A Brief Refresher on Object Oriented Concepts

CSCI 2134: Software Development

Agenda

- Lecture Contents
  - UML
  - Inheritance
  - Polymorphism
  - Abstract classes
  - Interfaces
- Brightspace Quiz
- Readings:
  - This Lecture:
  - Next Lecture: Chapter 5

Scenario

- Your program is composed of many classes
- Problem: Looking at all the code to see how the classes are related and work together is really hard
- Idea: We need a way to diagram our classes in an intuitive way to show the relationships between them
- Solution: Use UML: Unified Modelling Language

```
public class SortedList {
  private int size = 0;
  private String [] list =
                    new String[5];
  …
```

```
public class Student {
  private String name;
  private String email = "N/A";
  private int credits;

  public Student(String name) {
    this.name = name;
  }
  …
```

```
public class AppendableString {
  private String appStr;

  public AppendableString(String s) {
    appStr = s;
  }
  ...
```

UML: Simple Class Diagrams

- Every class is represented by a box with the name of the class
- If we don't need to know what the class does, or what's in it, that's all we need.
- But many times we want to know what's inside the class

```
public class Matrix {
  private int height;
  private int width;
  private double [][] matrix;
  …
```

UML: Detailed Class Diagrams

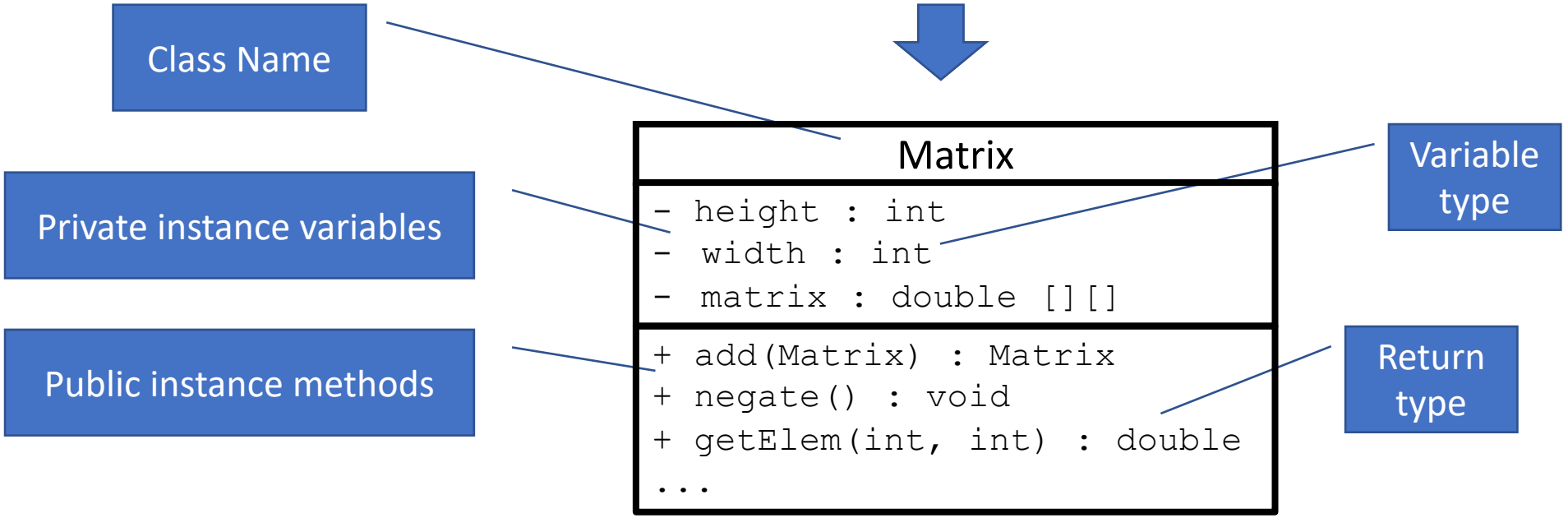
- Every class is represented by a box with the
  - Name of the class
  - Variables in the class (optional)
  - Methods in the class (optional)
- The prefix indicates the access level
  - + means public
  - – means private

```
public class Matrix {
  private int height;
  private int width;
  private double [][] matrix;
  …
```

Matrix
- height : int - width : int - matrix : double [][]
+ add(Matrix) : Matrix + negate() : void + getElem(int, int) : double ...

UML: Detailed Class Diagrams

```
public class Matrix {
  private int height;
  private int width;
  private double [][] matrix;
  …
```



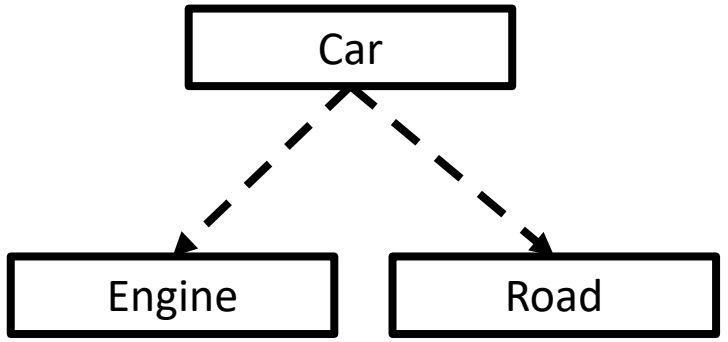
Relationships Between Classes

There are several different relationships between classes that may occur:

