OOPS - Introduction

Object Oriented Programming

What is OOP? - Introduction

This series will not be language -specific
Examples will be not require knowledge of the language being used

What is OOP?

In order to understand what object oriented programming is, It’s best to first understand what objects are

In order to understand what objects are, it’s best to first understand what primitive data types are

What is OOP? - Primitive Data

Primitive data types store single, simple values

Examples:

- Byte
- Int
- Float
- Boolean
- Double
- Char

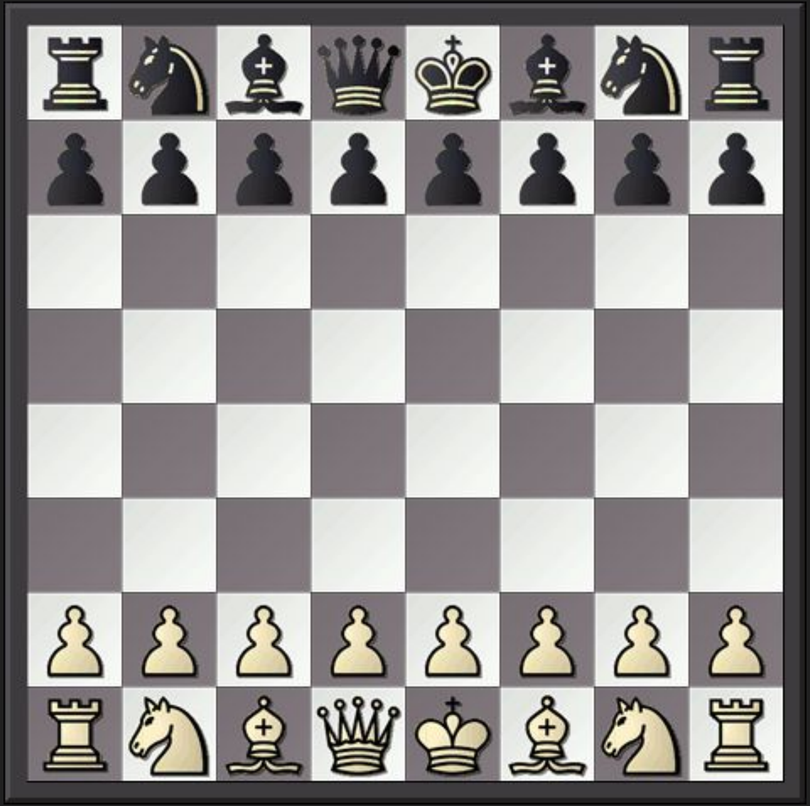
What is OOP? - Primitive Data

Program at the time weren’t very complicated compared to today standard

As programs became larger and more complex, only using primitive data types wasn’t sufficient anymore

Programmers began to need to group similar pieces of data together

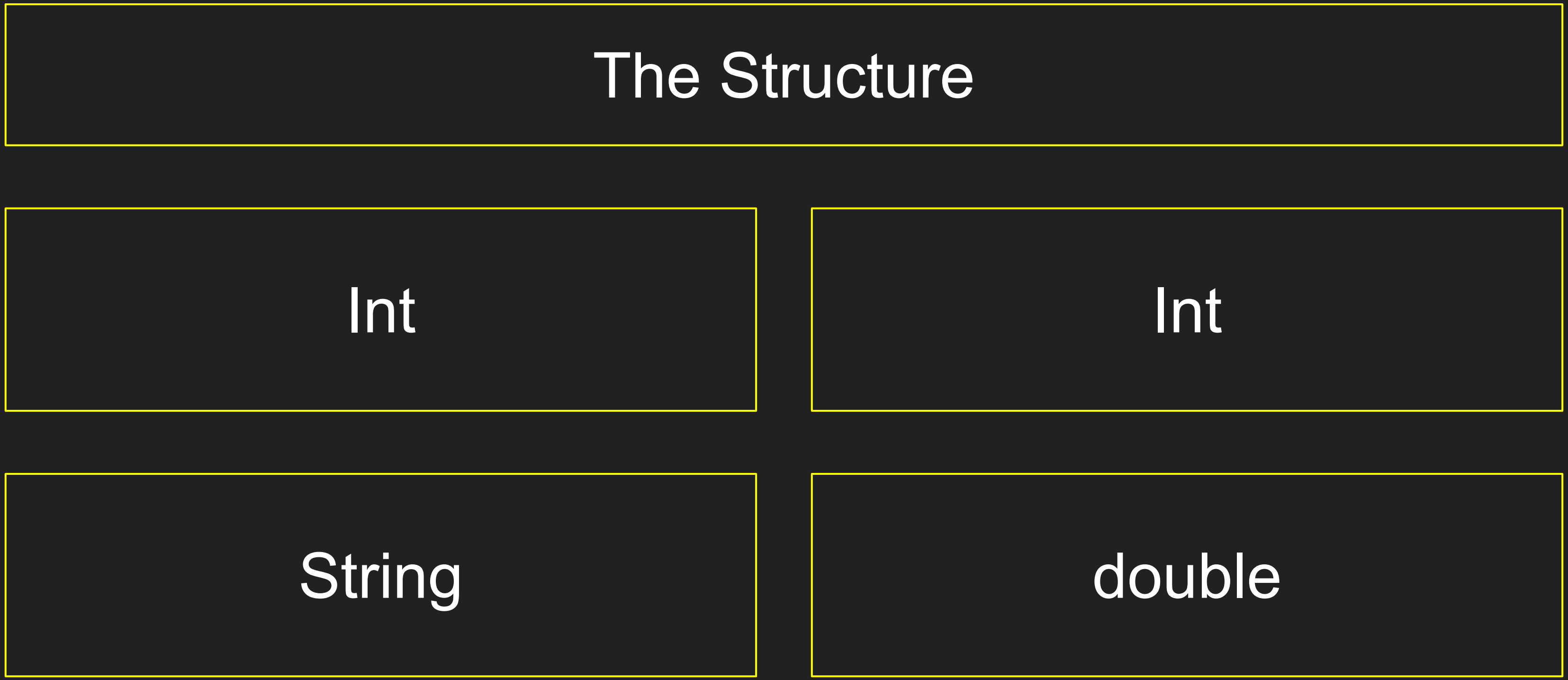
What is OOP? - Chess Example



What is OOP? - The Structure

The Structure	The Array
Stores many pieces of data	Stores many pieces of data
Can store different types of data	Cannot store different types of data

