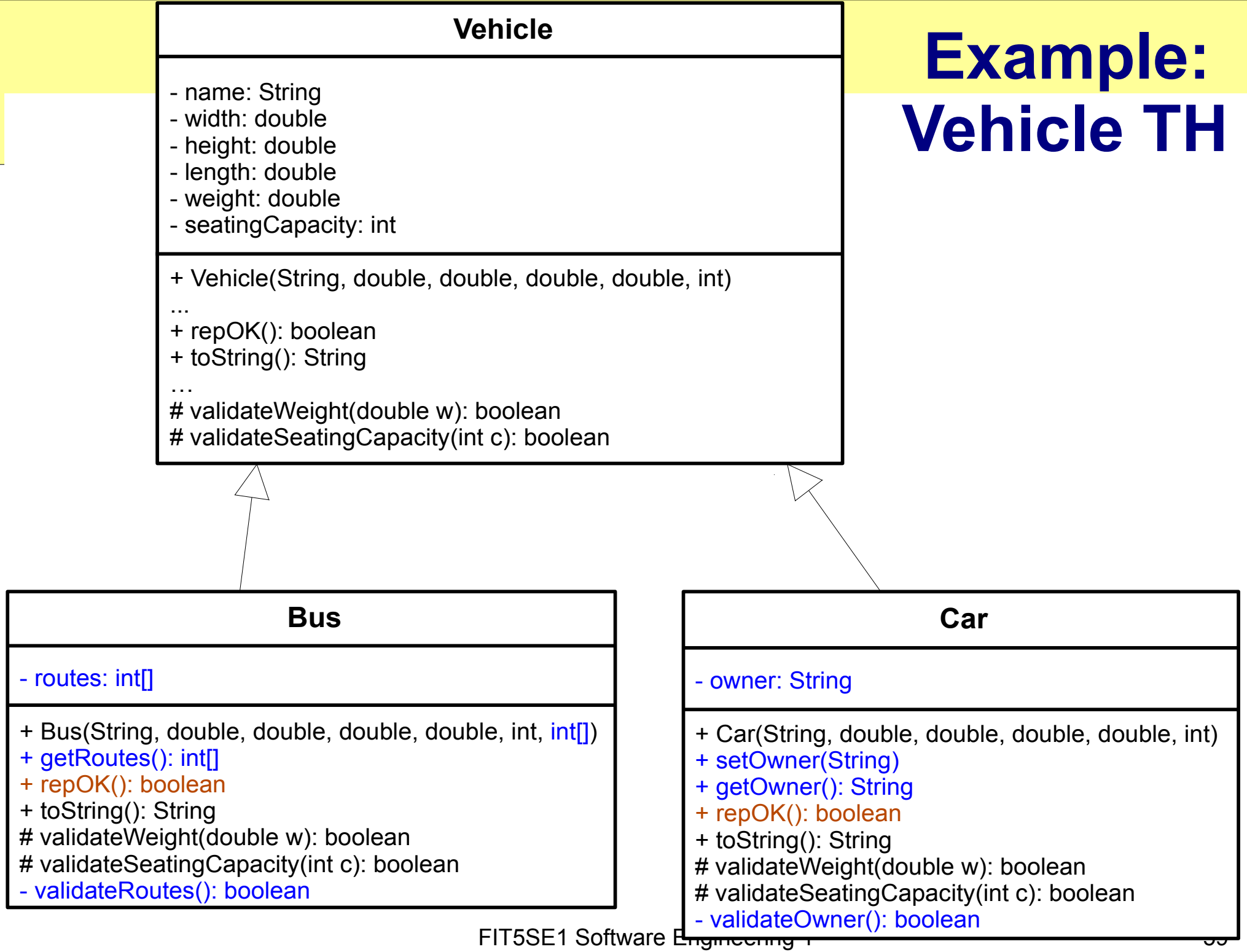
.**Bus**

- Note:
  - `Bus` uses the `implement` keyword
  - `comparePriorityTo` invokes other methods to get data

Subtypes with additional attributes

- Design specification of the subtype needs to take into account the additional attributes:
  - class header specification: attributes, abstract properties, abstraction function, rep invariant
  - constructors may need to take extra argument(s) (depending on domain constraint(s))
  - new operations may be needed, e.g. getter/setter
  - supertype's operations may need to be overriden