- Dependency
- Aggregation
- Nesting
- Inheritance
- Implementation

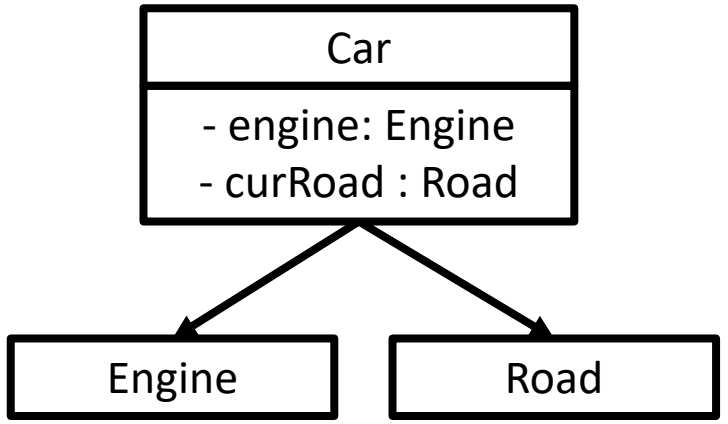
Dependency

- Relationship: “knows about”
- Classes use other classes in their implementation
  - Passed as **parameters**
  - Instantiated
  - **Returned** by methods
- A class being used (usually) does not know about its user.
- This is a unidirectional relationship, e.g.,
  - Engine does not know about Car
  - Road does not know about Car



Association

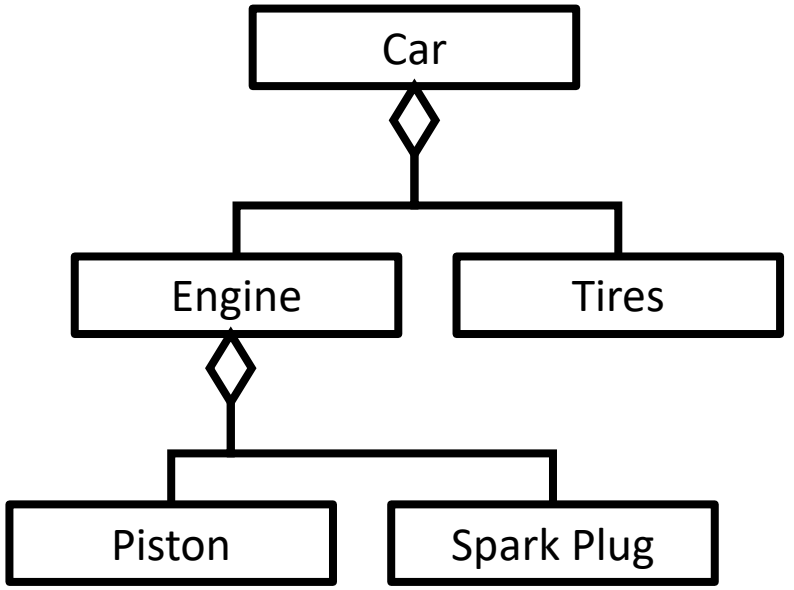
- A stronger form of dependency
- Relationship: “directly uses”
- Classes use other classes as class or instance variables
- A class being used is *intimately known by* its user.
- A class being used (usually) does not know about its user.
- Typically, unidirectional relationship



Aggregation


© bojan fatur/iStockphoto.

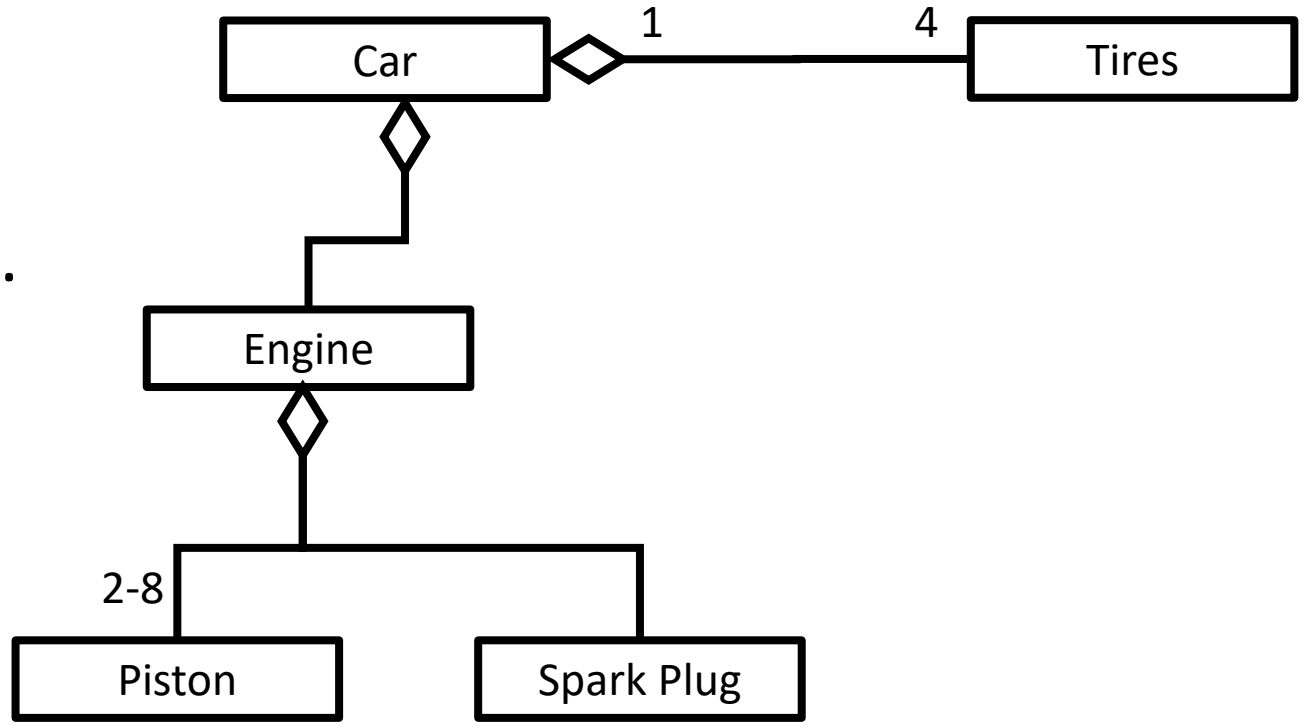
- Relationship: “**has a**” or “**owner of**”
- This is a stronger version of association.
- **Objects** of one class contain **objects** of another class
- It implies the “using object” is actually the “owning object”
- E.g., a Car is an aggregation of an Engine, Tires, and other parts
- Note: A class may use a collection to store multiple objects



