VT Academy
Thinking Ahead, Beyond Boundaries

Object Oriented Programming

Object Oriented Programming in C#

Session 4

Objectives

- Concept of OOP
- Class and object
- Class members

Concept of OPP

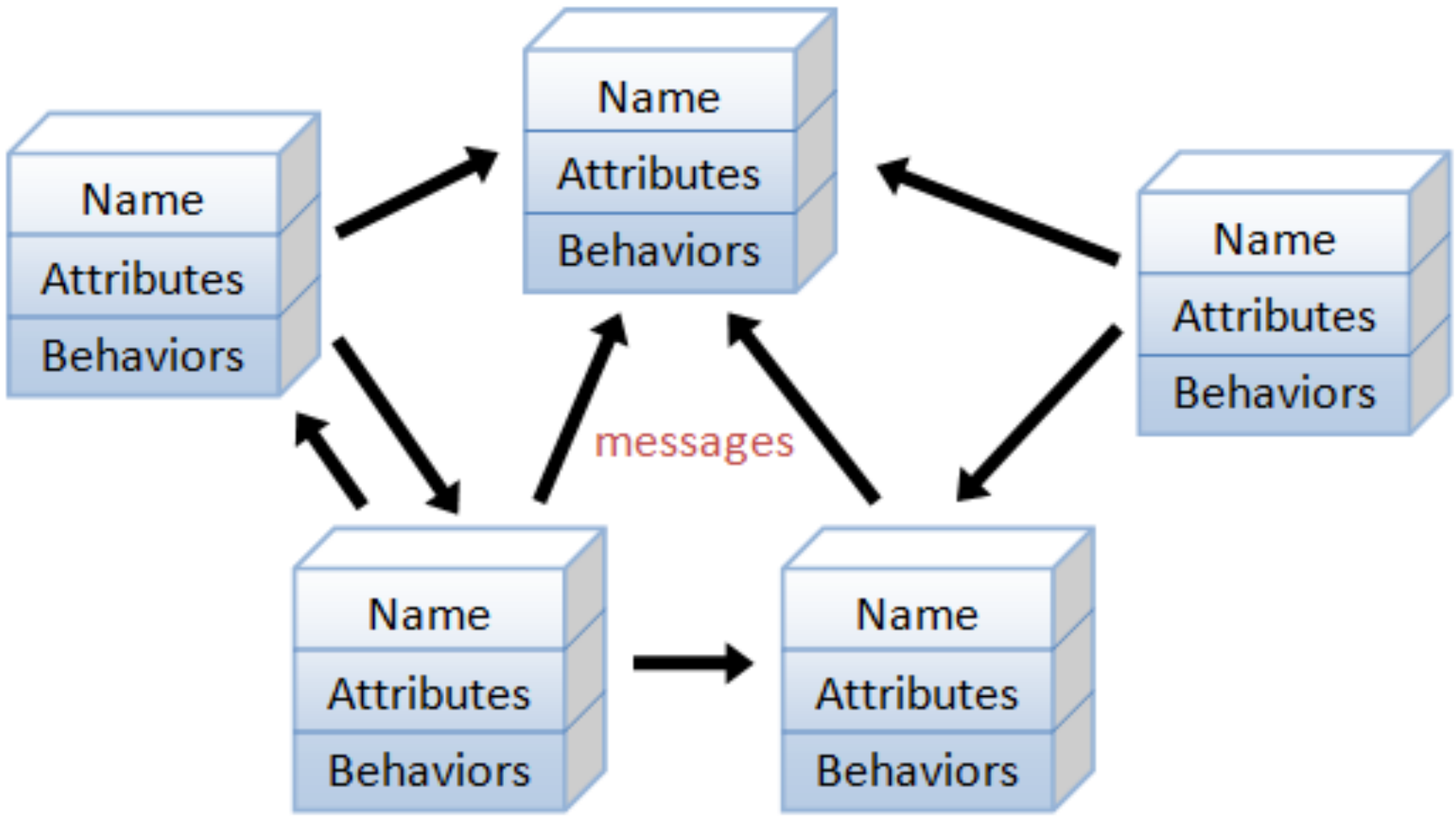
- **Object-Oriented Programming** (**OOP**) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as *attributes;* and code, in the form of procedures, often known as *methods.*

- A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self").

