# Introduction to OOP

Nemanja Mićović, ISWiB 2019

# ISWIB 2019

14-22.july
2019
ISWIB
International student week in Belgrade
the future
is calling

# About the author

- Teaching assistant at Faculty of Mathematics
- PhD student of Machine Learning
- AI engineer at Lebo Apps GmbH
- GNU/Linux enthusiast
- Passionate gamer
- Main organizer of RISK (risk.matf.bg.ac.rs)
- Contact:
  - Email: `nemanja_micovic@matf.bg.ac.rs`
  - LinkedIn
  - GitHub
  - Website
  - Instagram

# Course outline

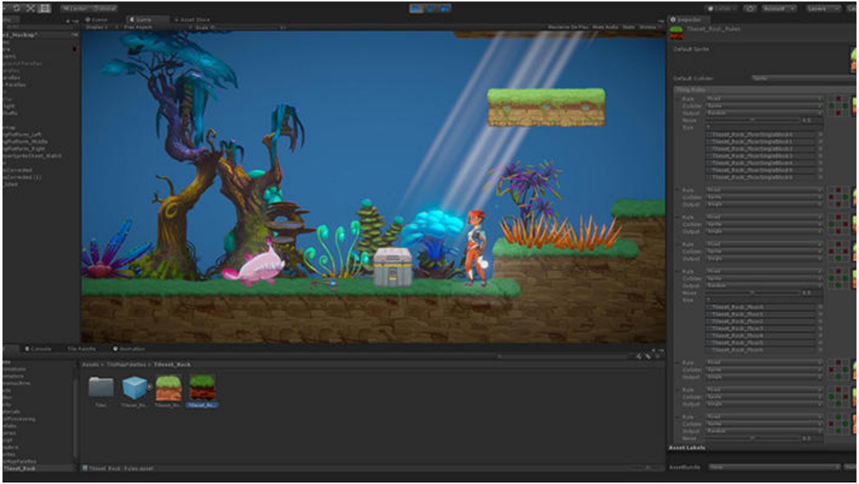
- Object oriented programming (day 01)
- Introduction to Unity - 2D (day 02)
- Introduction to Unity - 3D (day 03)
- Projects (day 04)
- Projects (day 05)

# Object oriented programming (day 01)

- Programming language C#
- Basics of object oriented programming
- Learning to use Visual Studio

# Introduction to Unity - 2D (day 02)

- Introduction to Unity and 2D



# Introduction to Unity - 3D (day 03)

- TO DECIDE

# Projects (day 04 and day 05)

# Day 01 - Introduction to (Object Oriented) Programming

# Language C#

- Developed by Microsoft
- A general purpose multi-paradigm programming language
- Used by Unity engine
- Has syntax inspired by C, C++ and Java
- Will be very familiar to you if you've done some C, C++ or Java before

# Language C#

```
using System;
namespace HelloWorld
{
    class Hello
    {
        public static void Main()
        {
            // This is a comment
            Console.WriteLine("Hello World!");
        }
    }
}
```

# Statements

- Statements are commands that the programmer issues
- Computer executes the statements and produces some result
- It's important that we are very precise - because computer is a machine
- In real world, statements would be something like:
  - `turn aroud`
  - `get up`
  - `start running`
  - `turn off the light`
- In C#, a statement is **terminated** by a semicolon `;`

# Statements

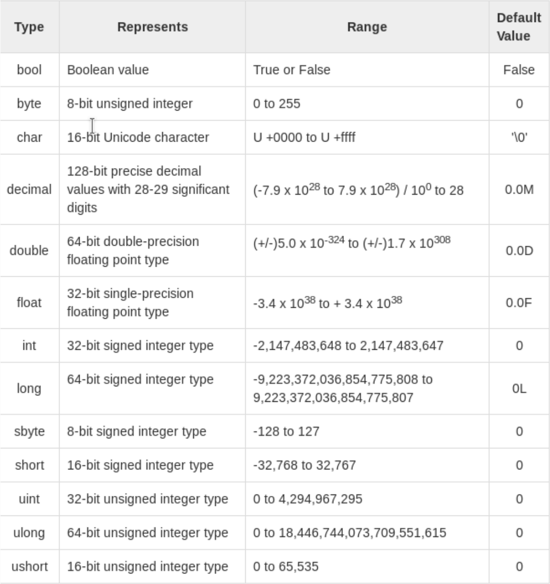
```
using System;
namespace HelloWorld
{
    class Hello
    {
        public static void Main()
        {
            int x = 10;                 // statement 1
            int y = 20;                 // statement 2
            Console.WriteLine(x + y);   // statement 3
        }
    }
}
```

