VIA University College

DNP-A18

C# Programming

Inheritance and types in C#

Inheritance

How to expand beyond the first class in C#

Types

Types are everywhere. Learn how to use and convert them

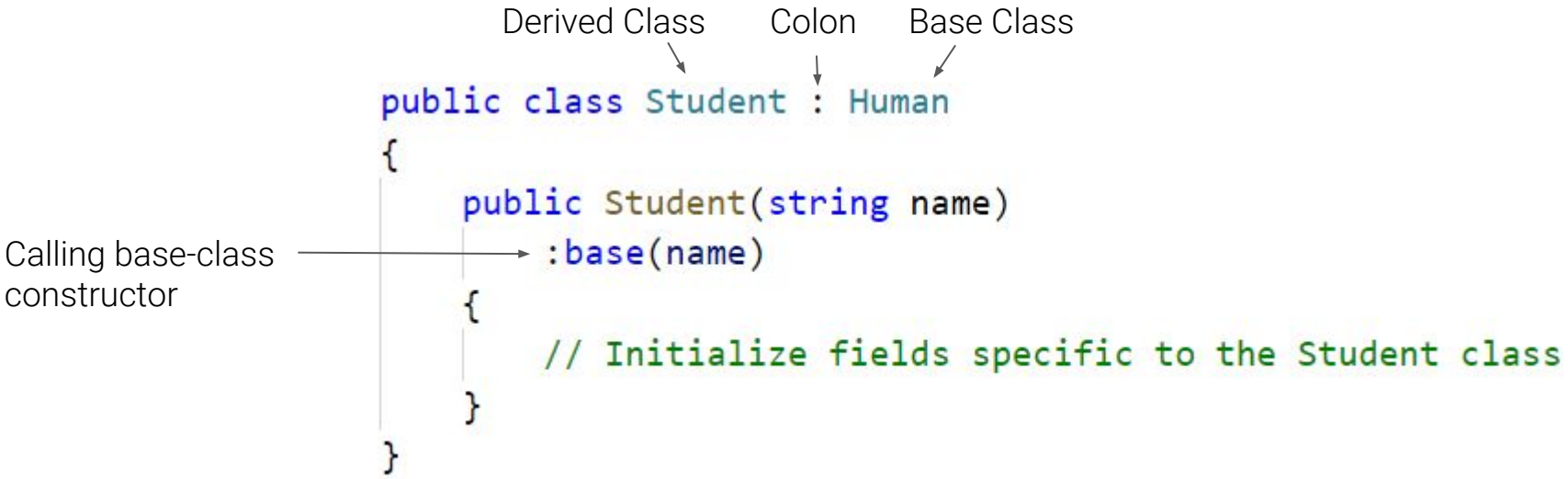
Working with C#

Learn about concepts that make writing clean code easier

Exercises

Get familiar with inheritance and types in C#

Inheritance



Forgot about inheritance?

Method Overriding

Modifying the implementation of an inherited method

```
public class Student
{
    public virtual void Learn()
    {
        // Default implementation
    }
}

public class DNPStudent : Student
{
    public override void Learn()
    {
        // New implementation
        base.Learn(); // optionally call parent method!
    }
}
```

Hiding Inherited Members

The new keyword

Hides a member that is inherited from the base class.

```
public class BaseClass
{
    public int x;
    public void Invoke()
    {
        // base class implementation
    }
}
public class DerivedClass : BaseClass
{
    new public void Invoke()
    {
        // derived class implementation
    }
}
```

```
public class BaseClass
{
    public static int x = 55;
    public static int y = 22;
}
public class DerivedClass : BaseClass
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseClass.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
```

Abstract Classes and Members

Indicates missing or incomplete implementation
Abstract classes cannot be instantiated

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

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

If a member is declared as abstract, the containing class needs to be declared as abstract too

Abstract members do not include implementation

Derived (concrete) classes must implement all abstract members in the base abstract class

Abstract Classes and Members

What's the problem?

```
class Pokemon
{
    public abstract void Fight();
}
```

Abstract Classes and Members

What's the problem?

```
abstract class Student
{
    public abstract void Learn(){}
}
```

Sealed Classes and Members

Sealed modifier prevents derivation of classes or overriding of methods

```
public sealed class Circle : Shape
{
    public override void Draw()
    {
        System.Console.WriteLine("Drawing circle...");
    }
}
```

```
public class Circle : Shape
{
    public sealed override void Draw()
    {
        System.Console.WriteLine("Drawing circle...");
    }
}
```

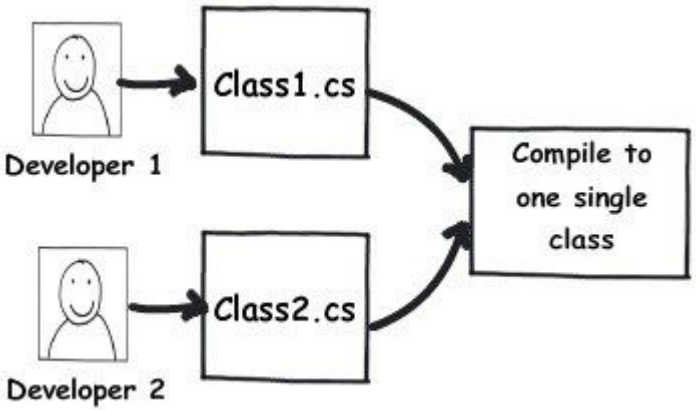
Sealed modifier can only be applied to methods that are overriding a virtual method on the base class

Sealed classes are slightly faster but hardly ever used

Partial Classes

It is possible to split the definition of a class or a struct, an interface or a method over two or more source files.

Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.




```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

[Partial Classes and Methods]

Interfaces


```
public interface IFruit
{
    void Grow();
}
```

Naming Convention
All interfaces in .NET start with an I

No method bodies

No access-modifier on methods (they are all public)

Implementing an interface

```
public class Apple : IFruit
{
    public void Grow()
    {
        // Bring to teacher when fully grown
    }
}
```

Interfaces

A class can implement multiple interfaces

```
public class Apple : IFruit, IEdible, IPluckable
{
    public void Grow()
    {
        //...
    }
    public void Eat()
    {
        //...
    }
    public void Pluck()
    {
        //...
    }
}
```

An interface can extend multiple interfaces

```
public interface ILego : IShape, ISellable
{
    void Build();
}
```

A class **CAN NOT** extend multiple classes

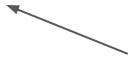
Interfaces

What's the problem?

```
interface IPhone
{
    void Call();
}
abstract class SmartPhone : IPhone
{
}
```

Implicit Type Conversion


```
byte b = 1; // 00000001
int i = b; //  00000000 00000000 00000000 00000001
```



What happens at runtime

```
int i = 1;
float f = i;
```

No data loss

Explicit Type Conversion

4 bytes

```
int i = 1;
byte b = i;
```

Why won't the second line compile?
Potential data loss!

In above case there is no data loss.
What if i = 200?
What if i = 300?

```
float f = 1.0f;
int i = (int)f;
```

Casting avoids compilation errors - you tell the compiler that you accept potential data loss

Non-compatible Types


```
string s = "42";
int i = (int)s;
```

Why won't this compile?

Use the Convert class or the Parse method

```
string s = "42";
int i = Convert.ToInt32(s);
int j = int.Parse(s);
```

Why "32"?

- ToByte()
- ToInt16()
- ToInt32()
- ToInt64()

Overflowing

The checked keyword is used to explicitly enable overflow checking for integral-type arithmetic operations and conversions

```
byte b = 255 + 2;
```

Compiler error

```
byte b = 255;
b = (byte) (b + 2);
```

No error, but what is the value of b?

```
// Checked expression
byte b = 255;
b = checked((byte) (b + 2));
```

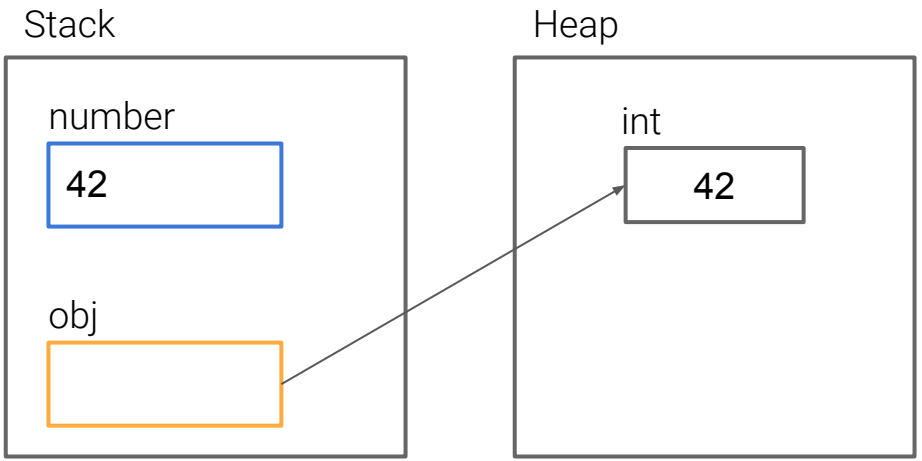
```
// Checked block
byte b = 255;
checked
{
    b = (byte) (b + 2);
}
```

Boxing

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

```
int number = 42;
object obj = number;
```

```
object obj = 42;
```



Unboxing

```
object obj = 42;
int number = (int)obj;
```

Upcasting & Downcasting


```
Circle circle = new Circle();
Shape upcast = circle;
Circle downcast = (Circle)upcast;
```

Upcasting (casting to a supertype) - OK, done implicitly!

Downcasting (casting to a subtype)

```
Pokemon pokemon = (Pokemon)shape; //throws InvalidCastException
```

Assumes Pokemon is derived from Shape

The **as** keyword prevents this

```
Pokemon pokemon = obj as Pokemon;
if (pokemon != null)
{
    //...
}
```

Type Compatibility

Evaluating type compatibility at runtime

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

The *is* keyword determines whether an object instance or the result of an expression can be converted to a specified type

expr is type

The *is* statement is also true if *expr* can be upcast to an instance of *type*

C# Reference

Conversion Operators

C# enables programmers to declare conversions on classes or structs so that classes or structs can be converted to and/or from other classes or structs, or basic types

