Classes

Subjects:

-Introduction
-Defining a Class and an Object
-Defining a class
-What`s the object?
-Accessing the Members of an Object
-An Object-Based Program Example

Introduction

You can freely declare and use all the intrinsic data types — such as `int`, `double`, and `bool` — to store the information necessary to make your program the best it can be. For some programs, these simple variables are enough. However, most programs need a way **to bundle related data into a neat package**.

C# provides arrays and other collections for gathering into one structure groups of ***like-typed variables, such as `strings` or `ints`***. A hypothetical college, for example, might track its students by using an array. But a student is much more than just a name — how should this type of program represent a student?

Some programs need to bundle pieces of data that logically belong together but aren't of the same type.

A college enrollment application handles students, each with her own **name**, **rank** (grade-point average), and **serial number.** Logically,

- ✓the **student's name** may be a `string`;
- ✓the **grade-point average**, a `double`;
- ✓the **serial number**, a `long`.

That type of program needs a way to bundle these three different types of variables into a single structure named **`Student`**. Fortunately, C# provides a structure known as the ***class for accommodating groupings of unlike-typed variables***.

Defining a Class and an Object

A class is a bundling of unlike data and functions that logically belong together into one tidy package. Classes are designed to represent ***concepts***.

Computer science models the world via structures that represent concepts or things in the world, such as bank accounts, games, customers, game boards, documents, and products. Analysts say that "**a class maps concepts from the problem into the program**".

For example, your problem might be **to build a traffic simulator that models traffic patterns for the purpose of building streets, intersections, and highways**. Any description of a problem concerning traffic would include the term ***vehicle*** in its solution. ***Vehicles have a top speed*** that must be figured into the equation. They also have **a weight**, and some of them **are clunkers**. In addition, **vehicles stop** and **vehicles go**. Thus, as a concept, ***vehicle is part of*** **the problem domain**.

A good C# traffic-simulator program would necessarily include the class **`Vehicle`**, **which describes the relevant properties of a vehicle**. The C# **`Vehicle`** class would have properties such as **`topSpeed`**, **`weight`**, and **`isClunker`**.

Class is a user-defined type that is composed of **field data** (often called **member variables**) and **members that operate on this data** (such as **constructors**, **properties**, **methods**, **events**, and so forth). Collectively, **the set of field data represents the "state" of a class instance** (otherwise known as an **object**).

Defining a class

An example of the class **`Vehicle`** may appear this way:

```
public class Vehicle
 {
    public string model; // Name of the model
    public string manufacturer; // Ditto
    public int numOfDoors; //The number of doors on the vehicle
    public int numOfWheels; // You get the idea.
 }
```

A class definition begins with the words `public class`, followed by the name of the class — in this case, `Vehicle`. Like all names in C#, the **name of the class is case sensitive**. C# doesn't enforce any rules concerning class names, but an **unofficial rule holds that the name of a class starts with a capital letter**.

Defining a class

```
public class Vehicle
 {
    public string model; // Name of the model
    public string manufacturer; // Ditto
    public int numOfDoors; //The number of doors on the vehicle
    public int numOfWheels; // You get the idea.
 }
```

The members of a class are variables

The class name is followed by a pair of open and closed braces. Within the braces, you have zero or more ***members***. ***The members of a class are variables*** **that make up the parts of the class**.

In this example, class `Vehicle` starts with the member `string model`, which contains the name of the model of the vehicle. The second member of this `Vehicle` class example is `string manufacturer`. The other two properties are the number of doors and the number of wheels on the vehicle.

Defining a class

As with any variable, make the names of the members as descriptive as possible.

```
public class Vehicle
  {
     public string model; // Name of the model
     public string manufacturer; // Ditto
     public int numOfDoors; //The number of doors on the vehicle
     public int numOfWheels; // You get the idea.
  }
```

The **public** **modifier** in front of the class name **makes the class universally accessible throughout the program**. Similarly, the `public` **modifier in front of the member names makes them accessible to everything else in the program**. Other modifiers are possible.

The class definition should describe the properties of the object that are salient to the problem at hand.

What`s the object?