What is OOP? - The Structure



The Structure

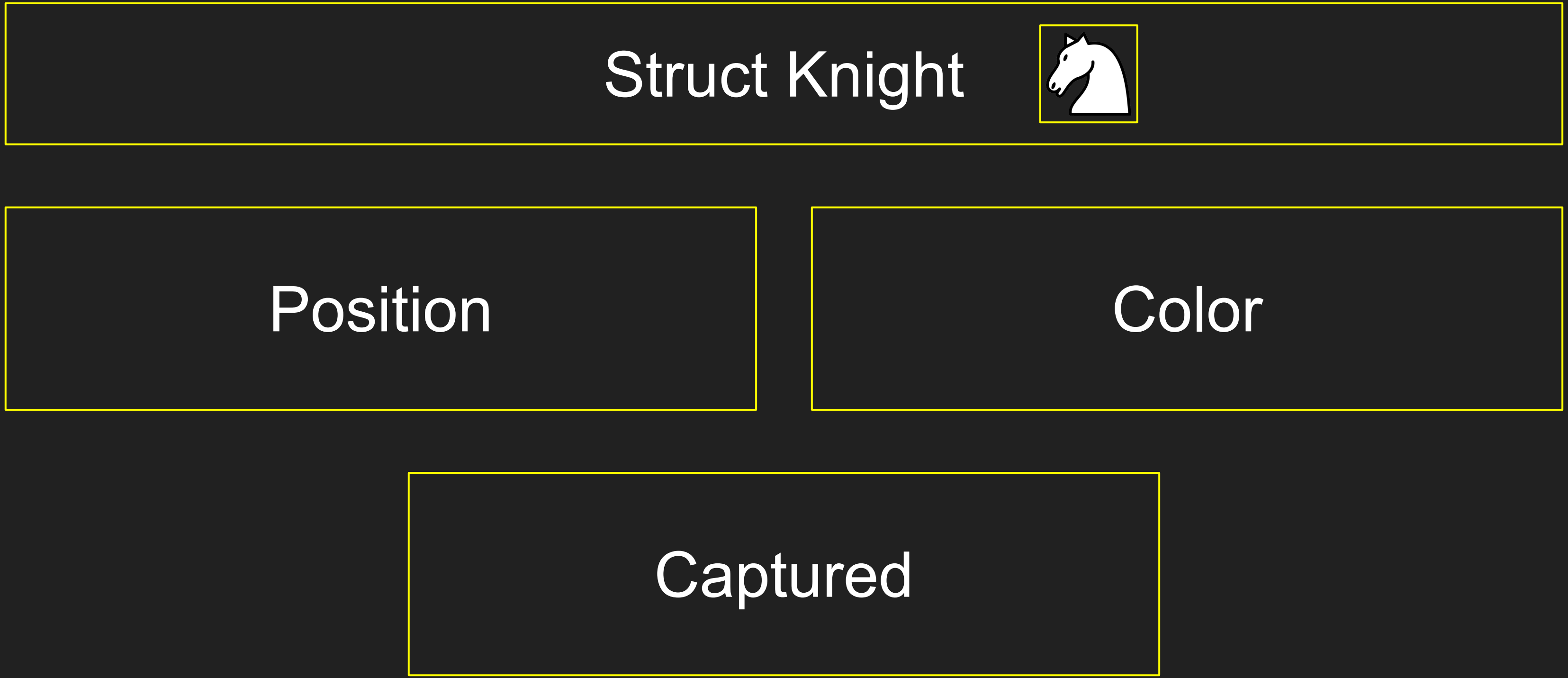
Int

Int

String

double

What is OOP? - The Structure



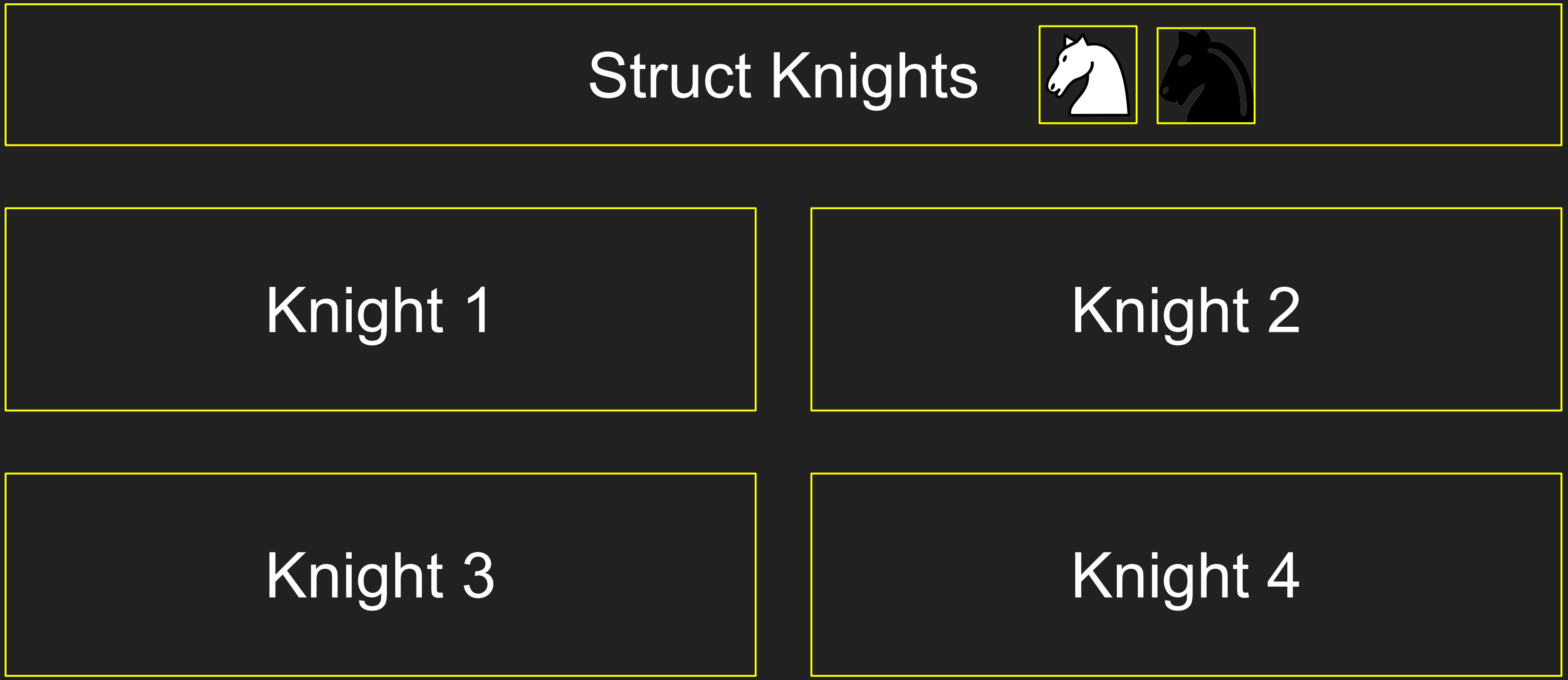
Struct Knight

Position

Color

Captured

What is OOP? - The Structure



Struct Knights ♘ ♞

Knight 1

Knight 2

Knight 3

Knight 4

What is OOP? - The Structure

Struct WhiteKnights ♘	Struct BlackKnights ♞
Knight 1	Knight 2
Knight 3	Knight 4

What is OOP? - The Structure

The main issue with structures is that you cannot define functions within one

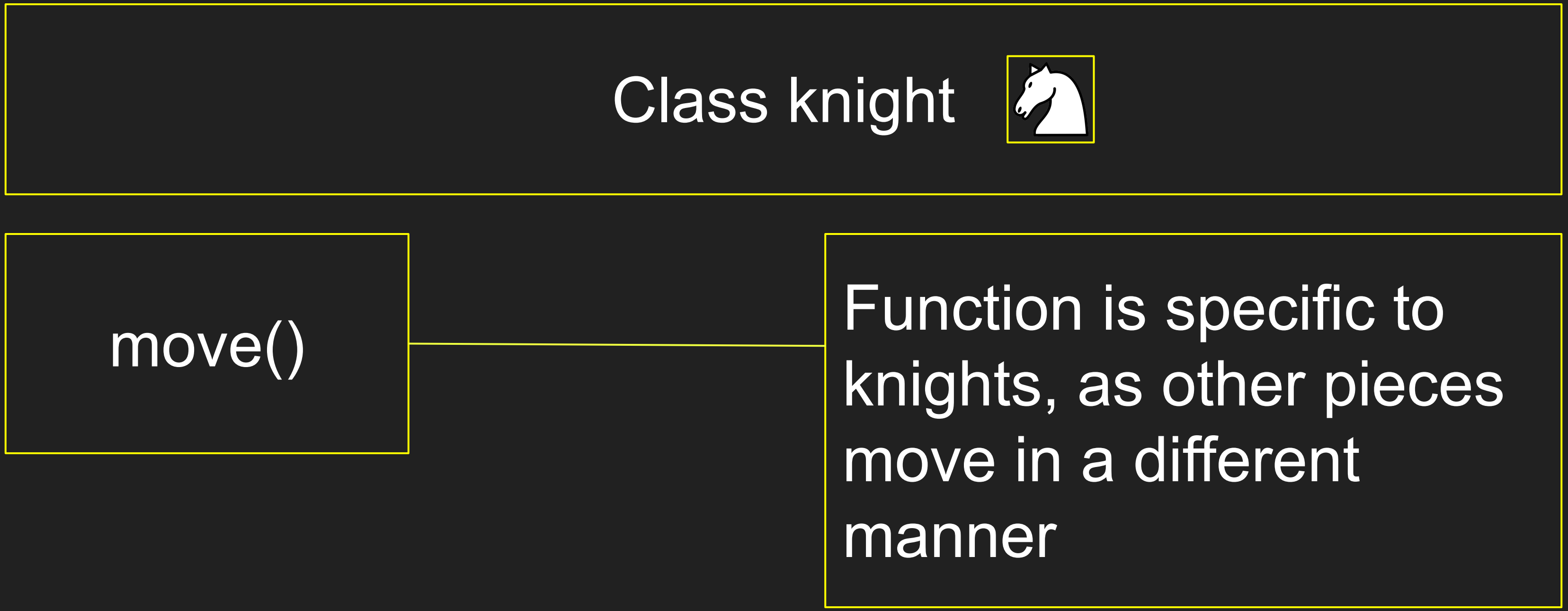
Thinking about the chess example, this prevents you from defining a function specific to the knights, such as their move function, within the structure

What is OOP? - Objects

Objects are instances of a class

Classes are templates for objects

What is OOP? - Classes



Class knight

move()

Function is specific to knights, as other pieces move in a different manner

What is OOP? - Classes

Class knight ♘

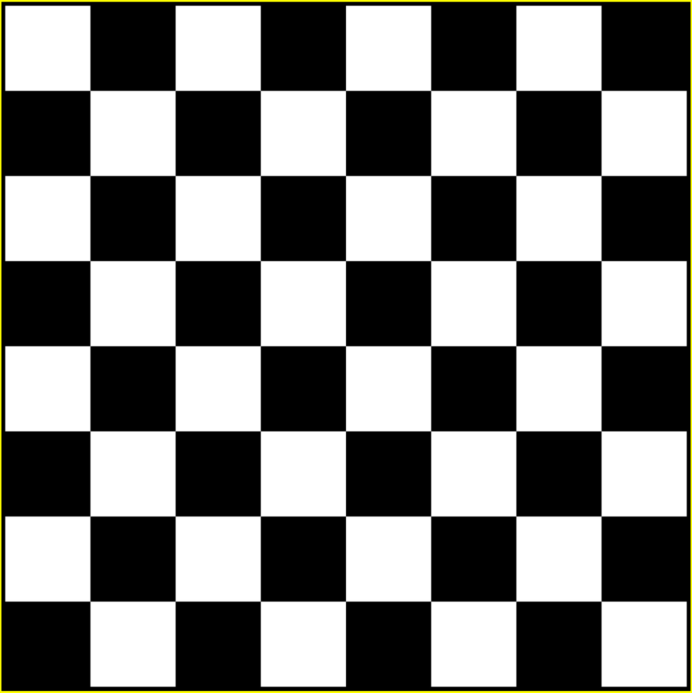
move()

position

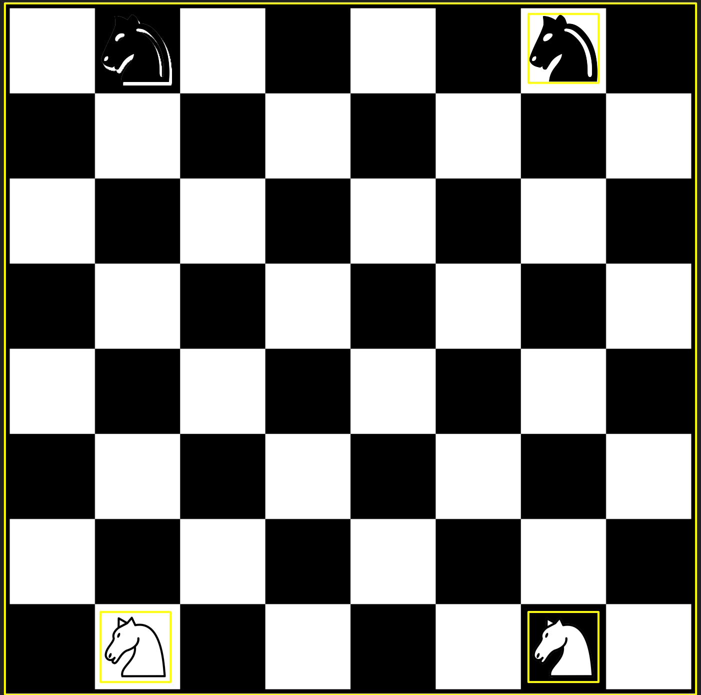
color

Color and position are not initialized, as different instances of a knight will have different values for these variables

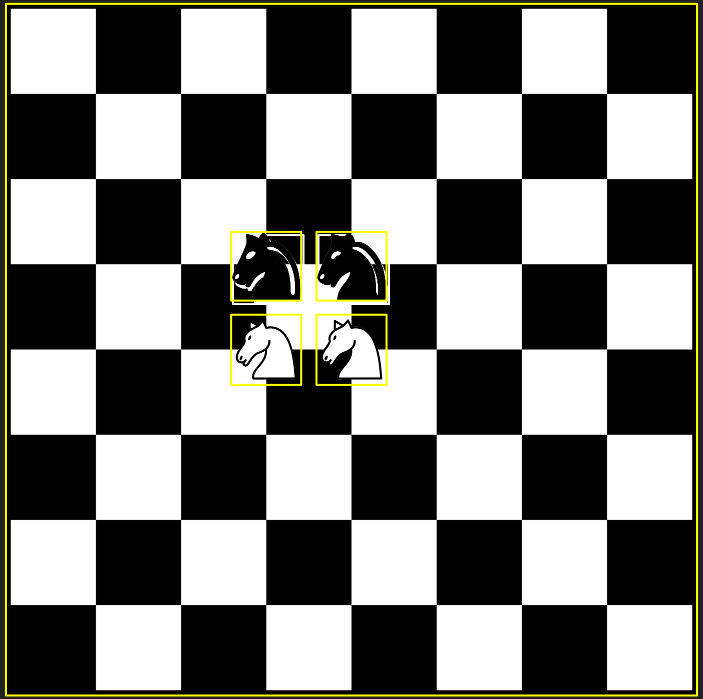
What is OOP? - Classes



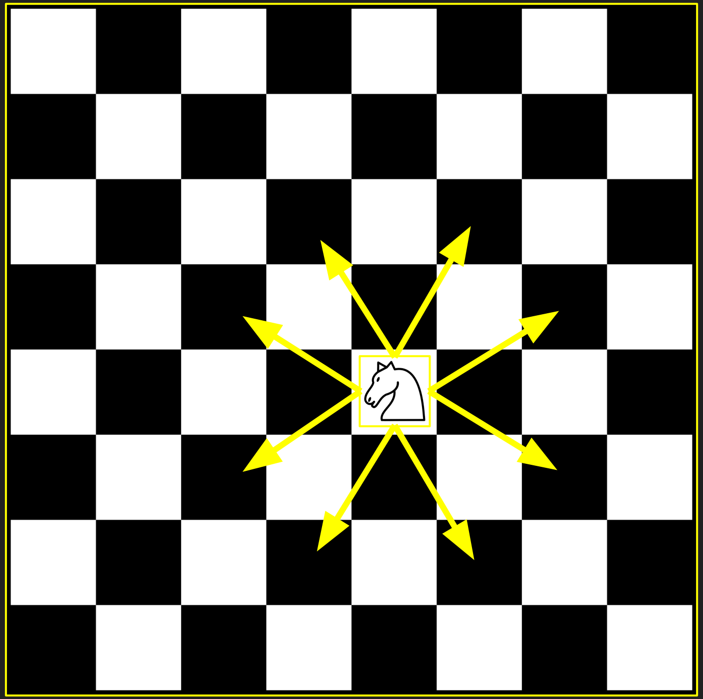
What is OOP? - Classes



What is OOP? - Classes



What is OOP? - Classes

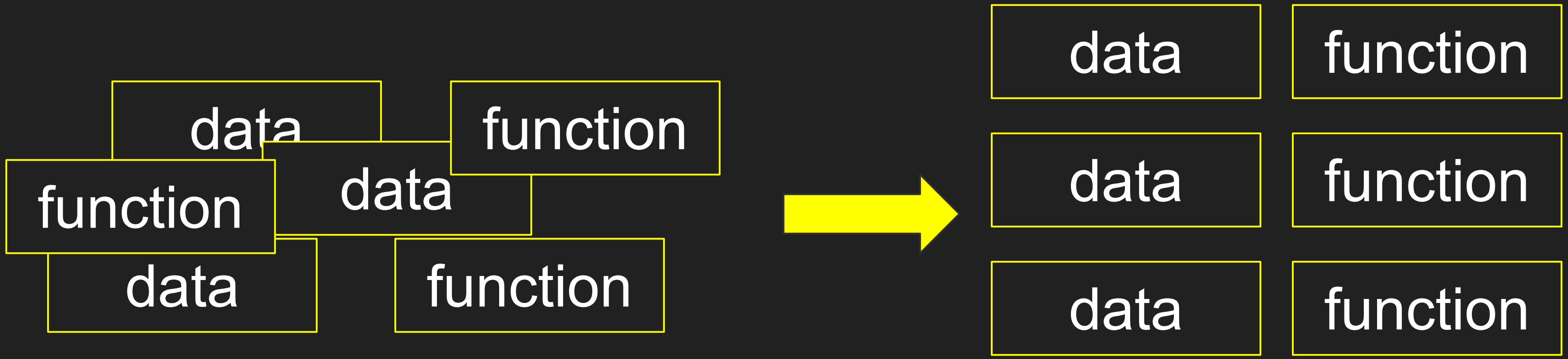


What is OOP? - Classes

Whereas the knight class represents any given knight, a knight, a knight object represents only one singular knight

What is OOP? - Classes

Object oriented programming helps programmers create complex programs by grouping together related data and functions



Series - Overview

Throughout the video we will explain OOP using its four main principles:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation - Introduction

Encapsulation refers to bundling data with methods that can operate on that data within a class

Essentially, it is the idea of hiding data within a class, preventing anything outside that class from directly interacting with it

Encapsulation - Introduction

This does not mean that members of other classes cannot interact at all with the attributes of another object

🚫 🚫 🚫

Encapsulation - Introduction

Members of other classes can interact with the attributes of another object through its methods

Remember, methods are the functions defined within the class

Encapsulation - Introduction

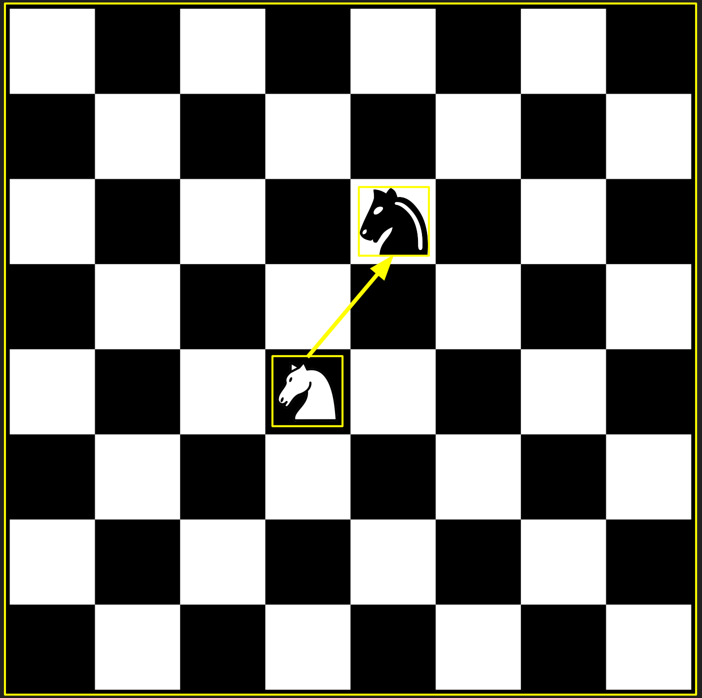
Getting Methods

Setting Methods

Retrieving information

Changing information

What is OOP? - Classes



piece.getColor()