# Types

- Language C# is a **typed** language
- This means that every variable we create needs to have a **type**

```
// Wrong! What is the type of x?
x = 5

// Correct! Type of x is Integer
int x = 5;
```

# Types



| Type | Represents | Range | Default Value |
|---|---|---|---|
| bool | Boolean value | True or False | False |
| byte | 8-bit unsigned integer | 0 to 255 | 0 |
| char | 16-bit Unicode character | U +0000 to U +ffff | '\0' |
| decimal | 128-bit precise decimal values with 28-29 significant digits | (-7.9 x $10^{28}$ to 7.9 x $10^{28}$) / $10^{0}$ to 28 | 0.0M |
| double | 64-bit double-precision floating point type | (+/-)5.0 x $10^{-324}$ to (+/-)1.7 x $10^{308}$ | 0.0D |
| float | 32-bit single-precision floating point type | -3.4 x $10^{38}$ to + 3.4 x $10^{38}$ | 0.0F |
| int | 32-bit signed integer type | -2,147,483,648 to 2,147,483,647 | 0 |
| long | 64-bit signed integer type | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 0L |
| sbyte | 8-bit signed integer type | -128 to 127 | 0 |
| short | 16-bit signed integer type | -32,768 to 32,767 | 0 |
| uint | 32-bit unsigned integer type | 0 to 4,294,967,295 | 0 |
| ulong | 64-bit unsigned integer type | 0 to 18,446,744,073,709,551,615 | 0 |
| ushort | 16-bit unsigned integer type | 0 to 65,535 | 0 |

# Types

- We also call the previously shown types with the name **primitive types**
- Why? They are **primitive** :)
- They don't represent anything complex or composite
- From primitive types, in Unity we will be mostly be using `bool`, `char`, `int` and `float`
- There are also more complex types that we represent with **classes**
- We will get to that a bit later - but it's the **main topic** of the object oriented programming

# Variables

- With variables we wish to **store** some information at some convenient place
- We also wish to be able to **retrieve** that information when needed
- Variables have a **name** and a **type**
- A **type** is what we’ve seen before - for example `int`
- A **name** is the name of the variable - for example `x`

```
int x = 10;
int y;
float z = 3.1f;
char w = 'a';
```

# Variables - exercise

**Task 01**

Write a program that creates variables `x` and `y` with values 10 and 20. Write out the values `x*y`, `x+y`, `x-y` and `x/y`.

# If statement

- What if we wish to do **something** depending on a **certain condition**?
    - Close the door **if** it's cold, **otherwise** do nothing?
    - **If** today is Tuesday then learn C#, **otherwise** have fun
- We can accomplish this using the **if statement**

# If statement

```
static void Main(string[] args)
{
    int x = 4;
    bool even = x % 2 == 0;
    if (even)
    {
        Console.WriteLine("Number is even!");
    }
    else
    {
        Console.WriteLine("Number is not even!");
    }
}
```

# If statement - multiple branches

- We can also allow if to have multiple branches
- How much? It's up to you.

```
if (CONDITIION 1) STATEMENT 1
else if (CONDITIION 2) STATEMENT 2
else if (CONDITIION 2) STATEMENT 2
else if (CONDITIION 3) STATEMENT 3
...
else if (CONDITIION N) STATEMENT N
else STATEMENT N+1
```

# Loops

- Loops are a mechanism that allows us to execute a block of statements **multiple times**
- If you wanted to write `Hello` 10 times to standard output, how would you do it?

# Loops

```
Console.WriteLine("Hello");
Console.WriteLine("Hello");
Console.WriteLine("Hello");
Console.WriteLine("Hello");
Console.WriteLine("Hello");
Console.WriteLine("Hello");
Console.WriteLine("Hello");
Console.WriteLine("Hello");
Console.WriteLine("Hello");
Console.WriteLine("Hello");
```

# Loops

Or maybe this?