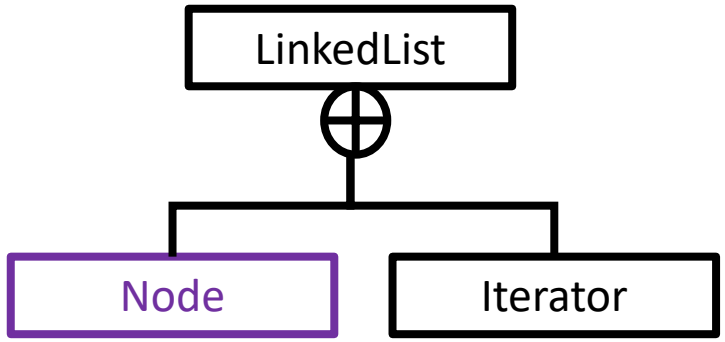
Multiplicity


© bojan fatur/iStockphoto.

- Association and aggregation in UML may have “multiplicity”.
- 1 Car has 4 Tires.
- An Engine may have 2-8 Pistons.
- Decorate the ends of an arrow to show multiplicity of the relationship.
- “At least 1” = “1..*”
- “Any number” = “*”



Nested Classes

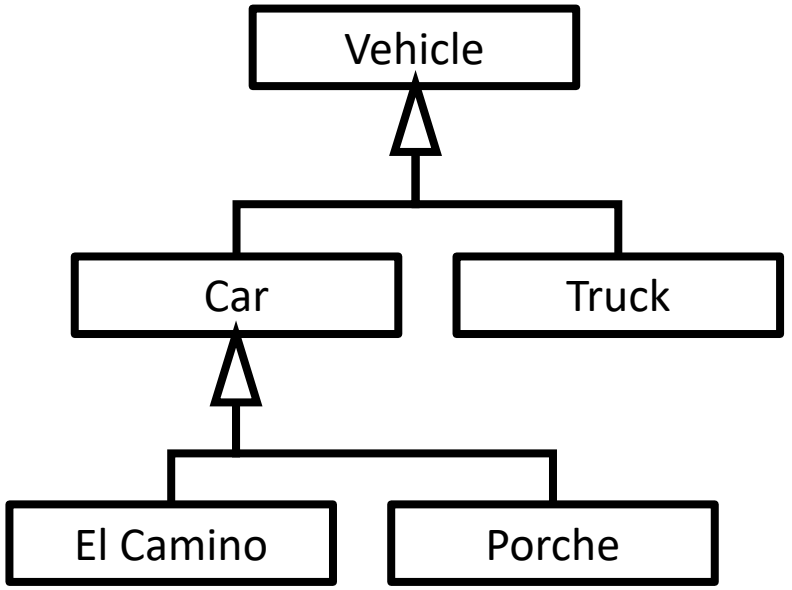


- Relationship: "**has a**"
- An even stronger form of aggregation.
- **Class** contains another nested **class**
- This is an aggregation of classes rather than objects.
- E.g. a LinkedList class defines a Node class within it

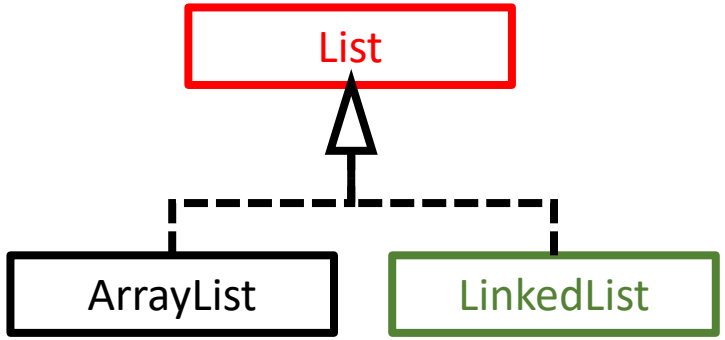
```
public class LinkedList<T> {
  private class Node {
    Node next;
    T value;
  }

  private Node head;
  …
```

Inheritance

- This is an “**is a**” relationship
- Between a more general class (superclass) and a more specific class (subclass)
- E.g.
  - El Camino is a Car
  - Porche is a Car
  - Car is a Vehicle



Implements



- This is an "**implements a**" relationship
- Between a class and an interface
- E.g.
  - *ArrayList* implements *List*
  - *LinkedList* implements *List*

- A special kind of inheritance when the superclass is an Interface

```
public class LinkedList<T> implements List {
   …
}
```

