# C#
# FUNDAMENTALS

SESSIE 2/4

# PLAN DAG 2

- Object georiënteerd programmeren
- Methoden
- Scope
- Encapsulation
- Toegangsbeperking
- Static
- Inheritance
- Polymorphisme
- Klassen
- Abstracte klasses
- Interfaces
- Partial

# CLASSES

- Blueprint from which objects are created
- Contain class members:
  - Field
  - Properties
  - Methods
  - Constructors
  - Finalizers

```
class Example
{
        public int X;
        public int GetFive()
        {
            return 5;
        }
}
```

# PROPERTIES

- Hybrid thing between a field and method
- Property is used like a field
- And implemented like a method with a set and a get accessor
- Get is for reading
- Set is for writing

```
public class Date
{
    private int _month;
    public int Month
    {
        get => _month;
        set
        {
            if ((value > 0) && (value < 13))
            {
                _month = value;
            }
        }
    }
}
```

# METHODS

Syntax of a method:

```
modifiers returnType NameOfMethod (Parameter List) {
        // method body
}
```

```
double currentAmountOfFuel = 0.0;

void FillFuelTank(double amountOfFuel) {

}
```

# METHODS

- **modifier returnType NameOfMethod (Parameter List) {**
- **// method body**
- **}**

- **modifier** − It defines the access type of the method and it is optional to use.
- **returnType** − Method may return a value. Void does not return value. Others MUST return values.
- **NameOfMethod** − This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List** − The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body** − The method body defines what the method does with the statements.

# METHODS

Simple method within a Car class.

```
class Car {
        bool hasStarted;

        void StartCar() {
                hasStarted = true;
        }

        void StopCar() {
                hasStarted = false;
        }
}
```

# METHODS - RETURN

Simple method within a Car class with return.

```
class Car {
        bool hasStarted;

        void StartCar() {
                hasStarted = true;
        }

        void StopCar() {
                hasStarted = false;
        }

        bool IsCarStarted() {
                return hasStarted;
        }
}
```

# METHODS - PARAMETERS

Simple method within a Car class with parameters.

```
class Car {
        bool hasStarted;
        double kilometersDriven;
        double currentAmountOfFuel ;

        void StartCar() { hasStarted = true; }

        void StopCar() { hasStarted = false; }

        bool IsCarStarted() { return hasStarted; }

        void DriveCar (double kilometersToBeAdded, double fuelSpent) {
                kilometersDriven = kilometersDriven + kilometersToBeAdded;
                currentAmountOfFuel = currentAmountOfFuel – fuelSpent;
        }

}
```

# EXERCISE

- Create a class car, give it some properties that make sense
- Add a method start, that return true when the engine has started successfully and false when it didn't.
- And: make it start successfully 7/10 times and make it fail to start 3/10 times.
- And also: add a method to increase speed, that increases the speed by the given input, but won't go over the maxspeed of the car.
- Make an instance of the Car, and start the car.

# METHOD OVERLOADING

- More than one version of a method
- Same name, but different parameters
- We've actually seen quite some of these already! E.g. WriteLine

# EXERCISE

- Add another method to your car class, make it overload the class method by having an extra parameter, namely chance the car will start

# CONSTRUCTOR

- Executed when an object is created
- More than one constructor is common
- Mainly used for initializing fields
- There is a default constructor if you don't create one, it takes 0 parameters

```
public class Person
{
   private string last;
   private string first;

   public Person(string lastName, string firstName)
   {
      last = lastName;
      first = firstName;
   }
}
```

# CONSTRUCTOR - INDEPTH

- Instance constructors
  - Used to create and initialize member variables of a class

- Expression body definition

```
public ClassName(string name) => Name = name;
```

- Static constructor
  - Initializes static members
  - Parameterless
  - If not provided, static members get default values
  - Only invoked the first time it's called

```
static Adult()
  {
     minimumAge = 18;
  }
```

- Private constructor
  - If a class only has private constructors, instances cannot be made from the outside
  - Often used for classes with only static members
  - Constructors without access member are private by default, but better to make it explicitly private