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Hello");
}
```

# Loops - for loop

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Hello");
}
```

- At the start, variable `i` is 0
- Then code inside the block is executed
- Then `i` is incremented (value increased by 1)
- Then `i` is checked against the condition `i < 10`, it's true as `i = 1`
- So next **iteration** happens and code inside the block is executed
- Then `i` is incremented (value increased by 1)
- Then `i` is checked against the condition `i < 10`, it's true as `i = 2`
- So next **iteration** happens and code inside the block is executed

# Loops - for loop

- We keep on going until. . .
- Then code inside the block is executed
- Then `i` is incremented (value increased by 1)
- Then `i` is checked against the condition `i < 10`, it's true as `i = 9`
- Then code inside the block is executed
- Then `i` is incremented (value increased by 1)
- Then `i` is checked against the condition `i < 10`, it's false as `i = 10`
- The loop stops and program continues after the loop

# Loops - while loop

```
i = 0;
while (i < 0)
{
    Console.WriteLine("Hello");
    i++;
}
```

- A **while** loop is a bit different and simpler
- It keeps on executing until the **condition** is true
- Usually it's used when we **don't know** the amount of required iterations
- Does the code above remind you of something?

# Functions

- Functions are a basic building block of our programs
- Function is a block of code that we give a name to
- Ignore the word `static` for now

```
static int Add(int a, int b)
{
    return a + b;
}
static void ShowFirst(int n)
{
    for (int i = 0; i < n; i++)
    {
        Console.WriteLine(i+1);
    }
}
```

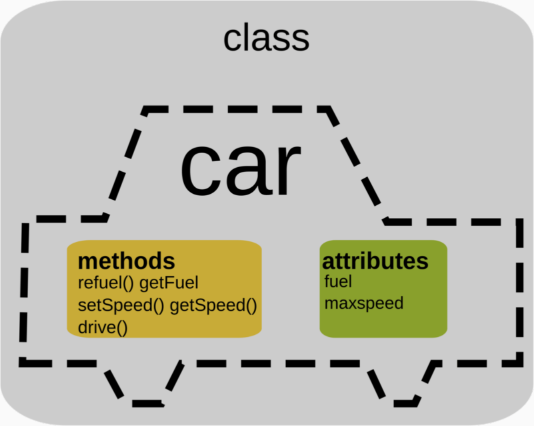
# Functions

- It’s very good to break our code into multiple functions
- As program grows we break it down into multiple files also
- But how do we describe complicated things with these simple constructs?
- Using Object Oriented Programming, it’s much easier to do it! Let’s check it out!

# Object Oriented Programming (OOP)

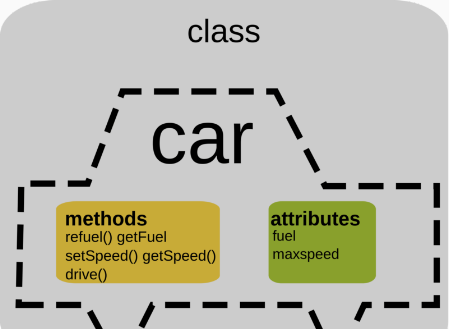
# The OOP

- So far we have learned about types like `int`, `float` and similar
- We also mentioned that we can construct more complex types. How?
- By constructing a **class**!



# Classes and objects

- A class is a template that **describes** an **object**
- A class defines:
  - Attributes (we also call them properties)
  - Methods (they are basically functions)
- A **attribute** is some data that the object contains
- A **method** is a function that can get invoked on an object



# The OOP

- We use classes to create templates for the objects that exist in *our world*
- For example, if we are creating:
  - music program - classes are `Song`, `Artist`, `Album`, `Playlist`...
  - video game - classes are `Player`, `Enemy`, `NPC`, `Tree`, `Ball`...
- Objects are concrete examples of a class, we also call them **instances**
- Our program is an **interaction of objects**
- This is similar to real world actually!

# Class - constructor

- A Class has a **constructor** (shorthand is `ctor`)
- A constructor is a **special function** that allows us to **instantiate** an object

```java
public class Point {
    // Class attributes
    private float x, y;

    // Constructor (ctor)
    public Point(float someX, float someY)
    {
        x = someX;
        y = someY;
    }
}
```

# Class - constructor

- How do we instantiate an object?
- We use the keyword `new`
- This invokes the constructor from the class

```
static void Main(string[] args)
{
    Point p = new Point(2.3f, 4.1f);
    Console.WriteLine(p);
}
```

# Class - methods

- Methods are services that an object gives us to use
- For example, we can ask a point to move in 2D space

```
public class Point {
    private float x, y;
    public Point(float someX, float someY) {
        x = someX;
        y = someY;
    }
    public void translate(float dx, float dy) {
        x += dx;    // update x by vector dx
        y += dy;    // update y by vector dy
    }
}
```