Example: Vehicle TH



Bus

ch7.vehiclesextra.**Bus**

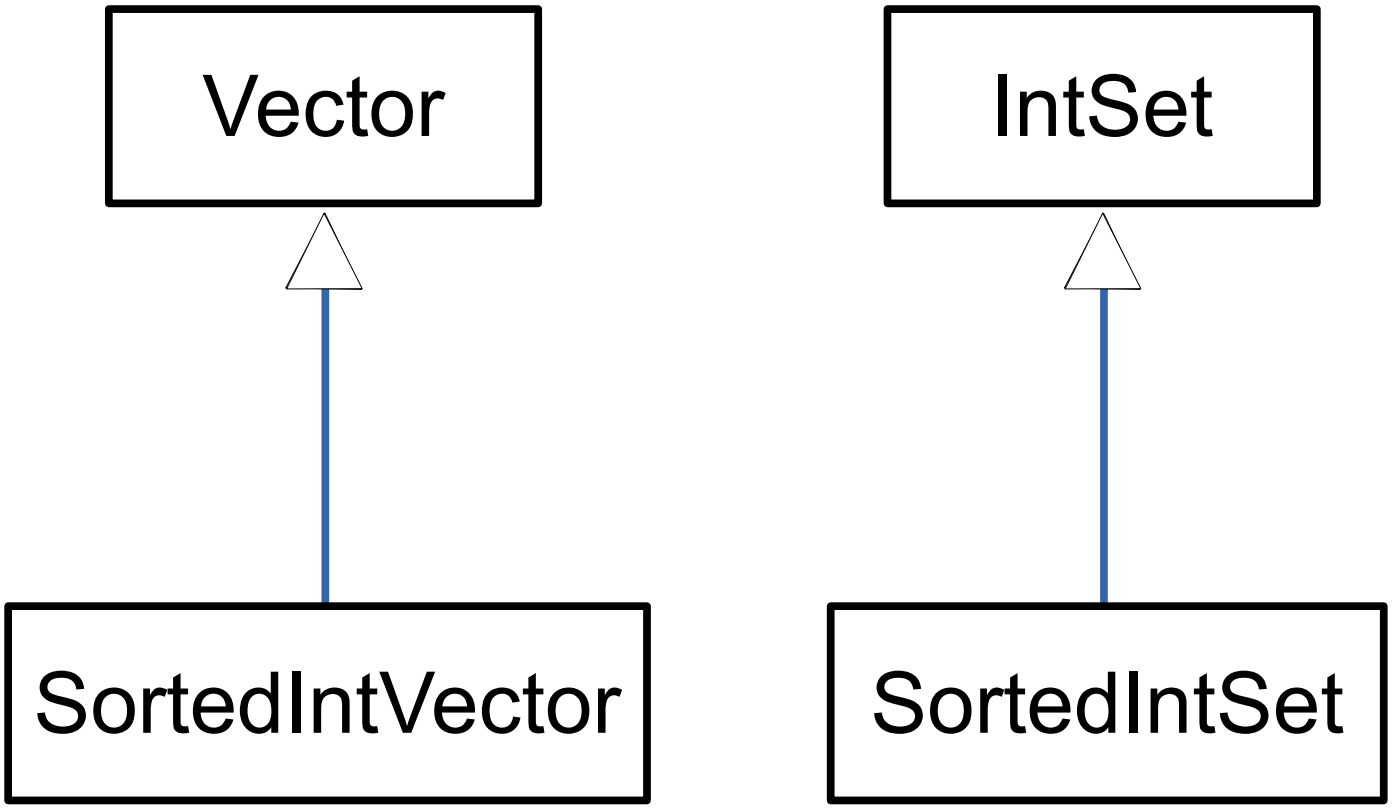
- Note:
  - abstract properties use function `length` over array
  - constructor takes an extra argument
  - `getRoutes`: return a copy of `routes`
  - `repOK`: first invoke `super`'s then invoke `validateRoutes`
  - `validateRoutes`: validate `routes` against abstract properties

Car

ch7.vehiclesextra.**Car**

- Note:
  - abstract properties use function `length` over string
  - `setOwner`: validate argument by invoking `validateOwner` before setting
  - `repOK`: first invoke `super`'s then invoke `validateOwner`
  - `validateOwner`: validate `owner` against abstract properties

Collection class type hierarchy examples



Vector.add

```
/**
 * @effects appends o to the end of this
 */
public boolean add(Object o)
```

SortedIntVector.add

```
/**
 * @effects <pre> if this is empty OR o is >= all elements
 *                   of this
 *                   super.add(o)
 *                 else
 *                   insert o at the position i in this s.t
 *                     xk <= o for all 0 <= k <= i-1 and
 *                     xj > o for all i+1 <= j < this.size
 *           </pre>
 */
public boolean add(Object o)
```

Is this a correct overriding method ?

