CS100
Introduction to Programming

Lecture 14. Object-Oriented Programming: Inheritance

Learning objectives

- Understand the different object relationships
- Learn how to implement inheritance
- Understand & define variable/function access
- Learn about overloading

Outline

- Code Reuse
- Object Relationships
- Inheritance
  - What is Inherited
  - Handling Access
- Overriding

Code Reuse

- Important to successful coding

- Efficient
  - no need to reinvent the wheel
- Error free (more likely to be)
  - code has been previously used/test

Code Reuse Examples

- What are some ways we reuse code?
  - Functions
  - Classes
  - Inheritance – will be covered today

- Any specific examples?
  - calling accessor/mutator functions inside a constructor

Outline

- Code Reuse
- Object Relationships
- Inheritance
  - What is Inherited
  - Handling Access
- Overriding

Refresher on Objects

- ***objects*** are what we call an ***instance*** of a ***class***

- For example:
  - **`Date`** is a class
  - **`today, helloween`**, etc. could be variables of type **`Date`**
  - We say that **`today`** and **`helloween`** are **`Date`** objects

Object Relationships

- Two types of object relationships

- The “is-a” relationship
  - inheritance

- The “has-a” relationship
  - composition
  - aggregation

} both are forms of association

Inheritance Relationship

a Car ***is-a*** Vehicle

- this is called ***inheritance***

Inheritance Relationship

a Car ***is-a*** Vehicle

- the Car class ***inherits*** from the Vehicle class

- Vehicle is the general class, or the ***parent class***
- Car is the specialized class, or ***child class***, that inherits from Vehicle

Inheritance Relationship Code

```
class Vehicle {
  public:
    // functions
  private:
    int    m_numAxles;
    int    m_numWheels;
    int    m_maxSpeed;
    double m_weight;
    // etc
} ;
```

all Vehicles have axles, wheels, a max speed, and a weight

Inheritance Relationship Code

```
class Car {

} ;
```

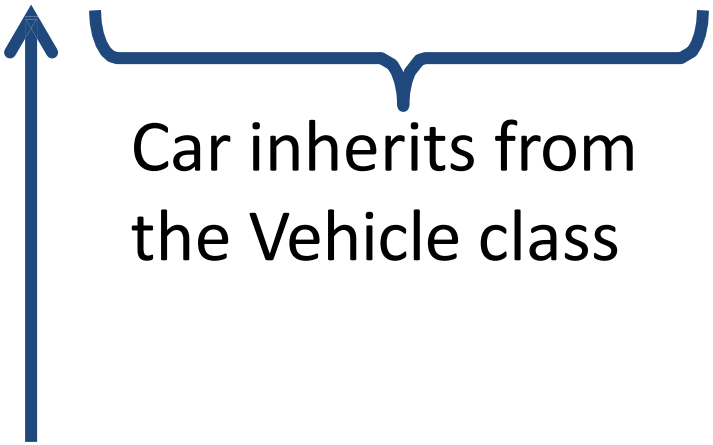
Inheritance Relationship Code

```
class Car: public Vehicle {

} ;
```

Car inherits from
the Vehicle class

Inheritance Relationship Code

```
class Car: public Vehicle {

} ;
```



Car inherits from
the Vehicle class

don’t forget the
colon here!

Inheritance Relationship Code

```
class Car: public Vehicle {
  public:
    // functions
  private:
    int    m_numSeats;
    double m_MPG;
    string m_color;
    string m_fuelType;
    // etc
} ;
```

all Cars have a number of seats, a MPG value, a color, and a fuel type

Inheritance Relationship Code

```
class Car:
  public Vehicle { /*etc*/ };
class Plane:
  public Vehicle { /*etc*/ };
class SpaceShuttle:
  public Vehicle { /*etc*/ };
class BigRig:
  public Vehicle { /*etc*/ };
```

Composition Relationship

a Car ***has-a*** Chassis

- this is called ***composition***

Composition Relationship

a Car ***has-a*** Chassis

- the Car class ***contains*** an object of type Chassis

- a Chassis object is part of the Car class
- a Chassis cannot “live” out of context of a Car
  - if the Car is destroyed, the Chassis is also destroyed

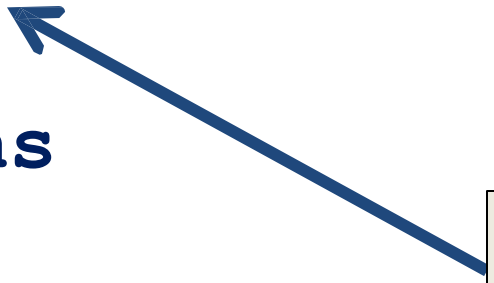
Composition Relationship Code

```
class Chassis {
  public:
    //functions
  private:
    string m_material;
    double m_weight;
    double m_maxLoad;
    // etc
} ;
```

all Chassis have a material, a weight, and a maxLoad they can hold

Composition Relationship Code

```
class Chassis {
   public:
      //functions
   private:
      string m_material;
      double m_weight;
      double m_maxLoad;
      // etc
} ;
```



also, notice that there is no inheritance for the Chassis class

Composition Relationship Code

```
class Car: public Vehicle {
  public:
    //functions
  private:
    // member variables, etc.



} ;
```

Composition Relationship Code

```
class Car: public Vehicle {
  public:
    //functions
  private:
    // member variables, etc.

    // has-a (composition)
    Chassis m_chassis;
} ;
```

Aggregation Relationship

a Car ***has-a*** Driver

- this is called ***aggregation***

集合

Aggregation Relationship

a Car ***has-a*** Driver

- the Car class is ***linked to*** an object of type Driver

- Driver class is not directly related to the Car class
- a Driver **can** live out of context of a Car
- a Driver must be "contained" in the Car object <u>via a pointer</u> to a Driver object

Aggregation Relationship Code

```
class Driver: public Person {
  public:
    // functions
  private:
    Date   m_licenseExpire;
    string m_licenseType;
    // etc
} ;
```

Driver itself is a child class of Person

Driver inherits all of Person's member variables (Date m_age, string m_name, etc.) so they aren't included in the Driver child class

Aggregation Relationship Code

```
class Car: public Vehicle {
  public:
    //functions
  private:
    // member variables, etc.



} ;
```

Aggregation Relationship Code

```
class Car: public Vehicle {
  public:
    //functions
  private:
    // member variables, etc.

    // has-a (aggregation)
    Driver *m_driver;
} ;
```