# Class - methods

- Please do note that we **call** a method **on an object**!

```
Point p = new Point(1.0f, 2.0f);
p.translate(3f, 4f);
// NOT! translate(p, 3f, 4f) or translate(3f, 4f)...
```

# Class - visibility specifiers

- So far you've probably noticed things like `private` and `public`
- What are they?
- They are only used for things that are class related
- They handle the problem of **visibility** of a function, class or variable
- If something is `private`, then we can't see it outside of its scope
- If something is `public`, then everyone can see it

# Class - visibility specifiers

- Outside of class `Point`, you **can't** see variables `x` and `y`!

```java
public class Point {
    // Class attributes
    private float x, y;

    // Constructor (ctor)
    public Point(float someX, float someY)
    {
        x = someX;
        y = someY;
    }
}
```

# C# Coding Standards

**Variables:**

- Variable names should start with lowercase
- If a name is formed of multiple words, use camelcase notation

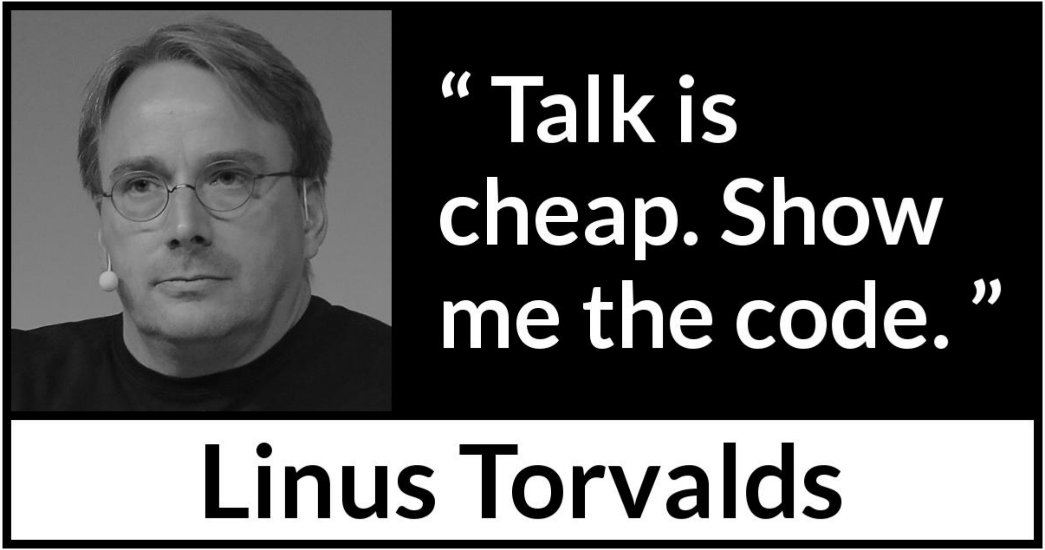
**Functions:**

- Function name should start with a Capital case letter
- If a name is formed of multiple words, use camelcase notation
- Same for class methods

**Class attributes:**

- Name should be prefixed with an underscore _
- If a name is formed of multiple words, use camelcase notation

# Live coding - Classes - A small Gaming example



“ Talk is cheap. Show me the code. ”

