OOP Fundamentals

Part 1

OOP

- In most traditional programming languages (Pascal, Fortran, Basic, PL/I, Cobol, C) the world is divided into code and data.
- You write code to manipulate data.
- But programmers have also searched for ways to organize code and data. e.g., putting all related functions into same source file.
- C has the concept of a **struct** that lets us organize data.
- See C program that calculates day of year values

OOP

- C uses a struct to group related data together.
- struct Date
- {
- int year;
- int month;
- int day;
- };
- You can then define a variable of type Date like so:
- struct Date today;

OOP

- You initialize the fields in the struct like so:
- today.year = 2009;
- today.month = 2;
- today.day = 20;
- In C, you can also define the struct and initialize like so:
- struct Date birthDate = {1985, 10, 12};

To write your day of year function, you might write a helper function that determines whether a particular date is a leap year:

```
int IsLeapYear(int year)
{
  return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
}
```

Your day of year function makes use of IsLeapYear:

```
int DayOfYear(struct Date date)
{
  static int MonthDays[12] = {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334};
  return MonthDays[date.month – 1] + date.day  +
         ((date.month > 2) && IsLeapYear(date.year));
}
```

OOP

- Notice in C that structs can only contain data. So the code and the data are still separate.
- But since IsLeapYear and DayOfMonth functions are closely related to the Date struct, it makes sense to consolidate them inside the Date structure itself.
- Doing so turns it into a C++ program [see program CppDate.cpp]

OOP

- Notice code bulk is smaller – no arguments to functions which can now rightly be called **methods**. They can reference the structure fields directly because they are part of the structure.
- Note also struct keyword is eliminated from the declaration of myDate in Main().
- It now looks like myDate is a normal variable of type Date.
- In OOP, the myDate variable can now be said to be an **object** of type Date, or an **instance** of Date.
- Finally, note that DayOfYear can be called directly in main…by using a period and method name.

OOP

- **From a meta perspective, in C we were asking a function to crunch some data for us. Now we are asking the Date structure to return *its* DayOfYear.**
- **This is one aspect of object oriented programming…consolidating code and data into a single unit.**
- **But in C++, the consolidating unit is not a struct, it's a class. So we would turn keyword struct into class and add public: as first line in class.**
- **class Date**
- **{**
- **public:**
- **as before**
- **}**
- **In a C++ struct, everything is public and can be accessed in main.**
- **In a class, everything (by default) is private. So if want to call methods directly, we must make them public.**

OOP

- Consult the C# version of this program.
- Date is a **class**.
- This class has 5 **members**: 3 data members (year, month, day) are called **fields**.
- 2 code members are called **methods**.
- Variable myDate is an **object** of type Date. It's also called an **instance** of the Date class.

OOP

- A class is the encapsulation of data and the methods that operate on that data.
- It serves as a blueprint in the sense that we can make any number of concrete instances of this class (called objects), each of which has the data and methods defined in the blueprint.
- What a class can contain:

OOP

- **Fields**
  - member variable that holds data
  - can apply static, readonly, and const
- **Methods**
  - Actual code that executes
- **Properties**
  - aka *smart fields* because they are actually methods that look like fields to a class's clients.

OOP

- **Constants**
  - A field with a value that cannot change
- **Indexers**
  - Like a smart array
  - It's a data member that lets an object be indexed through set and get methods
- **Attributes**
  - Values enclosed in square brackets that provide additional information about a target entity
- **Events**
  - Something that causes a piece of code to execute
- **Operators**
  - You can, through operator overloading, add standard mathematical operators to a class to enable you to write more intuitive code

- **Access Modifiers**
- **public** - member is accessible from outside the class and its hierarchy
- **protected** - member not visible outside the class and can only be accessed by the class and its descendents
- **private** - cannot be accessed outside the class at all. This is the default access if none is specified.
- **Internal** - visible only within the current compilation unit. It is like a hybrid of public and protected depending on where the code resides

OOP

- **The Main Method**
- Every C# application must have a Main() method as a method in one class that's defined as *public* and *static*
- This is the entry point of the program
- It does not matter which class – nor does it affect the order of compilation

