Polymorphism

Polymorphism is the third pillar of object-oriented programming, after encapsulation and inheritance. 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.

Polymorphism provides a way for a subclass to define its own version of a method defined by its base class, using the process termed ***method overriding.*** If a base class wants to define a method that may be (but does not have to be) overridden by a subclass, it must mark the method with the *virtual* keyword. When a subclass wants to change the implementation details of a virtual method, it does so using the override keyword.

Virtual methods enable you to work with groups of related objects in a uniform way. For example, suppose you have a drawing application that enables a user to create various kinds of shapes on a drawing surface. You do not know at compile time which specific types of shapes the user will create. However, the application has to keep track of all the various types of shapes that are created, and it has to update them in response to user mouse actions. You can use polymorphism to solve this problem.

```
public class Shape
{
        public int X { get; private set; }
        public int Y { get; private set; }
        public int Height { get; set; }
        public int Width { get; set; }
        public virtual void Draw()
        {
                Console.WriteLine("Performing base class drawing tasks");
        }
}
class Rectangle : Shape
{
        public override void Draw()
        {
                // Code to draw a rectangle...
                Console.WriteLine("Drawing a rectangle");
                base.Draw();
        }
}
```

```
class Triangle : Shape
{
        public override void Draw()
        {
                // Code to draw a triangle...
                Console.WriteLine("Drawing a triangle");
                base.Draw();
        }
}
```

When a method is marked with the virtual keyword, the method provides a default implementation that all derived types automatically inherit. If a child class so chooses, it may override the method but does not have to.

Sometimes you might not want to seal an entire class but simply want to prevent derived types from overriding particular virtual methods. In this case, you can effectively **seal** that method.

```
public class Employee
{
        …
        public virtual void GiveBonus(float amount)
        {
                …
        }
}
class SalesPerson : Employee
{
        …
        public override sealed void GiveBonus(float amount)
        {
                …
        }
}
```

If you try to override the GiveBonus() method in the other class derived from SalesPerson class, you will receive compile-time error.

Abstract keyword

The **abstract** modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes, not instantiated on its own. Members marked as abstract must be implemented by classes that derive from the abstract class.

Abstract classes *have the following features:*

- An abstract class cannot be instantiated.
- An abstract class may contain abstract methods and accessors.
- It is not possible to modify an abstract class with the sealed modifier because the two modifiers have opposite meanings. The sealed modifier prevents a class from being inherited and the abstract modifier requires a class to be inherited.
- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

```
abstract class Shape
{
        public abstract int GetArea();
}
class Square : Shape
{
        public int Side { get; set; }

        public Square(int side)
        {
                Side = side;
        }
        public override int GetArea()
        {
                return Side * Side;
        }
}
static void Main(string[] args)
{
        Square squareInstance = new Square(12);
        Console.WriteLine($"Area of the square = {squareInstance.GetArea()}");
}
```

When a class has been defined as an abstract base class (by the abstract keyword), it may define any number of abstract members. Abstract members can be used whenever we want to define a member that does not supply a default implementation but must be accounted for by each derived class. By doing so, we enforce a polymorphic interface on each descendant, leaving them to contend with the task of providing the details behind our abstract methods. An abstract base class's polymorphic interface simply refers to its set of virtual and abstract methods. This is much more interesting than first meets the eye because this trait of OOP allows us to build easily extendable and flexible software applications.

Although we cannot directly create an instance of an abstract class, it is still assembled in memory when derived classes are created. Base classes (abstract or not) are useful, they contain all the common data and functionality of derived types. Thus, it is perfectly fine (and common) for abstract classes to define any number of constructors that are called indirectly when derived classes are allocated.

Use the abstract modifier in a method or property declaration to indicate that the method or property does not contain implementation.

Abstract methods *have the following features:*

- An abstract method is implicitly a virtual method.
- Abstract method declarations are only permitted in abstract classes.
- Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({ }) following the signature. For example:

  ```
  public abstract void MyMethod();
  ```

  The implementation is provided by a method *override*, which is a member of a non-abstract class.
- It is an error to use the *static* or *virtual* modifiers in an abstract method declaration.

Abstract properties *behave like abstract methods, except for the differences in declaration and invocation syntax.*

- It is an error to use the abstract modifier on a static property.
- An abstract inherited property can be overridden in a derived class by including a property declaration that uses the *override* modifier.

An abstract class must provide implementation for all ***interface*** *members.* An abstract class, that implements an interface, might map the interface methods onto abstract methods. For example:

```
interface I
{
        void M();
}
abstract class C : I
{
        public abstract void M();
}
```

Member Shadowing

C# provides a facility that is the logical opposite of method overriding, termed shadowing. *If a derived class defines a member that is identical to a member defined in a base class, the derived class has shadowed the parent's version.* To address this issue, we have a few options. We can update the base class' method using the keyword virtual, and the derived class' method using the keyword override. As an alternative, we can include the new keyword to the current method in derived class. Doing so explicitly states that the derived type's implementation is intentionally designed to effectively ignore the parent's version. We can also apply the new keyword to any member type inherited from a base class (field, constant, static member, or property).