```
class SampleClass
{
    public static explicit operator SampleClass(int i)
    {
        SampleClass temp = new SampleClass();
        // code to convert from int to SampleClass...

        return temp;
    }
}
```

must be declared static

Conversions are defined like operators and are named for the type to which they convert

implicit
Conversion occurs automatically when required

explicit
Conversion requires a cast

Either the type of the argument to be converted, or the type of the result of the conversion, but not both, must be the containing type

Conversion Operators

Nullable Types

Value types cannot be null
(e.g. a boolean is either true or false, never null)

DateTime is a value type

Members
GetValueOrDefault()
HasValue()
Value

When to use?
When dealing with databases!

```
Nullable<DateTime> date = null;
```

Shorthand

```
DateTime? date = null;
```

```
DateTime date2 = date ?? DateTime.Today;
```

Null-coalescing operator
Returns the left-hand operand if the operand is not null; otherwise it returns the right hand operand.

Nullable Types

Null-Conditional Operator


```
string title;

if (post != null)
    title = post.Title;
```

This can be simplified...

```
var title = post?.Title;
```

Another example, using an indexer

```
Post first = null;
if  (posts != null)
    first = posts[0];
```

Simplifies to

```
var first = posts?[0];
```

Null-Conditional Operator

Accessing members in a chain

```
var body = post?.Comments[0].Body.Substring(0,100);
```

Multiple times in an expression

```
var count = 0;

if (post != null)
{
    if (post.Tags != null)
    {
        count = post.Tags.Count;
    }
}
```

Simplifies to

```
var count = post?.Tags?.Count;
```

nullable int unless we use null-coalescing operator

[Null-conditional Operators]

Properties

A class member that encapsulates a getter/setter for accessing a field.
To create a getter/setter with less code!

```
public class Person
{
    private DateTime _birthdate;

    public void SetBirthdate(DateTime birthdate)
    {
        _birthdate = birthdate;
    }

    public DateTime GetBirthdate()
    {
        return _birthdate;
    }
}
```

same as

```
public class Person
{
    private DateTime _birthdate;

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

~same as

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

Auto-implemented property

Initializing Properties


```
public class Post
{
    public DateTime DateCreated { get; }
    public Collection<Comment> Comments { get; }

    public Post()
    {
        DateCreated = DateTime.Now;
        Comments = new Collection<Comment>();
    }
}
```

You can also directly initialize a property without creating a constructor:

```
public class Post
{
    public DateTime DateCreated { get; } = DateTime.Now;
    public Collection<Comment> Comments { get; } = new Collection<Comment>();
}
```

Indexers

Indexers allow instances of a class or struct to be indexed *just like arrays*.

The indexed value can be set or retrieved without explicitly specifying a type or instance member.

Indexers resemble properties except that their accessors take parameters.

Key	Value
“Gabrielle”	“Gaby”
“Elvis”	“El”
“Jakob”	“Jake”

Example: NickNames class

```
class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}
```

Indexer

Usage

```
var stringCollection = new SampleCollection<string>();
stringCollection[0] = "Hello, World";
Console.WriteLine(stringCollection[0]); // outputs "Hello, World"
```

Working with Time

DateTime
Represents an instant in time, typically expressed as a date and time of day.

DateTime is immutable.

```
var dateTime = new DateTime(2018, 24, 12);
var now = DateTime.Now;
var today = DateTime.Today;
var tomorrow = now.AddDays(1);
var yesterday = now.AddDays(-1);

System.Console.WriteLine("Hour: " + now.Hour);
System.Console.WriteLine("Minute: " + now.Minute);
System.Console.WriteLine(now.ToString("yyyy-MM-dd HH:mm"));
```

```
var dateTime = new DateTime(2016, 3, 14);
dateTime.AddYears(1);
Console.WriteLine(dateTime.Year);
```

What is the value?

Time Format Strings

DateTime Structure

Working with Time

TimeSpan
Represents a time interval.

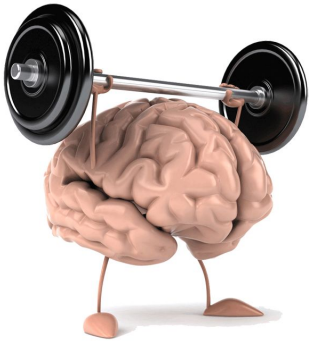
```
// Creating TimeSpans
var timeSpan = new TimeSpan(2,30,55); //hours, minutes, seconds

var timeSpan1 = new TimeSpan(1,0,0);
var timeSpan2 = TimeSpan.FromDays(1);

var start = DateTime.Now;
var end = DateTime.Now.AddMinutes(42);
var duration = end - start;

// Accessing properties
System.Console.WriteLine("Minutes: " + timeSpan.Minutes);
System.Console.WriteLine("Total Minutes: " + timeSpan.TotalMinutes);
```

TimeSpan Structure

Debugging Applications

How to debug

Exercises



List of code snippets using OmniSharp