# EXERCISE

- Add a constructor to your car that takes parameter for the color and the brand

- Create the car using your constructor

# VARIABLE SCOPE

Scope depends on the level the variable is declared:

- Class level: called field member, scope is the entire class
- In a method: called local variable, scope is the entire method
- In a block: scope is the block

# ACCESS MODIFIERS

- Public
- Private
- Protected
- Internal
- Protected internal

| visibility / keyword | Containing Classes | Derived Classes | Containing Assembly | Anywhere outside the containing assembly |
|---|---|---|---|---|
| public | yes | yes | yes | yes |
| protected internal | yes | yes | yes | no |
| protected | yes | yes | no | no |
| private | yes | no | no | no |
| internal | yes | no | yes | no |

# OBJECT ORIENTED PROGRAMMING (OOP)

## ENCAPSULATION

*Everyone knows how to access and use classes. How this works is not relevant.*

## INHERITANCE

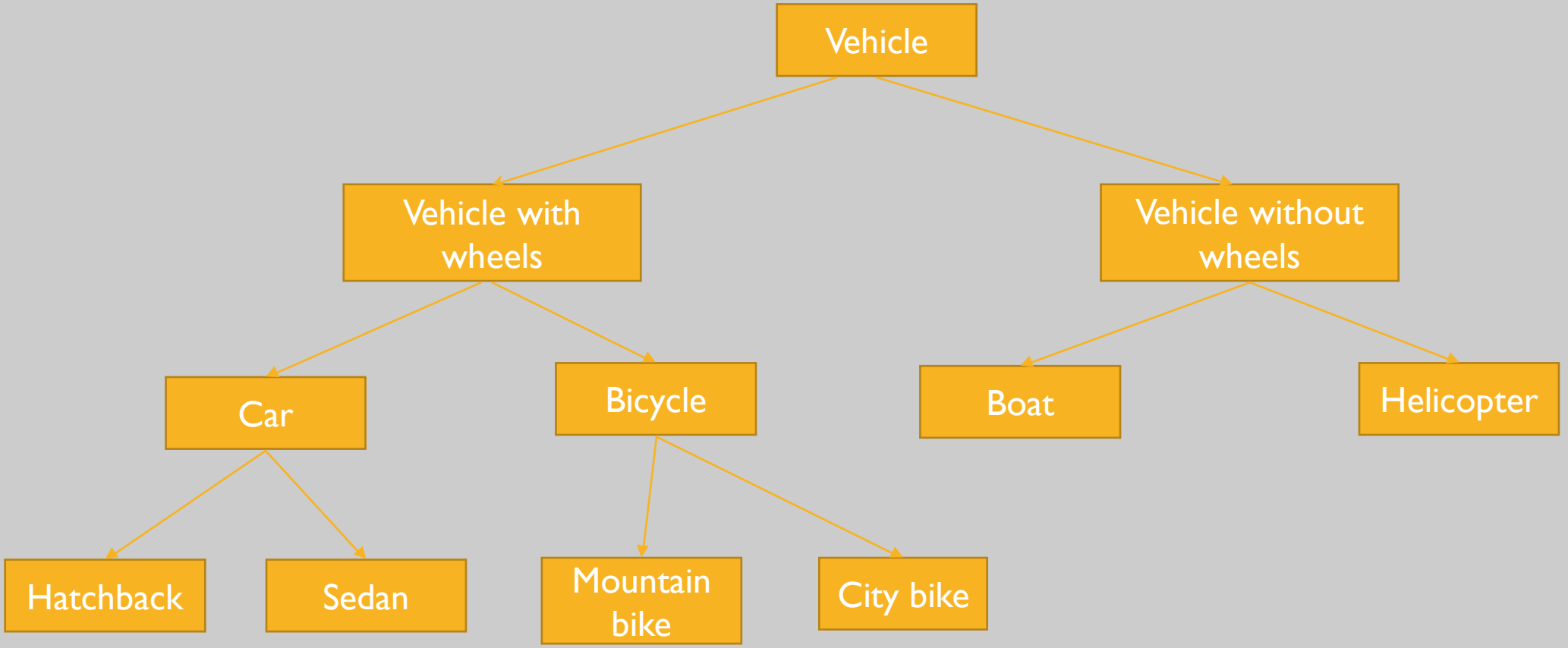
*Classes can have subclasses that inherit properties and functionalities.*