- In OOP, computer programs are designed by making them out of objects that interact with one another.

- Significant object-oriented languages include Java, C++, C#, Python, PHP, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Scala, Common Lisp, and Smalltalk.

Concept of OPP



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

SOP vs OOP

- **Object-Oriented Programming** (**OOP**) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as *attributes;* and code, in the form of procedures, often known as *methods.*

- A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self").

- In OOP, computer programs are designed by making them out of objects that interact with one another.

- Significant object-oriented languages include Java, C++, C#, Python, PHP, Ruby, Perl, Object Pascal, Objective-C, Dart, Swift, Scala, Common Lisp, and Smalltalk.

OOP vs SOP

Structured Oriented Programming (SOP)	Object Oriented Programming (OOP)
SOP is designed which focuses on process / logical structure and then data required for that process	OOP is designed which focuses on data
SOP follows top-down approach	OOP follows bottom-up approach
SOP is also known as Modular Programming and a subset of procedural programming language	OOP supports inheritance, encapsulation, abstraction, polymorphism, etc
In SOP, programs are divided into small self contained functions	In OOP, programs are divided into small entities called objects
SOP is less secure as there is no way of data hiding	OOP is more secure as having data hiding feature
SOP can solve moderately complex programs	OOP can solve any complex programs.
SOP provides less reusability, more function dependency	OOP provides more reusability, less function dependency
Less abstraction and less flexibility	More abstraction and more flexibility

Features of OOP

VTC Academy
Thinking Ahead, Beyond Boundaries

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

What is Object?

- Objects are key to understanding *object-oriented* technology. They can be physical or logical entities.
- Examples of real-world objects: dog, desk, television, bicycle, account, employee, etc