Linus Torvalds

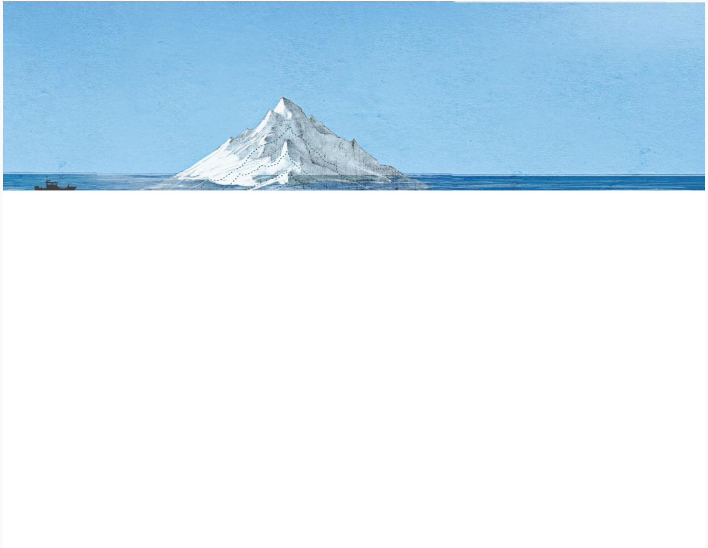
# OOP - Advanced concepts

# The Pillars of OOP

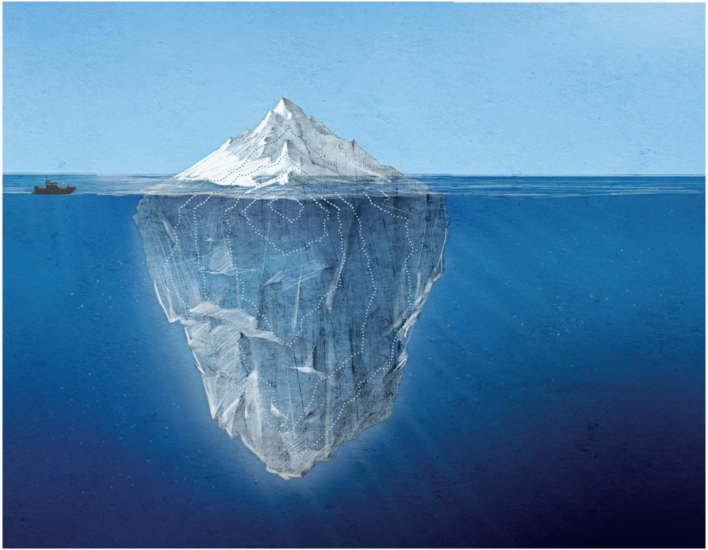
- The three main pillars of OOP are:
    - Encapsulation
    - Polymorphism
    - Inheritance

# Encapsulation

# Encapsulation



# Encapsulation



# Encapsulation

- Sometimes, we only see the *tip of the iceberg*
- And also, lots of times, we only **wish** to see the *tip*
- It makes using libraries and foreign code **much easier**

*Encapsulation is a principle by which we wish to hide internal logic from the outside world*.

- It's acomplished by:
  - Showing a *public interface* to the user (exposed via `public`)
  - And hiding internal logic (hidden via `private`)

# Inheritance

# Inheritance

- Let's assume we need to create a class for both a `Player` and `Enemy`
  - Both of them will have a position in our game world?
  - Both of them will have health?
- So, do we create two classes that both have the same things?
- Or, maybe there's a better way?

# Inheritance

- Inheritance allows us to **extend** the functionality of a class
- We say that class `B` **inherits** class `A`.
- This means that `B` **is** also `A`.
- But `A` is not `B`.
- `B` is a *special* case of `A`
- You confused?

# Inheritance

- Inheritance allows us to **extend** the functionality of a class
- We say that class `Husky` **inherits** class `Dog`.
- This means that `Husky` **is** also `Dog`.
- But `Dog` is not `Husky` (not every dog is a Husky!)
- `Husky` is a *special* case of `Dog`
- Not confused anymore?

# Inheritance in C#

```
class GameObject
{
    private Vec2 _position;

    public GameObject(Vec2 position)
    {
        _position = position
    }
}
```

# Inheritance in C#

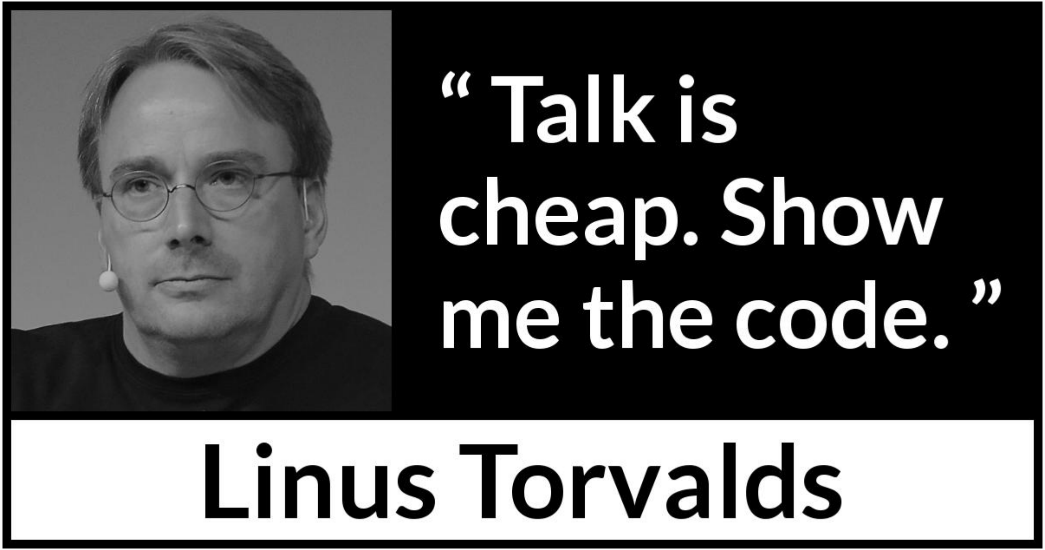
```
class Player: GameObject
{
    public Player(Vec2 position) : base(position) { }
}

class Enemy: GameObject
{
    public Enemy(Vec2 position) : base(position) { }
}
```

