Module 1 Day 9

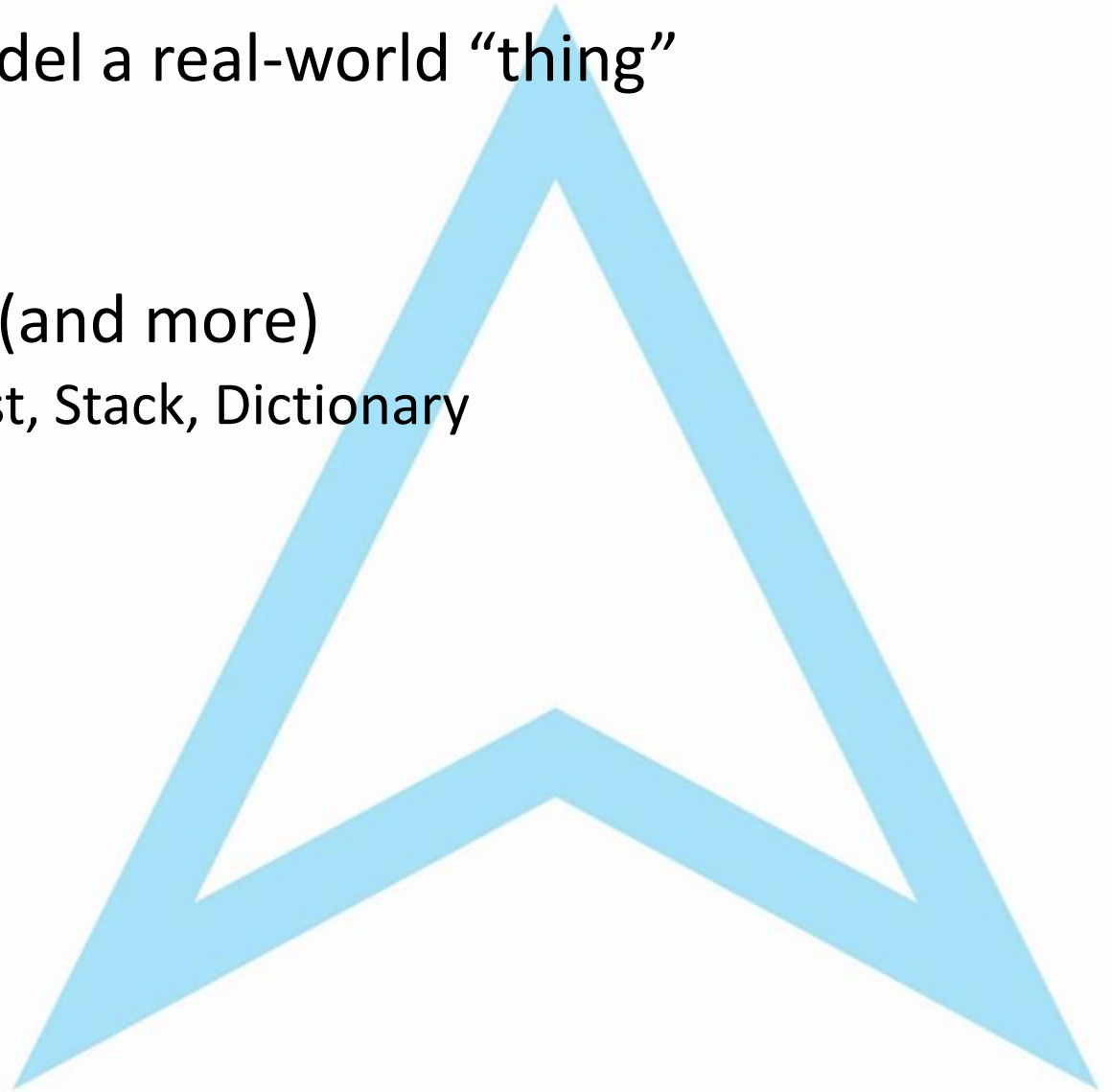
Introduction to Classes

What makes an application?

- Program Data
  - ✓ Variables & .NET Data Types
  - ✓ Arrays
  - ✓ More Collections (list, dictionary, stack, queue)
  - ❖ Classes and objects (OOP)
- Program Logic
  - ✓ Statements and expressions
  - ✓ Conditional logic (if)
  - ✓ Repeating logic (for, foreach, do, while)
  - ✓ Methods (functions / procedures)
  - ❖ Classes and objects (OOP)
  - ❑ Frameworks (MVC)
- Input / Output
  - User
    - ✓ Console read / write
    - ❑ HTML / CSS
    - ❑ Front-end frameworks (HTML / CSS / JavaScript)
  - Storage
    - ❑ File I/O
    - ❑ Relational database
    - ❑ APIs