## POLYMORPHISM

*A class can behave in different ways depending on how it's accessed and implemented.*

Object Oriented Programming means that the objects that have a role in the application are organized in classes. Classes can be considered blueprints for all the objects in an application.

# OOP EXAMPLE


Vehicle
Vehicle with wheels
Vehicle without wheels
Car
Bicycle
Boat
Helicopter
Hatchback
Sedan
Mountain bike
City bike

# EXERCISE OOP

- Create an OOP scheme for shapes
- What is the main parent class?
- Make sure to include at least the following shapes: rectangle, square, trapezoid, equilateral triangle, right-angled triangle and a circle
- What methods should the shapes have?
- And what properties?
- At what level are the methods and properties defined?

# INHERITANCE

- Is-a relationship between classes
- Derived classes can only have one direct base class
- All class members from base class are inherited by derived class, except for (don't worry about these now):
  - Private members
  - Constructors
  - Finalizer (destructor)

# CONSTRUCTORS AND INHERITANCE

- Constructors invoke contructor of the base class implictly, if not done explicitly

```
class Circle : Shape
{
    public Circle(double radius) : base(radius, 0) // calling the constructor of the base class
    {
    }
}
```

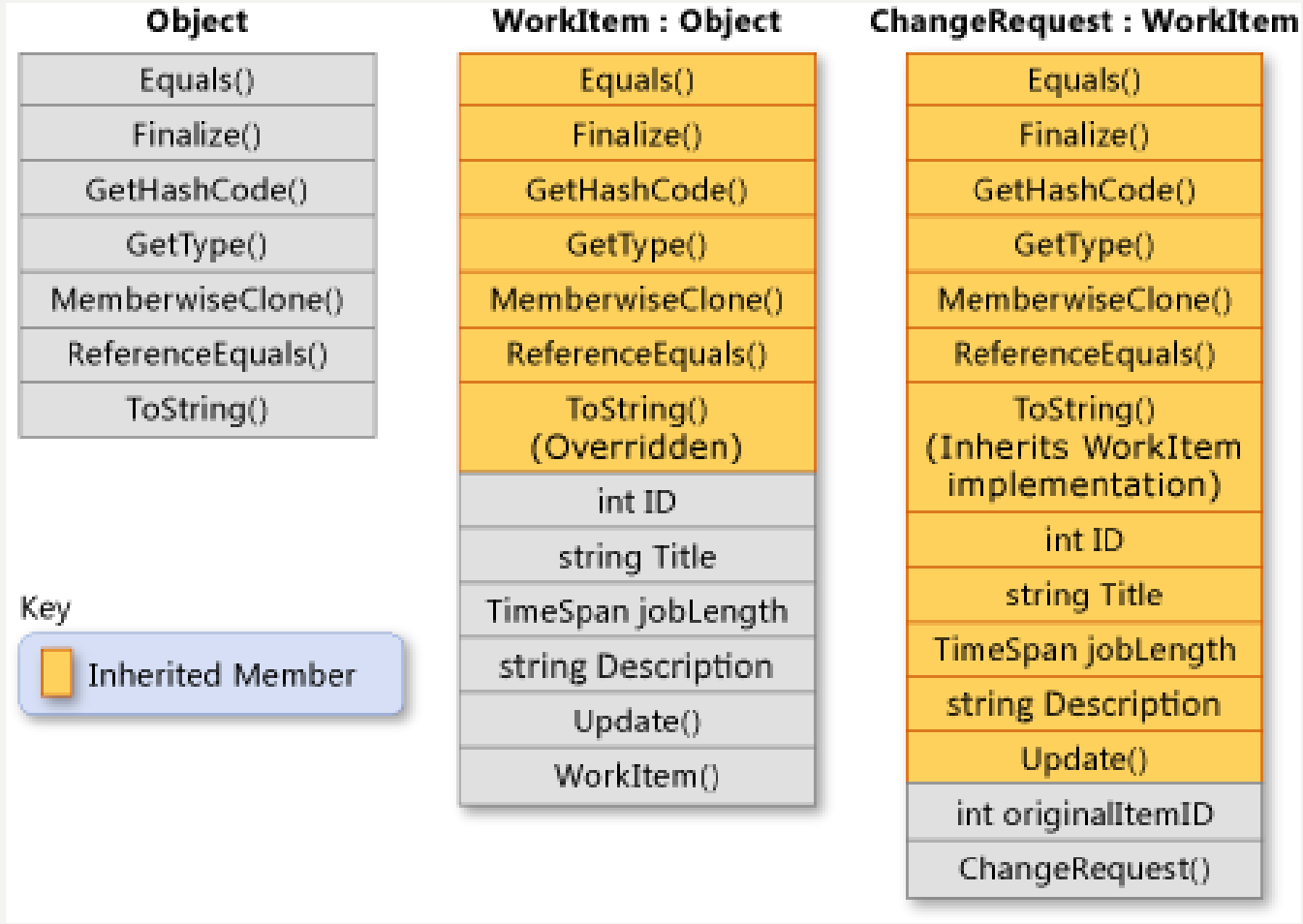
# CONSTRUCTOR CAN INVOKE ANOTHER CONSTRUCTOR IN THE CLASS

```
class Circle : Shape
{
    public Circle() : this(2.0) // calling the constructor below
    {
    }

    public Circle(double radius) : base(radius, 0) // calling the constructor of the base class
    {
    }

}
```