A **class object** is declared in a similar (but not identical) fashion to declaring an intrinsic object such as an `int`. The term ***object is used universally to mean a "thing"***. An `int` variable is an `int` object. A vehicle is a `Vehicle` object. The following code segment creates a car of class `Vehicle`:

```
Vehicle myCar;
myCar = new Vehicle();
```

The first line declares a variable `myCar` of type `Vehicle`, just like you can declare a `somethingOrOther` of class `int`. (***Yes, a class is a type, and all C# objects are defined as classes***.)

The `new Vehicle()` command creates a specific object of type `Vehicle` and stores the location into the variable `myCar`. The **`new`** operator creates a **new block of memory** in which your program can store the properties of `myCar`.

In C# terms, you say that `myCar` is an object of class `Vehicle`. You also say that `myCar` is an instance of `Vehicle`. In this context, ***instance means "an* example of"** or **"one of"**. You can also use the word *instance as a verb, as in instantiating* `Vehicle`. *That's what* ***new*** *does.*

What`s the object?

Compare the declaration of `myCar` with that of an `int` variable named `num`:

```
int num;
num = 1;
```

```
Vehicle myCar;
myCar = new Vehicle();
```

The first line declares the variable `num`, and the second line assigns an already created constant of type `int` into the location of the variable `num`.

The intrinsic `num` and the object `myCar` are stored differently in memory. The constant `1` doesn't occupy memory because both the CPU and the C# compiler already know what `1` is. Your CPU doesn't have the concept of `Vehicle`. **The `new Vehicle` expression allocates the memory necessary to describe a vehicle to the CPU, to C#.**

Accessing the Members of an Object

Each object of class `Vehicle` has its own set of members. The following expression stores the number `1` into the `numberOfDoors` member of the object referenced by `myCar`:

```
myCar.numOfDoors = 1;
```

Every C# operation must be evaluated by type as well as by value. The object `myCar` is an object of type `Vehicle`. The variable `myCar.numOfDoors` is of type `int`. (Look again at the definition of the `Vehicle` class.) The constant `1` is also of type `int`, so **the type of the variable on the right side of the assignment operator matches the type of the variable on the left**.

Similarly, the following code **stores a reference** to the `strings` describing the model and manufacturer name of `myCar`:

```
myCar.manufacturer = "BMW";
myCar.model = "Isetta";
```

An Object-Based Program Example

The simple **`VehicleDataOnly`** program performs these tasks:

- **Define the class** `Vehicle`.
- **Create an object** `myCar`.
- **Assign properties to** `myCar`.
- **Retrieve those values from the object for display**.

Here's the code for the `VehicleDataOnly` program:

```
// VehicleDataOnly -- Create a Vehicle object, populate its
// members from the keyboard, and then write it back out.
using System;
    namespace VehicleDataOnly
        {
        public class Vehicle
            {
            public string model; // Name of the model
            public string manufacturer; // Ditto
            public int numOfDoors; // The number of doors on
                                   //the vehicle
            public int numOfWheels; // You get the idea.
            }
```