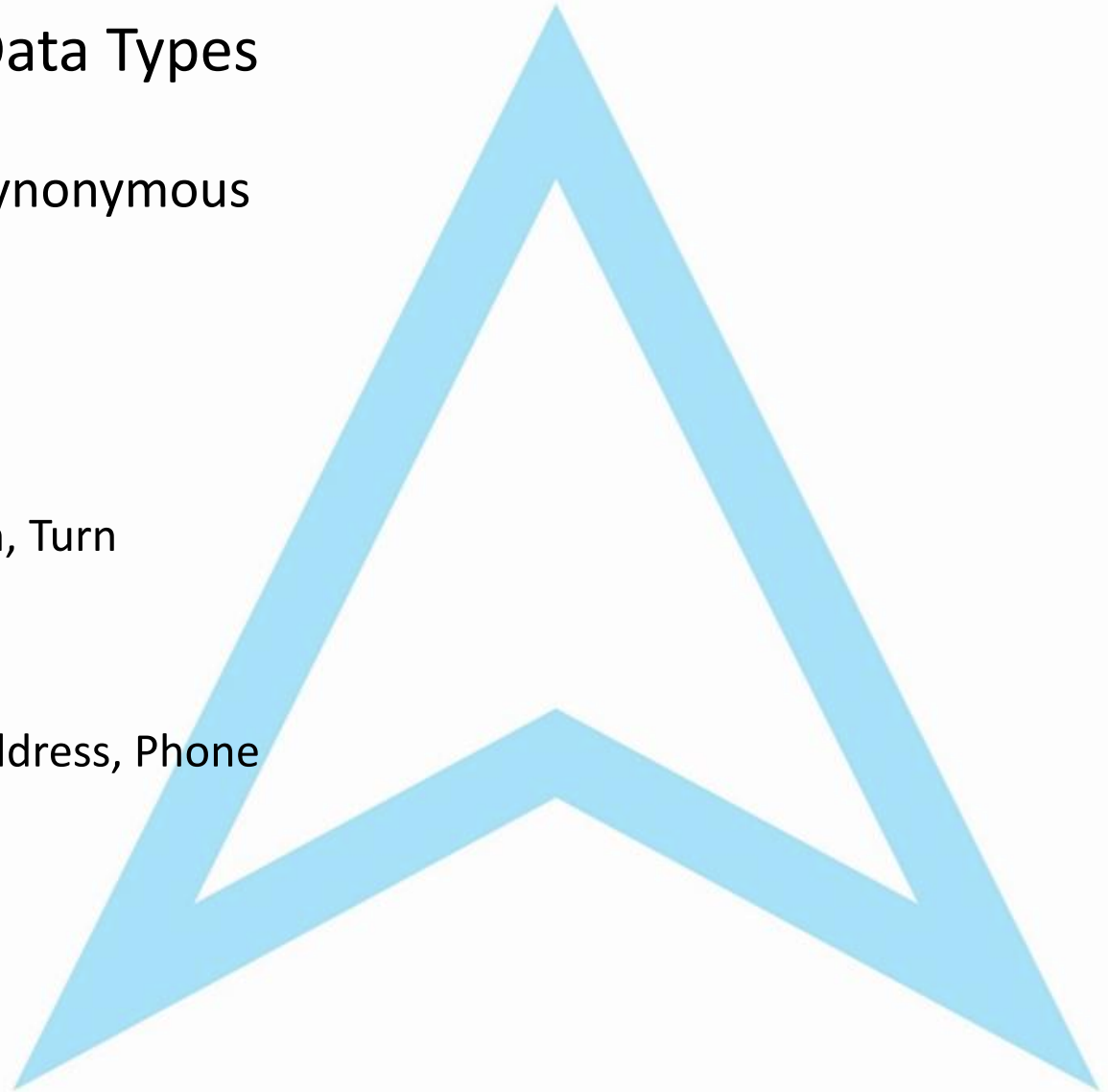
Classes

- Combine Data and Behavior to model a real-world “thing”
  - Data: Variables / properties
  - Behavior: Methods
- So far, we have used these classes (and more)
  - int, double, string, Console, Array, List, Stack, Dictionary



Classes

- Now we are going to write our own Data Types
  - These are called Classes in OO parlance
  - Remember that Classes and Types are synonymous
- e.g., Car
  - Data - describes it - adjectives
    - Make, model, color, Engine State, Gear
  - Behavior – what it can do - verbs
    - Start, Change Gear, Speed Up, Slow Down, Turn
- e.g., Contact
  - Data
    - First Name, Last Name, Birthday, Email Address, Phone
  - Behavior
    - Send Mail, Call, Text