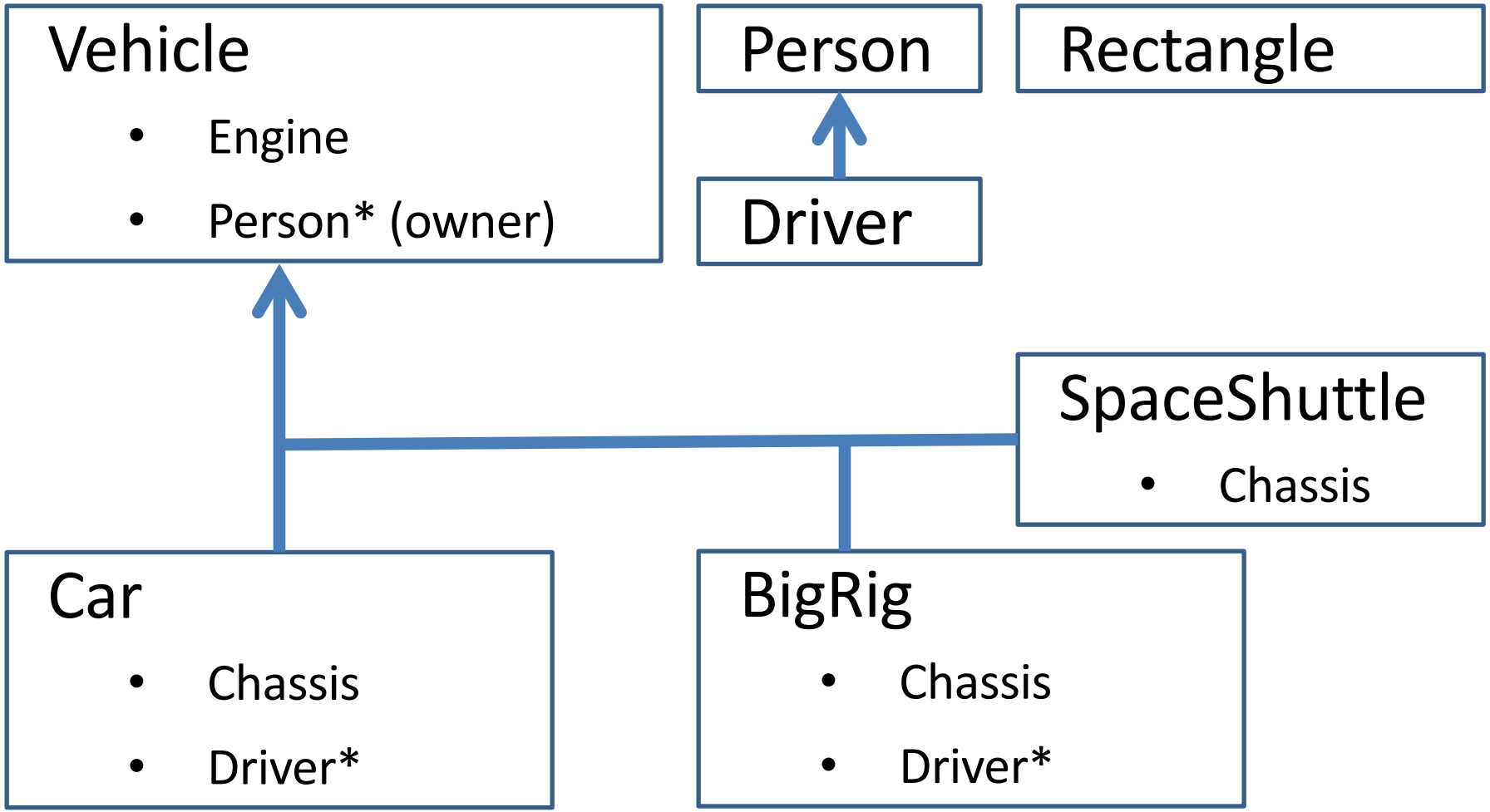
Visualizing Object Relationships

- on paper, draw a representation of how the following objects relate to each other
- make sure the type of relationship is clear

- Car
- Vehicle
- BigRig
- Rectangle
- SpaceShuttle

- Engine
- Driver
- Person
- Owner
- Chassis

Visualizing Object Relationships



Outline

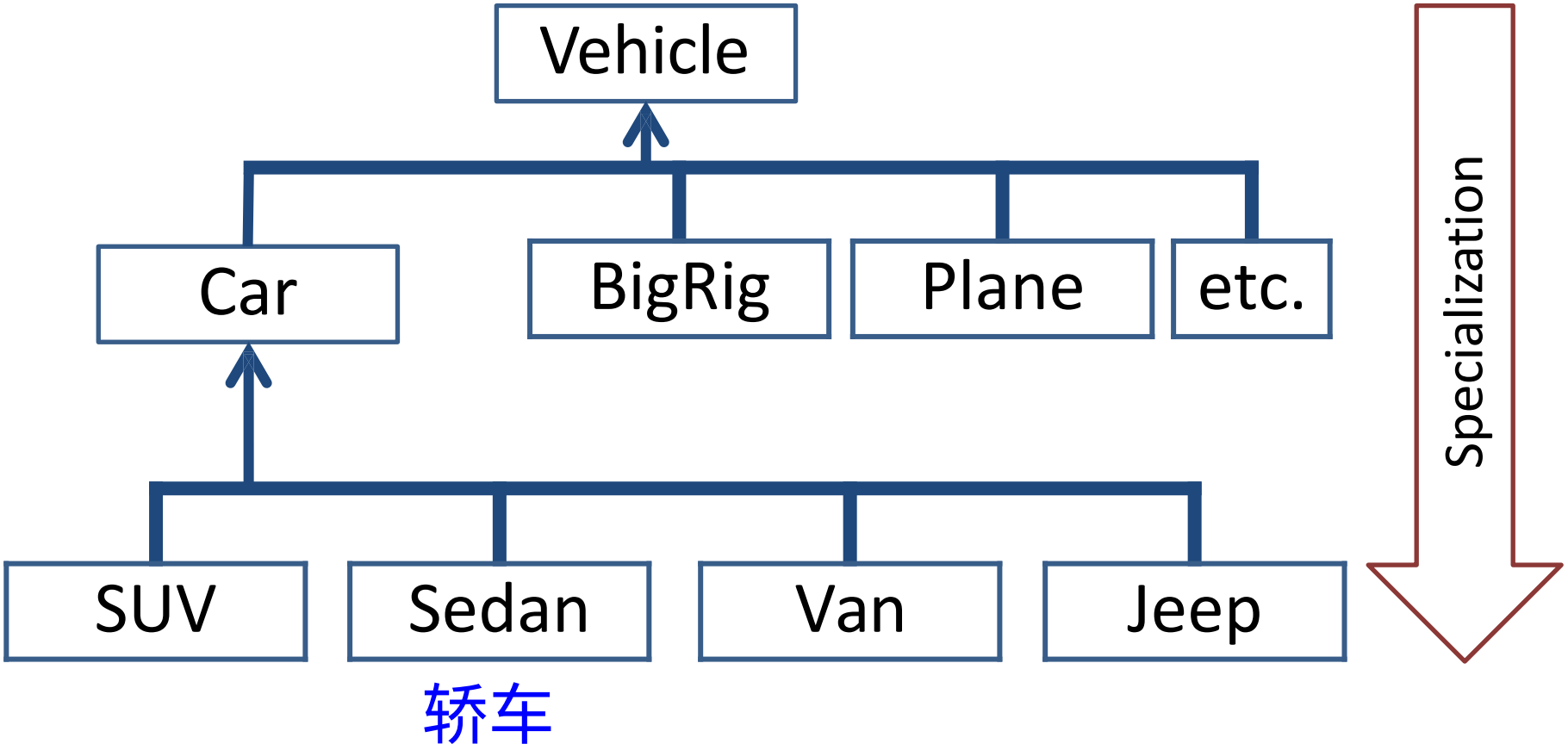
- Code Reuse
- Object Relationships
- Inheritance
  - What is Inherited
  - Handling Access
- Overriding

Inheritance Access Specifiers

- inheritance can be done via:
  public, ***private***, or ***protected***
  - We will be using only ***public***

- you can also have multiple inheritance
  - where a child class has more than one parent
  - an example will be covered in the tutorial

Hierarchy Example



Hierarchy Vocabulary

- **more general class** (e.g., Vehicle) can be called:
  - parent class
  - base class
  - superclass
- **more specialized class** (e.g., Car) can be called:
  - child class
  - derived class
  - subclass

Hierarchy Details

- parent class contains all it has in common with its child classes (less specialized)
  - Vehicle has a maximum speed, a weight, etc. because all vehicles have these

- member variables and functions of the parent class are inherited by **all** of its child classes

Hierarchy Details

- child classes can use, extend, or replace the parent class behaviors

Hierarchy Details

- child classes can **use**, extend, or replace the parent class behaviors

- use
  - the child class takes advantage of the parent class behaviors exactly as they are
    - like the mutators and accessors from the parent class