# METHOD OVERRIDING

- Child class has a different implementation of the method
- This is only possible if in the parent class the method has the keyword “virtual”

```
class ParentClass {
    public virtual void DoSomething(){
    }
}

class ChildClass : ParentClass {
    public override void DoSomething(){
    }
}
```


Object
Equals()
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals()
ToString()
WorkItem : Object
Equals()
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals()
ToString()
(Overridden)
int ID
string Title
TimeSpan jobLength
string Description
Update()
WorkItem()
ChangeRequest : WorkItem
Equals()
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals()
ToString()
(Inherits WorkItem
implementation)
int ID
string Title
TimeSpan jobLength
string Description
Update()
int originalItemID
ChangeRequest()
Key
Inherited Member

# OOP PROGRAMMING EXERCISE

- Create the schema of the last exercise and turn it into C# classes
- Please have a good look at the following principles
  - Apply inheritance correctly
  - Use the naming conventions for all elements
  - Use the correct data types
  - Use overriding

Bonus: create a main method in which you ask the user in the console (or if you rather have a form, also fine) and let the user pick a shape and instantiate one with the properties given by the user

# POLYMORPHISM

- Polymorphism is the ability of an object to take on many forms.
- Poly = multiple
- Morph = change

- Any Object in Java that has more than one IS-A relationship is polymorphic.
- Any class you define inherits from Object, so that class is a $ClassName and an Object.
- So... By default all objects in Java are polymorphic.

- We have already used this concept yesterday:

- Dog is a Dog
- Dog is a Mammal
- Dog is an Animal
- Dog is an Object

- Animal is an Animal
- Animal is an Object

```
public class Animal {
}

public class Mammal extends Animal {
}

public class Reptile extends Animal {
}

public class Dog extends Mammal {
}
```

# POLYMORPHISM

- The only way to access an object is through its reference variable

- Animal lassie = new Dog();

- Lassie is the reference variable of type Animal in this case. We can only access the properties and methods of that class.
- A reference variable can refer to any object of its declared type or its subtypes.