OOP

- Common programming paradigm:
- class Employee { }
- class AppClass
- {
- public static void Main() {}
- }
- Employee is the problem domain class and will be instantiated in Main
- Second class contains the required entry point

OOP

- Can access command line arguments by declaring Main as
- public static void Main(string[] args)
- {
- foreach (string arg in args)
- System.Console.WriteLine("Argument: {0}", arg);
- } // note args[0] is NOT program name
- You can also define Main as
- public static int Main() { code… return 0; }
- Useful for console apps, rare for GUI apps

OOP

- **Constructors**
- Method that's called whenever an instance of an object is created.
- Benefit: Ensure that an object gets proper initialization before it's used.
- When a user creates an object, that object's constructor is first called and must return before the user can perform any other work with the object.

OOP

- A constructor
- Always has the same name as its class
- Never declares a return type (compile error if try)
- Objects are always created with the **new** operator
- SomeClass sc = new SomeClass(optional args)
- This allocates memory for the object and initializes any variables.

OOP

- Any instance variables not explicitly initialized in a constructor are set to defaults during construction process (zero for numbers, false for bool, null for objects).
- If no constructor is provided for a class, the complier provides a “default one” with no code.
- BUT…if there are constructors, and none is a public no-argument one, attempting to call such a no-argument constructor will reside in a compile error.

OOP

- Repeat:
- **A constructor can be called with no arguments only if there are no constructors for the class (in which case the compiler provided default one is used) or if the class defines a public, no argument constructor.**

- Very common to overload a Constructor. e.g., you have a Circle class and you want to draw it:
- public Circle(Point center, int radius); // point and radius
- public Circle (int x1, int y1, int x2, int y2); //bounding rect
- One constructor in a class can call another constructor with:
- public Circle(Point center, int radius):this(20, 20, 40, 40)

OOP

- **Static modifier**
- Static modifier was added to IsLeapYear in the C++ program
- By default, all variables and methods are **instance** unless they are declared with the static keyword.
- Instance – all objects get their own copy of the data
- Static – one copy of variable shared by all instances

OOP

- Sometimes you will see C# code that uses notation like **Classname**.method (e.g., Console.WriteLine()) and other times you see **objectVariable**.method (e.g., myDate.DayOfYear())
- That is the difference that comes from adding static to a method definition.
- In the Console class, WriteLine is defined as
- public static void WriteLine(string value)

OOP

- A static method belongs to the class itself (rather than to an object of that class).
- To call it, you must always use the Class.method syntax rather than Object.method syntax.
- **A static method can be called even when no instances of the class exist.**
- All of the methods in the Console class are static, so you never have to do *new* for a Console instance.

OOP

- In fact, if you tried to do this: myDate.IsLeapYear()
- it would not compile.
- You must do: Date.IsLeapYear(1997)
- i.e., since IsLeapYear was declared as static, you must use the class name to invoke it.

OOP

- Fields can also be defined as static, in which case one copy of the data item is shared among all instances of the class. e.g.,
- class CountMe
- {
- static int count = 0;
- public CountMe() { count++; } // increment in constructor
- }
- This class can increment the count variable each time a new instance of such a class is created. Since it's shared by all instances (i.e., one copy), it will reflect the current count of live instances.
- Nonstatic variables are not shared. Each instance of the class gets their own copies.

OOP

- **Constants Vs. Read-Only Fields**
- 2 rules for constants
- Value must be specified at compile time (explicitly or defaulted by compiler)
- Value must be a literal
- public const double pi = 3.1415;
- **By default, const members are static.**
- Thus, you don't really have to create a new instance of a class to access its constants.

OOP

- If you have need for a constant whose value is NOT known at compile time, you can use a **readonly** field and initialize it at run time.
- You can set the value for readonly members in only 1 place, and that is in the constructor. Once set, cannot be changed.
- public readonly int ScreenWidth;
- In constructor, can do this:
- this.ScreenWidth = SomeValue;

OOP

- Note: readonly fields, unlike constants, are **instance** members, so class must be newed before have access to it.
- If you wanted a readonly field to be static, you declare it this way:
- public static readonly int ScreenWidth;

OOP

- Then you create a special constructor called a **static constructor** – used to initialize static fields, readonly or otherwise.
- So one constructor for the class is declared:
- static ClassName() //static constructor
- {
- this.ScreenWidth = value;
- }
- Note the lack of an access modifier on the static constructor
- The static constructor (if present) executes before any normal constructors.