Objects

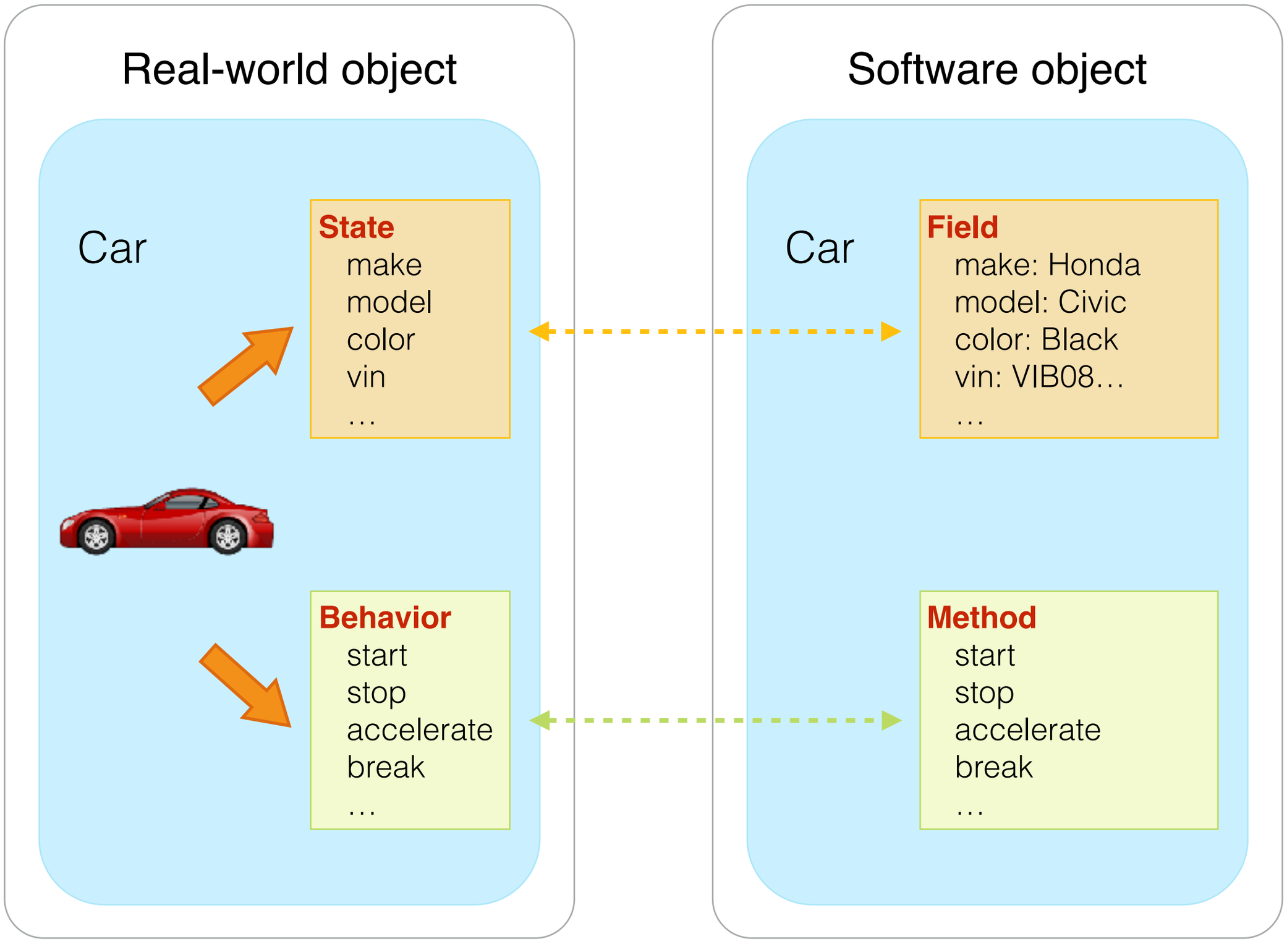
Objects

- Real-world objects share two characteristics: *state* and *behavior*
- Example:
  - Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail).
  - Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes)
- Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of OOP - Data Abastaction

Objects

- Software objects are conceptually similar to real-world objects: they too consist of state and related behavior
- An object stores its state in *fields* (variables in some programming languages) and exposes its behavior through *methods* (functions in some programming languages)
- Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication
- Hiding internal state and requiring all interaction to be performed through an object's methods is known as *data encapsulation* — a fundamental principle of object-oriented programming

Objects


Real-world object
Car
State
make
model
color
vin
…
Behavior
start
stop
accelerate
break
…
Software object
Car
Field
make: Honda
model: Civic
color: Black
vin: VIB08…
…
Method
start
stop
accelerate
break
…

What is Class?