Upcast / Downcast

By Upcast we encapsulate fields and methods of derived class. Upcast is also a type of classic Polymorphism. If there are either the first type of classic polymorphism (virtual/override keywords) and the second type (Upcast/Downcast), so the first type is dominant.

Upcast is implicit so that it is safe. What do we mean by safe? Well, we can happily cast Derived class to Base class and expect all the properties and methods of Base class to be available. When we do upcast we are only able to use methods found in Base class, that is, Derived class has inherited all the properties and methods of Base class.

```
DerivedClass instanceDer = new DerivedClass();

BaseClass instanceBase = (BaseClass)instanceDer;
        // or
BaseClass instanceBase = instanceDer;
```

The opposite side of the coin to upcast is **downcast**. We have to be very careful when doing downcast, beacause in order to do downcast, previously we should have done upcast. Downcast is potentially unsafe, because you could attempt to use a method that the derived class does not actually implement. With this in mind, downcast is always explicit, that is, we are always specifying the type we are downcasting to.

```
DerivedClass instanceDown = (DerivedClass)instanceBase;
```

As keyword

Be aware that explicit casting is evaluated at runtime, not compile time. It is worth pointing out, for the time being, that when you are performing an explicit cast, you can trap the possibility of an invalid cast using the try and catch keywords. ***C# provides the as keyword to quickly determine at runtime whether a given type is compatible with another***. The as operator is being used to perform certain types of conversions between compatible reference types or nullable types.

```
class BaseClass
{
        public override string ToString()
        {
                return "BaseClass";
        }
}
class DerivedClass : BaseClass
{
        public override string ToString()
        {
                return "DerivedClass";
        }
}
class AnotherClass
{
        public override string ToString()
        {
                return "AnotherClass";
        }
}
static void Main(string[] args)
{
        object obj1 = new BaseClass();
        object obj2 = new DerivedClass();
        object obj3 = new AnotherClass();

        Console.WriteLine(obj1 as BaseClass);      //  BaseClass
        Console.WriteLine(obj2 as BaseClass);      // DerivedClass
        Console.WriteLine(obj3 as BaseClass);      //  null
}
```

The as operator is like a cast operation. However, **if the conversion isn't possible, as returns null instead of raising an exception**. Note that the **as operator performs only reference conversions, nullable conversions, and boxing conversions**. The as operator can't perform other conversions, such as user-defined conversions, which should instead be performed by using cast expressions.

```
expression as type
```

The code is equivalent to the following expression except that the expression variable is evaluated only one time.

```
expression is type ? (type)expression : (type)null
```

Is keyword

In addition to the as keyword, the C# language provides the is keyword to determine whether two items are compatible. Unlike the as keyword, however, the is keyword returns false, rather than a null reference, if the types are incompatible.

The is keyword evaluates type compatibility at runtime. It determines whether an object instance or the result of an expression can be converted to a specified type.

expr is type

where *expr* is an expression that evaluates to an instance of some type, and *type* is the name of the type to which the result of *expr* is to be converted. **The is statement is true if *expr* is non-null and the object that results from evaluating the expression can be converted to *type*; otherwise, it returns false.**

```
if (obj is Person)
{
        // Do something if obj is a Person.
}
```

The **is** statement is **true** if:

- *expr* is an instance of the same type as *type*.
- *expr* is an instance of a type that derives from *type*. In other words, the result of *expr* can be upcast to an instance of *type*.
- *expr* has a compile-time type that is a base class of *type*, and *expr* has a runtime type that is *type* or is derived from *type*. The *compile-time type* of a variable is the variable's type as defined in its declaration. The *runtime type* of a variable is the type of the instance that is assigned to that variable.
- *expr* is an instance of a type that implements the *type* interface.

The is keyword generates a compile-time warning if the expression is known to always be either true or false.

expr cannot be an anonymous method or lambda expression. It can be any other expression that returns a value.

Starting with C# 7.0, the is and switch statements support pattern matching.

```
BaseClass obj = new BaseClass();
switch (obj)
{
        case DerivedClass d:
                // do smth
                break;
        case BaseClass b:
                // do smth
                break;
        case null:
                // do smth
                break;
        default:
        break;
}
```

The is keyword supports the following patterns:

- Type pattern, which tests whether an expression can be converted to a specified type and, if it can be, casts it to a variable of that type.
- Constant pattern, which tests whether an expression evaluates to a specified constant value.
- var pattern, a match that always succeeds and binds the value of an expression to a new local variable.