OOP

- **Adding behavior to your classes: Methods**
- By default, all fields and methods in a class in C# is **private** if no access modifier is found.
- Normally, OOP says: **keep data private and provide public methods for clients of the class to use in working with the data.**
- So in our Date class, we could simply remove the public declarations from month, day, year and they would be private.

OOP

- But now, cannot initialize them directly in Main as we did before. Instead, we could do a couple of things:
- **1.) Add a constructor that accepted initialized values:**

```
public Date(int year, int month, int day)
{
  this.year = year;
  this.month = month;
  this.day = day;
}
```

- In Main: myDate = new Date(1997, 10, 12);
- Since method arguments have the same name as instance variables, must use the "this" keyword to disambiguate.
- "this" is always a reference to the object whose code is currently executing.

OOP

- **2.) Write public accessor methods that let Date class users get and set these values:**

```
public void SetMonth(int month)
{
   this.month = month;
}
public int GetMonth()
{
   return month;
}
```

- Users of date objects then do calls such as:
- myDate.SetMonth(8); and int y = myDate.GetMonth();

OOP

- In both cases (constructors getting values and accessor methods), you can write code to **protect the validity and integrity** of your data so no month, e.g., is less than 1 or greater than 12. Likewise for days and years, if needed.

```
public Date(int year, int month, int day)
{
  if ((year < 1950) || (year > 2010))
     this.year = 1950;   // set a default
  else
     this.year = year;
  if ((day < 1) || (day > 31))
     this.day = 15;  //set a default
  else
     this.day = day;
etc. for month
}
```

OOP

- We can do similar things in our getXXX and setXXX methods.
- This ensures that a Date can never be invalid either initially or later when changed.

OOP

- **Properties**
- C# provides for a more elegant way of dealing with a classes private data through what are Called **properties** (aka smart fields).
- With properties, a client can access a class's fields as though they were public without knowing whether accessor methods exist.

OOP

- So instead of writing public SetMonth() and GetMonth methods, we could write 1 property that does both:
- public int Month // convention is name props with Caps and var with small
- { // properties never take any arguments, nor have () for empty
- get
- {
- return month;
- }
- set
- {
- if (value < 1 || value > 12)
- month = 1; //set a default
- else
- month = value; // ← note keyword value has the argument
- }
- }

OOP

- Property is called Month – with this the client can write
- myDate.Month = 12 to set it
- and int x = myDate.Month to get it

OOP

- Note: in setter method above be careful not to assign the value to the Property name (Month) because this would cause the setter method to be called indefinitely. !!
- Note: Setter does not take any arguments. The value being passed is automatically placed in a variable named ***value*** that is accessible inside the setter method.

OOP

- You can make a property **read only** simply by omitting the set part of a property.
- Properties can have modifiers abstract, virtual, or override, so they can be inherited.
- These apply at the property level…cannot be specified at the getter and setter levels.

OOP

- Now, with properties in our Date class that verify the integrity of year, day and month, our Constructor code can just use them:
- public Date (int year, int month, int day)
- {
- Year = year; //set property to argument value
- // validity checking is done in property method
- Month = month;
- Day = day;
- }

OOP - Summary

- Create a class which is a template or pattern for an entity which encapsulates the:
- variables / attributes / aspects of this kind of thing. Code outside the object generally does not know the names of data members, nor has direct access to these variables.
- methods / functions / behaviors which this kind of thing can do. Code outside object does not know how these methods work. Some methods may be hidden "helper" functions.
- create one or more instances of this class. Each instance is an **object**.

OOP - Summary

- Objects are initialized & created by calling the constructor (with appropriate arguments when necessary).
- We invoke the object's behaviors by calling its methods.
- Each instance of a class (= each object) has its **own data values** (called instance variables since each instance of the class has its own copy); i.e., one Date has 1/4/1990 and another Date has 2/3/2000, etc.)

OOP - Summary

- **BUT**, each instance of a class **shares the methods**; i.e., one copy of method code for each class instance.
- Remember that static variables are SHARED across all object instances.
- Static methods cannot access non-static variables.