C# Intermediate Course

Classes

- Should be named with **Pascal Case**

Constructors

```
public class Customer
{
                    ...

    public Customer()
    {
        Orders = new List<Order>();
    }

    public Customer(int id) {
        this.Id = id;
    }

    public Customer(int id, string name)
        : this(id)
    {
        this.Name = name;
    }
}
```

- `: this()` calls the first constructor before the overloaded one
  - Should be used minimally, as adds complexity to control flow

Object Initializers

- Syntax for quickly initializing an object without the need to call one of its constructors

```
var person = new Person {
  FirstName = "Miller",
  LastName = "Anderson"
};
```

Methods

The Params Modifier

```
public class Calculator
{
  public int Add(params int[] numbers){}
}

var result = calculator.Add(new int[]{ 1, 2, 3, 4 });
var result = calculator.Add(1, 2, 3, 4);
```

The Ref Modifier

- Should be avoided when possible due to 'code smell' (Mosh' opinion)

```
public class Weirdo
{
  public void DoAWeirdThing(ref int a)
  {
    a += 2;
  }
}

var a = 1;
weirdo.DoAWeirdThing(ref a); // a = 3;
```

The Out Modifier

- Should be avoided when possible due to 'code smell' (Mosh' opinion)

```
public class MyClass
{
  public void MyMethod(out int result)
  {
    result = 1;
  }
}

int a;
myClass.MyMethod(out a); // a = 1
```

Fields

- Variables stored at class level

Initialization

```
public class Customer
{
  public List<Order> orders = new List<Order>();
}
```

- Can be initialized at class level without need for constructor

Read-only Fields

```
public class Customer
{
  readonly List<Order> orders = new List<Order>();
}
```

Access Modifiers

> A way to control access to a class and / or its members

- Public
  - Accessible from everywhere
- Private
  - Accessible from only the class
- Protected
  - Accessible only from the class and its derived / child classes
- Internal
  - Accessible only from the same assembly (or class library)
- Protected Internal
  - Accessible only from the same assembly or any derived classes
  - Rarely used

Properties

- A class member that encapsulates a getter/setter for accessing a field

```
public class Person
{
  private DateTime _birthdate;

  public DateTime Birthdate
  {
    get { return _birthdate;}
    set { _birthdate = value; }
  }
}
```

Auto-implemented Properties

```
public class Person
{
  public DateTime Birthdate { get; set; }
}
```

- Internally creates a private field

Indexers

```
var array = new int[5];
array[0] = 1;

var list = new List<int>();
list[0] = 1;
```

```
var cookie = new HttpCookie();
cookie.Expire = DateTime.Today.AddDays(5);

cookie["name"] = "Miller";
// Preferable over:
cookie.SetItem("name", "Miller");
```

```
public class HttpCookie
{
  public string this[string key]
```

```
    {
      get { ... }
      set { ... }
    }
  }
```

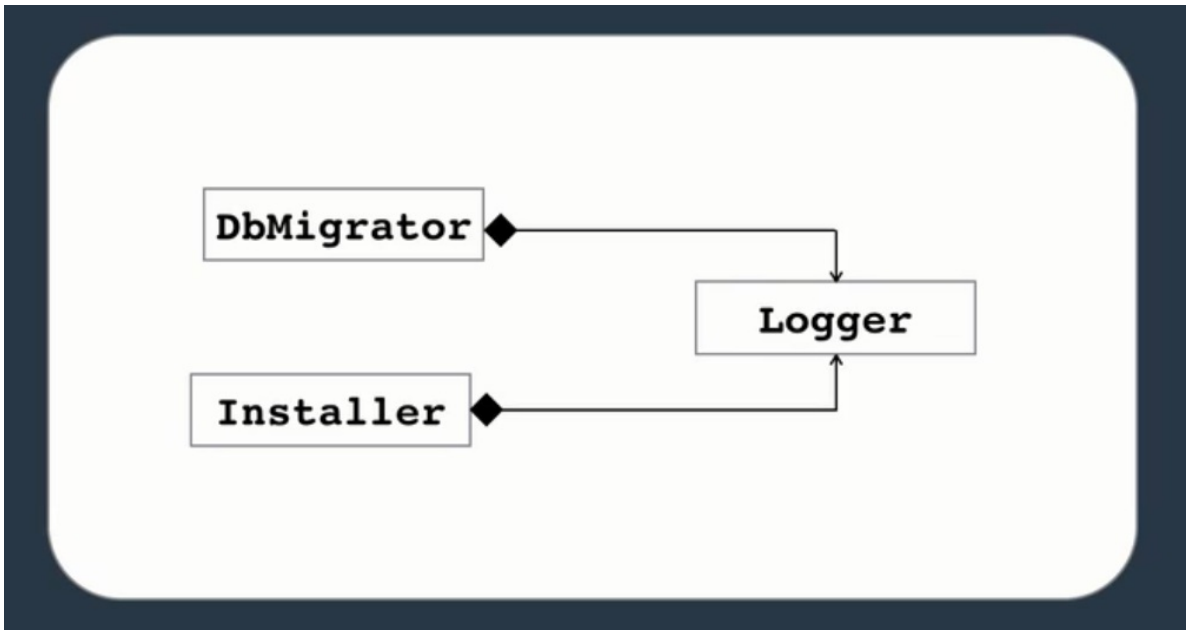
- Use a `Dictionary<>` in instances where data lookup depends on a `key`
- Use `List<>` in instances where data lookup depends on `index`