VTC Academy
Thinking Ahead, Beyond Boundaries

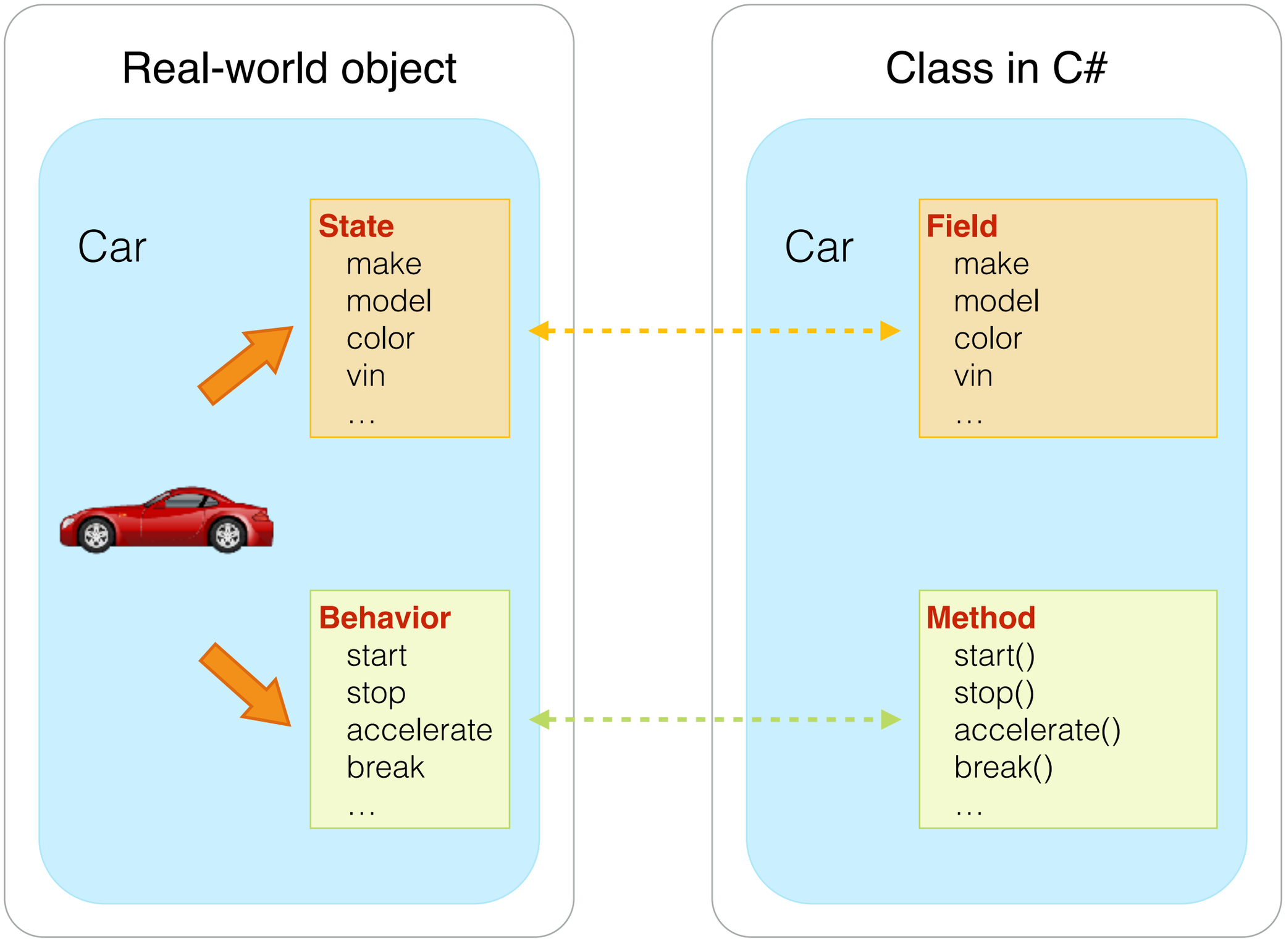
“Classes act as templates from which an instance of an object is created at run time. Classes define the properties of the object and the methods used to control the object's behavior

- Definition by Google -

Classes

- Classes provide the structure for objects
    - Define their prototype, act as template
- Classes define:
    - Set of attributes
        - Represented by variables and properties
        - Hold their state
- Set of actions (behavior)
    - Represented by methods
    - A class defines the methods and types of data associated with an object