Checks the color of any given piece from anywhere in the program

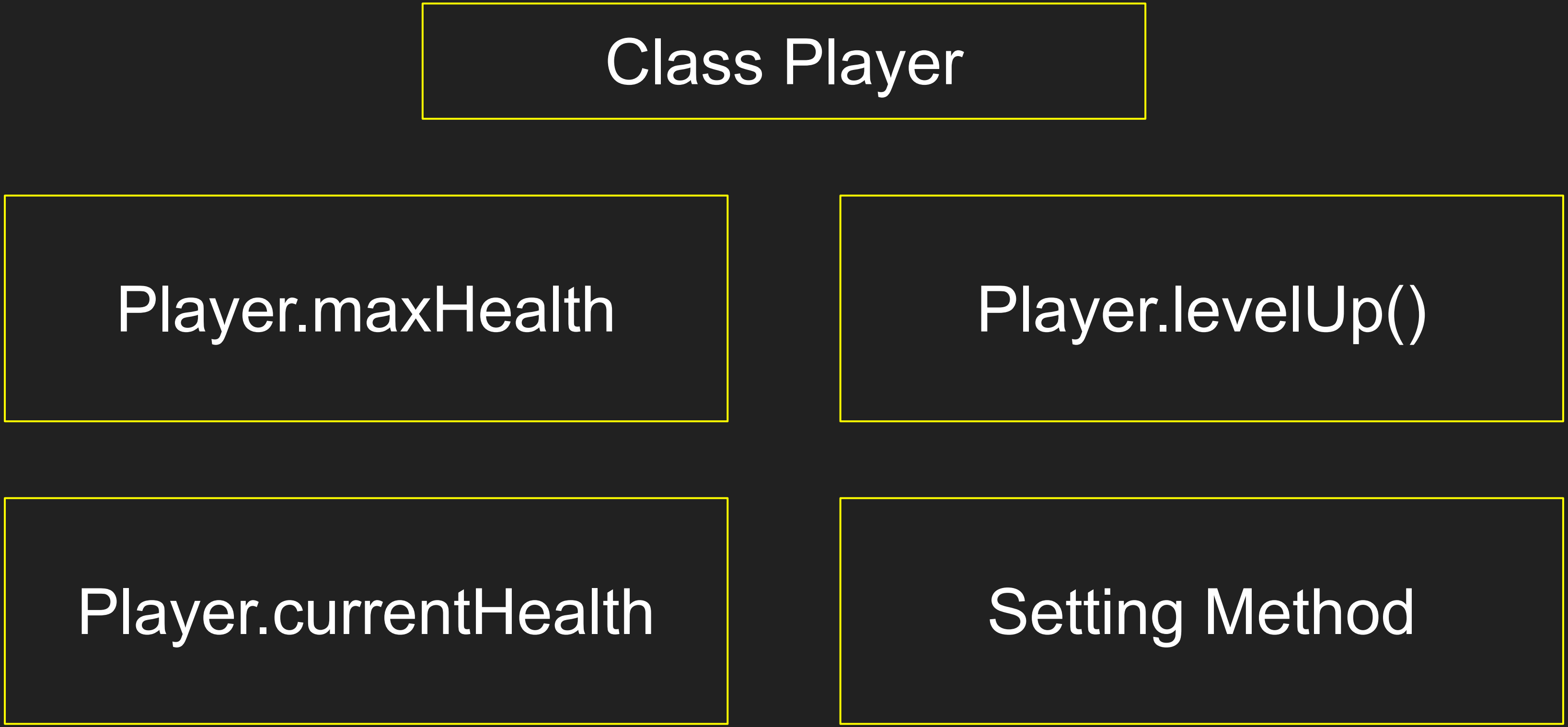
Encapsulation - Introduction

Setting methods also allow the programmer to easily keep track of attributes that depend on one another

Method 1

Method 2

Encapsulation - Game Example



Class Player

Player.maxHealth

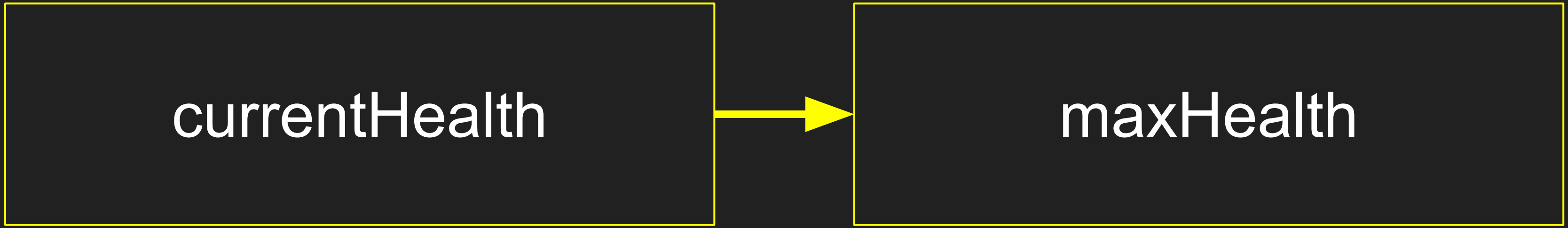
Player.levelUp()

Player.currentHealth

Setting Method

Encapsulation - Game Example

The setting method allows both attributes to be changed as they should, rather than requiring you to individually change them



Encapsulation - Game Example

The below example ensures that the change to the attribute is within the bounds of what is allowed

```
def setCurrentHealth(newHealth):
	player.currentHealth = newHealth
	if player.currentHealth > player.maxHealth:
		player.currentHealth = player.maxHealth
```

Encapsulation - Methods

You may also want some attributes to be “read only” from the outside

- To do this, you would define a getter method but not a setter method
- The variable could only be referenced, not changed

Encapsulation - Game Example

Class Piece

Piece.rank

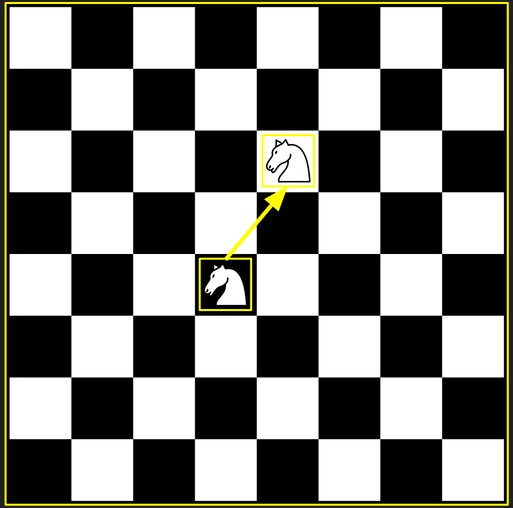
Player.file

Encapsulation - Game Example

Class Piece

Piece.rank

Player.file



Encapsulation - Game Example

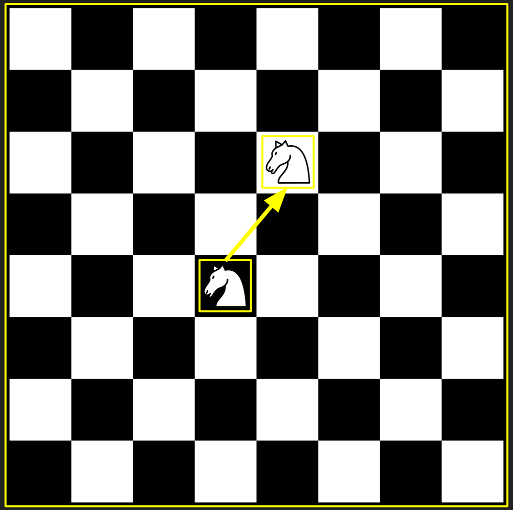
Class Piece

Piece.rank

=newRank

Player.file

=newFile



Encapsulation - Game Example

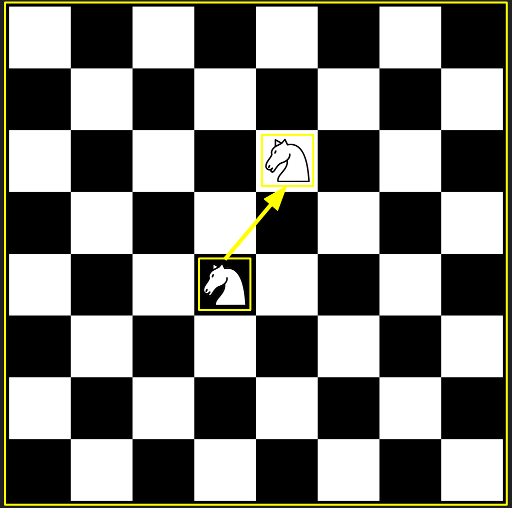
Class Piece

Piece.rank

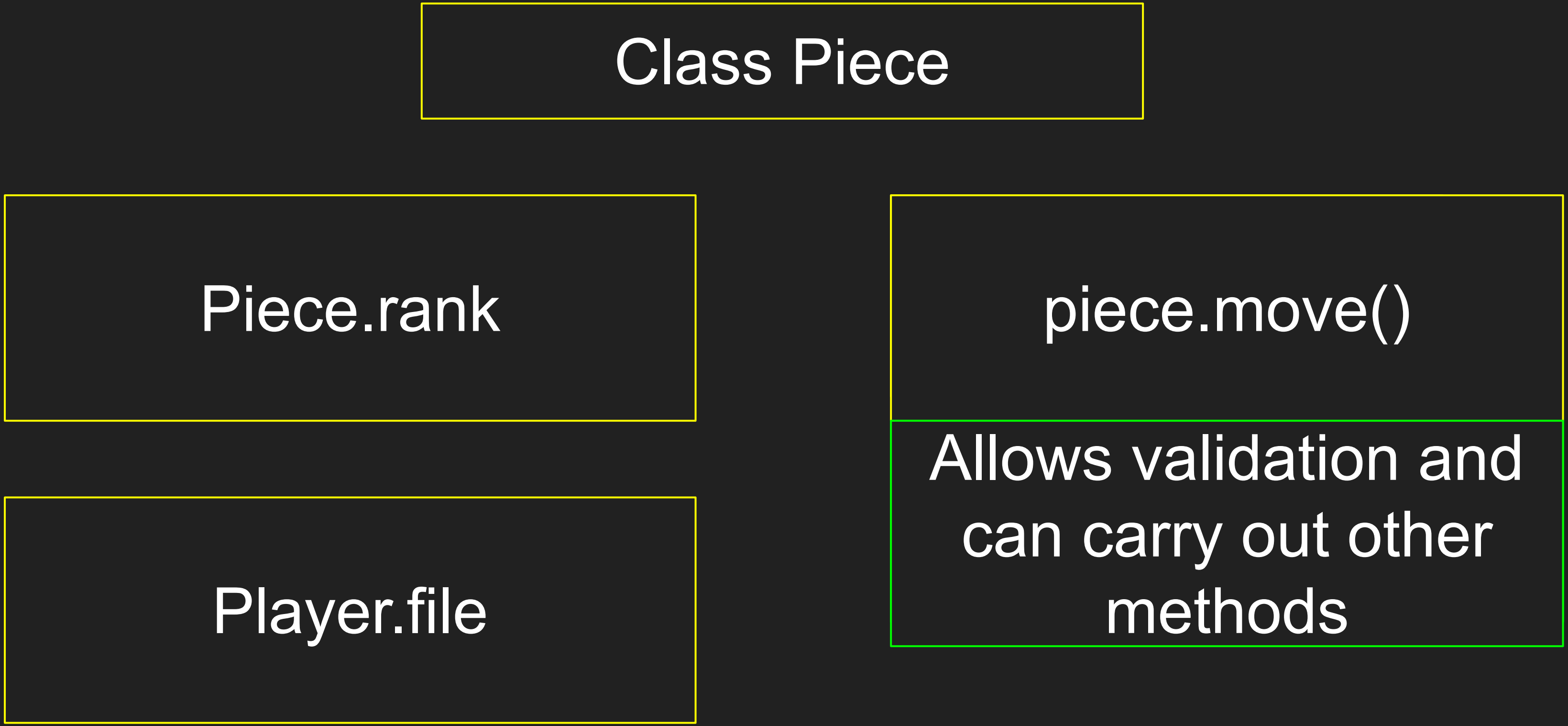
=n nk

Player.file

=newFile



Encapsulation - Game Example



Class Piece

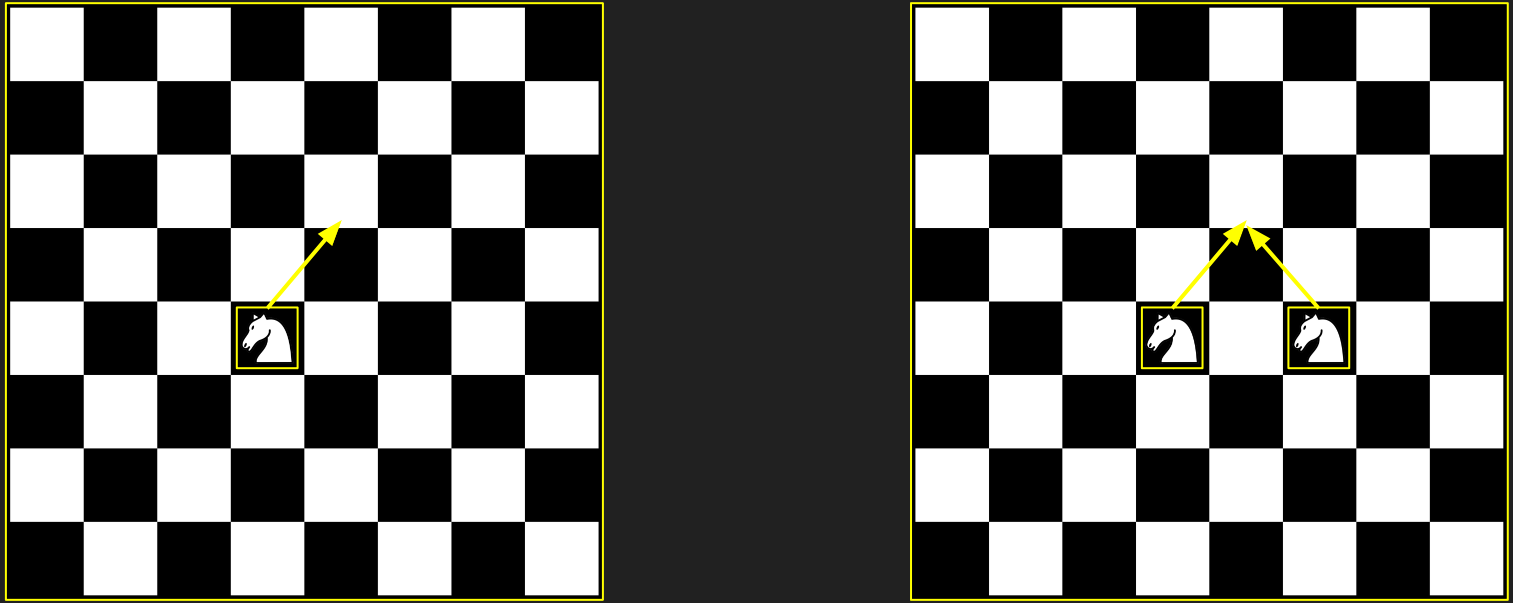
Piece.rank

Player.file

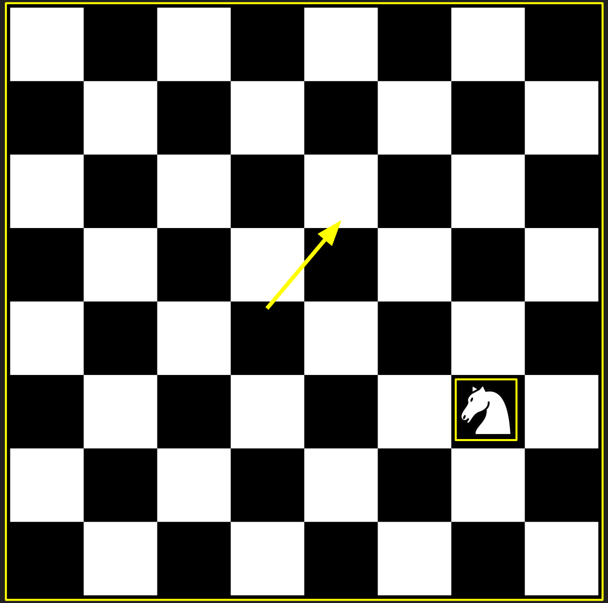
piece.move()