IntSet.insert

```
/**
 * @modifies  <tt>this</tt>
 * @effects   adds x to this, i.e.
 *            </tt>this_post = this + {x}</tt>
 */
public void insert(int x)
```

SortedIntSet.insert

```
/**
 * @modifies  <tt>this</tt>
 * @effects   <pre>adds x to this, i.e.
 *            this_post = this + {x},
 *            such that x is greater than all elements
 *            before and smaller than all elements
 *            after it</pre>
 */
public void insert(int x)
```

Is this a correct overriding method ?

Abstract class

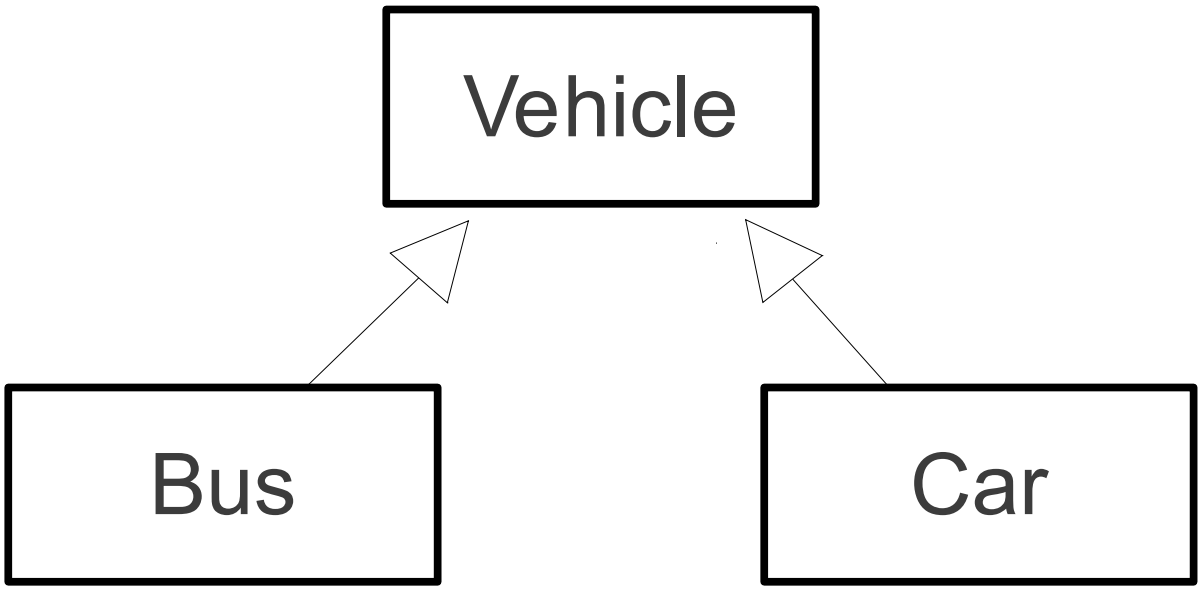
- A super-type that cannot be instantiated
  - though still have constructors
- Provides either partial or full implementation
- Partial implementation must contain *abstract methods*

Which class in the Vehicle TH would be made abstract

Multiple implementations

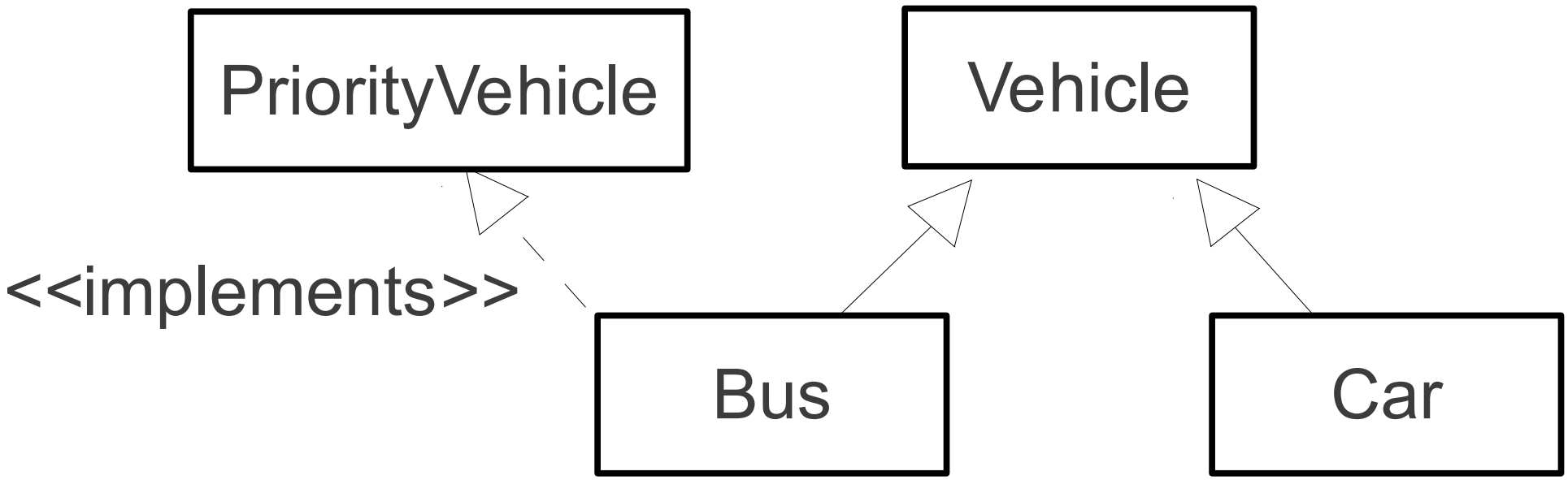
- A restricted TH: subtypes have same behaviour
- Subtype is an implementation of the supertype
- Using code is written using supertype except for creating objects
- Subtypes are placed in the same package
- Subtypes may refer to one another