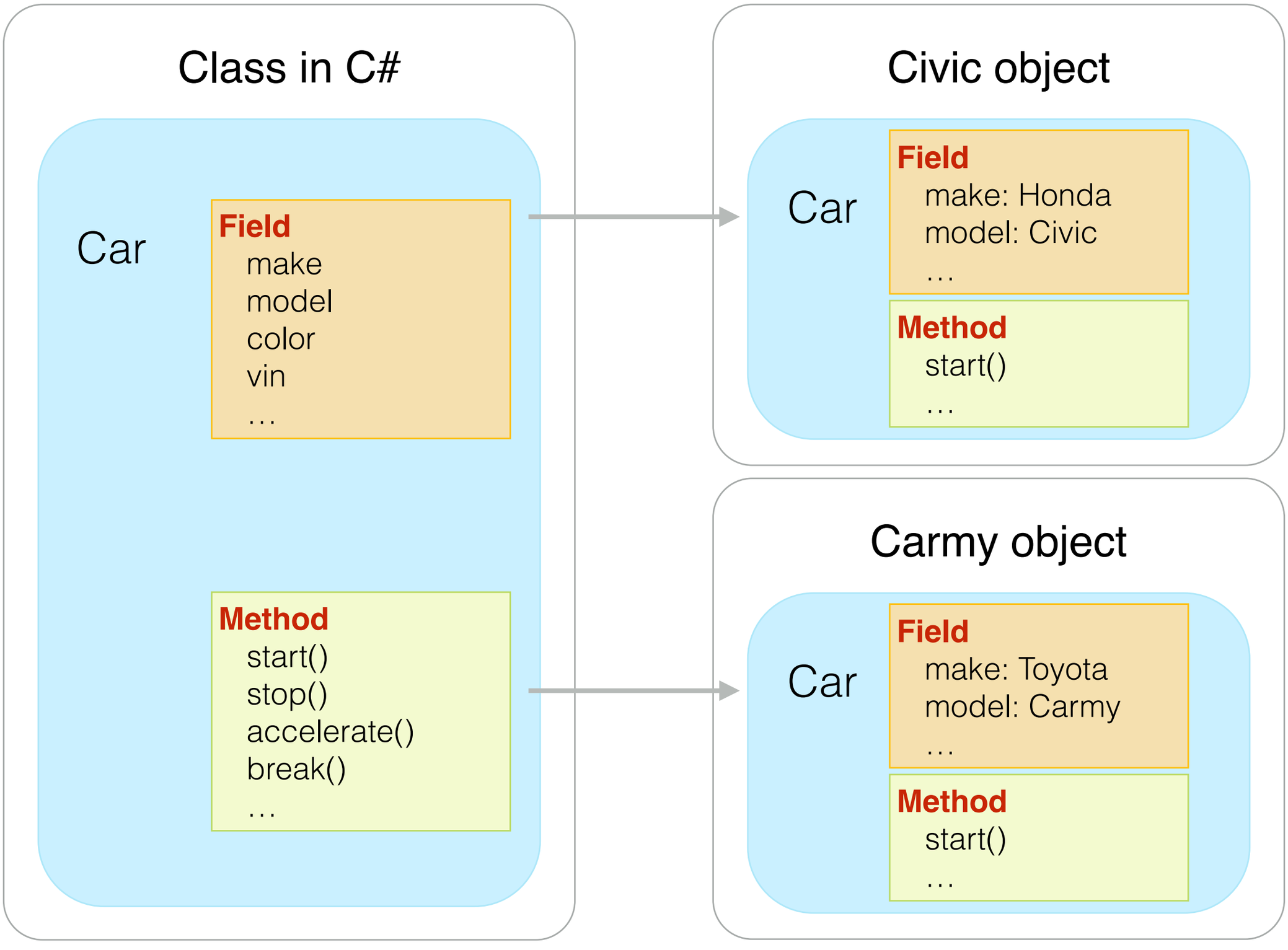
Classes


Real-world object
Car
State
make
model
color
vin
…
Behavior
start
stop
accelerate
break
…
Class in C#
Car
Field
make
model
color
vin
…
Method
start()
stop()
accelerate()
break()
…

Objects in C# Program

- An object is a concrete instance of a particular class
- Creating an object from a class is called instantiation
- Objects have state
  - Set of values associated to their attributes
- Example:
  - Class: Car
  - Objects: Civic Car, Honda Car

Objects in C# Program



Classes in C# Program

- Basic units that compose programs
- Implementation is encapsulated (hidden)
- Classes in C# can contain:
  - Fields (member variables)
  - Properties
  - Methods
  - Constructors
  - Inner types
  - Etc. (events, indexers, operators, …)

Defining a Class

- A class definition starts with the keyword class followed by the class name; and the class body enclosed by a pair of curly braces
- Syntax:

```
<access specifier> class  class_name
{
    // member variables
    <access specifier> <data type> variable1;
    <access specifier> <data type> variable2;
    ...
    // member methods
    <access specifier> <return type> method1(parameter_list)
    {
        // method body
    }
    <access specifier> <return type> method2(parameter_list)
    {
        // method body
    }
    ...
}
```

Defining a Class

- Example:

```
public class Car
{
    // fields of Car class
    public string make;
    public string model;
    public string color;
    public string vin;

    // methods of Car class
    public void Start()
    {
        Console.WriteLine("Car is starting!”);
    }
    public void Stop()
    {
        Console.WriteLine("Car is stoping!”);
    }
}
```