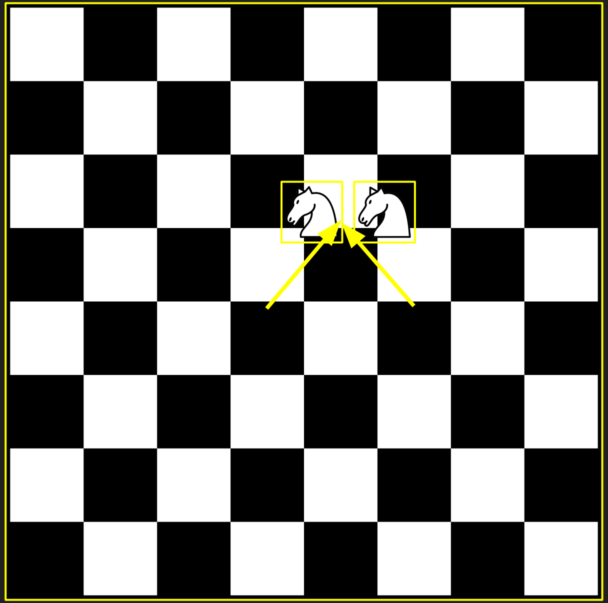
Allows validation and can carry out other methods

Encapsulation - Game Example



Encapsulation - Game Example





Encapsulation - Information Hiding

It’s generally best to not allow external classes to directly edit an object’s attributes

This is very important when working on large and complex programs

Each piece should not have access to or rely on the inner working of other sections of code

Encapsulation - Information Hiding

This idea will be revisited very soon

Information hiding, or keeping the data of one class hidden from external classes, helps you keep control of your program and prevent it from becoming too complicated

Encapsulation - Overview

Encapsulation is a vital principle in Object Oriented Programming

Encapsulation:

- Keeps the programmer in control of access of data
- Prevents the program from ending up in any strange or unwanted states

Abstraction - Introduction

This segment we will be looking at the next of the four principles of object - oriented programming: Abstraction

Abstraction refers to only showing essential details and keeping everything else hidden

Abstraction - Car Example

How the steering wheel steers the car

How the gas and brake pedals affect the car

How much gas your car has

Important to most people

Abstraction - Car Example

How exactly the car functions internally

As long as you understand the outcome, the process is not very important to you

Important to most people

Abstraction - Explanation

This idea extends to object-oriented programming

The classes you create should act like your car. Users of your classes should not worry about the inner details of those classes

Abstraction - Explanation

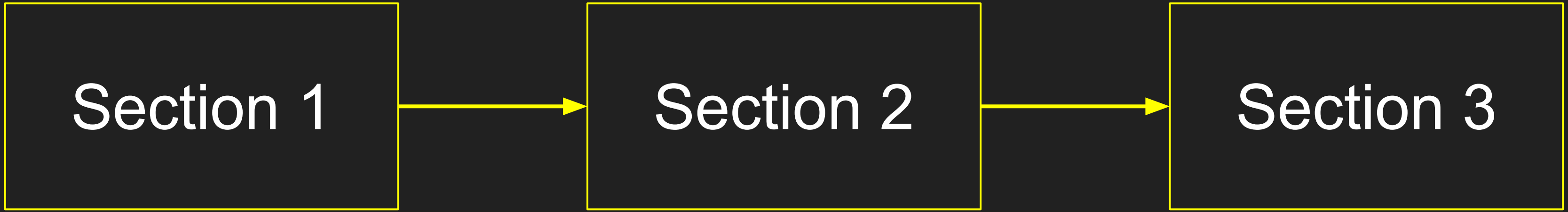
This is similar to encapsulation that was discussed last video

Classes should not directly interact with other classes' data

This is very important when working on your program incrementally

Abstraction - Explanation

This is very important when working on your program incrementally



Section 1

Section 2

Section 3

Abstraction - Interface and Implementation

Interface

Implementation

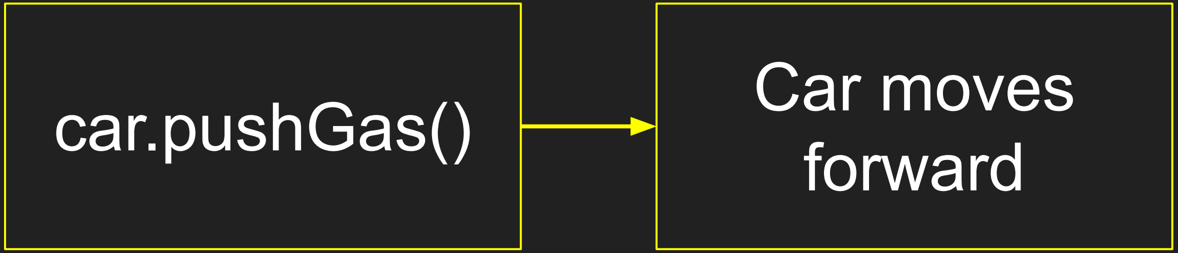
Abstraction - Interface and Implementation

The interface refers to the way sections of code can communicate with one another

This typically is done through methods that each class is able to access

Abstraction - Interface and Implementation

The implementation of these methods, or how these methods are coded, should be hidden