UML Relationship Symbols

Relationship	Symbol	Line	Orientation	Arrow Tip
Dependency		Dashed	To	Solid
Association		Solid	To	Solid
Aggregation		Solid	From	Diamond
Nested Class		Solid	From	Circle-Plus
Inheritance		Solid	From	Triangle/Open
Interface Implementation		Dashed	From	Triangle/Open

Motivation: Inheritance

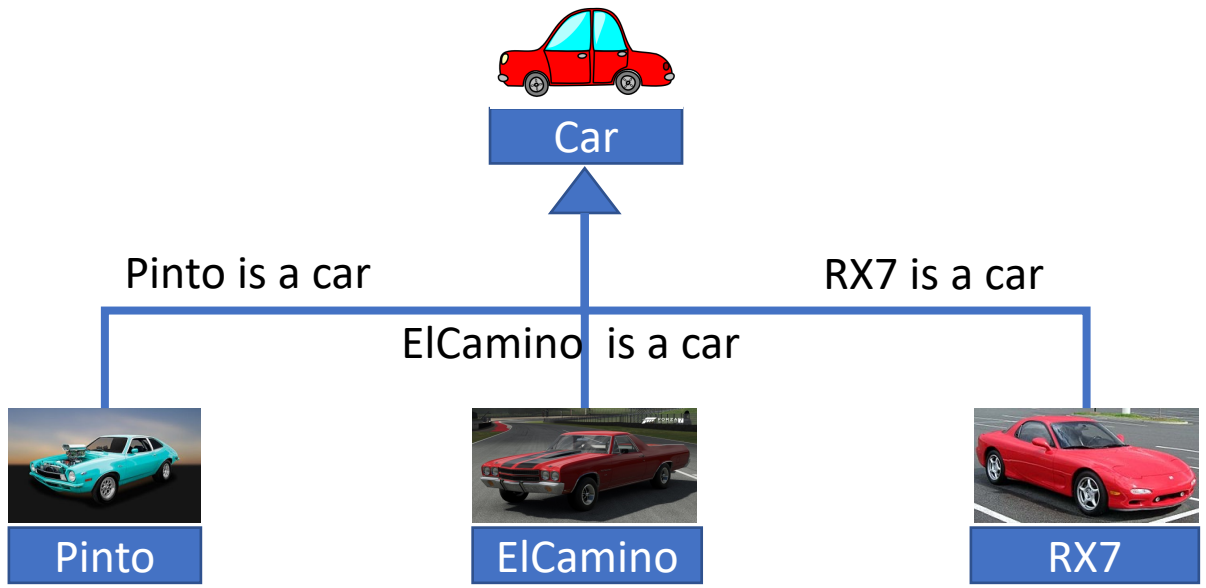
- Suppose you wanted to create a car racing game
- You decide to create a class for each kind of race car in your game
- You have a lot of cars in your game
- What's the problem?
  - All the classes are the same
  - All of these are cars

```
public class FordPinto {
  private int speed;
  private int weight;
  private Color color;

  public void turn(int dir) { …
  public void accelerate() { …
  public void brake() { …
```

```
public class ElCamino {
  private int speed;
  private int weight;
  private Color color;

  public void turn(int dir) { …
  public void accelerate() { …
  public void brake() { …
```

```
public class MazdaRX7 {
  private int speed;
  private int weight;
  private Color color;

  public void turn(int dir) { …
  public void accelerate() { …
  public void brake() { …
```

All of These are Cars



Observations

- In many cases classes in our programs represent a specific kind of general object
  - E.g., The Ford Pinto, Chevy El Camino, and Mazda RX7 are all cars
- Many traits are shared by all these cars because they are all cars.
  - E.g., Weight, colour, speed, turn, accelerate, brake, etc
- It would be much more efficient to specify a car class and then specialize it.
- Should not repeat the same properties in all classes.

An Inheritance Hierarchy

- All variables and methods shared by all cars are placed in a Car class
- Each of the specific car classes extends the car class with attributes specific to that car

```
public class Car {
  private int speed;
  private int weight;
  private Color color;

  public void turn(int dir) { …
  public void accelerate() { …
  public void brake() { …
```

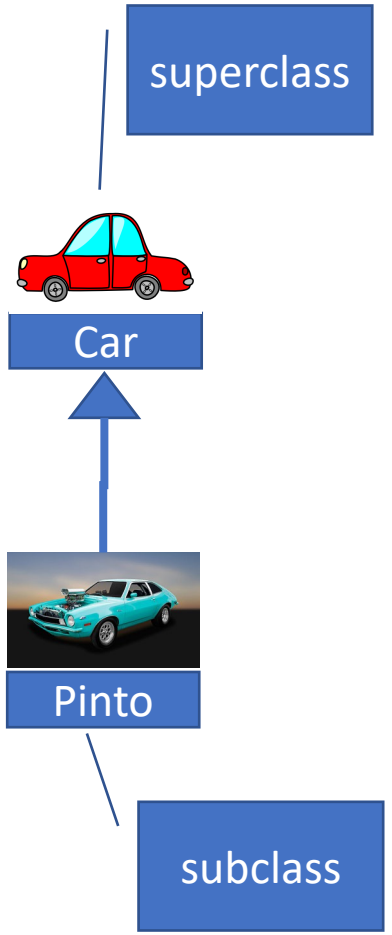
```
public class FordPinto extends Car {

}
```

```
public class ElCamino extends Car {

}
```