Class Coupling

- A measure of how interconnected classes and subsystems are

Inheritance

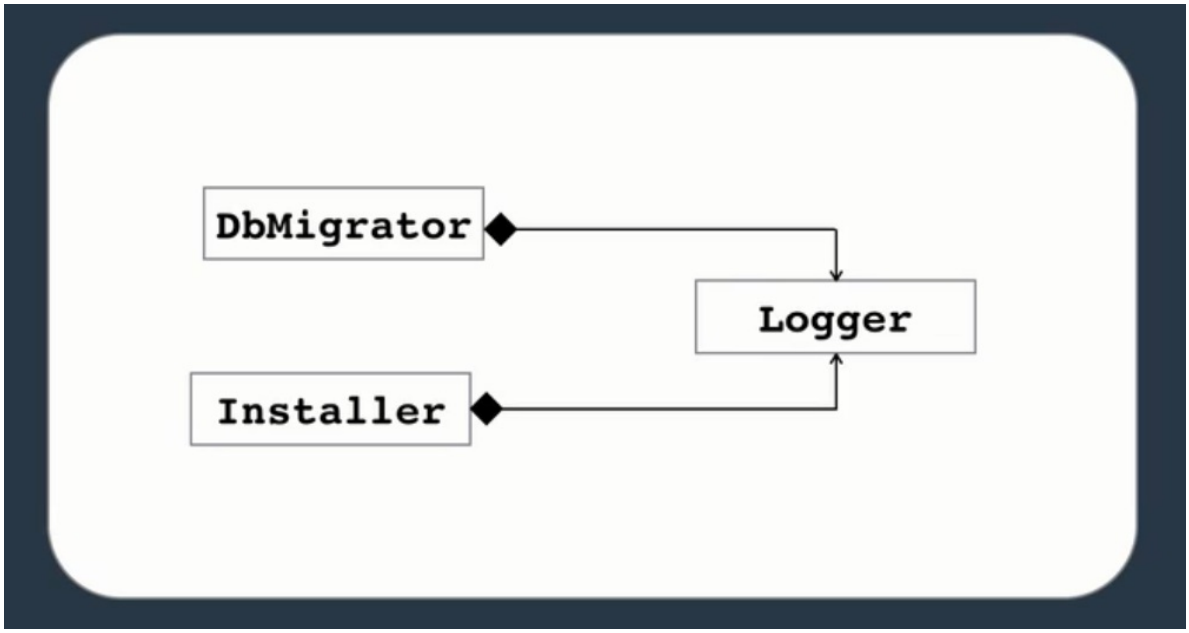
- A kind of relationship between two classes that allows one to inherit code from another
- Have an '*Is-A*' relationship:
  - A Car is a Vehicle
- Code re-use
- Polymorphic behavior



Composition

- A kind of relationship between two classes that allows one to contain the other
- Have a '*Has-A*' *relationship:*
  - A Car has an Engine

- Flexibility
- A means to loose-coupling



Problems with Inheritance

- Easily abused by amateur designers / developers
- Large hierarchies
- Fragility
- Tight coupling
- Any inheritance relationship can be translated to composition