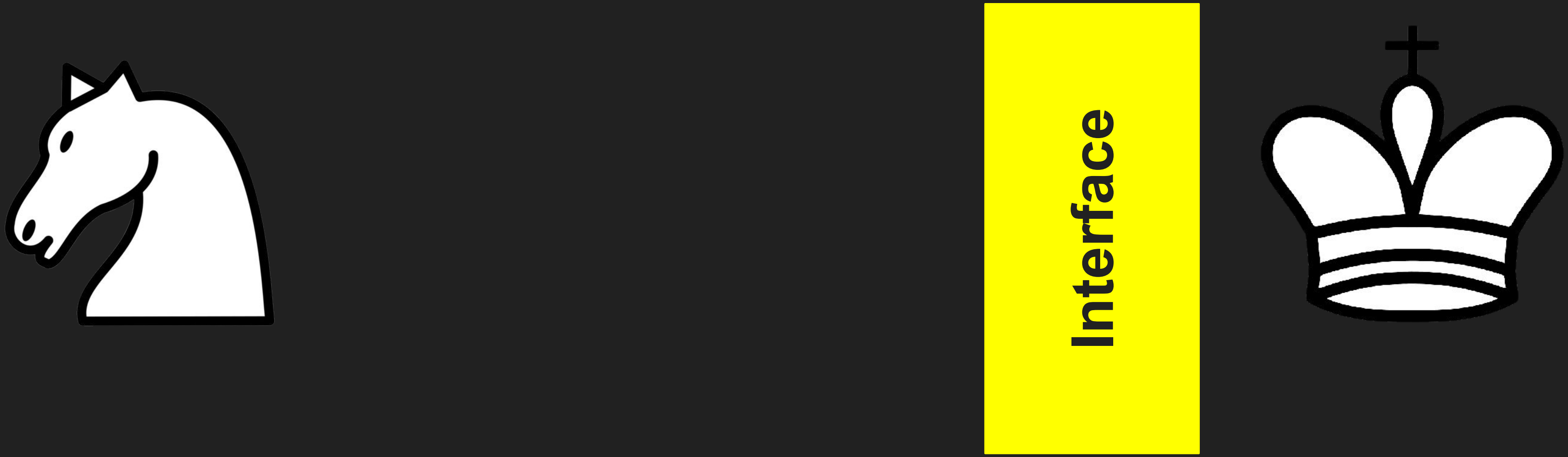


car.pushGas() → Car moves forward

The “how” is not important

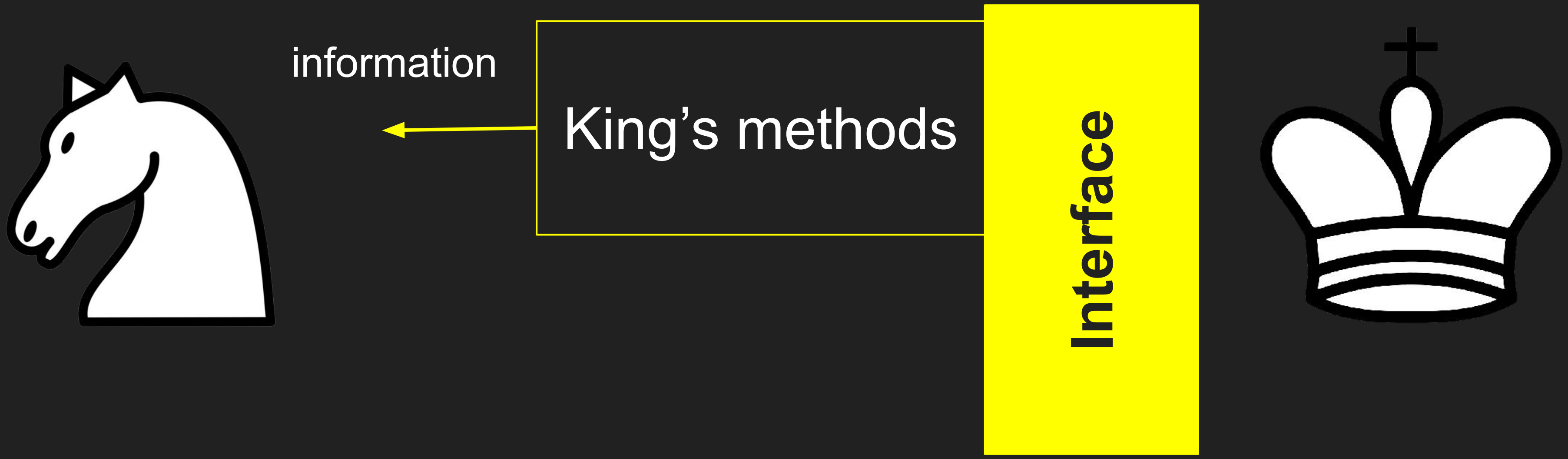
Abstraction - Chess Example

Abstraction - Chess Example



Interface

Abstraction - Chess Example

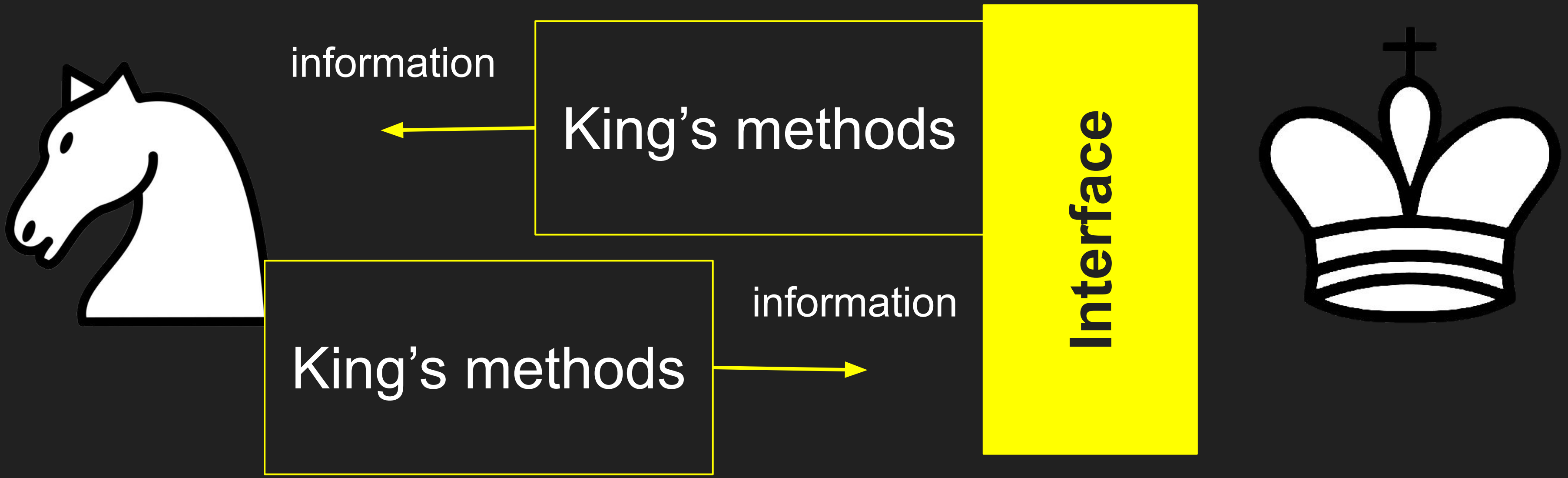


information

King’s methods

Interface

Abstraction - Chess Example



information

King's methods

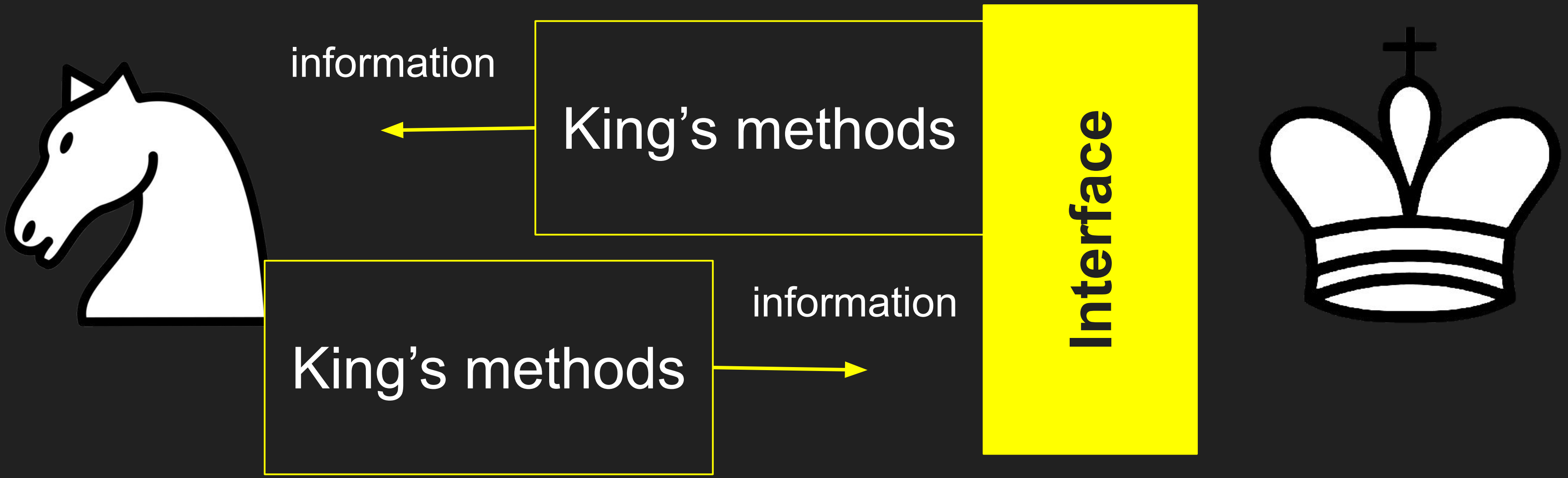
King's methods

information

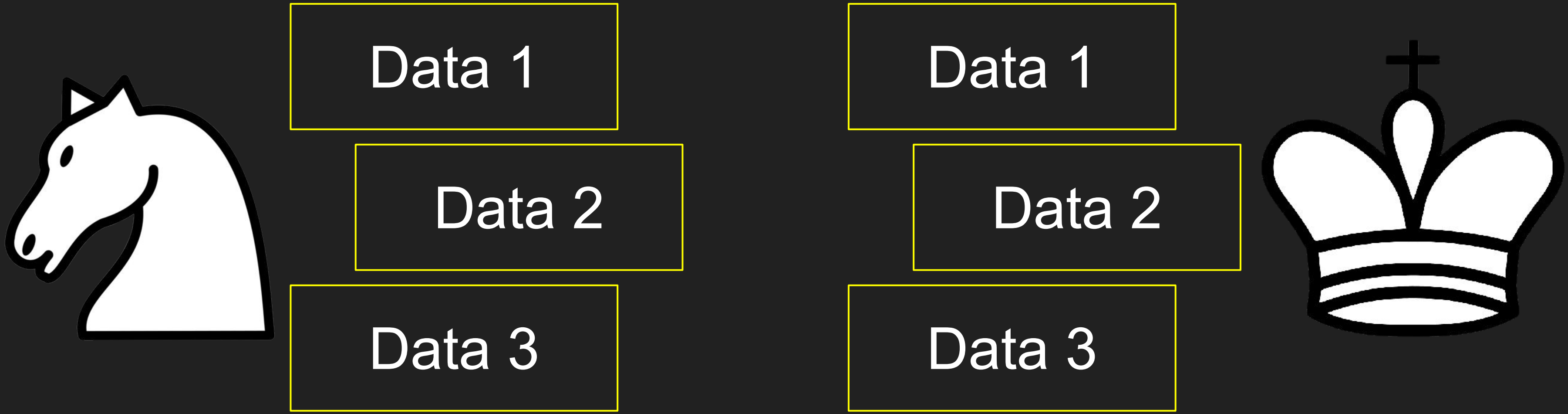
Interface

Abstraction - Chess Example

King’s implementation is not important to you



Abstraction - Interface and Implementation



Data 1

Data 2

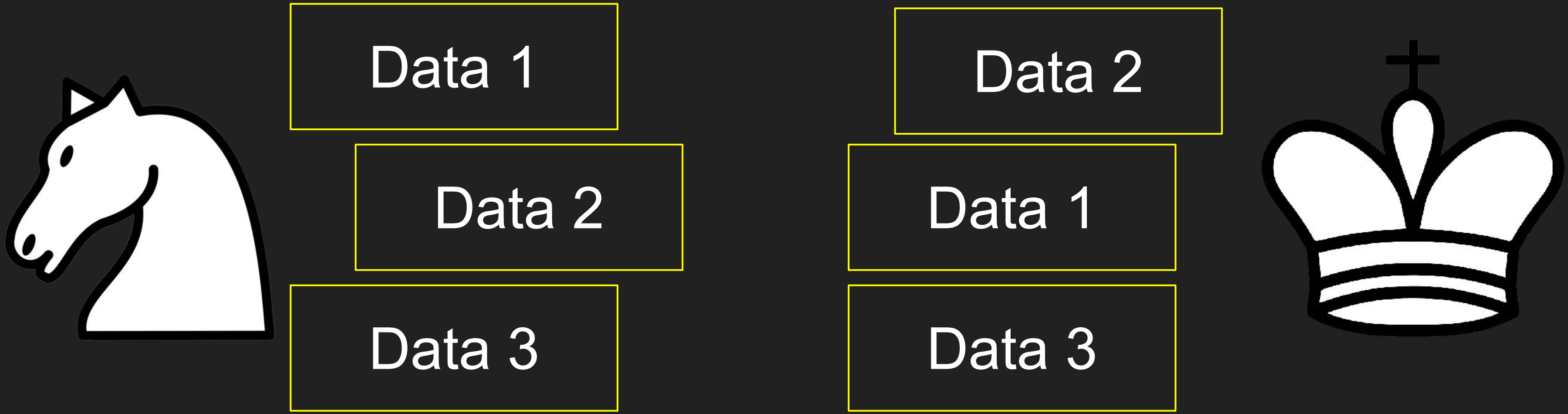
Data 3

Data 1

Data 2

Data 3

Abstraction - Interface and Implementation



Data 1

Data 2

Data 3

Data 2

Data 1

Data 3

Abstraction - Interface and Implementation

Data
a 2
Data
Data 2
ta 1
ata 3

Abstraction - Interface and Implementation

If classes are entangled, then one change creates a ripple effect that causes many more changes

Creating an interface through which classes can interact ensures that each piece can be individually developed

Abstraction - Overview

Abstraction allows the program to be worked on incrementally and prevents it from becoming entangled and complex

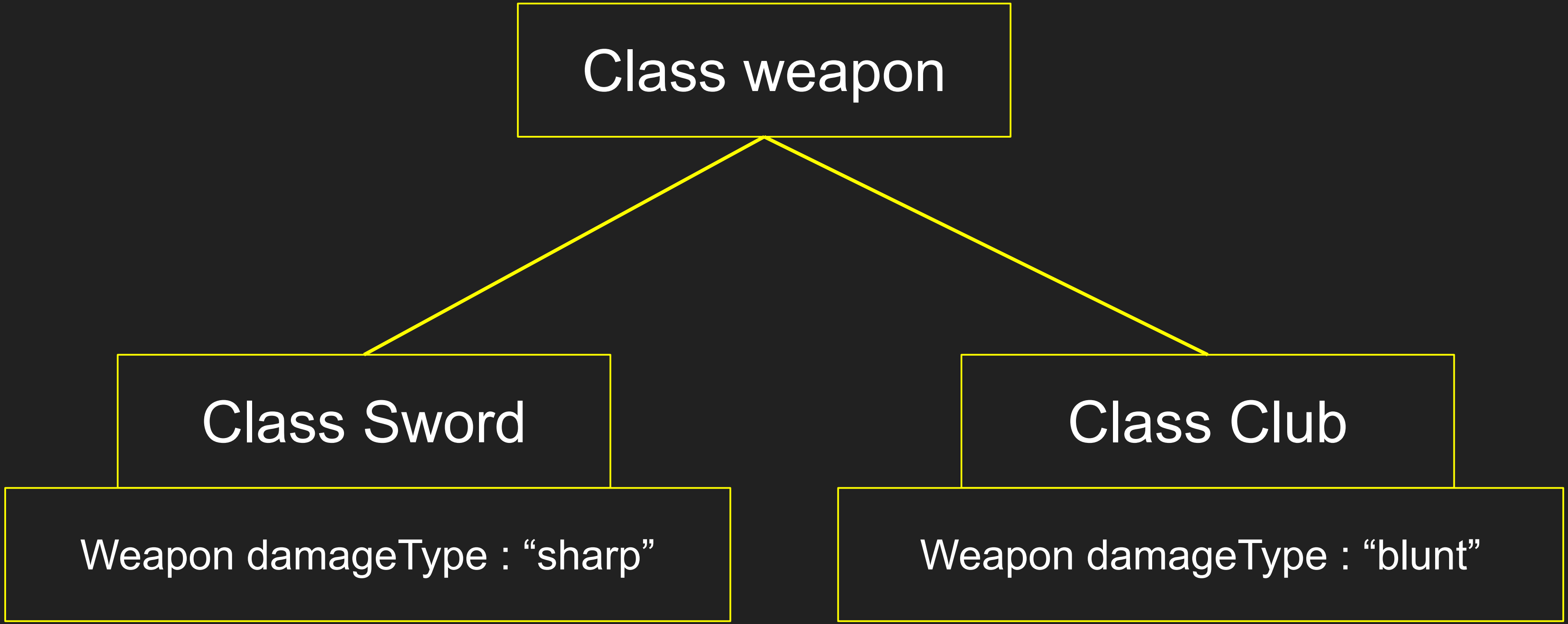
Determine specific points of contact that can act as an interface between classes, and only worry about the implementation when coding it

Inheritance - Introduction

This segment we will be looking at the next of the four main principles of object - oriented programming: Inheritance

Inheritance is the principle that allows classes to derive from other classes.

Inheritance - Game Example



Class weapon

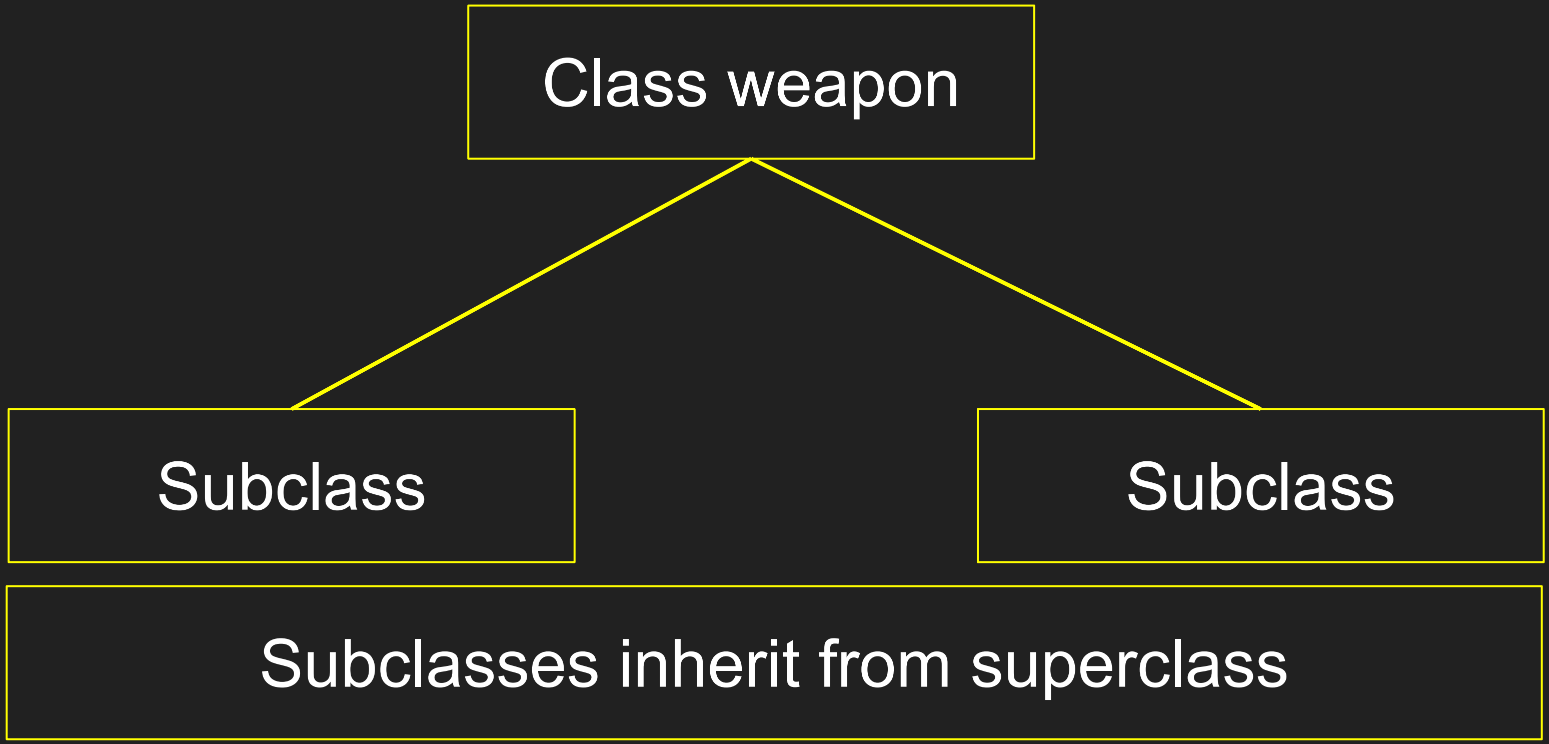
Class Sword

Weapon damageType : “sharp”

Class Club

Weapon damageType : “blunt”