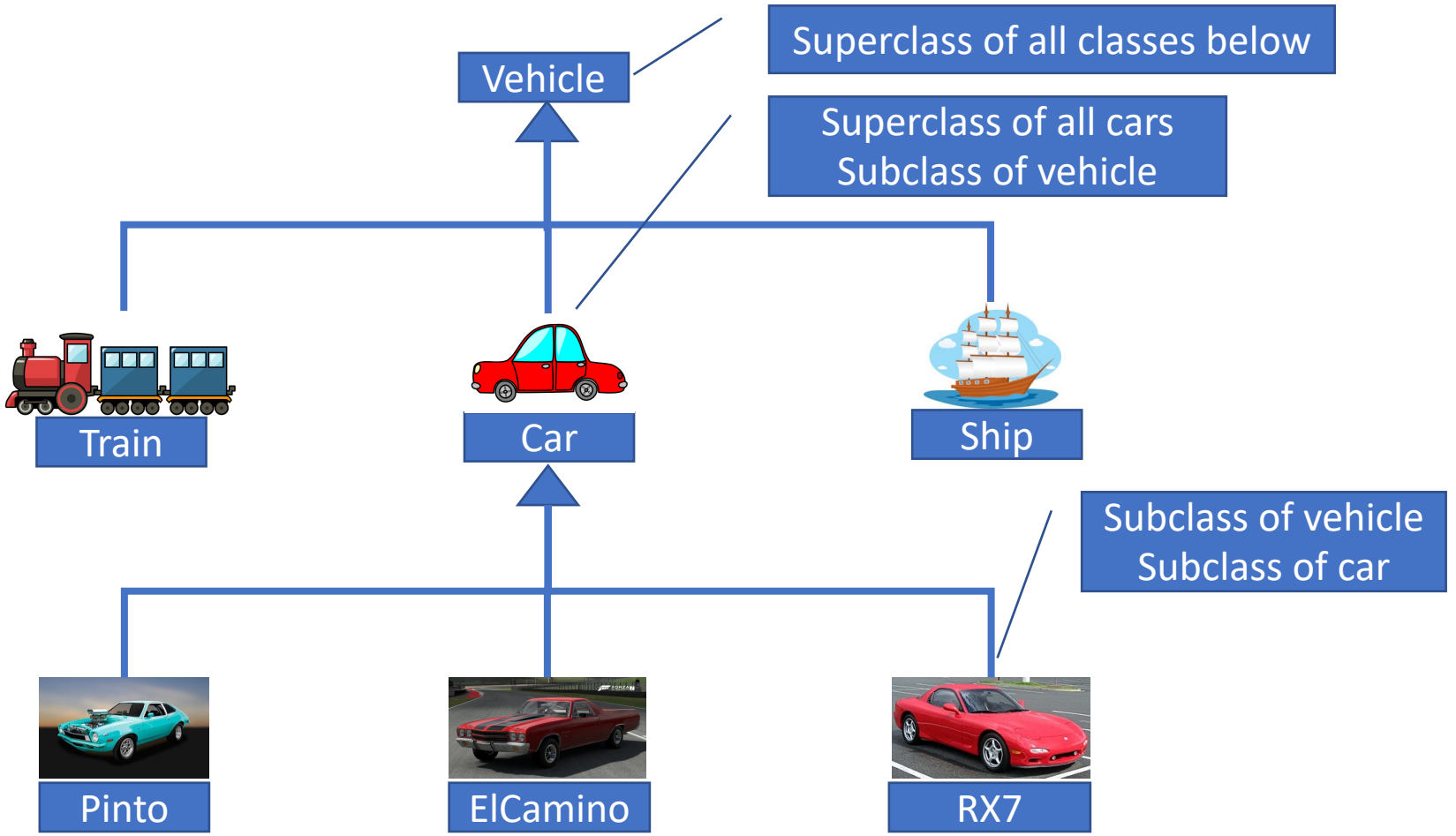
```
public class MazdaRX7 extends Car {

}
```

Subclass and Superclass

- Definition: If class B extends class A then
  - B is a subclass of A
  - A is a superclass of B
- Example:
  - Pinto is a subclass of Car
  - Car is a superclass of Pinto



Bigger Example: Subclass and Superclass



Inheritance

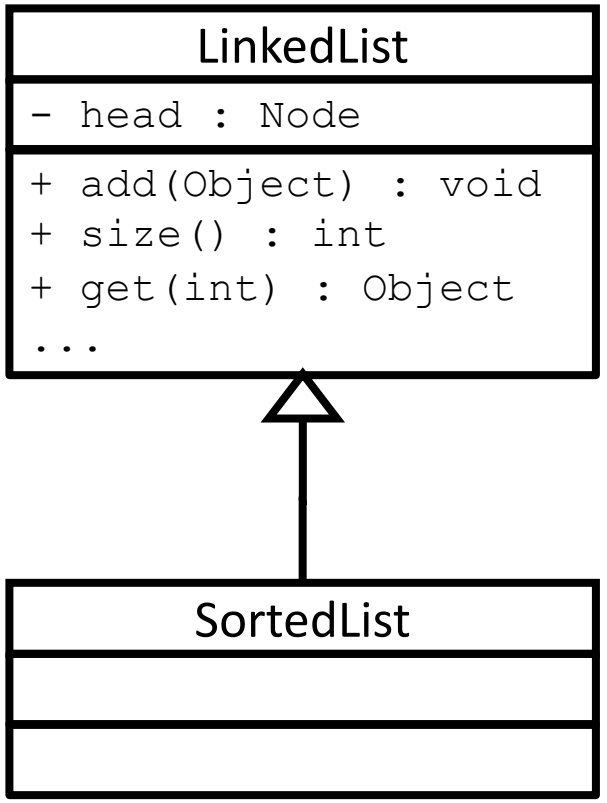
- **Key Idea**:
  - A subclass inherits **all** attributes of its superclass.
  - It has access to the accessible attributes of its superclass
    - All public instance variables
    - All public methods
    - All protected instance variables
    - All protected methods
    - **Not** private variables or methods
- **Key Idea**: any subclass will (at minimum) provide the same public interface as its superclass

  I.e., the same set of public methods

Inheritance Example

```
class LinkedList {
  …
  private Node head;

  public void add(Object item) { …
  public int size() {…
  public Object get(int i) {…
  …
```

```
class SortedList extends LinkedList {
  …
```



Rule: Inherited attributes are not shown in UML diagrams

Why is Subclassing Useful?