An Object-Based Program Example

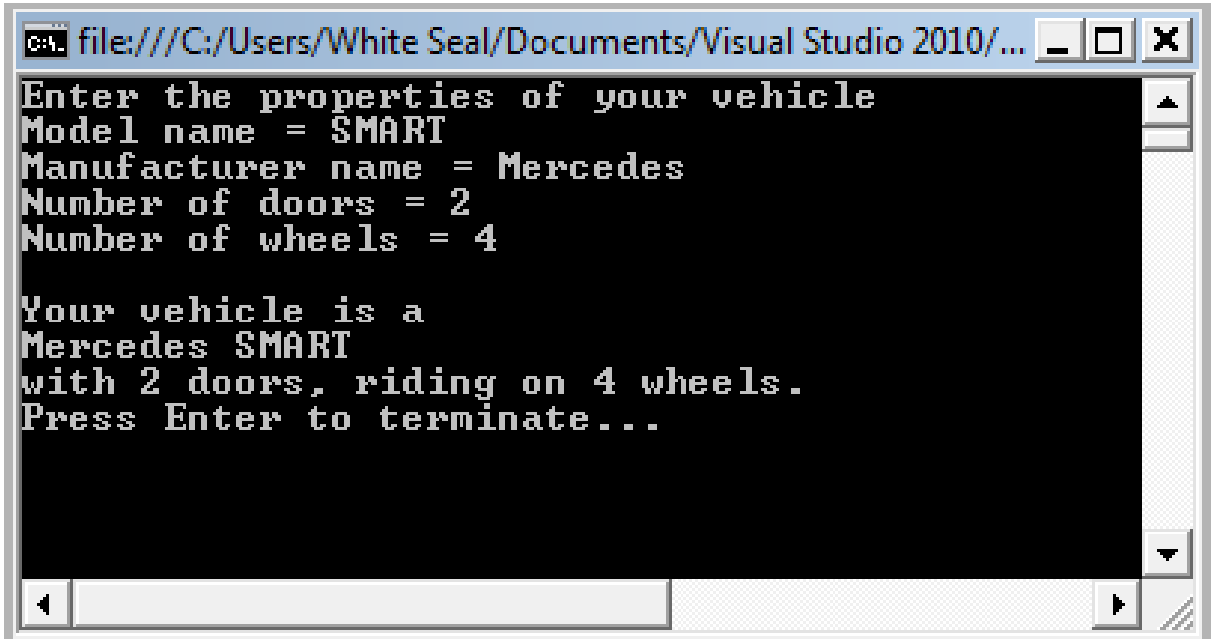
```
public class Program
{
    // This is where the program starts.
    static void Main(string[] args)
    {
        // Prompt user to enter name.
        Console.WriteLine("Enter the properties of your
        vehicle");
        // Create an instance of Vehicle.
        Vehicle myCar = new Vehicle();
        // Populate a data member via a temporary variable.
        Console.Write("Model name = ");
        string s = Console.ReadLine();
        myCar.model = s;
        // Or you can populate the data member directly.
        Console.Write("Manufacturer name = ");
        myCar.manufacturer = Console.ReadLine();
```

An Object-Based Program Example

```
        // Enter the remainder of the data.
        // A temp is useful for reading ints.
        Console.Write("Number of doors = ");
        s = Console.ReadLine();
        myCar.numOfDoors = Convert.ToInt32(s);
        Console.Write("Number of wheels = ");
        s = Console.ReadLine();
        myCar.numOfWheels = Convert.ToInt32(s);
        // Now display the results.
        Console.WriteLine("\nYour vehicle is a ");
        Console.WriteLine(myCar.manufacturer    +    "    "    +
        myCar.model);
        Console.WriteLine("with " + myCar.numOfDoors + " doors,
        "
        + "riding on " + myCar.numOfWheels
        + " wheels.");
        // Wait for user to acknowledge the results.
        Console.WriteLine("Press Enter to terminate...");
        Console.Read();
        }
    }
}
```

An Object-Based Program Example

The program listing begins with a definition of the `Vehicle` class. **The definition of a class can appear either before or after** `class Program` **— it doesn't matter**.
The program creates an object `myCar` of class `Vehicle` and then populates each field by reading the appropriate data from the keyboard. (The input data isn't — but should be — checked for legality.) The program then writes `myCar`'s properties in a slightly different format. The output from executing this program appears this way:



The calls to `Read()` as opposed to `ReadLine()` leave the cursor directly after the output string, which makes the user's input appear on the same line as the prompt. In addition, inserting the newline character `'\n'` in a write generates a blank line without the need to execute `WriteLine()` separately.

Discriminating between Objects

A program can create numerous objects of the same class, as shown in this example:

```
Vehicle car1 = new Vehicle();
car1.manufacturer = "Studebaker";
car1.model = "Avanti";
// The following has no effect on car1.
Vehicle car2 = new Vehicle();
car2.manufacturer = "Hudson";
car2.model = "Hornet";
```

Creating an object `car2` and assigning it the manufacturer name `Hudson` has no effect on the `car1` object (with the manufacturer name `Studebaker`). In part, the ability to discriminate between objects is the real power of the class construct. The object associated with the Hudson Hornet can be created, manipulated, and dispensed with as a single entity, separate from other objects, including the Avanti.

Can You Give Me References?