Inheritance - Game Example



Class weapon

Subclass

Subclass

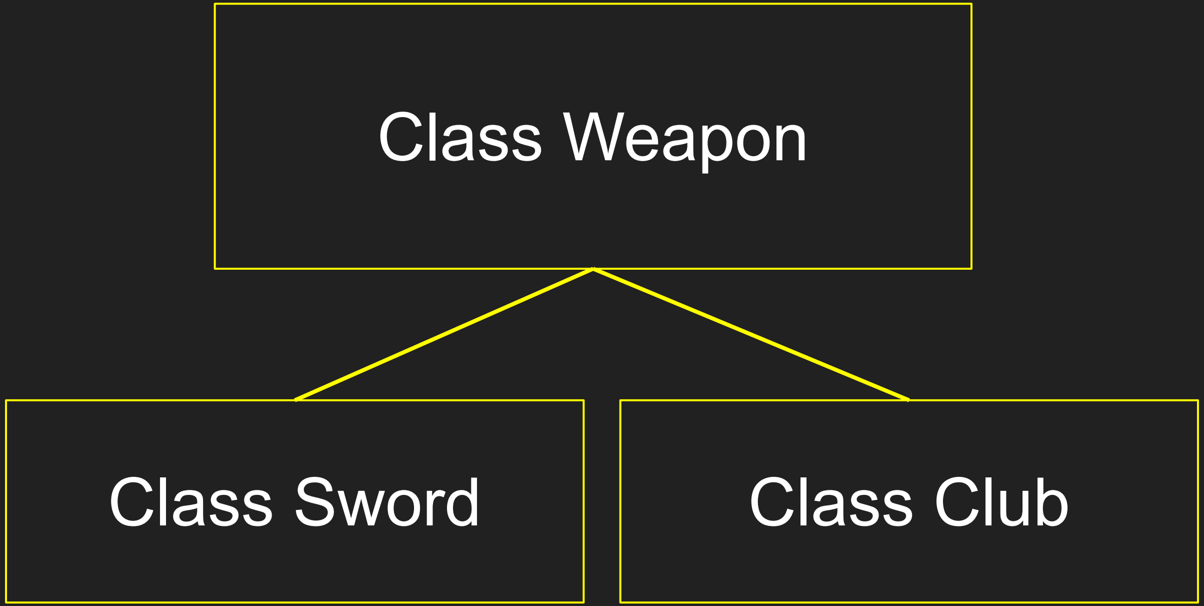
Subclasses inherit from superclass

Inheritance - Game Example

Any sword or club would require the methods and attributes present in the weapon class in order to function

In most cases the class hierarchy you create will have many more layers with many more classes in each layer

Inheritance - class Hierarchy

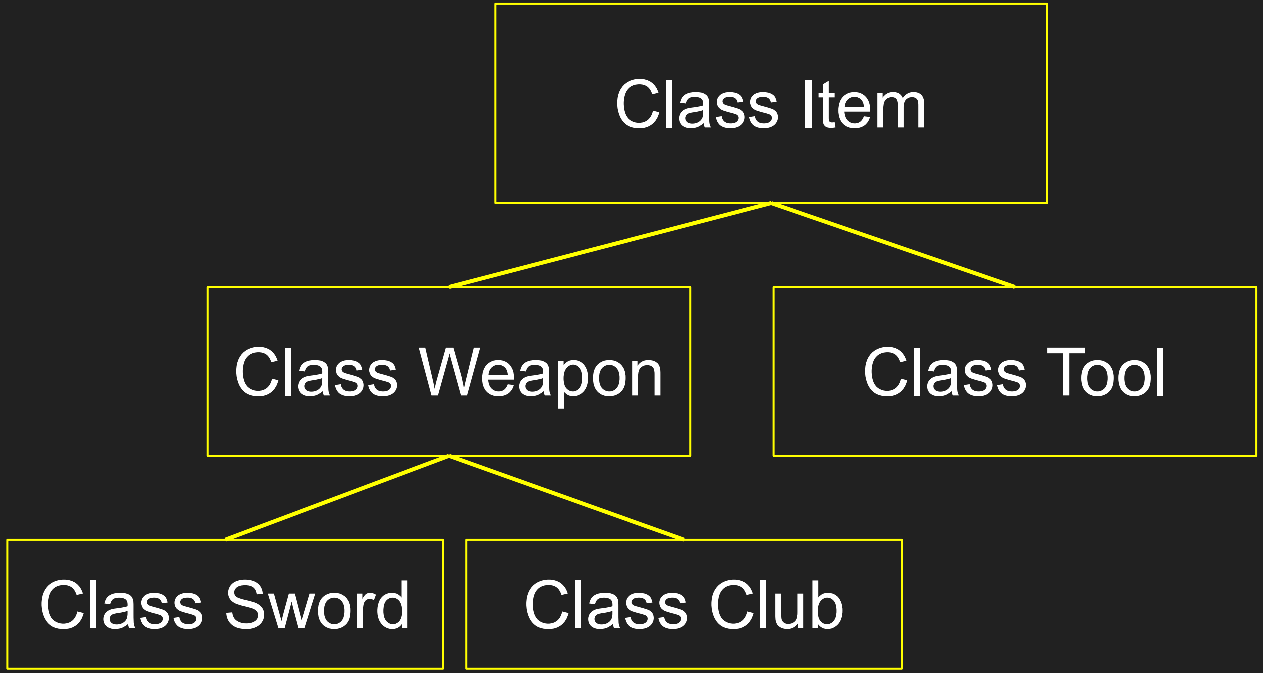


Class Weapon

Class Sword

Class Club

Inheritance - Class Hierarchy


Class Item

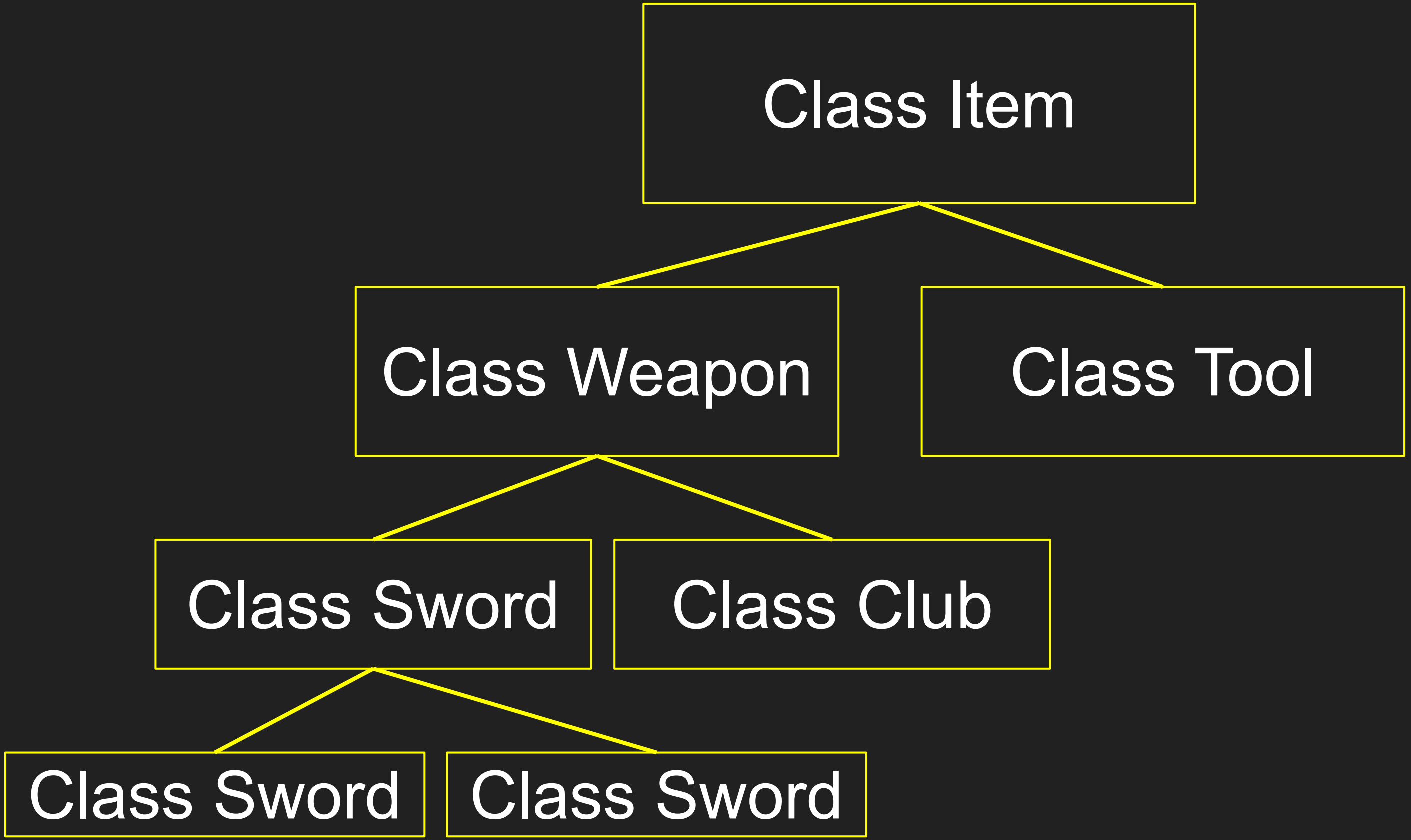
Class Weapon

Class Tool

Class Sword

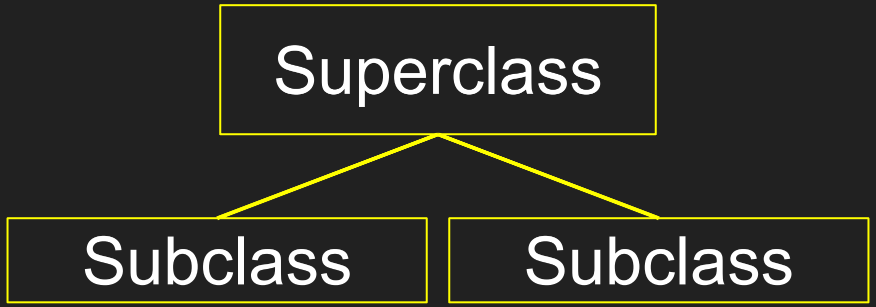
Class Club

Inheritance - Class Hierarchy



Inheritance - Class Hierarchy

The class hierarchy acts as a web of classes with different relationships to one another



Inheritance - Access Modifiers

Access modifiers change which classes have access to other classes, methods, or attributes

The three main access modifiers we will be covering are:

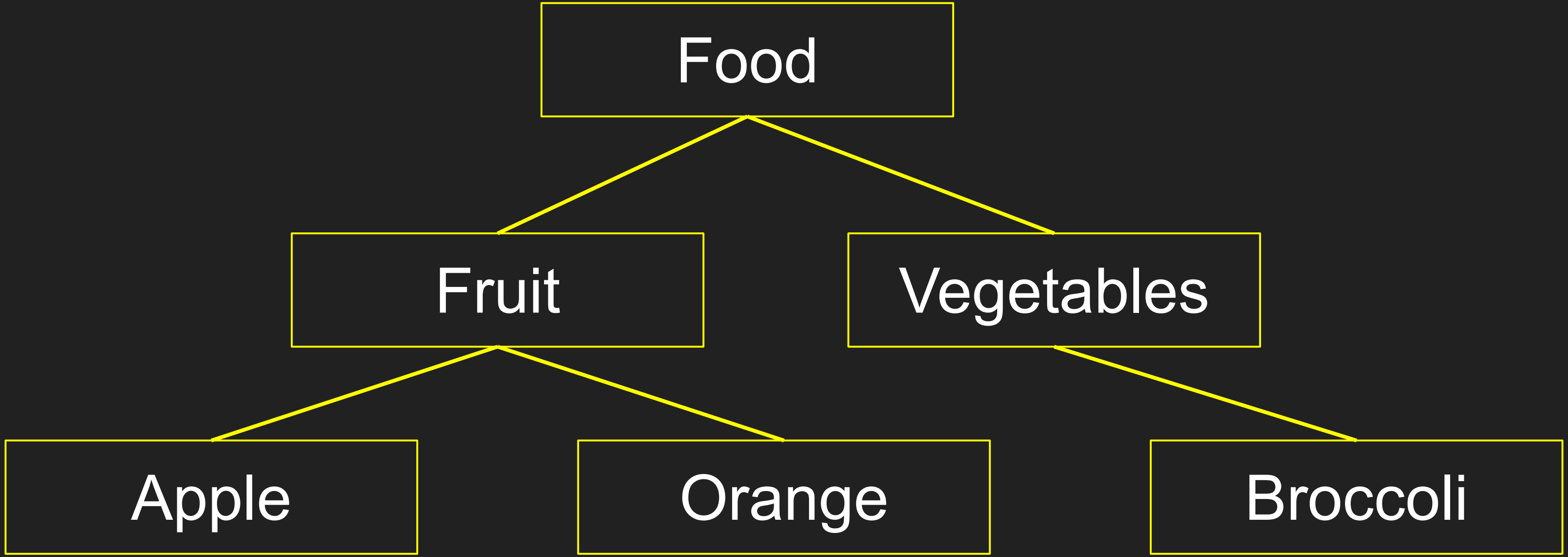
- Public
- Private
- Protected

Inheritance - Access Modifiers

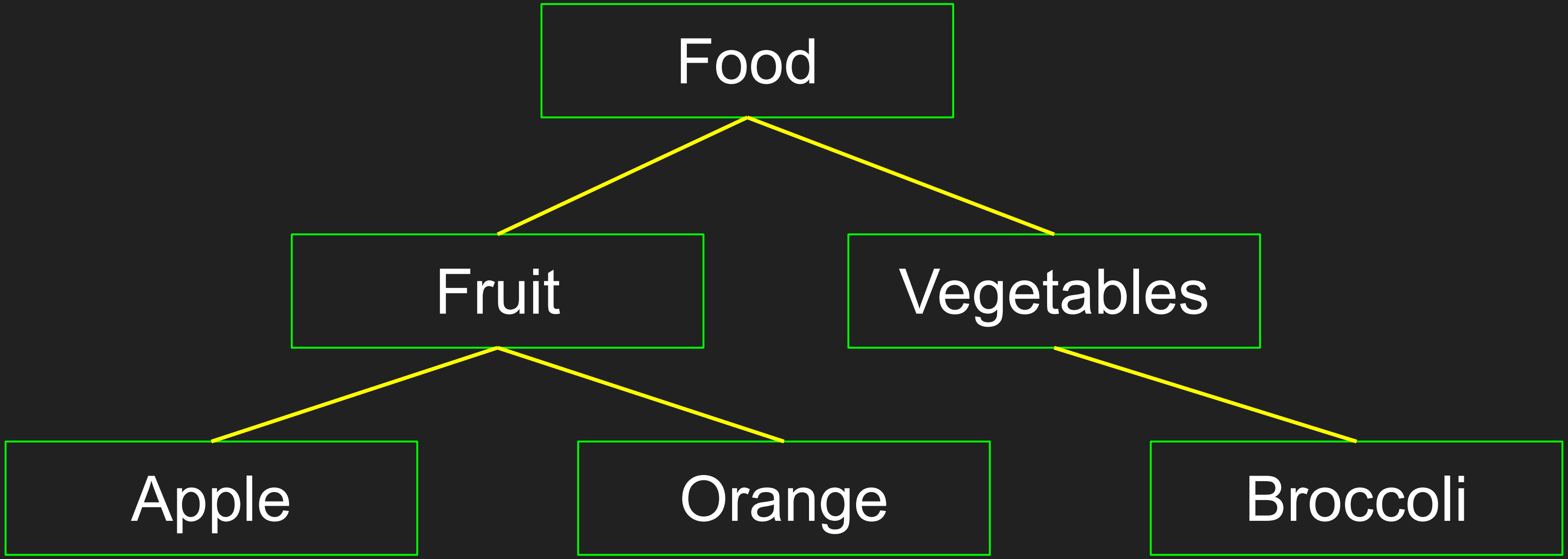
Public members can be accessed from anywhere in your program