- Subclassing (extending) classes is useful because we create a more specific type from a more general one by:
  - Adding methods or variables to the subclass
  - Overriding methods of the subclass

Overriding Methods

- **Idea:** If a subclass inherits a method that does not provide the required functionality, the subclass can provide its own method, with the same signature
- For example: the `add()` method for the subclass `SortedList` has to add items in sorted order

The Substitution Principle

- *Definition*: An object of a subclass can be used anywhere an object of a superclass is expected.

- Why?
  - Every subclass has same public interface as the superclass.
- Analogy: If you know how to drive a general car you can drive a specific car because a specific (subclass) car has the same interface
  - Steering wheel and pedals
- Example: This is legal…

```
Car car = new Porche();
Object obj = new String("Hello");
```

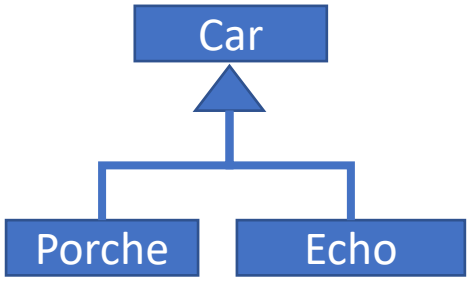
- This is not legal:

```
Porche car = new Car();
```

Porche is a subclass of *Car* so this assignment is ok.

In Java, all classes are subclasses of Object. So, this is ok

What is Polymorphism?



- Question: Does every car behave the same way when you press the gas pedal?
- Answer: No, it depends on the car
  - E.g., A Porche 911 will have a very different response from a Toyota Echo.
  - But both Porche 911 and a Toyota Echo are of type Car.

- Polymorphism: when a single class/interface can be used to represent multiple different classes.
- Typically, a superclass-type variable references a subclass object.
- `Car myCar = new Porche()`

Polymorphism and methods

- Which operation to perform is based on the **actual type** of an object.
- Suppose both the **Porche** and **Echo** classes override the `accelerate()` method
- Which method gets called?

```
Car c = new Porche();
c.accelerate();
```

- What about if

```
Car c = new Echo();
c.accelerate();
```

```
public class Car {
  protected int speed;
  protected int weight;
  protected Color color;

  public void turn(int dir);
  public void accelerate() { …
  public void brake(Boolean b);
```

```
public class Porche extends Car {
  public void accelerate() {
    speed += 10;
  }
  ...
}
```

```
public class Echo extends Car {
  public void accelerate() {
    speed += 1;
  }
  ...
}
```

One Step Further

- Question: Which accelerate() method is called here?

```
void onGreen(Car myCar) {
  myCar.brake(false);
  myCar.accelerate();
}
```

- Options:
  a) The accelerate() method in Car
  b) The accelerate() method in Porche
  c) The accelerate() method in Echo
  d) It depends…
- Answer: It depends … on the class of the object that myCar refers to.

Dynamic Dispatch (Dynamic Method Lookup)

- **Key Idea**: Java uses dynamic dispatch
- Definition: Dynamic dispatch means that the method to be executed is determined **dynamically at runtime** depending on the type of the object, not the type of the variable.
- **Analogy**: If you are handed an unlabeled drink, you don't know what it is until you drink it.

Why Is Polymorphism Useful?

- Scenario:
  - You are working on your racing game.
  - You want to add the TeslaZ race car.
- Your solution:
  - Add a subclass of Car

