Information Hiding in OO Languages

- Classes support information hiding
  - Data members of a class can be declared as private
    - Hides implementation details of the class
  - Create accessor methods to access data
    - Typically get__() and set__(), which are public
    - “Getters and setters”
  - Other classes access data via get() and set() method
    - So long as the interface to get and set stay the same, the class can change how it represents its data
    - Information hiding permits implementations to change without affecting using classes
  - But, is tedious to always access data via accessors
    - x = foo.getX() is more tedious than x = foo.x;
    - Wouldn’t it be great to preserve benefits of accessors, while also retaining the syntax of direct access?
C# Properties

• Provide procedural access to data
  – Like accessors

• But have syntax similar to direct variable access
  – foo.X instead of foo.getX();

• Minor feature, but provides improvement in readability of programming

Travis thumbs his nose at private property
www.flickr.com/photos/sillygwailo/492070136/
C# Property Syntax

```csharp
[access-modifiers] return-type property-name
{
    get
    {
        ... sequence of statements ending in a return (or throw)
    }
    set
    {
        ... sequence of statements
    }
}
```

- Get accessor returns value of same type as “return type”
- Set accessors have implicit parameter named “value”
  - Use to set internal data representation
- Properties can be public, private, protected
  - Public: any class can access, private: only that class, protected: that class and children
- By convention, property names have initial capital (“X” to access “x”)
public class GameInfo
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
}

// Test code
GameInfo g = new GameInfo();

// Call set accessor
g.Name = "Radiant Silvergun";

// Call get accessor
System.Console.Write(g.Name);
Automatic Properties

• Very often, properties are straightforward getters and setters
  – Get accessor just reads value of one variable
  – Set accessor just writes the value of one variable

• Creating properties in this situation is very mechanical

• With **automatic properties**, the compiler creates these straightforward get and set accessors
  – New to C# 3.0
public class GameInfo
{
    public string Name {get; set;}
}

// Test code
GameInfo g = new GameInfo()

// Call set accessor
g.Name = “Radiant Silvergun”;

// Call get accessor
System.Console.Write (g.Name);

This property behaves the same as the first property example, two slides ago.

A private class variable, *private String name*, is automatically created.
Arrays

- Array is an *indexed* collection of *objects*
  - Index means you use `array[i]` syntax to access members
  - Recall that types like int, string, float, etc. are objects
  - Can even have arrays of arrays
- Unlike C++, in C# array is an object
- Arrays have many useful properties and methods
  - Properties:
    - Length: length of array
    - Rank: number of dimensions of array
  - Methods:
    - Sort(): sorts values in one dimensional array
    - BinarySearch(): searches one dimensional array
    - Clear(): sets range of elements to 0 or null reference
    - Reverse(): reverses elements in one dimensional array
    - ... and many others
Declaring an Array

type[] array-name;

Example:

int[] numbers;

numbers = new int[3];

Technically, just creates a variable (numbers) that will hold a reference to an array-of-integers object.

Creates an array-of-integers object, of length 3, which are initialized to 0.

Array numbering follows C conventions

– First element is numbers[0]
– Upper bound is 2, so 3rd element is numbers[2]
Arrays of Reference Types

public class GameInfo
{
    public string gameName;
}

GameInfo[] myGArray = new GameInfo[2];

- Creating a “new” array of a reference type
  - Just creates a series of null references
  - Need to assign object instances to array elements
Arrays of Reference Types (2)

```csharp
public class GameInfo
{
    public string gameName;
}

GameInfo[] myGArray = new GameInfo[2];

GameInfo A = new GameInfo();
GameInfo B = new GameInfo();

myGArray[0] = A;
myGArray[1] = B;
```

There are only two instances of class GameInfo.

There are four reference variables that point to GameInfo.

- A
- B
- myGArray[0]
- myGArray[1]
Initializing Arrays

int[] anIntArray = new int[3] { 2, 4, 6 }

OR

int[] anIntArray = { 2, 4, 6 }

• Both syntaxes have identical behavior
• Can also initialize reference types:

string[] aStringArray = { “The”, “Quick”, “Brown”, “Fox” }

AClass[] AClassArray = { new AClass(), new AClass(), new AClass() }
Two Dimensional Arrays

type[,] array-name

int[,] myTwoDimArray = new int[2, 3];

• Can have two (and more) dimensional arrays
• Also possible to initialize
  – Implicitly sets bounds of the array

// Create a 4 x 3 array
int[,] myTwoDimArray =
{
  {0, 1, 2}, {3, 4, 5}, {6, 7, 8}, {9, 10, 11}
}
string[] aStringArray = { “Cherry”, “Apple”, “Banana”, “Peach” };  

// Sort elements  
Array.Sort( aStringArray );  
// Elements now: Apple, Banana, Cherry, Peach  
Array.Reverse( aStringArray );  
// Elements now: Peach, Cherry, Banana, Apple

• Call Array.Sort(), passing your array to sort a one dimensional array  
• Call Array.Reverse() to reverse elements
Looping in C#

• C# has four looping constructs
  – for
    • for (j = 0; j < 5; j++) { ... }
    • Classic loop syntax in C-like languages
    • Possibility of off-by-one errors in array indexing
  – foreach
    • foreach (int j in intArray)
    • Eliminates array indexing errors, no need to create index variable before statement
  – while
    • while (j < 5) { ...; j++; }
    • Loop until an event occurs
  – do ... while
    • do {...; j++;} while (j < 5)
    • Uncommon, perform action, then do condition check
Foreach Statement

```csharp
string[] aStringArray = { "Cherry", "Apple", "Banana", "Peach" }; // Sort elements
Array.Sort( aStringArray );
foreach (string s in aStringArray)
    System.Console.Write("{0} : ", s);
// Output: "Apple : Banana : Cherry : Peach : "
```
List

• Arrays have problem that you must know how many elements you want in advance
  – This is not always known
• List class is collection with variable size
  – Dynamically increases in size if needed
  – When an array reaches its capacity, need to create new array, and copy all elements from old array to new array
  • Ugh!
Creating a List

List<type> listname

Example:

List<string> stringList = new List<string>();  // Create list of string.  
Don’t forget ()

stringList.Add ( “Quick” );
stringList.Add ( “Brown” );
stringList.Add ( “Fox” );

foreach (string s in myStringList)                     // Lists work with
  System.Console.Write("{0} ", s);

• Add elements with Add() method
• Clear() removes all elements from list
• Remove() removes first element from list
• Sort() sorts the list
• Count property: number of elements in list
Queue, Stack, Dictionary

- C# provides queue, stack, and dictionary
- Queue: first-in, first-out
  - Enqueue(), Dequeue(), Peek()
- Stack: last-in, first-out
  - Push(), Pop(), Peek()
- Dictionary
  - Holds set of key, value pairs
  - Permits lookup of a value given the key
  - Example use: extensible character attribute system
    - Keys: strings, names of attribute
    - Value: int, value of specific attribute
• Chapter 4 (Classes and Objects)
• Chapter 9 (Arrays, Indexers, and Collections)
  from pp. 155-176
  in *Programming C# 3.0*

• If you are behind in the reading, you need to catch up fast.
  – You are already 5-10 hours behind.
  – It is hard to catch up after this point