This includes anywhere both inside of the class hierarchy it is defined as well as outside in the rest of the program

Inheritance - Access Modifiers



Inheritance - Access Modifiers

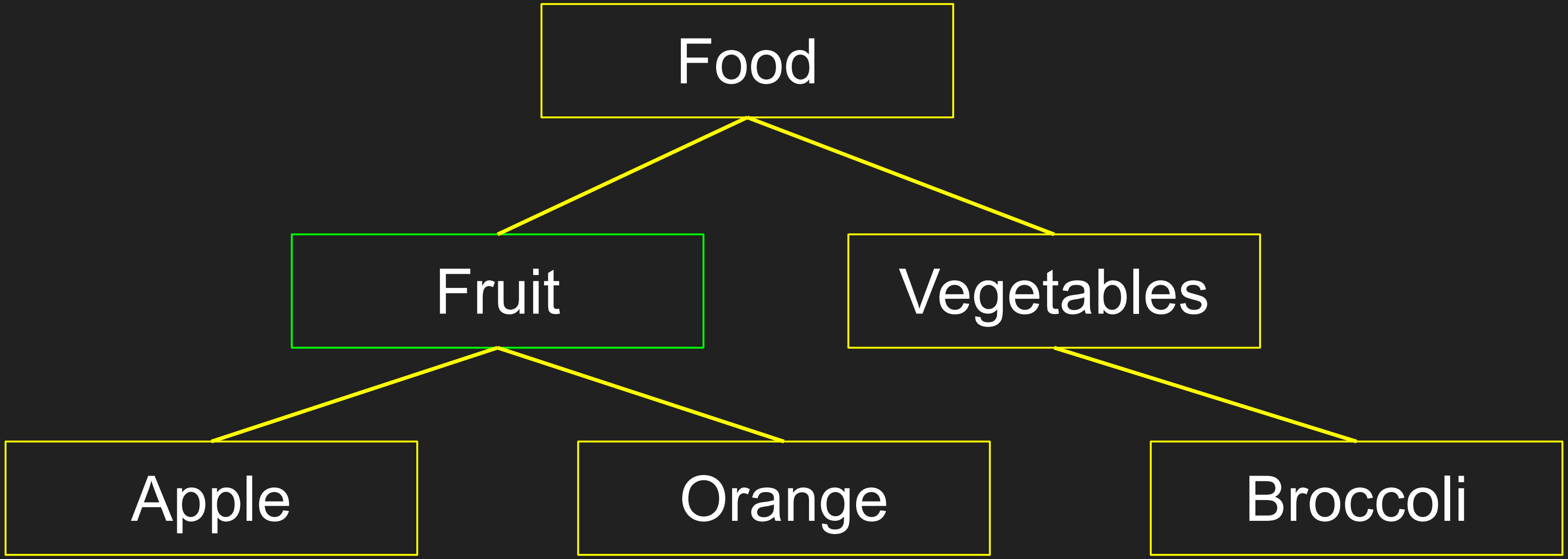


Inheritance - Access Modifiers

Private members can only be accessed from within the same class that the member is defined

This allows you to create multiple private members of the same name in different locations so that they do not conflict with one another

Inheritance - Access Modifiers

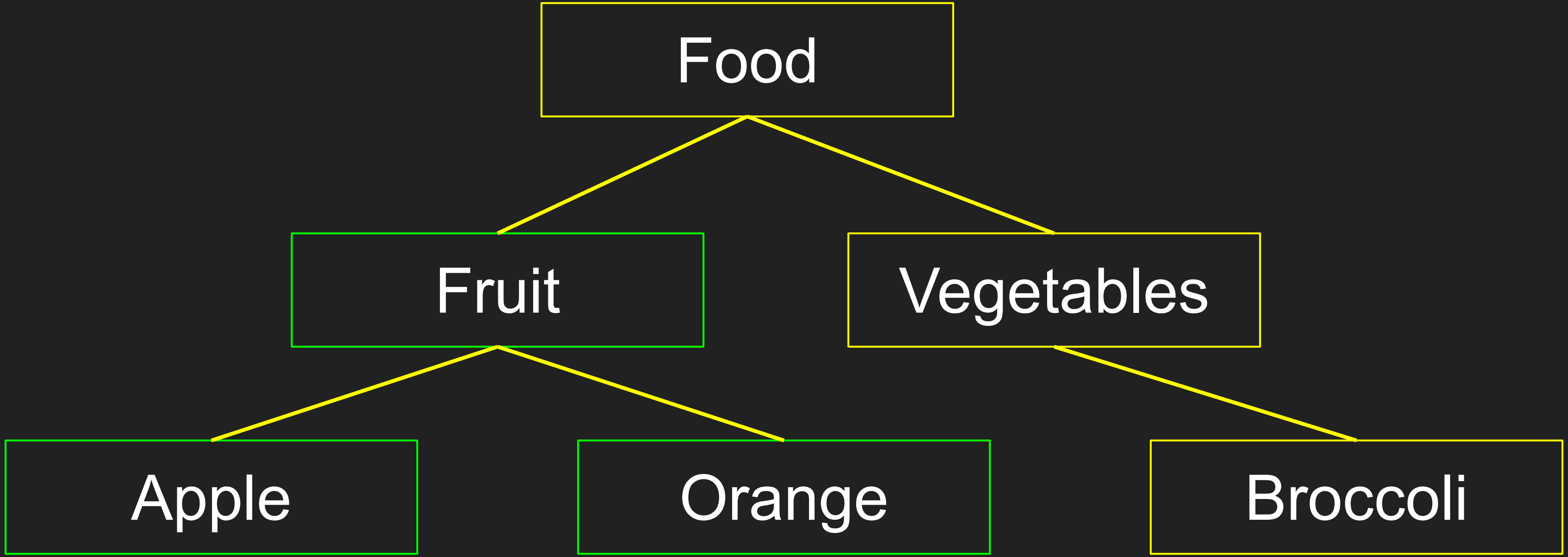


Inheritance - Access Modifiers

Private members can only be accessed from within the class it is defined, as well as any subclasses on that class.

This essentially makes protected members private to the hierarchy in which they are defined

Inheritance - Access Modifiers



Polymorphism - Introduction

Polymorphism describes methods that are able to take on many forms

There are two types of polymorphism

The first type is known as dynamic polymorphism

Polymorphism - Introduction

Dynamic polymorphism occurs during the runtime of the program

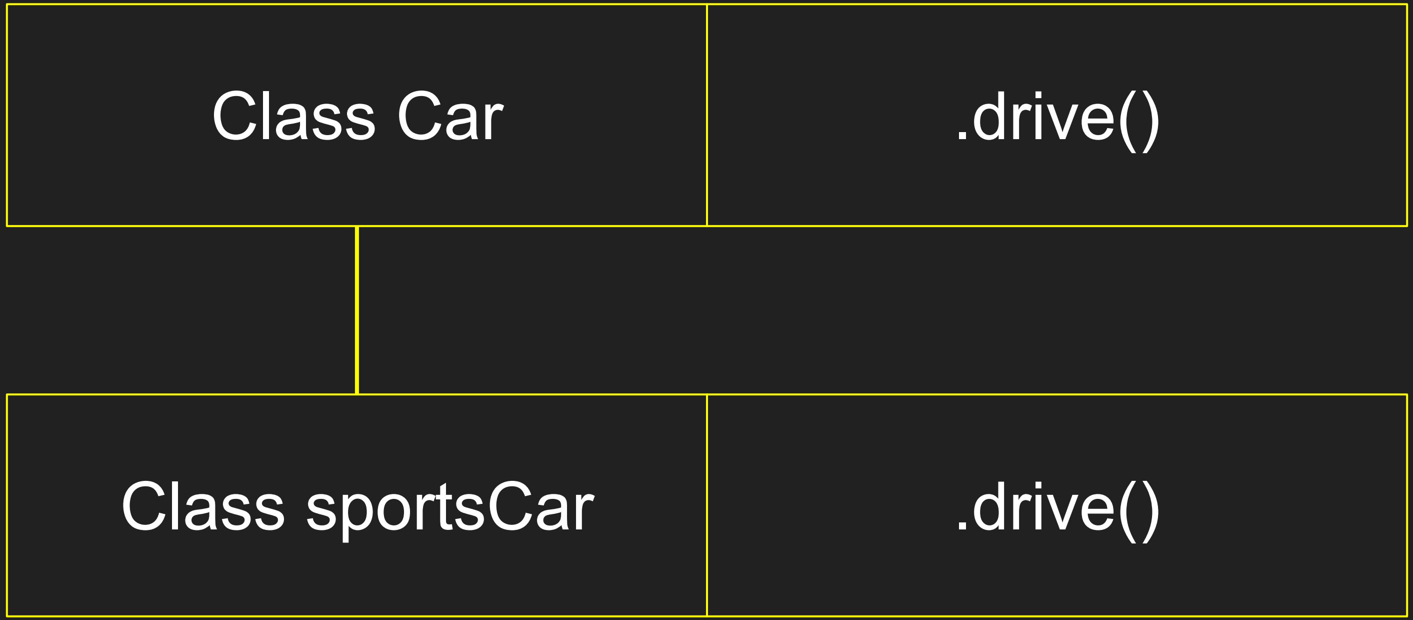
This type of polymorphism describe when a method signature is in both a subclass and a superclass

Polymorphism - Introduction

The methods share the same name but have different implementation

The implementation of the subclass that the object is an instance of overrides that of the superclass

Polymorphism - Car Example



Class Car

.drive()

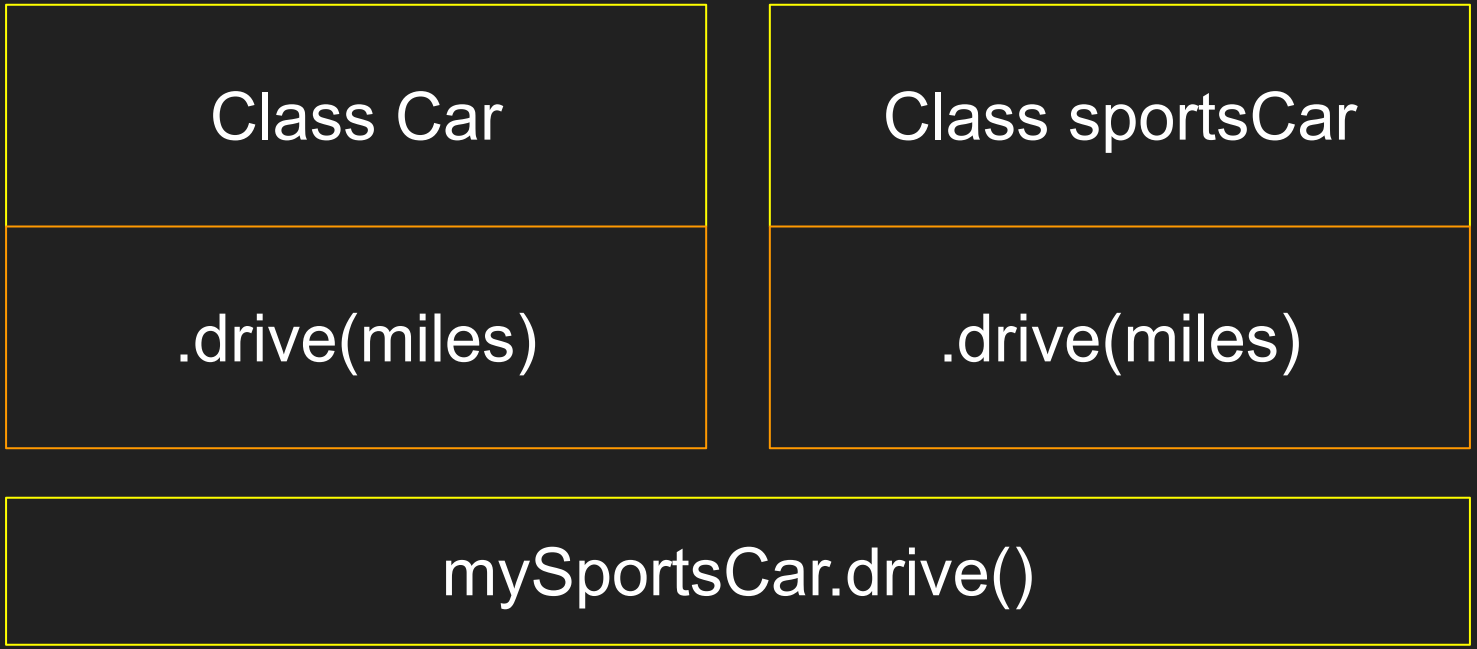
Class sportsCar

.drive()

Polymorphism - Car Example

Class Car	Class sportsCar
.drive(miles)	.drive(miles)
{ Car.gas -= 0.04 * miles }	{ Car.gas -= 0.02 * miles }