Classes – Fields: One way to store data

- Data stored inside each object instance
- A variable declaration, at the class level
- "member variable"
- Usually private (accessible only by this class, but not by other classes)
- For public data, we usually use Properties

```
public class Car
{
    /*****************************************************
     * Private fields (not properties)
     * ***************************************************/
    private int maxSpeed = 120;
```

Properties

(Data)

Classes - Properties

- Automatic Properties
- Derived Properties

```
// Type (class) to represent an automobile
public class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int Year { get; set; }
    // A derived property for the age of the car
    public int Age
    {
        get
        {
            return DateTime.Now.Year - this.Year;
        }
    }
```

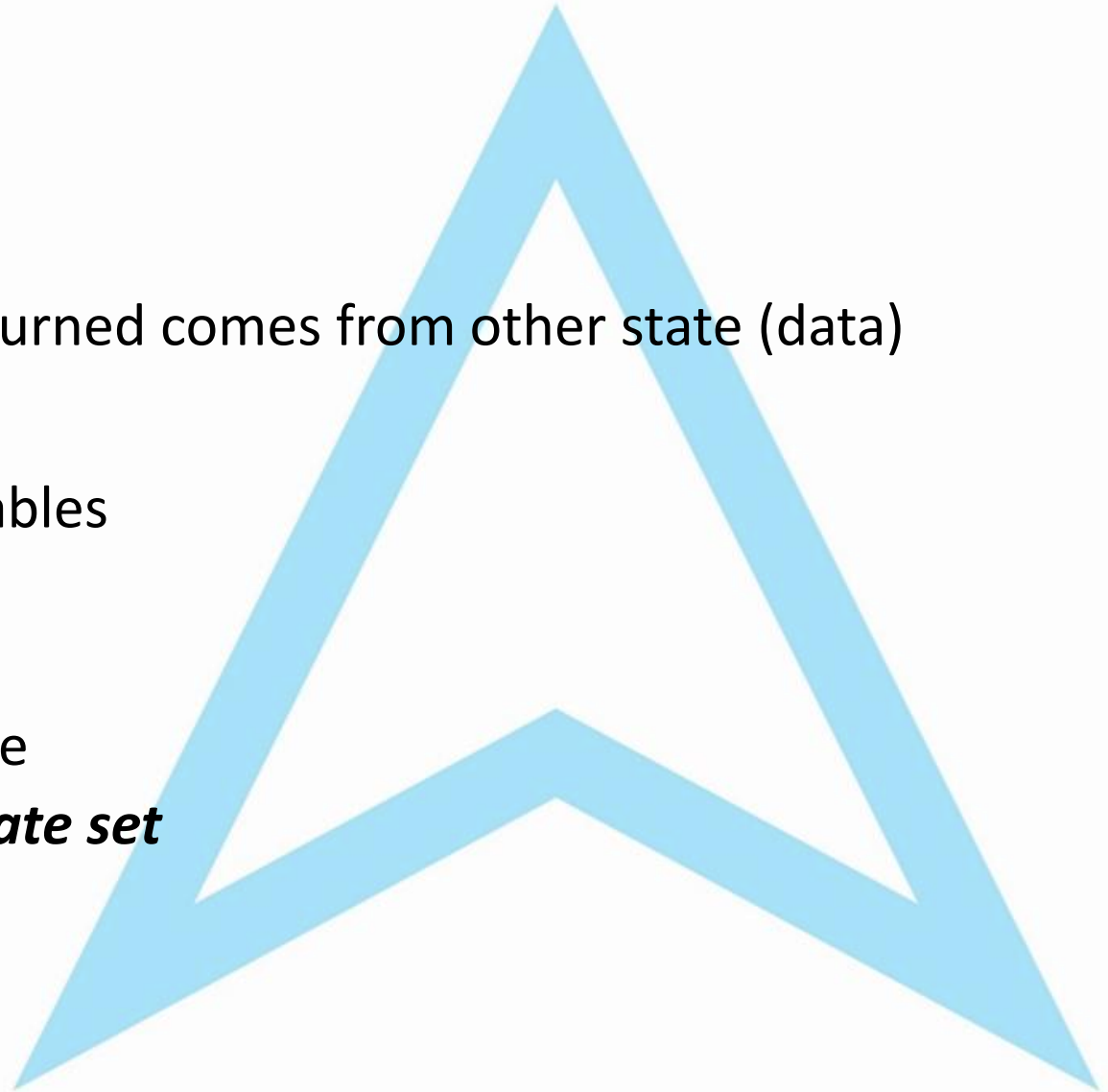
Classes - Properties

- Properties “backed by” a private variable (field)

```
private string gear;
public string Gear
{
    get
    {
        return this.gear;
    }
    set
    {
        // Check to make sure the gear can be set, based on where it is now...
        // Then, set it...
        this.gear = value;
    }
}
```