Example: `Vehicle`



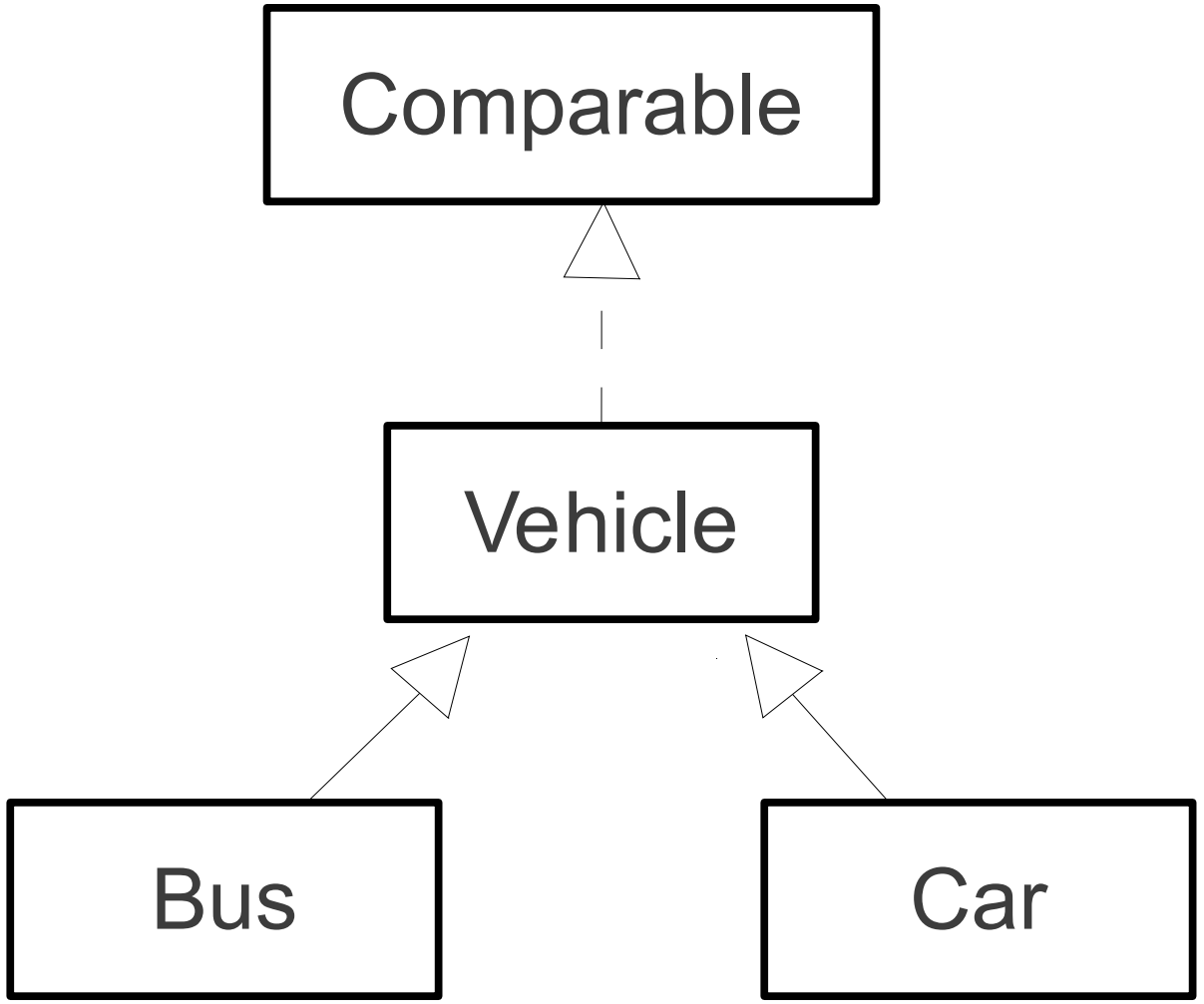
- `Bus` and `Car`:
  - have same behaviour
  - provide concrete implementations