Hierarchy Details

- child classes can use, **extend**, or replace the parent class behaviors

- extend
  - the child class creates entirely new behaviors
    - a **`RepaintCar()`** function for the Car child class
    - mutators/accessors for new member variables

Hierarchy Details

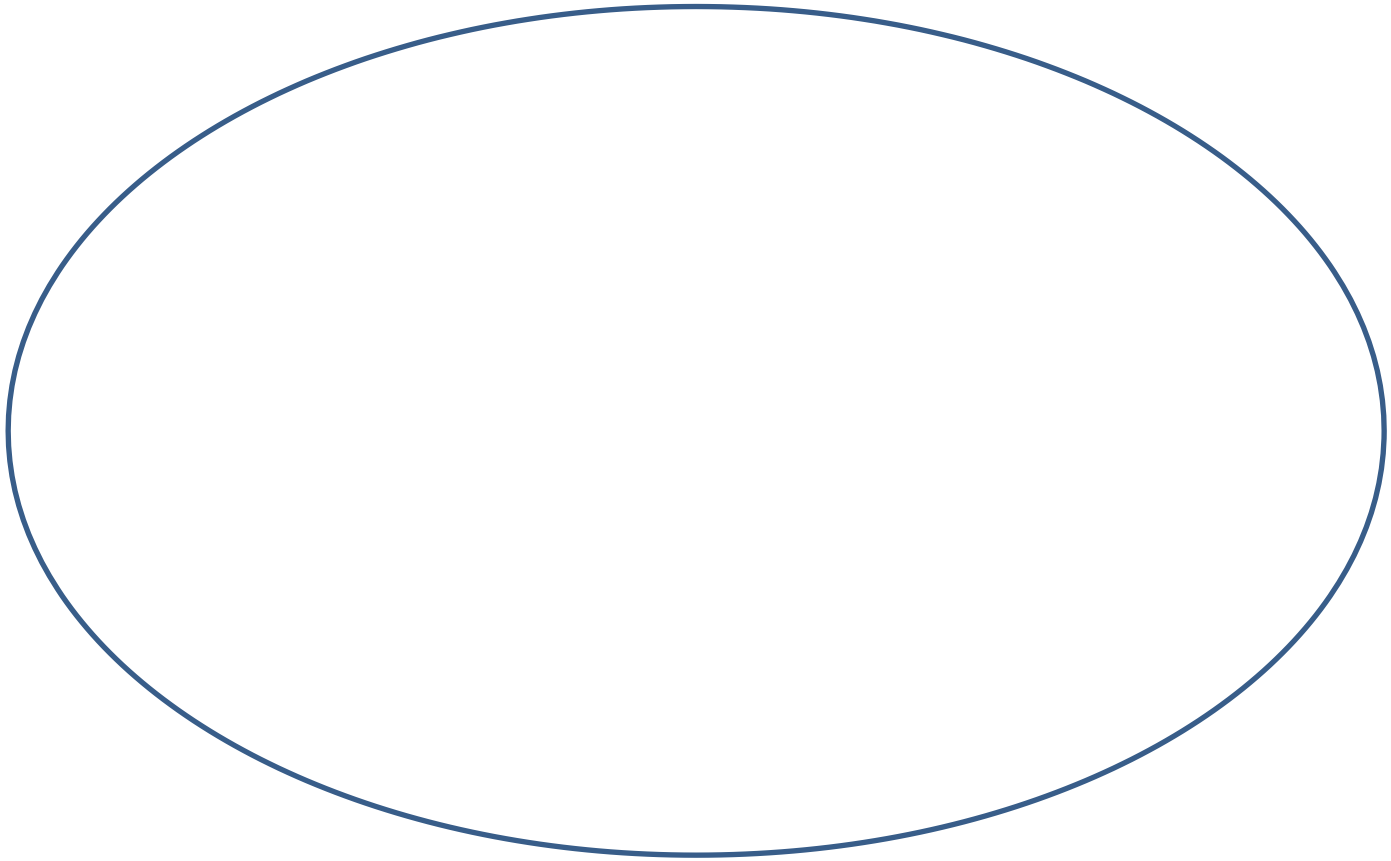
- child classes can use, extend, or **replace** the parent class behaviors

- replace
  - child class overrides parent class's behaviors
    - (we'll cover this later today)

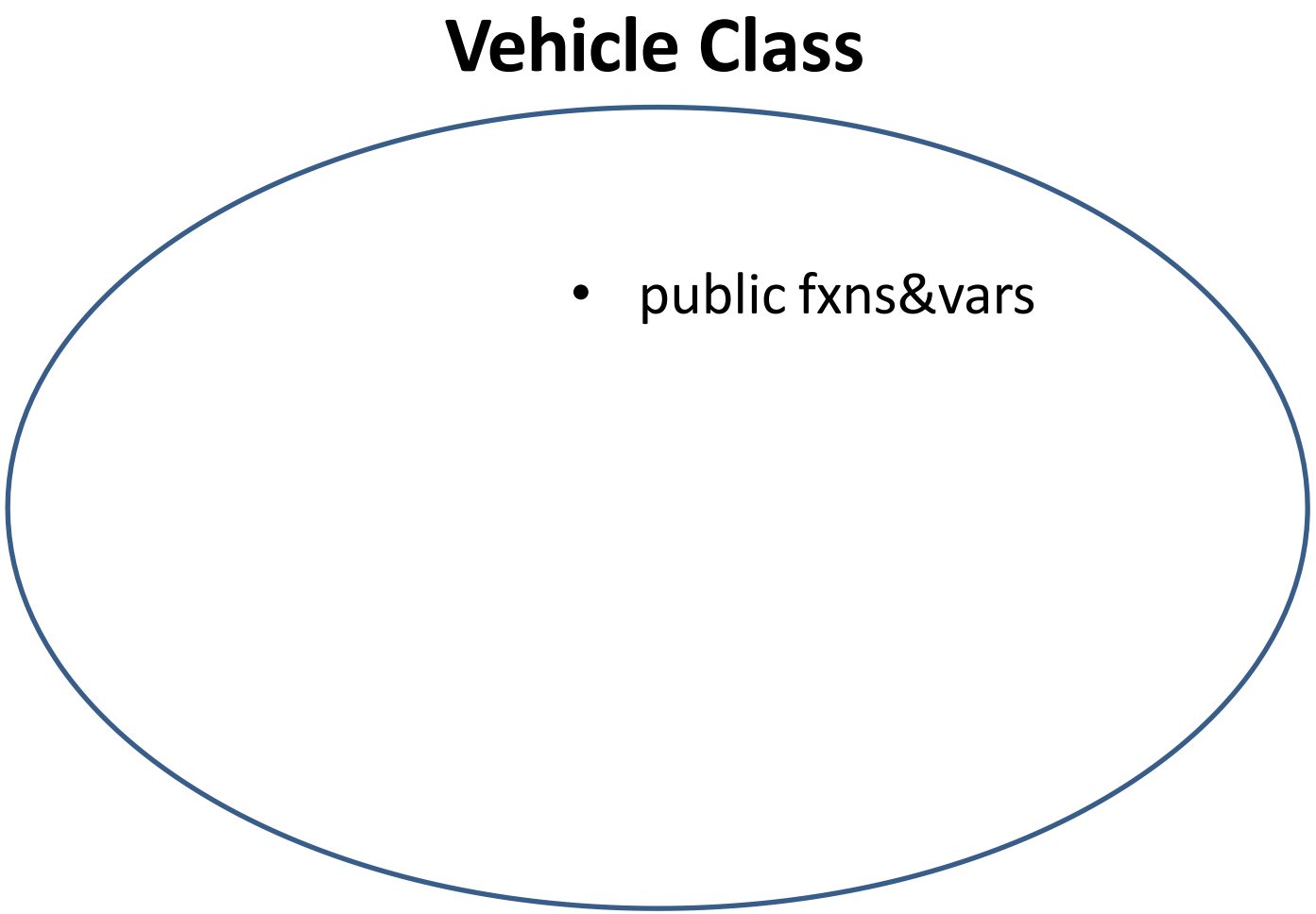
Outline

- Code Reuse
- Object Relationships
- Inheritance
  - What is Inherited
  - Handling Access
- Overriding