Classes - Properties

- Automatic Properties
  - { get; set; }
- Derived properties
  - “getter” with no setter; the value returned comes from other state (data)
- Access modifiers
  - For class and for properties and variables
  - *private* and *public* (for now)
  - Users of the object can “see” public
  - Only the class itself can access private
  - We can have a ***public get*** with a ***private set***



Properties Summary

Property	Get	Set	Notes
Automatic	✔	✔	`public int Age { get; set; }`
Derived	✔ with code	✘	`public string Color { get { // Code goes here… return "some value"; } }` "getter" uses other data to calculate and return a value
"backed"	✔	✔	Stores value in a private member variable. The Get and Set methods access the private field.
Get-private-set	✔	✔ (private)	`public int Speed { get; private set; }` Outsiders can see it, only this class can change it
Read-only	✔	✘	`public string Suit { get; }` Value can ONLY be set inside class constructor

Methods

(Behavior)

Classes - Methods

- Methods provide “behavior”. We’ve written lots of these.
- The **this** keyword allows access to the data held by *this instance* of the class

```
// Public can see the speed, but cannot set it directly
public int Speed { get; private set; }

// Accelerate 1 mph
public int Accelerate()
{
    // Check if car is in gear, then set speed
    this.Speed++;
    return this.Speed;
}
```

Classes - Constructors

- Special method that is invoked as the object is being instantiated
- Same name as the class, and NO return type
- Can accept parameters
- If you don't define one, a "default constructor" exists automatically

```
// Constructor for a Car
public Car(int year, string make, string model)
{
    this.Year = year;
    this.Make = make;
    this.Model = model;
    this.gear = "P";
}
```

Read-only - Properties

- Only get;
- Can only be set from the class constructor

```
// Vehicle ID.  If this changes after a car is constructed, that would be fraud
1 reference
public string VIN { get; }
// Vehicle Make (Chevy). Never changes.
2 references
public string Make { get; }
// Vehicle Model (Corvette). Never changes
2 references
public string Model { get; }
// Year the car was built. Never changes.
3 references
public int Year { get; }
```

Method Overloading

- Change behavior of a method based on how it is called
- Define another method:
  - With the **same name**
  - With a **different set of parameters**, as defined by their data type and order
    - Differing by *parameter name only* will not make it different

```
// Accelerate 1 mph
public int Accelerate()
{
    return Accelerate(1);
}

// Accelerate a certain number of mph. Can be + or -.
public int Accelerate(int amount)
{
    // check if car is in gear, then set speed
    this.Speed += amount;
    return this.Speed;
}
```

Method Overloading

A method overloaded MUST have the same name, plus:

- Overloaded methods MUST change the argument list
- Overloaded methods MAY change the return type
- Overloaded methods MAY change the access modifier

Encapsulation

- The action of enclosing something in or as if ***in a capsule***
- From Wikipedia
  - A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.
  - A language mechanism for restricting direct access to some of the object's components.
- Mike's words
  - Bundling stuff together which goes together (as in classes)
    - Models real-world as closely as possible - **maintainable**
  - Not showing outsiders any more than they need to know (access modifiers)
    - **Loosely couples** your system; makes system more **flexible**

Encapsulation

- Access modifiers help us encapsulate

```
public int Property { get; set; }          //public set
public int Property { get; }               //readonly set w/in constructor
public int Property { get; private set; } //private set
```

- Read-only Properties
  - Value can *only* be set by the object in its constructor
  - After that, the value cannot change
- How can we better encapsulate the Card class?
  - Which properties should be set only when the card is created?
  - Which properties should be set only by the Card class itself?
  - Which properties should be freely available to be set by the public?

Lecture Code Goals

- Card
  - Create a Card class to represent a standard American playing card
  - Properly encapsulate the Card class
- Deck
  - Create a Deck class to represent a standard deck of cards
  - How should we encapsulate the members of the Deck?
- Card Game
  - Deal a hand to each of 2 players
  - Print out the two hands
- Relationships
  - As important as the classes themselves is how these classes relate to one another
    - Reference

Card Game

- Design
  - What are the classes?
  - What are the properties and methods of each class? What is private and what is public?
  - What are the relationships between the classes?
- UML
  - Unified Modelling Language
  - Class Diagram expresses design

Static

- Member belongs to the class/type, *not* to individual object instances
  - "Class variable or method" vs. "instance variable or method"
- Property or field
  - The data is stored once, regardless of how many instances there are
  - "Shared" by all instances
  - You don't need an instance (object) to access the property
    - Console.ForegroundColor
    - DateTime.Now
- Method
  - You don't need an instance (object) to access the method
    - Console.Writeline
    - Math.Round
- Class
  - Only has static members; cannot be created (new'd)