# Inheritance in C#

- Class can inherit only **one** class
- Casting to parent class is often useful
- `object1` and `object2` give out only the public interface of the class `GameObject`

```
GameObject object1 = new Player(new Vec2());
GameObject object2 = new Enemy(new Vec2());
```

# Live coding - Inheritance - A small Gaming example



“ Talk is cheap. Show me the code. ”

Linus Torvalds

# Polymorphism

# Polymorphism

- As we've seen, we can cast objects of `B` onto parent class `A`

```
GameObject object1 = new Player(new Vec2());
GameObject object2 = new Enemy(new Vec2());
```

- This also allows a cool property - *hierarchical polymorphism*
- That is, we can define a function in class `A` and *redefine* it in class `B`

# Polymorphism

```
class GameObject
{
    private Vec2 _position;

    public GameObject(Vec2 position)
    {
        _position = position
    }

    public void Move(Vec2 movementVector)
    {
        _position += movementVector;
    }
}
```

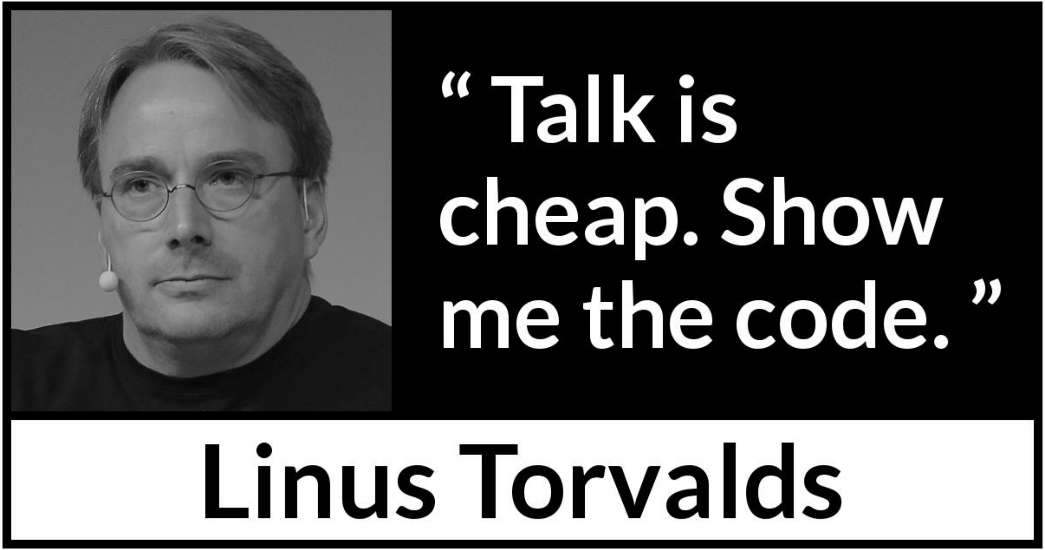
# Polymorphism

```
class Player: GameObject
{
    public Player(Vec2 position) : base(position) { }

    public override void Move(Vec2 movementVector)
    {
        // Add our own custom logic
        Console.WriteLn($"Player is moving by {movementVector}!");

        // Invoke the parent class implementation
        super.Move(movementVector);
    }
}
```

# Polymorphism

```
GameObject object = new Player(new Vec2());

// Call the .Move method of the `Player` class
object.Move(new Vec2(2f, 3f));

// RESULT: Player is moving by (2.0, 3.0)!
```

# Live coding - Polymorphism - A small Gaming example



“ Talk is cheap. Show me the code. ”

Linus Torvalds