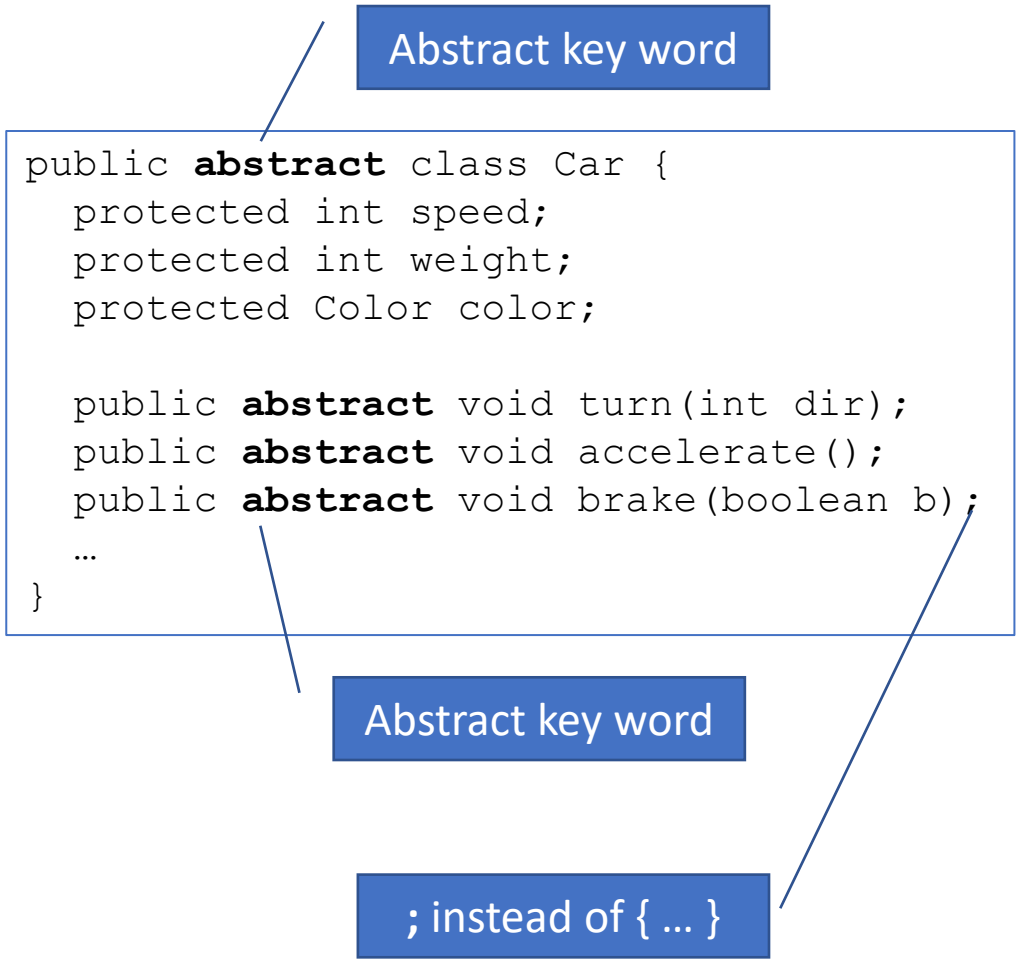
```
public class TeslaZ extends Car { …
```

  - Since all cars are subclasses of Car, your game can use the TeslaZ class with no further modification

    E.g., If you have your driver's license and I hand you the keys to a TeslaZ, you do not need to relearn how to drive
- Question: Which methods should TeslaZ override?

Abstract Classes

- Question: How do you force a subclass to override (provide) a specific method?
- Idea: In Java, a class can declare abstract methods, which do not have an implementation
- Definition: An abstract method does not have an implementation
- Definition: An abstract class has at least one abstract method and cannot be instantiated



Properties of Abstract Classes

- Important Idea: Abstract classes cannot be instantiated

  E.g. If car is an abstract class, then this is an error

```
Car myCar = new Car();
```

- A subclass of an abstract class must either:
  - Override all abstract methods, or
  - Be an abstract class itself
- The main purpose of an abstract class is to be subclassed, thus specifying a public interface for all subclasses

The Idea of Interfaces

- **Idea:** An *interface* defines **what** operations can be performed on an object (and not **how** these operations are implemented)
- Analogies:
  - Interface for a car: Steering wheel and pedals
  - Interface for a phone: 12 key pad
- In Java, all operations are performed via public methods
- An *interface* can be thought of as a promise that a class provides specific public methods

Wait a Moment …

- **Question:** Is that not the purpose of abstract classes? To force a subclass to implement specific methods?
- **Answer**: Yes, but…
- A class can only inherit from one class (in Java)
  - I.e., each class can only have one immediate superclass
- But a class may implement many different interfaces
  - E.g., a car can be used for both transportation and storage

1. Interfaces provide a way to require that a class implements specific methods
2. Abstract classes ensure specific methods are implemented ***and*** (potentially) give some default implementation

Specifying an Interface

- An interface looks very similar to a class
- Interfaces are “pure abstract classes”
- Like a class, the interface should be placed in a file of the same name
- The interface contains one or more method declarations
- Any class that *implements* the interface, must provide methods specified in the interface

```
public interface Drivable {
  public void turn(int dir);
  public void accelerate();
  public void brake(boolean b);
  …
}
```