The dot operator and **the assignment operator** are the only two operators defined on reference types:

```
// Create a null reference.
Vehicle yourCar;
// Assign the reference a value.
yourCar = new Vehicle();
// Use dot to access a member.
yourCar.manufacturer = "Rambler";
// Create a new reference and point it to the same object.
Vehicle yourSpousalCar = yourCar;
```

The first line creates an object `yourCar` without assigning it a value. **A reference that hasn't been initialized is said to point to the *null object*.**

***Any* attempt to use an uninitialized (null) reference generates an immediate error that terminates the program. The C# compiler can catch most attempts to use an uninitialized reference and generate a warning at build-time.**

Can You Give Me References?

```
// Create a null reference.
Vehicle yourCar;
// Assign the reference a value.
yourCar = new Vehicle();
// Use dot to access a member.
yourCar.manufacturer = "Rambler";
// Create a new reference and point it to the same object.
Vehicle yourSpousalCar = yourCar;
```

The second statement creates a new `Vehicle` object and assigns it to `yourCar`. **The last statement** in this code snippet assigns the reference `yourSpousalCar` to the reference `yourCar`. This action causes your `SpousalCar` to refer to the same object as `yourCar`. This relationship is shown in **Figure**.

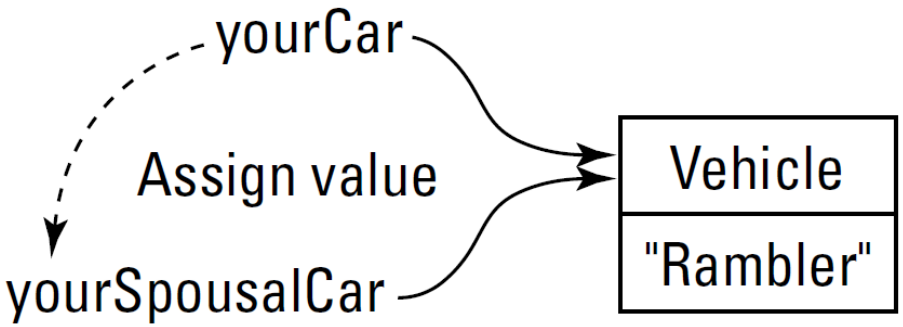


Figure: Two references to the same object

Can You Give Me References?

The following two calls have the same effect:

```
// Build your car.
Vehicle yourCar = new Vehicle();
yourCar.model = "Kaiser";
// It also belongs to your spouse.
Vehicle yourSpousalCar = yourCar;
// Changing one changes the other.
yourSpousalCar.model = "Henry J";
Console.WriteLine("Your car is a " + yourCar.model);
```

Executing this program would output `Henry J` and **not** `Kaiser`. Notice that `yourSpousalCar` doesn't point to `yourCar`; rather, both `yourCar` and `yourSpousalCar` refer to the same vehicle.

Can You Give Me References?

In addition, the reference `yourSpousalCar` would still be valid, even if the variable `yourCar` were somehow "lost" (if it went out of scope, for example), as shown in this chunk of code:

`null` is the default value for all reference types of variables.

```
// Build your car.
    Vehicle yourCar = new Vehicle();
    yourCar.model = "Kaiser";
// It also belongs to your spouse.
    Vehicle yourSpousalCar = yourCar;
// When your spouse takes your car away ...
        yourCar = null;
// yourCar now references the "null object".
// yourSpousalCar still references the same vehicle
Console.WriteLine("your car was a " + yourSpousalCar.model);
```

Executing this program generates the output `Your car was a Kaiser`, even though **the reference `yourCar` is no longer valid**. **The object is no longer *reachable from the reference yourCar***. *The object* doesn't become completely unreachable until both `yourCar` and `yourSpousalCar` are "lost" or nulled out.

At that point the C# ***garbage collector steps in and returns the space formerly used by that particular*** `Vehicle` object to the pool of space available for allocating more `Vehicles` (or `Students`, for that matter).

Can You Give Me References?

Making one *object variable* (***a variable of a reference type***, *such as* `Vehicle` or `Student`, rather than one of a simple type such as `int` or `double`) **point to a different object makes storing and manipulating reference objects in arrays and collections quite efficient**.

Each element of the array stores a reference to an object, and when you swap elements within the array, you're just moving references, not the objects themselves.

References have a fixed size in memory, unlike the objects they refer to.