- Once a variable is declared, it cannot be changed.
- It can, however be reassigned to other reference variables (if it is not final).

- Animal lassie = new Dog();
- Mammal mammal = lassie;

- You can assign down, not up.

```
Dog dog = new Dog();
Mammal mammal = dog;
Animal animal = dog;
Object object = dog;

Object otherObject = new Object();
Animal otherAnimal = otherObject;
```

# POLYMORPHISM EXERCISE

- Add a new car class: SportCar
- Make SportCar extend Car

- Give SportCar ánd Car a new method: sayVroom()

- SportCar will print: "sportcar vrooming"
- Car will print: "car vrooming"

- Make a new class App.
- Create a main method.
- Make an instance of Car, with reference type Car
- Make an instance of SportCar, with reference type Car
- Call sayVroom on both.
- What will it say?

- Bonus: what happens when you make them of reference type Vehicle? Will it compile?

# STATIC

- Static keyword makes a member static
- Static class:
  - A static class can't be instantiated
  - This class can't have non static members
- Static method:
  - Class method
  - Are called using the classname, not the instance name
- Static field / property:
  - It is not bound to the instance of an object, but to the class

```
static class ClassName
{
    // static data members
   // static method
}
```

# SEALED

- On class level: avoid inheritance, a sealed class cannot be inherited from

- On method level: avoid overriding of a method, trying to override a sealed method creates a compiler error

```
class Parent { }

sealed class Child :  Parent { } //a sealed class can extend other classes
                                 //itself
```

# EXERCISE

- Method

- Add sealed to one of the methods in your shapes OOP program
- Try to override it
- Look at the pretty compiler error

- Class

- Add sealed to one of the base classes in your shapes OOP program
- Look at the pretty compiler error

- But what if you'd really have to add something to a class…?

# EXTENSION METHODS

A way of adding to sealed classes

First parameter is preceded by "this", it specifies the type for which the extension method applies

They are very powerful, this way you can even add to the .NET framework and third party classes

The only difference between a normal static method and an extension method is that the first parameter of the extension method specifies the type that it is going to operator on, preceded by the ***this*** keyword and that the extension method is always in a static class.

```
public static class Extension
{
    public static int WordCount(this string str)
    {
        return 42;
    }
}
```

# EXERCISE

- Use extension method to still add to the method to the sealed class of your OOP program

- Call the type specified after this and see that sealed is available

# ENCAPSULATION

- Pillar of OO
- Prevent security issues by other mistakes we might make
- Wrap data in a class
- Private fields and public properties (or other public get (accessor) / set (mutator) methods)
- Properties can be used to make data read-only or write-only as well

# EXERCISE

- Basic exercise:
  - Encapsulate the Car and Vehicle classes of the polymorphic exercise

- Advanced exercise:
  - Demonstrate the advantage of encapsulation

# ABSTRACT

- Methods → methods without implementation
- Properties → property with no implementation
- Events → don't worry about them now
- Indexers → used to index instances from a class (like arrays, don't worry about them now)
- Classes → can't be instantiated

```
public abstract class Animal
{
    public abstract string Something { get; }
}

class Mammal : Animal
{
    private string something;

    public override string Something
    {
        get
        {
            return something;
        }
    }
}

class Dog : Mammal
{
}
```

# ABSTRACT CLASSES

- Class that can't be instantiated
- Child classes that are not abstract must implement abstract methods from parents
- Class can be abstract without abstract methods
- Abstract class can have implemented methods

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;
    public Square(int n) => side = n;
    public override int GetArea() => side * side;
}
```

# EXERCISE

- Create an animal class, make it abstract
- Add an abstract method MakeSound to it
- Give it an abstract property Weight
- Derive two reasonable classes from Animal (e.g. butterfly and rabbit) and implement the necessary things
- In a Main method, instantiate your derived classes and call the MakeSound method
- Can you make the reference type of the abstract class?
- Can you instantiate the abstract class?

# INTERFACES

- Contract for behavior of a class
- Class implementing an interface (or multiple interfaces) must implement the members of the interface
- Interfaces can contain:
  - Methods
  - Properties
  - Indexers
  - Events
  - Constants
  - Static constructors
  - Nested types
  - Fields
  - Other access modifiers than public (public is default)
- Methods in interfaces can contain a body, a so called default implementation
- Convention, start name with I