Properties and Fields

- Fields are data members of a class
- Can be variables and constants
- Accessing a field doesn’t invoke any actions of the object
- Example:

```
public string make;
public string model;
public string color;
public string vin;
```

Methods

- A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main.
- Methods are blocks of code that can take parameters and may or may not return a value
- A method implements the behavior of an object, which can be accessed by instantiating the object of the class in which its defined and then invoking the method
- Methods are also known as functions, procedures, and subroutines

Defining Methods

- Syntax:

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
   // Method Body
}
```

- Elements of a method:
  - **Access Specifier**: This determines the visibility of a variable or a method from another class
  - **Return type**: The return type is the data type of the value the method returns. Use **void** if the method no need to return value.
  - **Method name**: Method name is a unique identifier
  - **Parameter list**: Enclosed between parentheses, the parameters are used to pass and receive data from a method. It is optional
  - **Method body**: Contains the set of instructions needed to perform a specific task

Defining Methods

- Example:

```
class NumberCalculator
{
   public int FindMax(int num1, int num2)
   {
      // local variable declaration
      int result;

      if (num1 > num2)
         result = num1;
      else
         result = num2;

      return result;
   }
   ...
}
```

C# Constructors

- A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class
- If you don't provide a constructor for your class, C# creates one by default that instantiates the object and sets member variables to the default values
- A constructor has exactly the same name as that of class and it does not have any return type
- A contstructor often contains the initialisation code of class members
- Syntax:

```
public Class_Name(parameter_list)
{
    // object member initialisation
}
```

C# Constructors

- Example:

```
public class Car
{
    // fields of Car class
    public string make;
    public string model;
    public string color;

    // contructors without arguments of Car class
    public Car()
    {
        Console.WriteLine("Object is being created!");
    }
    // contructors with arguments of Car class
    public Car(string _make, string _model)
    {
        model = _model;
        make = _make;
    }

    // methods of Car class
    ...
}
```

Creating Car Object

- You can create an object from a class using keyword **new**
- Creating Car object without arguments:

```
Car civic = new Car();
Car carmy = new Car();
```

- Creating Car object with arguments:

```
Car x6 = new Car("BMW", "X6");
```

Accessing Fields and Calling Methods

- Before accessing fields or calling methods, you need create an instance of the class
- Accessing object's fields:

```
Car civic = new Car();
civic.color = "Black";
string _color = civic.color;
Console.WriteLine("Color of civic car: {0}", _color);
```

- Calling methods without return value:

```
Car x6 = new Car("BMW", "X6");
x6.start();
```

- Calling methods with return value:

```
NumberCalculator numbercal = new NumberCalculator();
int result = numbercal.FindMax(10, 8);
```

Static Members

- A non-static class can contain static methods, fields
- The static member is callable on a class even when no instance of the class has been created
- The static member is always accessed by the class name, not the instance name
- Static methods cannot access non-static fields
- You declare static class members by using the **static** keyword before the return type of the member

Static Members Example

- Decalaring static methods:

```
public class Automobile
{
    // static field
    public static int NumberOfWheels = 4;

    // static method
    public static void Drive() { }
    public static event EventType RunOutOfGas;

    // other non-static fields and methods...
}
```

- To access a static class member, use the name of the class instead of a variable name to specify the location of the member:

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

Access Modifiers

- All types and type members have an accessibility level, which controls whether they can be used from other code in your assembly or other assemblies

- Types of acess modifiers:

  - **public**: The type or member can be accessed by any other code in the same assembly or another assembly that references it.

  - **private**: The type or member can be accessed only by code in the same class or struct.

  - **protected**: The type or member can be accessed only by code in the same class or struct, or in a class that is derived from that class.

  - **internal**: The type or member can be accessed by any code in the same assembly, but not from another assembly.

Summary

- **Object-Oriented Programming** (**OOP**) is a programming paradigm based on the concept of "objects", which may contain data fields and methods.
- An object in C# program is a concrete instance of a particular class
- The static member is callable on a class even when no instance of the class has been created
- A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class
- Types of acess modifiers: public, private, protected, internal