Classes That Contain Classes

The members of a class can themselves be references to other classes. For example, vehicles have motors, which have power and efficiency factors, including displacement. You could throw these factors directly into the class this way:

```
public class Vehicle
{
public string model; // Name of the model
public string manufacturer; // Ditto
public int numOfDoors; // The number of doors on the vehicle
public int numOfWheels; // You get the idea.
// New stuff:
public int power; // Power of the motor [horsepower]
public double displacement; // Engine displacement [liter]
}
```

Classes That Contain Classes

The motor is a concept of its own and deserves its own class:

```
public class Motor
{
public int power; // Power [horsepower]
public double displacement; // Engine displacement [liter]
}
```

You can combine this class into the `Vehicle` :

```
public class Vehicle
{
public string model; // Name of the model
public string manufacturer; // Ditto
public int numOfDoors; // The number of doors on the
//vehicle
public int numOfWheels; // You get the idea.
public Motor motor;
}
```

Classes That Contain Classes

Creating `myCar` now appears this way:

```
// First create a Motor.
      Motor largerMotor = new Motor();
      largerMotor.power = 230;
      largerMotor.displacement = 4.0;
// Now create the car.
      Vehicle friendsCar = new Vehicle();
      friendsCar.model = "Cherokee Sport";
      friendsCar.manufacturer = "Jeep";
      friendsCar.numOfDoors = 2;
      friendsCar.numOfWheels = 4;
// Attach the motor to the car.
      friendsCar.motor = largerMotor;
```

From `Vehicle`, you can access the motor displacement in two stages. You can take one step at a time, as this bit of code shows:

```
Motor m = friendsCar.motor;
Console.WriteLine("The motor displacement is " +
m.displacement);
```

Classes That Contain Classes

Or, you can access it directly, as shown here:

```
Console.Writeline("The motor displacement is " +
friendsCar.motor.displacement);
```

Either way, you can access the displacement only through the `Motor`.

Generating Static in Class Members

Most data members of a class are specific to their containing object, not to any other objects. Consider the `Car` class:

```
public class Car
{
public string licensePlate; // The license plate ID
}
```

Because the license plate ID is an ***object property, it describes each object of class*** `Car` ***uniquely***. For example, my car has a license plate that's different from yours; otherwise, you may not make it out of your driveway, as shown in this bit of code:

```
Car myCar = new Car();
myCar.licensePlate = "XYZ123";
Car yourCar = new Car();
yourCar.licensePlate = "ABC789";
```

However, **some properties exist that all cars share**. For example, the number of cars built is a property of the class Car but not of any single object. **These *class properties are flagged in C# with the keyword*** **`static`**.

Generating Static in Class Members

```
public class Car
{
public static int numberOfCars; // The number of cars built
public string licensePlate; // The license plate ID
}
```

Static members aren't accessed through the object. Instead, you access them by way of the class itself, as this code snippet demonstrates:

```
// Create a new object of class Car.
Car newCar = new Car();
newCar.licensePlate = "ABC123";
// Now increment the count of cars to reflect the new one.
Car.numberOfCars++;
```

The object member `newCar.licensePlate` is accessed through the object `newCar`, and the class (`static`) member `Car.numberOfCars` is accessed through the class `Car`. **All `Cars` share the same `numberOfCars` member, so each car contains exactly the same value as all other cars**.

Class members are static members. Nonstatic members are specific to each **"instance"** (each individual object) and are ***instance members***.

Defining `const` and `readonly` Data Members

One special type of static member is the `const` data member, **which represents a constant**. You must establish the value of a `const` variable in the declaration, and **you cannot change it anywhere within the program**, as shown here:

```
class Program
{
// Number of days in the year (including leap day)
public const int daysInYear = 366; // Must have initializer.
public static void Main(string[] args)
{
// This is an array.
int[] maxTemperatures = new int[daysInYear];
for(int index = 0; index < daysInYear; index++)
{
// . . .accumulate the maximum temperature for each
// day of the year . . .
}
}
}
```

Defining `const` and `readonly` Data Members

You can use the constant `daysInYear` in place of the value `366` anywhere within your program. The `const` variable is useful because it can replace a mysterious number such as `366` with the descriptive name `daysInYear` to enhance the readability of your program.

C# provides another way to declare constants — you can preface a variable declaration with the `readonly` modifier, like so:

```
public readonly int daysInYear = 366; //This could also be
//static.
```

As with `const`, after you assign the initial value, it can't be changed. The `readonly` approach to declaring constants is usually preferable to `const`. You can use `const` with class data members and inside class methods. But `readonly` isn't allowed in a method.

An **alternative convention** also exists **for naming constants**. Rather than name them like variables (as in `daysInYear`), many programmers prefer to **use uppercase letters separated by underscores**, as in `DAYS_IN_YEAR`. This convention separates constants clearly from ordinary read-write variables.

Thanks for attention