```
public interface ISomeInterface
{
    string Name {get; set;}
    bool Test(Object object);
}
```

# EXERCISE

- Create an interface for your OOP design for symmetric classes
- Make sure to choose the correct naming for this interface
- Add some relevant methods to this interface
- Make sure the symmetric classes implement the interface

# ABSTRACT CLASS VS INTERFACE

- Abstract class
- Default functionality
- Rules for deriving from a certain class (rules for being something), used when derived class are logically derived from the abstract class
- Reusable
- Class can only have one (direct) base class

- Interface
- Contract for behavior
- Loose coupling
- Separate the method definition from inheritance hierarchy
- Class can implement more than one interface

# PARTIAL

- C# special feature!
- Classes can be spread over multiple files
- Combined into one during compilation
- Awesome, but why would I want this?
  - Multiple developers can work on the same class simultaneously
  - UI design and business logic can be split up (winforms)
  - You can add to autogenerated code
  - Splitting up large classes can help maintenance (well…. Could?)

- Rules for partial classes:
- All files should use partial keyword right before the type (class, struct or interface) and have same name
- All the files need to be available during compilation
- Files need to be in same assembly and namespace (file names can differ)
- All partial definition should have the same access modifier
- If any part of the partial class is declared as an abstract, sealed, or base, then the whole class is declared of the same type.
- Inheritance: they cannot define different base classes

# EXERCISE

- Let's assume our Square class is getting way too big
- Split it up using partial
- Check if it's still working

- File A

```
partial class SomeClass {
   private string firstName = string.Empty;
   public string FirstName { get; set; }
}
```

- File B

```
partial class SomeClass {
   private string lastName = string.Empty;
   public string LastName { get; set; }
}
```

# STRUCT

- Value type to encapsulate data and related methods
- Value semantics:  Struct contains an instance of its own type
- Can't declare noargs constructor, not possible to override the default that produces the default value of the type
- Fields and properties can't be initialized at declaration (static and const can be)
- Can't derive or be derived (interfaces, can be implemented)
- Light weight alternative to class, use it for small things

Instantiating a struct:

- New keyword
- Or initialize all instance fields, before first usage

```
public struct Rectangle
{
        public double x;
        public double y;
}
    public static void Main()
    {
        Rectangle r;
        r.x = 3;
        r.y = 4;
        Console.WriteLine($"({r.x}, {r.y})");  // output: (3, 4)
    }
```

# WHEN TO USE A STRUCT?

- Controversial!

Some smart things to do:

- Don’t make them too big
- Use it for communicating with C / C++ code
- They are more like reference types than actual objects
- They don’t need to be inherited or inherit

# EXERCISE

- Have a look at your OOP application.
- Are there places where you could use a struct?
- Create a struct for a point that takes to coordinates

# EXERCISE: OOP DESIGN

- Draw the OOP design for animals
- Make sure to include: deer, mouse, whale, dolphin, snake, bird, balloon fish, rabbit, fly (and add some logical layers)
- Make sure that we can distinct predators and prey
- Add a method MakeSound, Move, Eat and ObtainOxygen to all of them
- Add fields for weight, height, color, max speed (choose the right data type and encapsulate them well)

# EXERCISE: PROGRAM YOUR DESIGN

- Code the previously drawn design

# WRAP UP DAG 2

- Object georiënteerd programmeren
- Methoden
- Scope
- Encapsulation
- Toegangsbeperking
- Static
- Inheritance
- Polymorphisme
- Klassen
- Abstracte klasses
- Interfaces
- Partial

# NOG VRAGEN?

Stel ze via Linkedin:

https://www.linkedin.com/in/maaikevanputten/

Of WhatsApp:

0683982426