The Comparable Interface

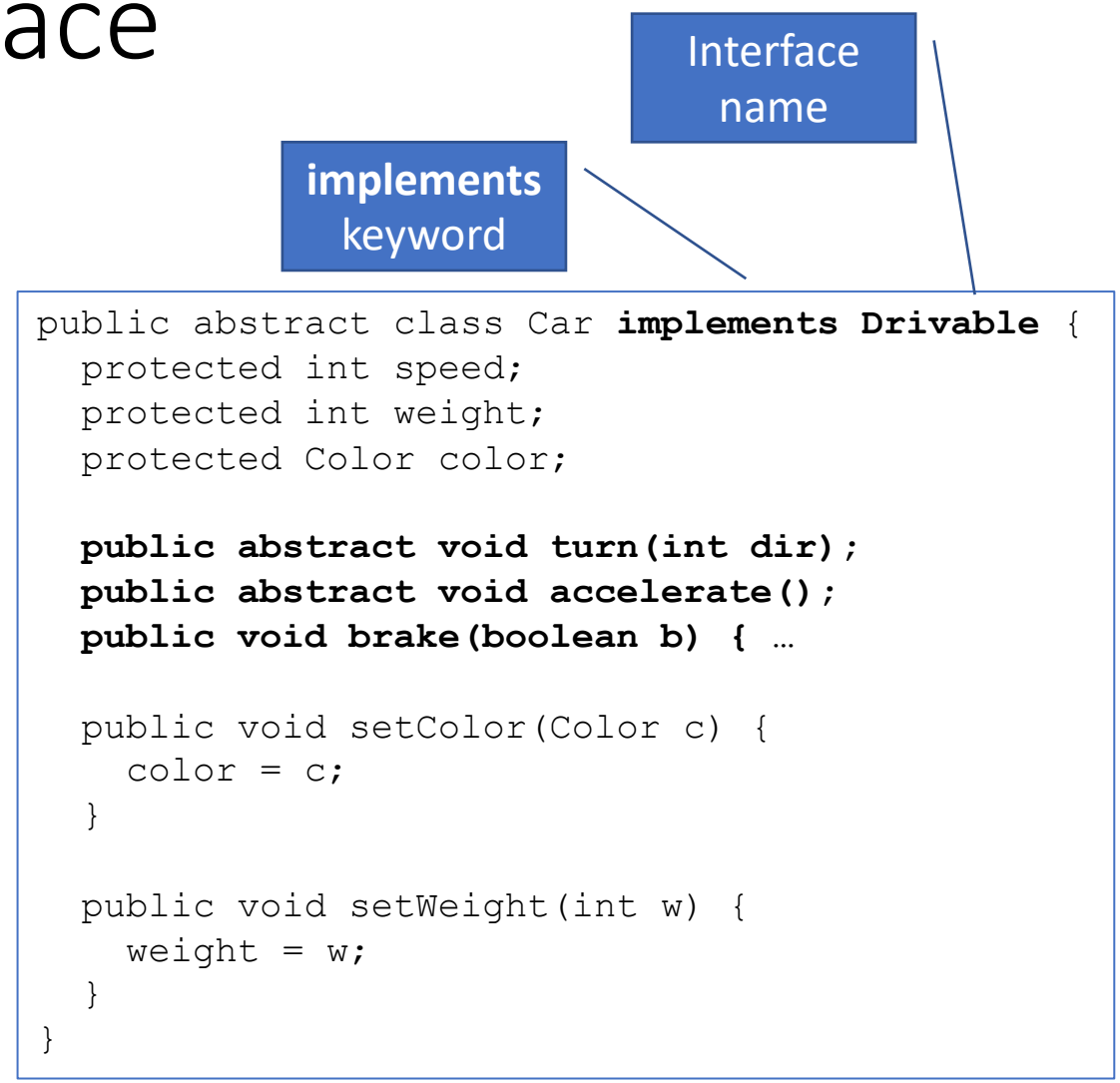
- The *Comparable* interface is one of the most commonly used interfaces in Java

```
public interface Comparable {
    int compareTo(Object other)
}
```

- It requires any class that implements it to have a **compareTo()** method that returns the relative order of objects:
  - Negative, if this < other
  - Positive, if this > other
  - 0, if this == other.

Implementing an Interface

- To *implement* an interface a class has to:
  - Identify that it implements the interface
  - Provide all the methods specified by the interface
- All subclasses also implement the same interface
- A class may implement **multiple** interfaces
  - The interfaces are listed in a comma separated list after the `implements` keyword



Why do we care about interfaces?

Abstract Data Types (ADT)

- **Definition: Abstract Data Type**

  A specification of the fundamental operations that characterize a data type, without supplying an implementation

  Horstmann, Cay S. *Big Java Late Objects, Enhanced eText, 2nd Edition*. Wiley, 2016-09-26. VitalBook file.

- An Abstract Data Type specifies the ***what***
  - What data it stores
  - What operations may be performed

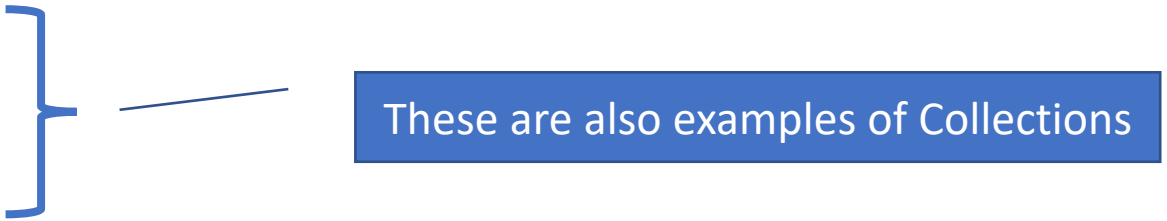
  It does not specify the how
  - How the data is stored
  - How the operations are implemented
- **Idea:** Both *interfaces* and *abstract classes* are ways to specify abstract data types in Java

Examples of Abstract Data Types

- Common Abstract Data Types
  - Lists
  - Sets
  - Maps
  - Stacks
  - Queues
  - Priority Queues
- All of these ADTs specify an interface, but do not reveal the implementation

Example: `List` Interface

- Classes that implement the `List` interface:
  - `ArrayList`
  - `LinkedList`
  - `Stack`
  - `Vector`



- Some methods that all these lists must provide

Method	Description
`boolean add(Object o)`	Appends the specified element to the end of this list (optional operation).
`boolean contains(Object o)`	Returns `true` if this list contains the specified element.
`Object get(int index)`	Returns the element at the specified position in this list.
`int indexOf(Object o)`	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`boolean remove(Object o)`	Removes the first occurrence of the specified element from this list, if it is present (optional operation).

Key 

Points

- UML class diagrams represent classes, their attributes, and relationships between classes.
- Modelling relationships in UML requires careful attention to the type and orientation of arrow used.
- Inheritance is key to: extending functionality, abstracting commonalities, and ensuring common interfaces.
- Polymorphism allows superclass types to refer to subclass objects
- Dynamic dispatch allows a programming language to dynamically decide which method to call based on the object's true type and not just its variable's type.
- Interfaces and abstract classes enforce that a subclass adheres to a set of public methods
- Abstract Data Types are fundamental types which specify **what** operations can be performed without specifying **how** the operations are performed.