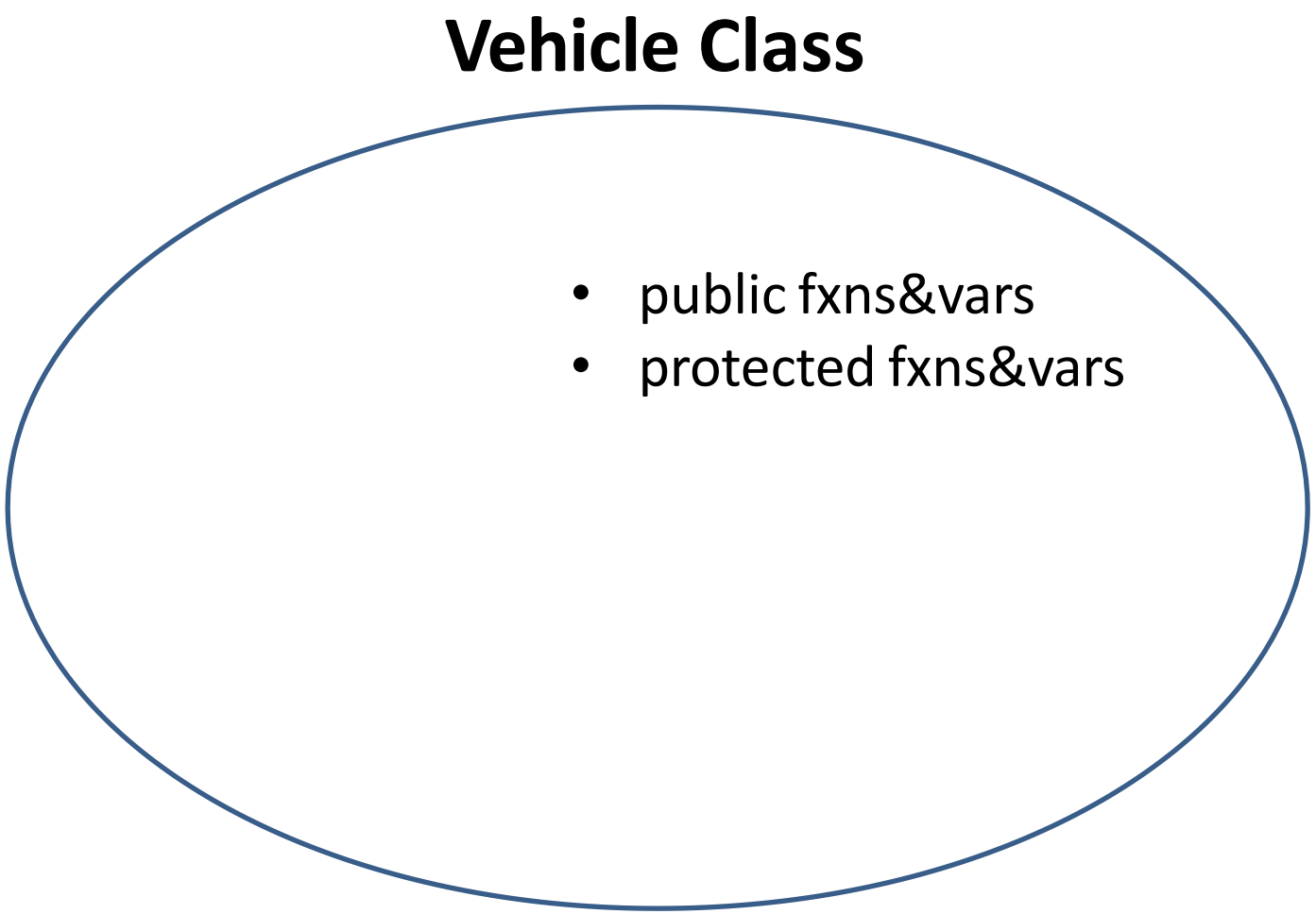
What is Inherited



What is Inherited

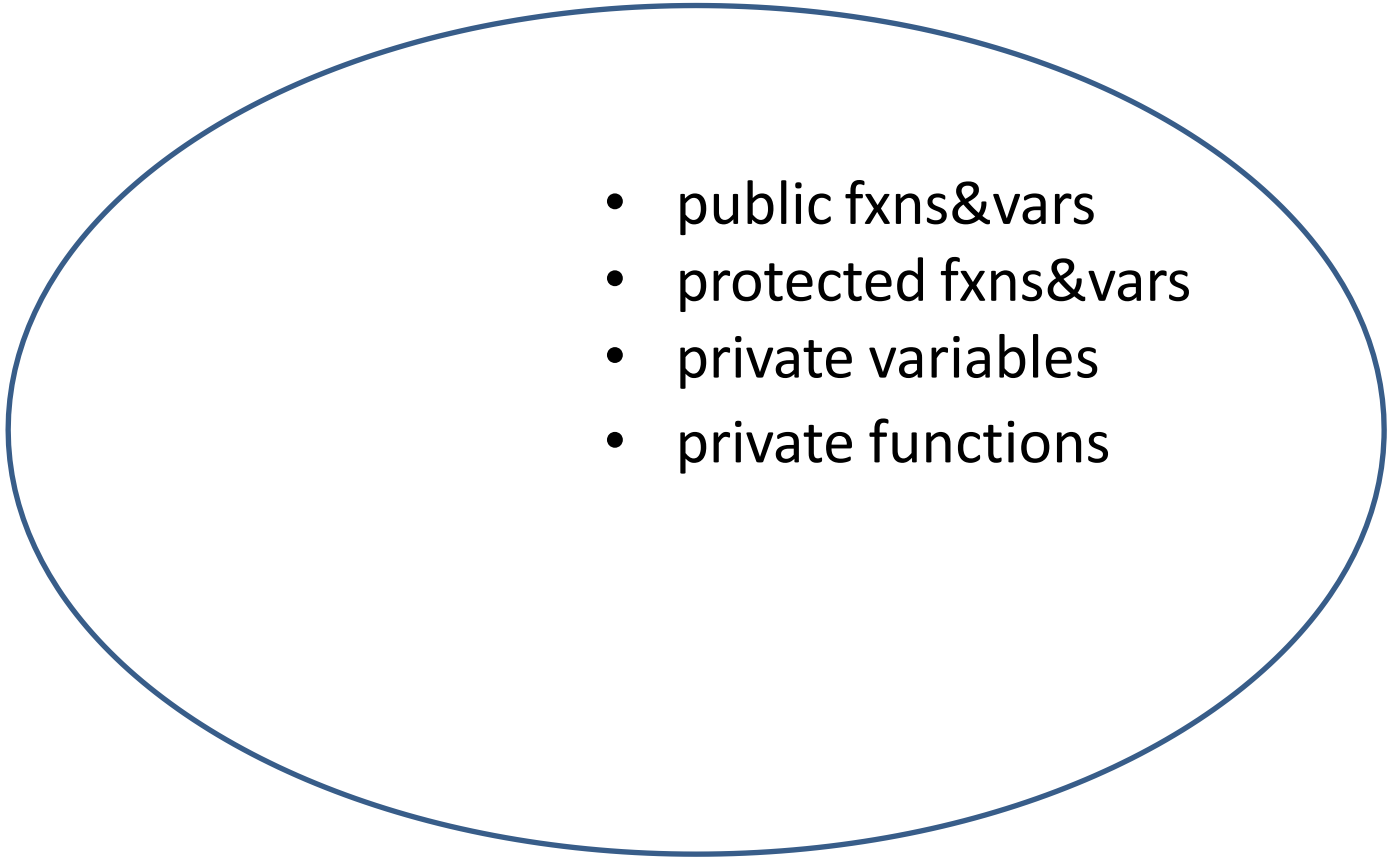


What is Inherited



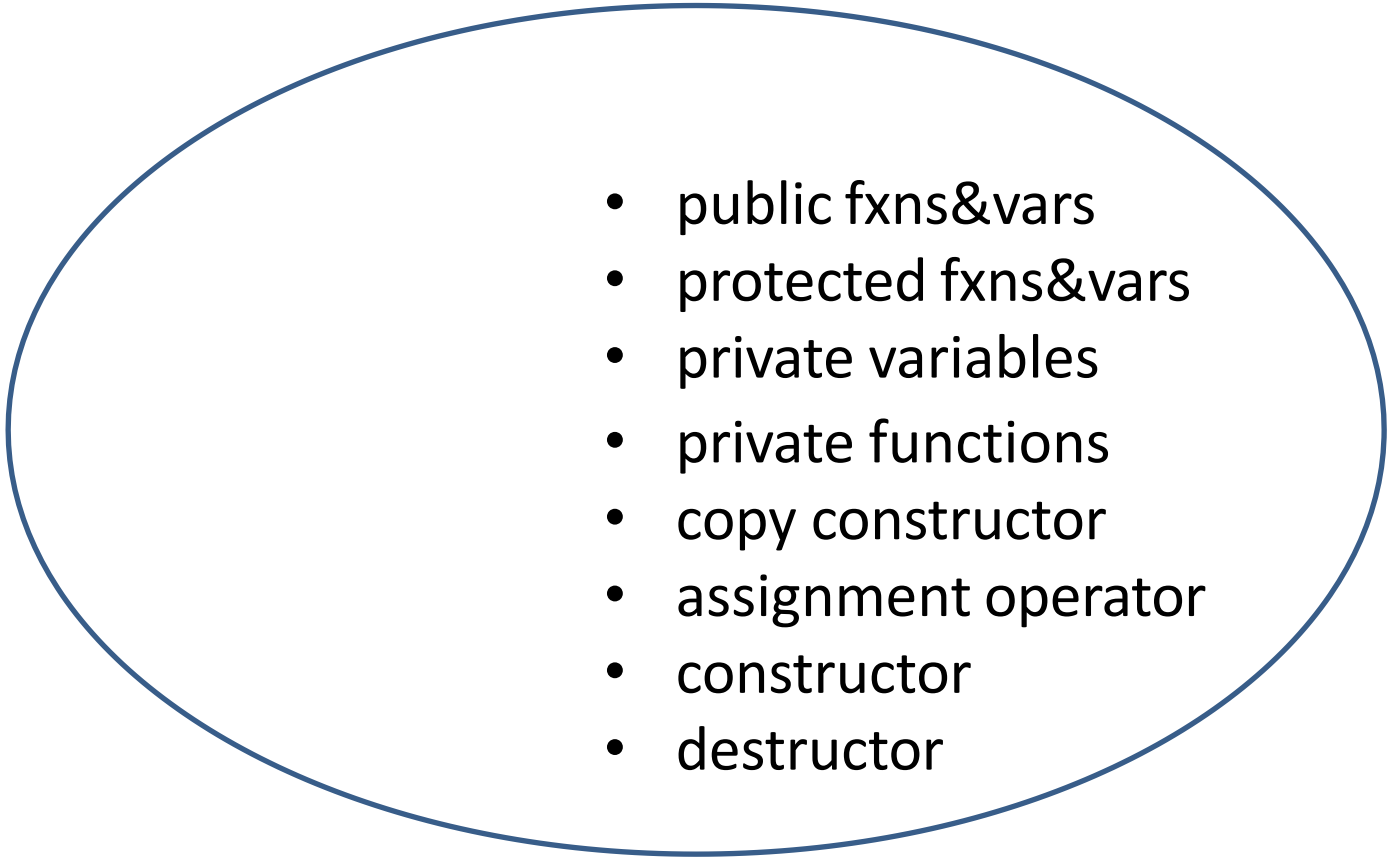
What is Inherited

Vehicle Class

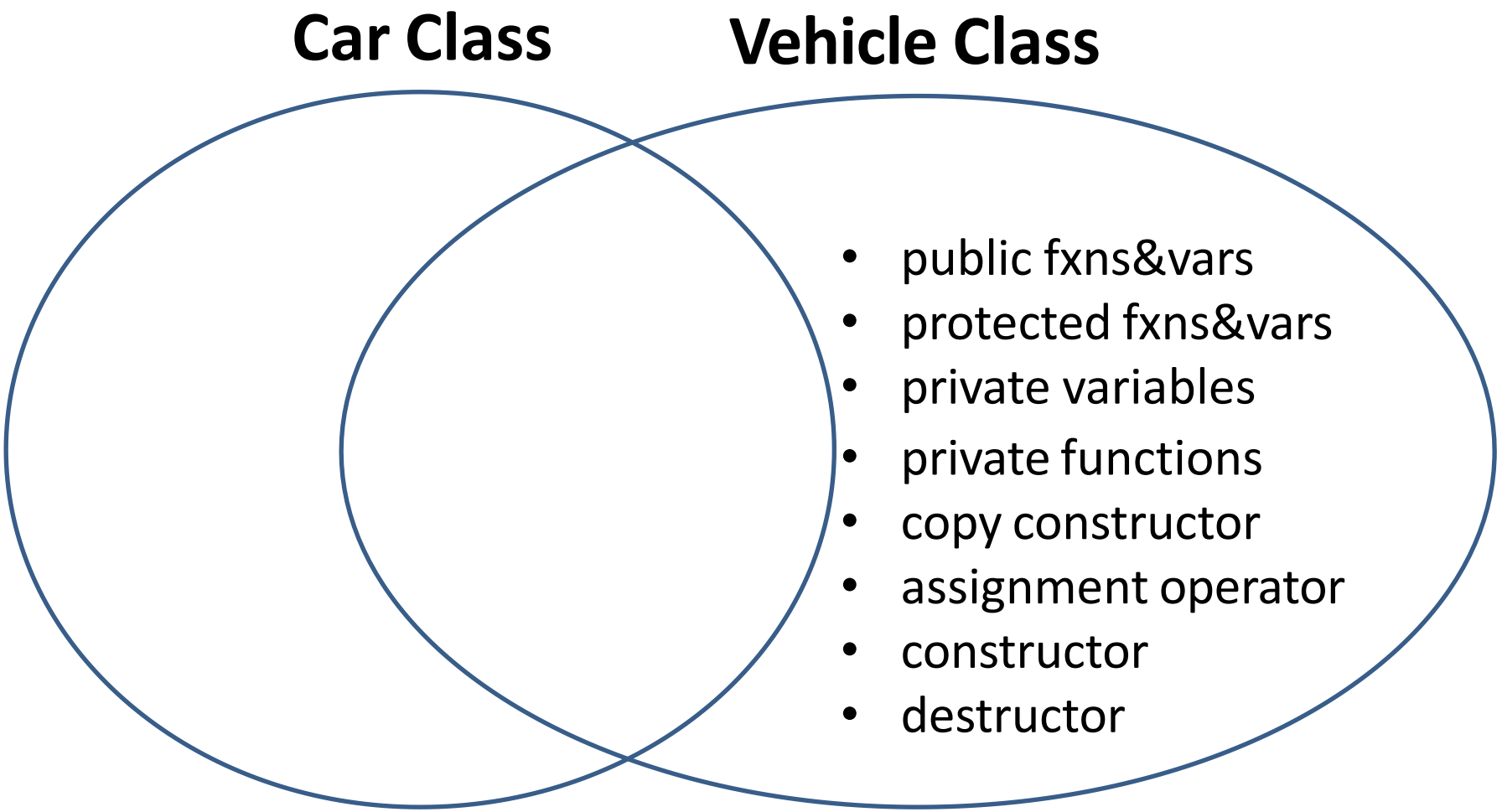


What is Inherited

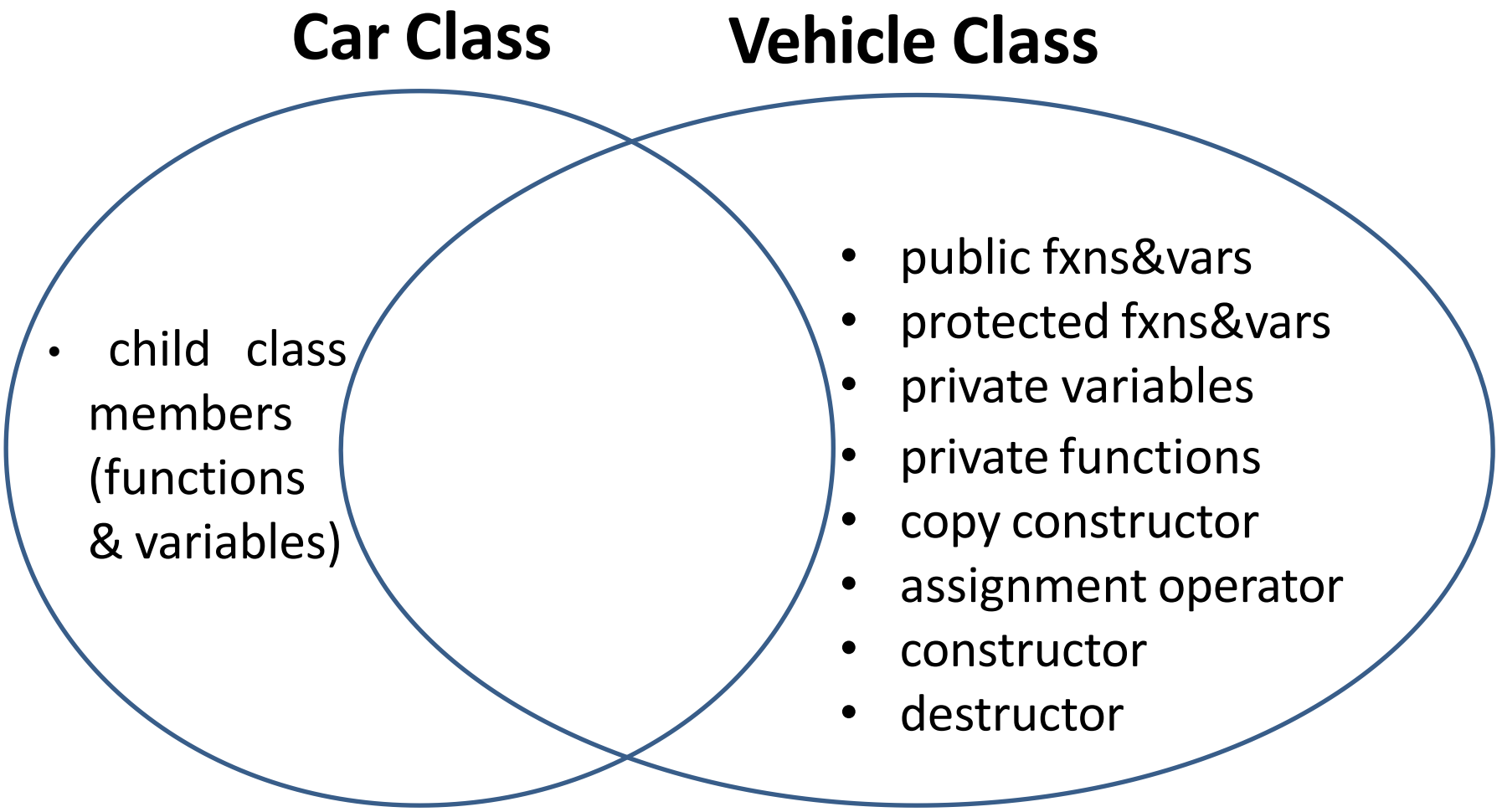
Vehicle Class



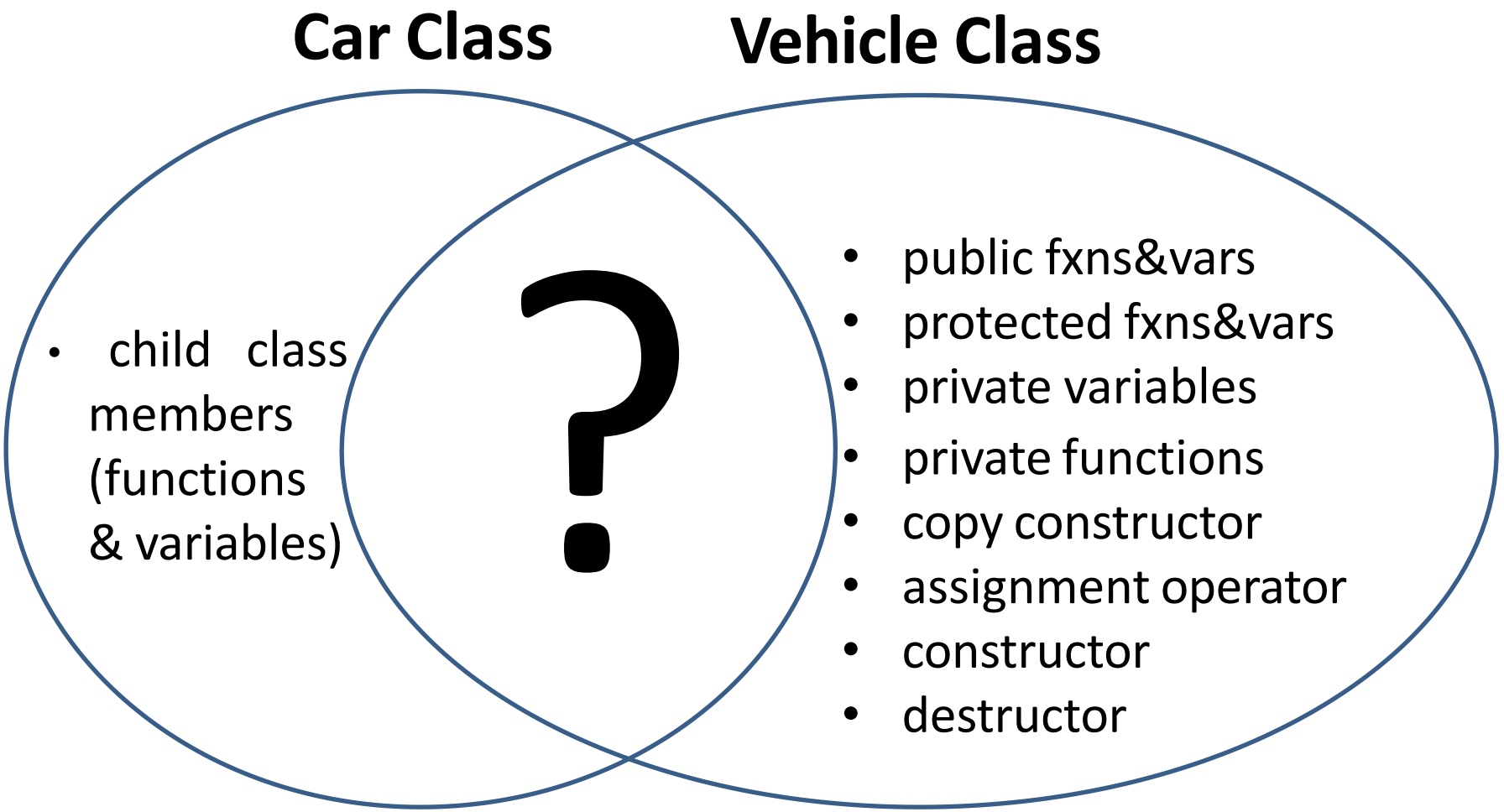
What is Inherited



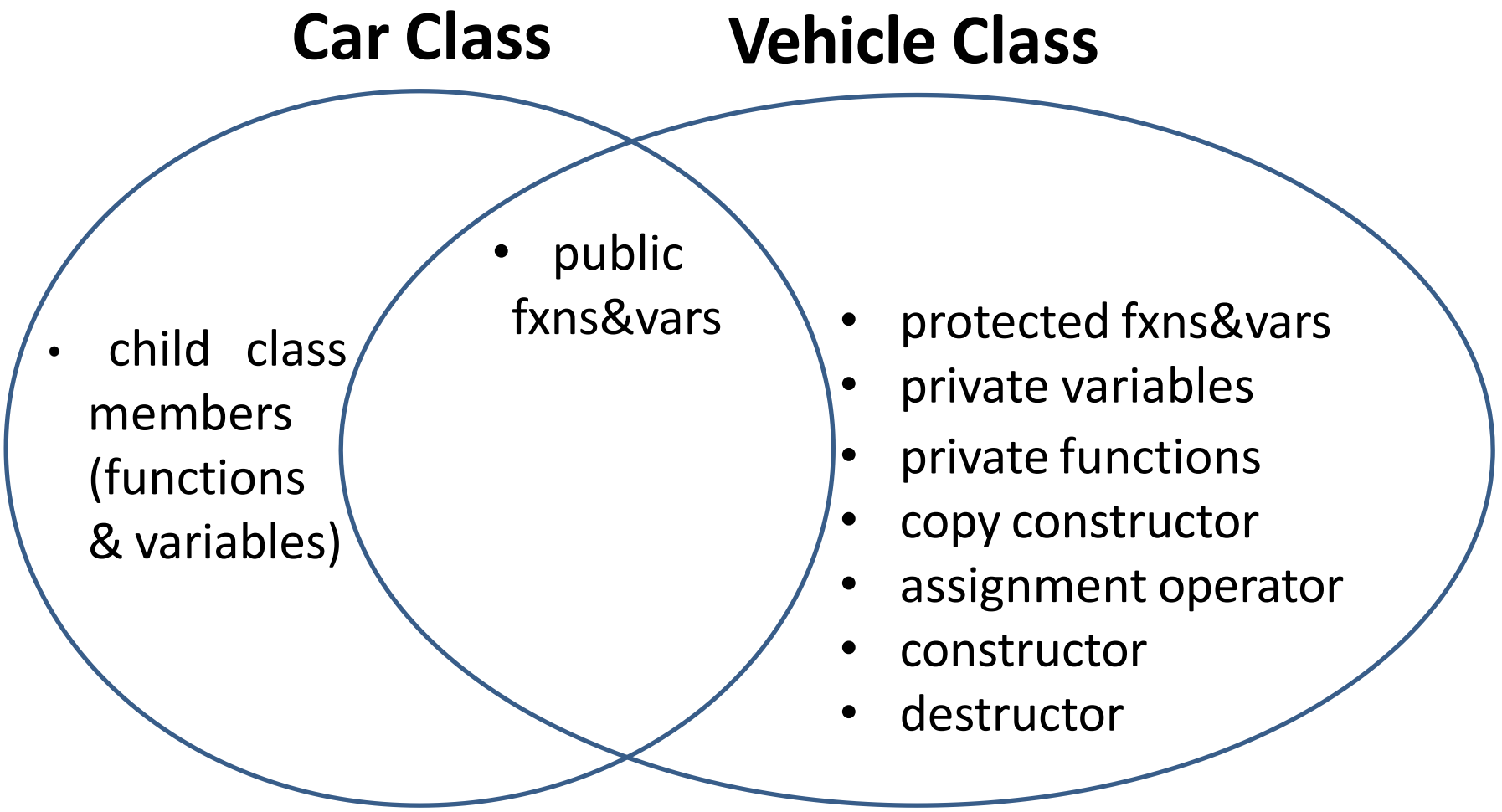
What is Inherited



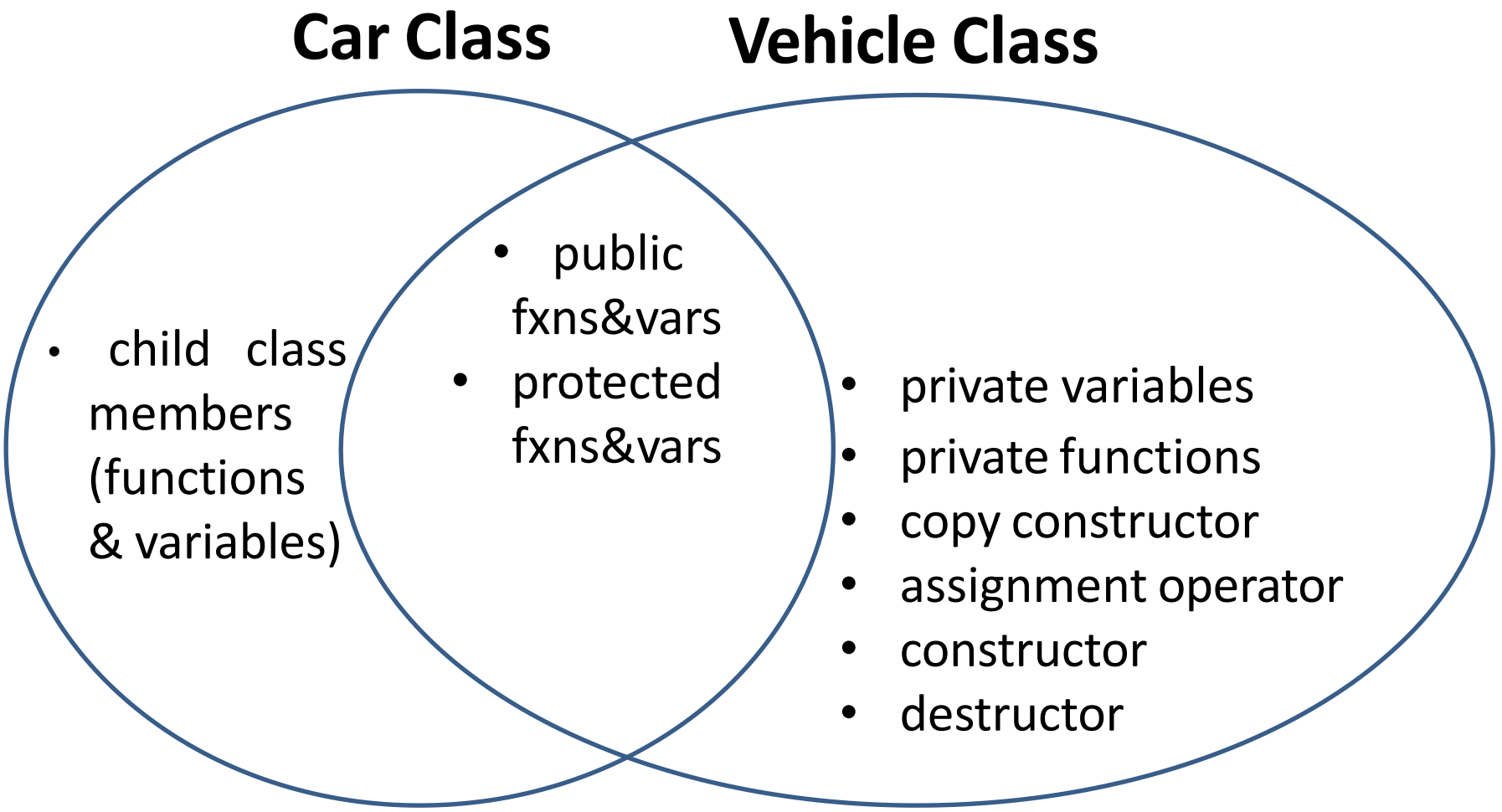
What is Inherited



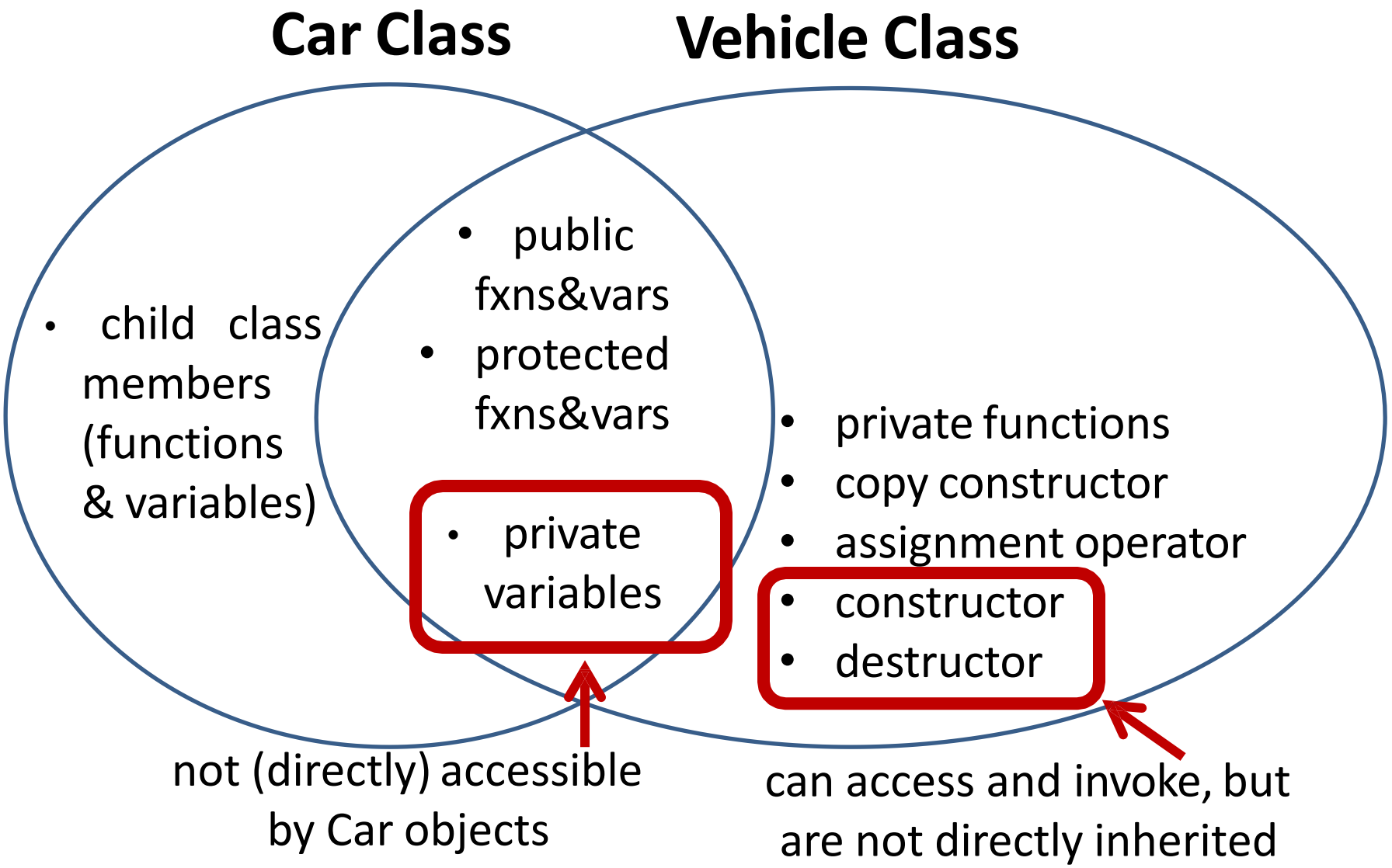
What is Inherited



What is Inherited



What is Inherited



Outline

- Code Reuse
- Object Relationships
- Inheritance
  - What is Inherited
  - Handling Access
- Overriding

Handling Access

- child class has access to parent class's:
  - public member variables/functions
  - protected member variables/functions
  - but *not* private member variables/functions
- how should we set the access modifier for parent member variables we want the child class to be able to access?

Handling Access

- we should not make these variables protected!

- leave them private!
- instead, child class uses protected functions when interacting with parent variables
  - mutators
  - accessors

Outline

- Code Reuse
- Object Relationships
- Inheritance
  - What is Inherited
  - Handling Access
- Overriding

Specialization

- child classes are meant to be more specialized than parent classes
  - adding new member functions
  - adding new member variables

- child classes can also specialize by ***overriding*** parent class member functions
  - child class uses **exact same function signature**

Overloading vs Overriding

- ***overloading***
  - use the same function name, but with different parameters for each overloaded implementation

- ***overriding***
  - use the same function name and parameters, but with a different implementation
  - child class method “hides” parent class method
  - **only possible by using inheritance**

Overriding Examples

- For these examples, the Vehicle class now contains these public functions:

```
void Upgrade();
void PrintSpecs();
void Move(double distance);
```

Overriding Examples

- For these examples, the Vehicle class now contains these public functions:

```
void Upgrade();
void PrintSpecs();
void Move(double distance);
```

- Car class inherits all of these public functions
  - it can therefore override them

Basic Overriding Example

- Car class overrides Upgrade()

```
void Car::Upgrade()
{
   // entirely new Car-only code
}
```

- when Upgrade() is called on a object of type Car, what happens?

Basic Overriding Example

- Car class overrides Upgrade()

```
void Car::Upgrade()
{
   // entirely new Car-only code
}
```

- when Upgrade() is called on a object of type Car, the Car::Upgrade() function is invoked

Overriding (and Calling) Example

- Car class overrides **and calls** PrintSpecs()

```
void Car::PrintSpecs()
{
  Vehicle::PrintSpecs();
  // additional Car-only code
}
```

- can explicitly call a parent's original function by using the scope resolution operator

Attempted Overloading Example

- Car class attempts to **overload** the function Move(double distance) with new parameters

```
void Car::Move(double distance,
               double avgSpeed)
{
   // new overloaded Car-only code
}
```

- but this does something we weren't expecting!

Precedence

- **overriding takes precedence over overloading**
  - instead of *overloading* the Move() function, the compiler assumes we are trying to *override* it

- declaring **`Car::Move(2 parameters)`**
- overrides **`Vehicle::Move(1 parameter)`**

- we no longer have access to the original **`Move()`** function from the Vehicle class

Overloading in Child Class

- to overload, we must have both original and overloaded functions in child class

```
void Car::Move(double distance);
void Car::Move(double distance,
               double avgSpeed);
```

- the “original” one parameter function can then explicitly call parent function