What is composition

Composition is another type of relationship between classes that allows one class to contain the other.

Composition is usually referred to as a **Has-A** relationship. For example, a car **has an** engine.

Just like inheritance, composition allows for code reuse. Composition is more flexible and is a means to designing loosely coupled applications.

We may have a logging class that's responsible for only logging to a text file. This class can then be used by any other class in our application to log anything, it's not coupled to a specific class to only log for that class.

Class composition

To create an association between classes, we instantiate a new class object inside the class we want to use it.

Syntax:

```
class Identifier
{
    // class body
}

class Identifier2
{
    Identifier id = new Identifier();

    id.Member();
}
```

Example:

```
using System;

namespace Classes
{
    class Program
    {
        static void Main(string[] args)
        {
            DataMigrator DataMig = new DataMigrator();

            DataMig.Migrate();

            Console.ReadLine();
        }
    }

    public class Logger
    {
        public void Log(string message)
        {
            Console.WriteLine(message);
        }
    }

    public class DataMigrator
    {
        // we need a new instance of
        // a Logger object to create
        // an association with the
        // Logger class
        Logger logger = new Logger();

        public void Migrate()
        {
            logger.Log("Migrating...");
        }
    }
}
```

In the example above, both classes are regular classes, but the **DataMigrator** class uses an instance of the **Logger** class inside of it. It's the same as calling one function from inside another.

With inheritance we would have access to everything in the **Logger** class. However, with composition we only have access to code that we specified. We have access to the **Logger** class, but only indirectly.

Composition over inheritance

If composition only gives us indirect access, why use it? The problem with inheritance is that it can be easily abused, which may lead to a large hierarchy of classes.

A large hierarchy of classes is fragile, if we change a class at the top of the hierarchy, any class that depends on it is affected and may need to be changed as well.

As an example, let’s consider that we have a parent class called **Animal** and some child classes that inherit from it.

Example:

```
class Animal
{
    public void Eat()
    {
        Console.WriteLine("Eating...");
    }

    public void Walk()
    {
        Console.WriteLine("Walking...");
    }
}

class Dog : Animal {}
class Cat : Animal {}
```

In the example above, we have a **Dog** and a **Cat** which can both walk. But, what if we add a fish?

Now the hierarchy needs to change, we need something like a **Mammal** class that inherits from **Animal** and only then we can inherit **Dog** and **Cat** from **Mammal**.

Favor composition

Any inheritance relationship can be translated to composition. For example, instead of **being an** Animal, the child classes now **have an** Animal.

Example:

```
class Animal
{
    public void Eat()
    {
        Console.WriteLine("Eating...");
    }
}

class Dog {}
class Cat {}
class Fish{}
```

In the example above, we forgot to give the animal class a **Walk()** function. With inheritance, we need to change the **Animal** class again, but we would still have the problem of the fish.

With composition, it's as simple as adding two new classes.

Example:

```
class Animal
{
    public void Eat()
    {
        Console.WriteLine("Eating...");
    }
}

class Walkable
{
    public void Walk()
    {
        Console.WriteLine("Walking...");
    }
}

class Swimmable
{
    public void Swim()
    {
        Console.WriteLine("Swimming...");
    }
}

class Dog {}
class Cat {}
class Fish{}
```

In the example above, we added a class for any animal that can walk, and a class for any animal that can swim.

The Cat and Dog classes can implement both **Walkable** and **Swimmable**, and the Fish class can implement **Swimmable**.

If we wanted to add a bird, we could simply add a Flyable class, and the bird could implement **Walkable**, **Swimmable** and **Flyable**.

The application is now loosely coupled, as each class stands on its own and is not dependent on another class.

Composition and inheritance pros and cons

Inheritance

- Pros: Reusable code, easy to understand
- Cons: Tightly coupled, can be abused, fragile

Composition

- Pros: Reusable code, flexibility, loosely coupled
- Cons: Harder to understand

We don’t mean that inheritance is a bad thing, it’s great and we will still need and use inheritance. Composition is just an alternative that we need to consider.

In most cases, we use both inheritance and composition throughout the application. We’ll learn more about composition with interfaces in the **Interfaces** tutorial.

Summary: Points to remember

- Composition is instantiating a class inside another instead of inheriting from it.
- We should consider favoring composition above inheritance as it allows for a loosely coupled application.

- Composition is a **Has-A** relationship, instead of the Is-A relationship from inheritance. For example, a car **has a** steering system.