What about this?



Is this TH a multi-implementation TH

… or this?

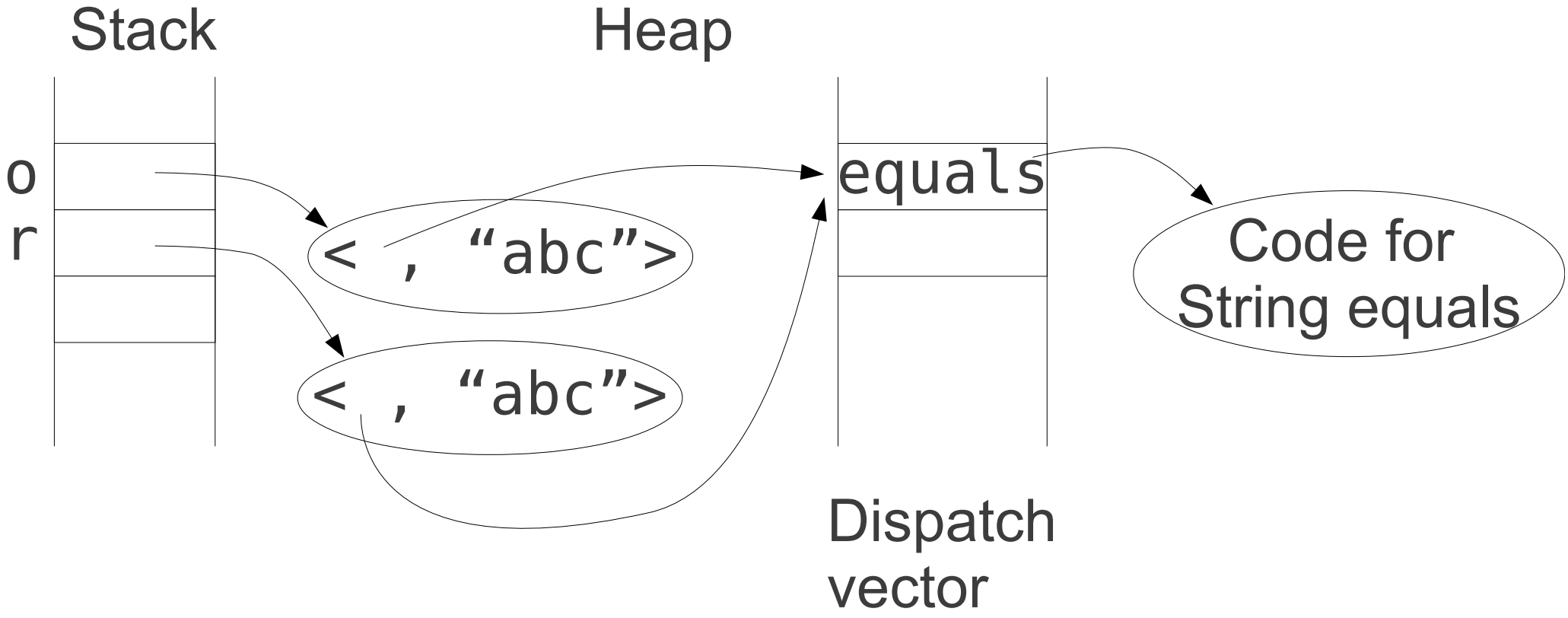


Is this TH a multi-implementation TH

Dispatching

- A run-time mechanism to find the right object to execute a method
- Each object has a pointer to a dispatch vector
- Dispatch vector contains references to the object methods
- Method invocation is dispatched to the target implementation

Dispatching example




```
String t = "ab";
Object o = t + "c";
String r = "abc";
boolean b =  o.equals(r);
```

Summary

- THs make program structure easier to understand
- THs enables shared specification for the related types
- THs supports extensibility
- Subtypes obey the substitution principle

Questions?