Polymorphism - Car Example



Class Car

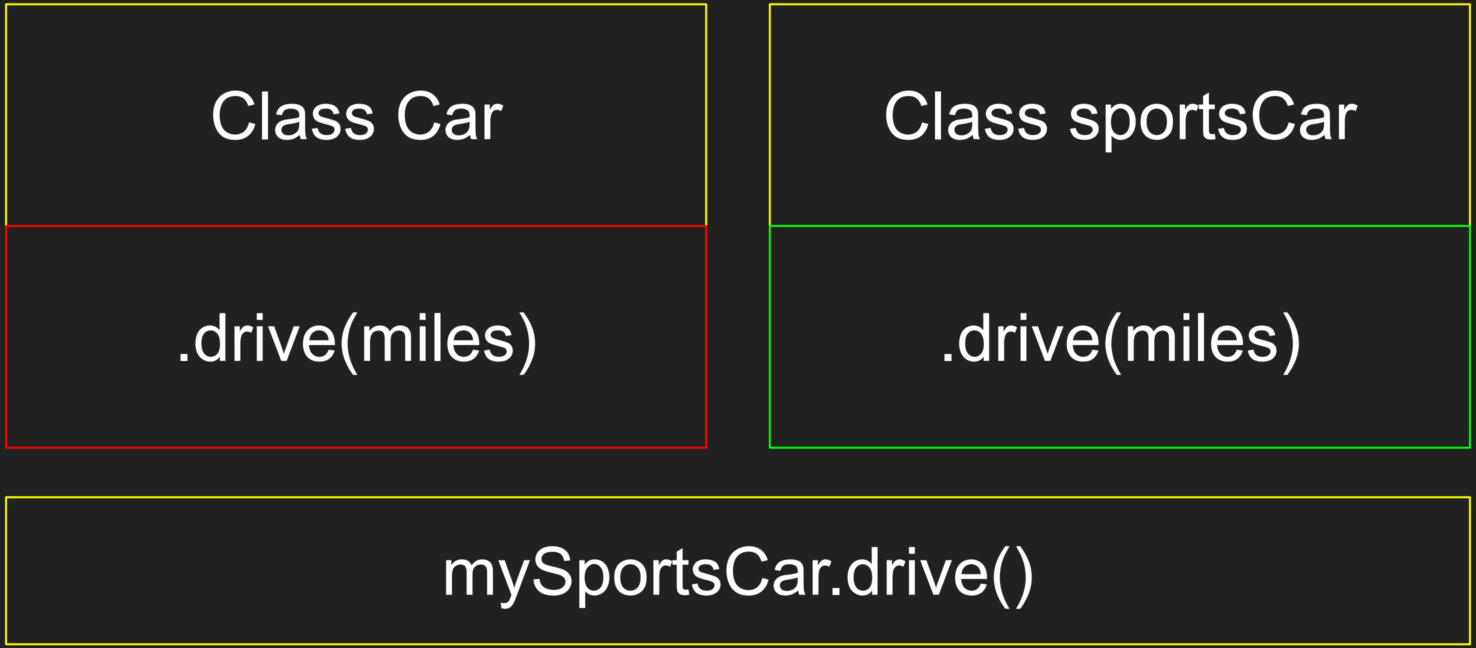
.drive(miles)

Class sportsCar

.drive(miles)

mySportsCar.drive()

Polymorphism - Car Example



Class Car

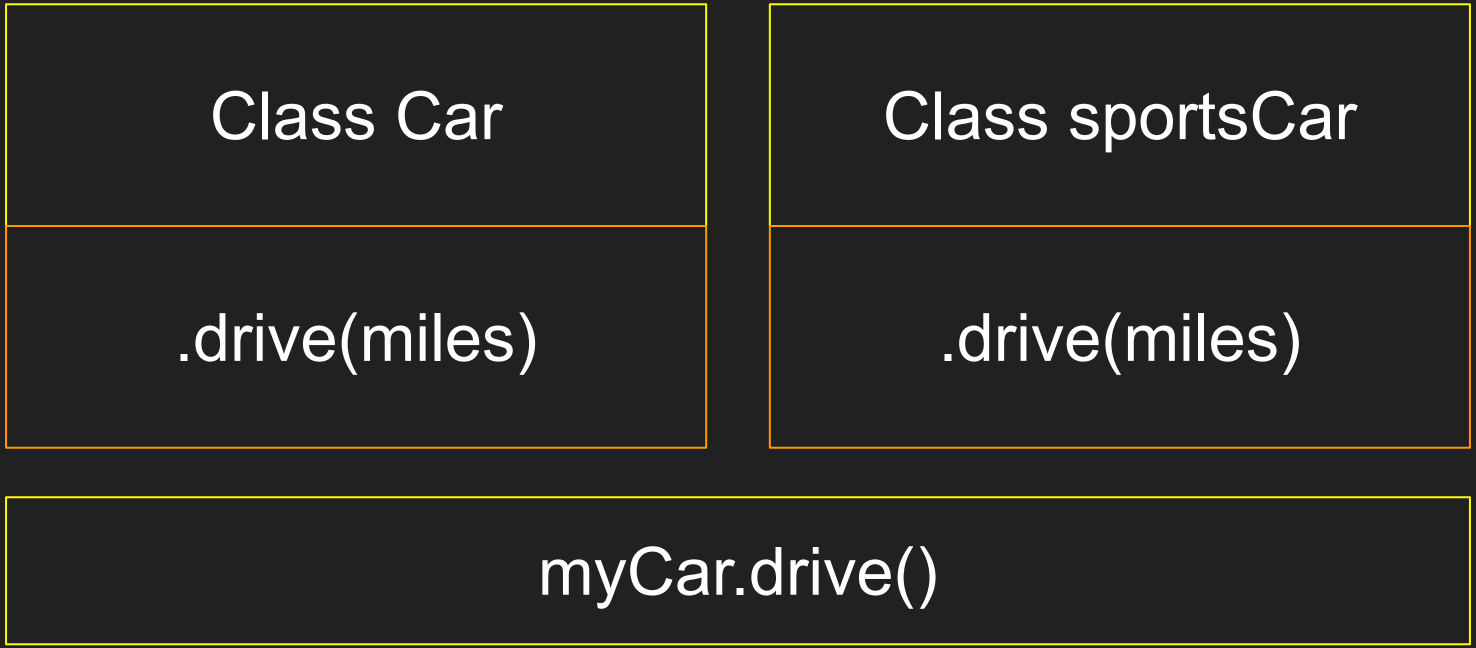
.drive(miles)

Class sportsCar

.drive(miles)

mySportsCar.drive()

Polymorphism - Car Example



Class Car

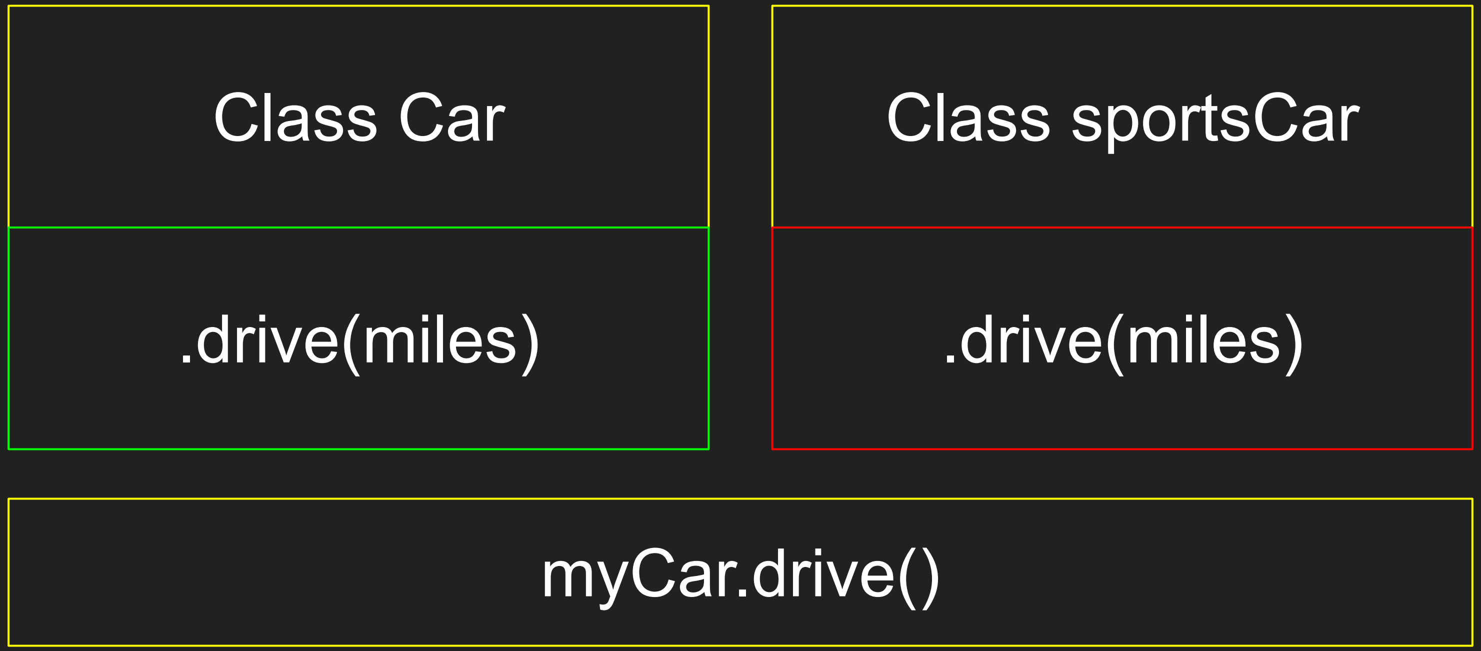
.drive(miles)

Class sportsCar

.drive(miles)

myCar.drive()

Polymorphism - Car Example



Class Car

.drive(miles)

Class sportsCar

.drive(miles)

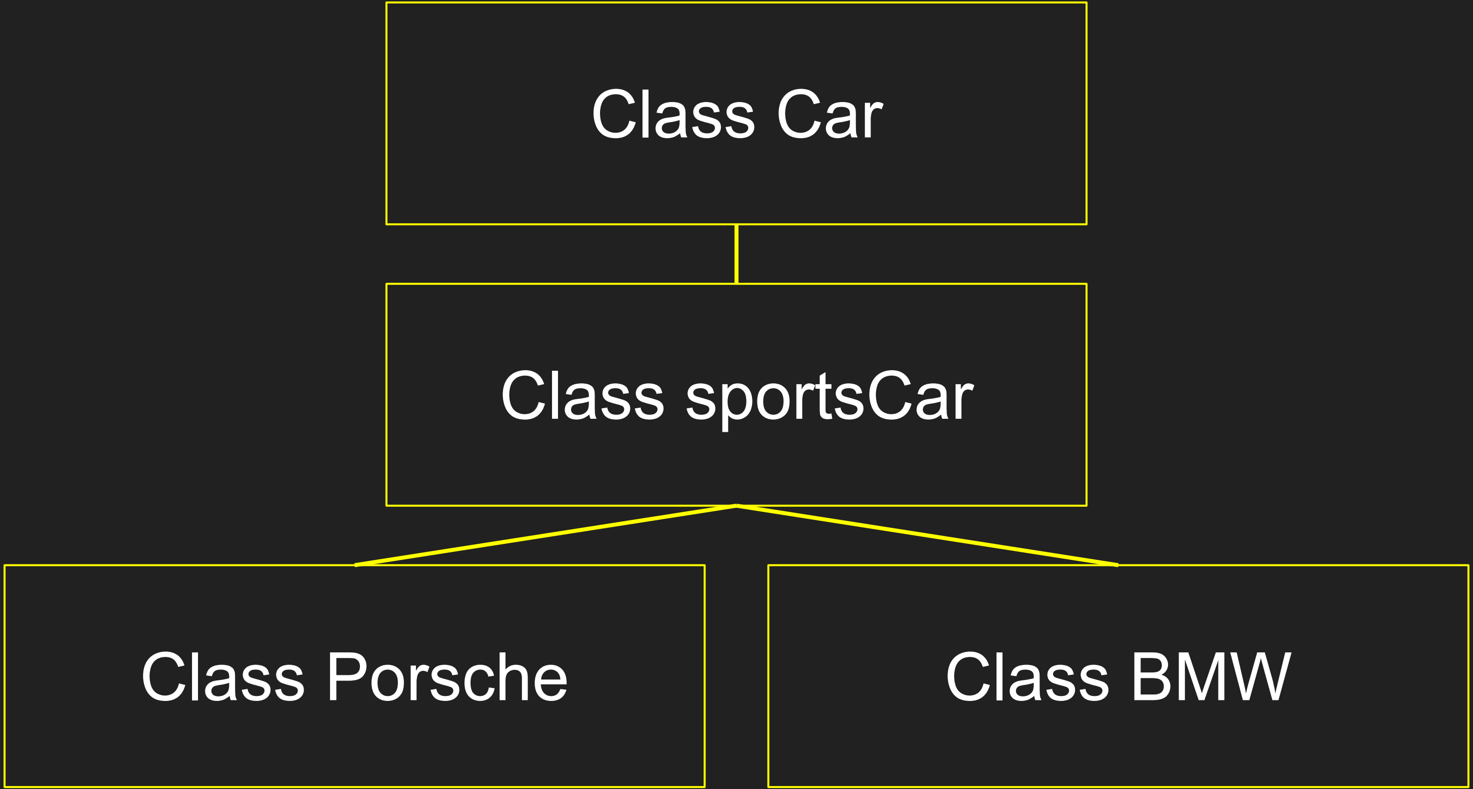
myCar.drive()

Polymorphism - Dynamic

This works because the form of the method is decided based on where in the class hierarchy it is called

The implementation of method signature that will be used is determined dynamically as the program is run

Polymorphism - Car Example



Class Car

Class sportsCar

Class Porsche

Class BMW

Polymorphism - Static

Static polymorphism occurs during compile-time rather than during runtime

This refers to when multiple methods with the same name but different arguments are defined in the same class

Polymorphism - Static

Ways to differentiate methods of the same name:

Polymorphism - Static

Ways to differentiate methods of the same name:

Different number of parameters

Polymorphism - Static

Ways to differentiate methods of the same name:

Different number of parameters

Different types of parameters

Polymorphism - Static

Ways to differentiate methods of the same name:

Different number of parameters

Different types of parameters

Different order of parameters

Polymorphism - Static

This is known as method overloading

Despite the methods having the same name, their signatures are different due to their different arguments

Polymorphism - Car Example

Class Car

1	.drive(int spd, string dest)
2	.drive(int spd, int dist)
3	.drive(string dest, int spd)

```
myCar.drive( 45, “Work”
```

Polymorphism - Car Example

Class Car

1	.drive(int spd, string dest)
2	.drive(int spd, int dist)
3	.drive(string dest, int spd)

myCar.drive(45, “Work”

Polymorphism - Car Example

Class Car

1	.drive(int spd, string dest)
2	.drive(int spd, int dist)
3	.drive(string dest, int spd)

myCar.drive(45, “Work”

Polymorphism - Car Example

Class Car

1	.drive(int spd, string dest)
2	.drive(int spd, int dist)
3	.drive(string dest, int spd)

myCar.drive(45, “Work”

Polymorphism - Car Example

Class Car

1	.drive(int spd, string dest)
2	.drive(int spd, int dist)
3	.drive(string dest, int spd)

```
myCar.drive( 45, “Work”
```

Polymorphism - Car Example

Class Car

1	.drive(int spd, string dest)
2	.drive(int spd, int dist)
3	.drive(string dest, int spd)

Polymorphism - Overview

Overall, polymorphism allows methods to take on many different forms

When utilizing polymorphism and method overloading, be sure that your are calling the correct form the method