Benefits of Composition

- Flexibility and loose coupling

Constructors and Inheritance

- Base class constructors are always executed first
- Base class constructors are not inherited

The Base keyword

```
public class Car : Vehicle
{
  public Car(String registrationNumber)
    : base(registrationNumber)
```

```
  {
    // Initialize fields specific to the car class
  }
}
```

- Invokes the constructor of the parent class (in this case, `Vehicle`)

Upcasting / Downcasting

- Conversion from a derived class to a base class (**upcasting**)
- Conversion from a base class to a derived class (**downcasting**)

Upcasting

```
Circle circle = new Circle();
Shape shape = circle;
```

- No conversion necessary

Downcasting

```
Circle circle = new Circle();
Shape shape = circle;

Circle anotherCircle = (Circle)shape;
```

The as keyword

- Used to cast an object as another without throwing a `CastExceptionError`

```
Car car = obj as Car;
if (car != null)
{
  ...
}
```

The is keyword

- Used to verify the type of an object

```
if (obj is Car)
{
  ...
}
```

Boxing / Unboxing

- Have a performance penalty

Value Types

- Stored on the stack (shorter lifespan / less memory)
- All primitive types

Reference Types

- Stored in the heap (longer lifespan / more memory)
- Any classes

Boxing

- The process of converting a value type instance to an object reference

```
int number = 10;
object obj = number;

// or
object obj = 10;
```

Unboxing

```
object obj = 10;
int number = (int)obj;
```

Method Overriding

- Modifying the implementation of an inherited method
- Used in polymorphism

```
public class Shape
{
  public virtual void Draw()
  {
    // Default implementation
  }
}

public class Circle : Shape
{
  public override void Draw()
  {
    // New implementation
  }
}
```

Abstract Classes and Members

Abstract Modifier

- Indicates that a class or member is missing implementation

```
public abstract class Shape
{
  public abstract void Draw();
}

public class Circle : Shape
{
  public override void Draw()
  {
    // Implementation for Circle
  }
}
```

Abstract Members

- Classes with `abstract` members must also be declared as `abstract`
- Do not include implementation

- Must implement all abstract members from base abstract class

Why use abstract?

- When you want to provide some common behavior while forcing other developers to follow your design

Sealed Classes and Members

- Opposite of abstract classes
- Prevents derivation of classes or overriding methods
- (rarely used)

Why?

- Slightly faster because of some run-time optimization

Interfaces

- A language construct similar to a class (syntactically) but is fundamentally different
- No access modifiers
- Think of an interface as a **contract** whose methods and properties must be implemented

```
public interface ITaxCalculator
{
  int Calculate();
}

public class TaxCalculator : ITaxCalculator
{
  public int Calculate(order){
    ...
  }
}
```

- Read as *TaxCalculator implements ITaxCalculator*
- *Not* the same as inheritance

Why?

- To build loosely-coupled applications
- Improve extensibility and testability of applications
- Promote loose-coupling between concrete classes that are dependent (inherit) another

Interfaces and Testability

- When unit testing classes, if that class has dependencies on other concrete classes, you can use an interface to reduce that coupling

`METHODNAME_CONDITION_EXPECTATION`

- Pattern for naming test methods

Interfaces and Extensibility

- **THEORY** - Extensibility allows for changing behavior of existing classes without changing said classes -- create new classes that implement desired behavior, that are referred to in concrete class

```
public class ConsoleLogger : ILogger
   {
        public void logError(string message)
        {
            Console.WriteLine(message);
        }

        public void LogInfo(string message)
        {
            Console.WriteLine(message);
        }
 }

public class DbMigrator
   {
        private readonly ILogger _logger;

        public DbMigrator(ILogger logger)
        {
            _logger = logger;
        }

        public void Migrate()
        {
            _logger.LogInfo("Migrating started at " + DateTime.Now);

            // ... details of migrating database
```

```
            _logger.LogInfo("Migrating finished at " + DateTime.Now);
        }
    }
```

Open Close Principle (OCP)

- Software applications should be open for extensibility but closed for modification

DRY Principle

- Don't Repeat Yourself

Interfaces are NOT for Multiple Inheritance

- Classes can *implement* multiple interfaces, not *inherit* them

Interfaces and Polymorphism