INHERITANCE

HOW TO CREATE FAMILIES OF RELATED TYPES TO REDUCE CODE DUPLICATION

INHERITANCE

Obtaining the qualities of other classes.

In object-oriented programming, inheritance enables new objects to take on the properties of existing objects. A class that is used as the basis for inheritance is called a *superclass* or *base class*. A class that inherits from a superclass is called a *subclass* or *derived class*. The terms *parent class* and *child class* are also acceptable terms to use respectively. A child inherits visible properties and methods from its parent while adding additional properties and methods of its own.

Subclasses and superclasses can be understood in terms of the **is a** relationship. A subclass **is a** more specific instance of a superclass. For example, an orange **is a** citrus fruit, which **is a** fruit. A shepherd **is a** dog, which **is an** animal. A clarinet **is a** woodwind instrument, which **is a** musical instrument. If the **is a** relationship does not exist between a subclass and superclass, you should not use inheritance. An orange **is a** fruit; so it is okay to write an *Orange* class that is a subclass of a *Fruit* class. As a contrast, a kitchen **has a** sink. It would not make sense to say a kitchen **is a** sink or that a sink **is a** kitchen !.

```
using System;
namespace inherInheritanceExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Dog fido = new Dog();
            fido.MakeNoise();
            fido.Sleep();
            Cat meow = new Cat();
            meow.Sleep();
            meow.MakeNoise();
        }
    }
    public class Animal
    {
        public void Sleep()
        {
            Console.WriteLine("Zzzzzz");
        }

        public virtual void MakeNoise()
        {
            Console.WriteLine("DurrrDurrr");
        }
    }
    class Dog : Animal
    {
        public override void MakeNoise()
        {
            Console.WriteLine("WoofWoof");
        }
    }
    class Cat : Animal
    {
        public override void MakeNoise()
        {
            Console.WriteLine("Meaaooooow");
        }
    }
}
```

In this example, we created a base class, Animal.
From the base class, we inherited two child classes, Dog and Cat.
We created a sleep function in the base class.

Since the child class inherits from base class, Dog and Cat need not declare the sleep function again.

However Dog and Cat make different noises, so they override MakeNoise function.

When we compile and see the output of the associated program we see that the output of each animals sleep method inherited from the base class has the same output.

```
PS C:\code\inher> dotnet run
WoofWoof
Zzzzzz
Zzzzzz
Meaaooooow
```

POLYMORPHISM

POLYMORPHISM

Present the same methods in varying ways.

The three types of polymorphism are overloading, parametric, and inclusion. Polymorphism helps to promote flexibility in designs by allowing the same method to have different implementations. In essence, you can leverage polymorphism to separate interface from implementation. It promotes code reuse and separation of concerns in your application.

Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.
Base classes may define and implement virtual *methods*, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

A running example of Parametric Polymorphism

```
using System;
namespace TestPolymorphism
{
    class OverloadingDemo
    {
        public int addNums(int i, int j)
        {
            return i + j;
        }

        public int addNums(int i, int j, int k)
        {
            return i + j + k;
        }

        static void Main(string[] args)
        {
            OverloadingDemo od1 = new OverloadingDemo();
            Console.WriteLine(od1.addNums(2, 3));
            Console.WriteLine(od1.addNums(2, 3, 4));
        }
    }
}
```

```
PS C:\code\inher> dotnet run
5
9
PS C:\code\inher>
```

Method Overloading

is the common way of implementing **polymorphism**. It is the ability to redefine a function in more than one form. A user can implement function **overloading** by defining two or more functions in a class sharing the same name. **C#** can distinguish the methods with different method signatures.

Parametric polymorphism

This refers to the form of polymorphism where you have more than one method in your class that has the same name but they differ in their method signatures.

Inclusion polymorphism

Inclusion polymorphism, or method overriding, can be achieved in C# using virtual methods. In method overriding, you have methods having identical signatures present in both the base and the derived classes. You would typically want to use virtual methods to implement run-time polymorphism or late binding.

DELEGATES AND LAMBDAS

DELEGATES

Adding alternate methods to objects

Delegates allow new methods to be injected into an object. This is achieved by packaging up the new method as a variable. The variable acts as the reference to a method and is similar to a the pointer concept in C / C++.

The delegate declaration determines the methods that con be referenced. A delegate can refer to a method which has the same signature as that of the delegate.

The syntax for declaring a delegate is :

delegate <return type> <delegate-name> <parameter list>

Once a delegate type is declared, a delegate object must be created with the **new** keyword and be associated with a particular method. When creating a delegate, the argument passed to the **new** expression is written similar to a method call, but without the arguments to the method.

LAMBDA EXPRESSONS

Blocks of code

Lambda expressions differ from Delegates in that it holds a method without a permanent name where as a Delegate is the name of a variable that hold a reference to a method.

Lambda expressions are code that can be represented either as a delegate, or as an expression tree that compiles to a delegate. The specific delegate type of a lambda expression depends on its parameters and return value.

A lambda expression uses =>, the lambda declaration operator, to separate the lambda's parameter list from its executable code. To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator, and you put the expression or statement block on the other side.

Named Method Delegate example.

```
using System;
public class Program
{
    private delegate void TestDel(string s);

    private static void Test(string s)
    {
        Console.WriteLine(s);
    }

    public static void Main()
    {
      TestDel td = new TestDel(Test);
        td("Hello World 1");

    }
}
```

```
User@DESKTOP-1Q2O7MI MINGW64 /c/code/rnd
$ dotnet run
Hello World 1
```

A statement lambda expression example

```
using System;
public class Program
{
    private delegate void TestDel(string s);

    public static void Main()
    {
    string message2 = "Hello World 8";
    TestDel td6 = s =>
        {
            Console.WriteLine(s);
            Console.WriteLine("Hello World 7");
            Console.WriteLine(message2);
        };
        td6("Hello World 6");
    }
}
```

```
User@DESKTOP-1Q2O7MI MINGW64 /c/code/rnd
$ dotnet run
Hello World 6
Hello World 7
Hello World 8
```

Gregory McLennan