Project ConsoleGames: Übersicht

EF IF

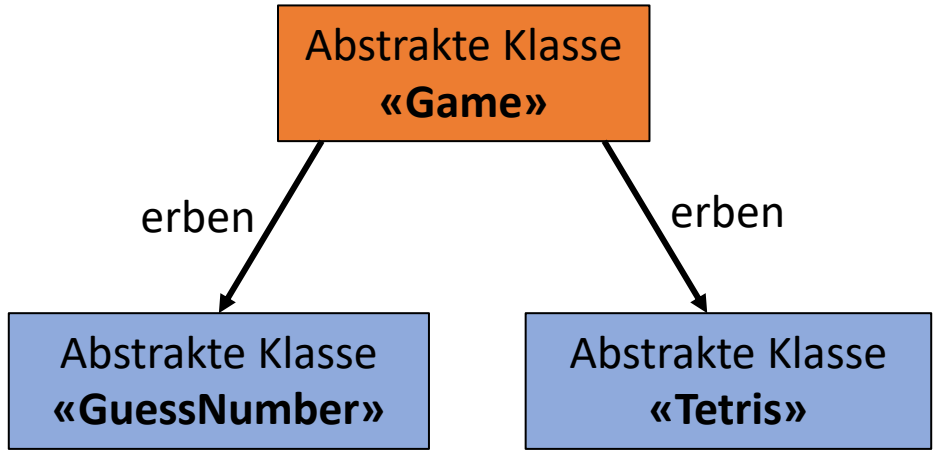
Ziel

- Ziele:
  - Sammlung mit Konsolen Games, alle (Gruppen) steuern eines bei.
  - Ein Programm, von welchem aus man die einzelnen Games starten kann.
  - Speicherung der High Scores.
- Probleme:
  - Wie führt man einzelne Games zusammen?
  - Wie stellt man sicher, dass Games einigermassen einheitlich sind?
- Antwort:
  - Objektorientiertes Programmieren!

OOP

- OOP: **O**bjekt**o**rientiertes **P**rogrammieren
- Grundidee: **Klassen** und **Objekte**
- Analogie: Häuser
- Die **Klasse** entspricht dem *Bauplan eines Hauses*.
- Wichtig: Nur weil es einen Plan von einem Haus gibt, gibt es noch lange kein Haus!
- **Objekt**: *konkretes Haus*, welches mit diesem Bauplan gebaut wurde:
  - Gewisse Dinge muss jedes Haus-Objekt haben, ansonsten ist es kein Haus (Dach, Haustüre,...)
  - Bei anderen Dingen können sich aber unterscheiden, z.B. Farbe, ...

OOP für ConsoleGames

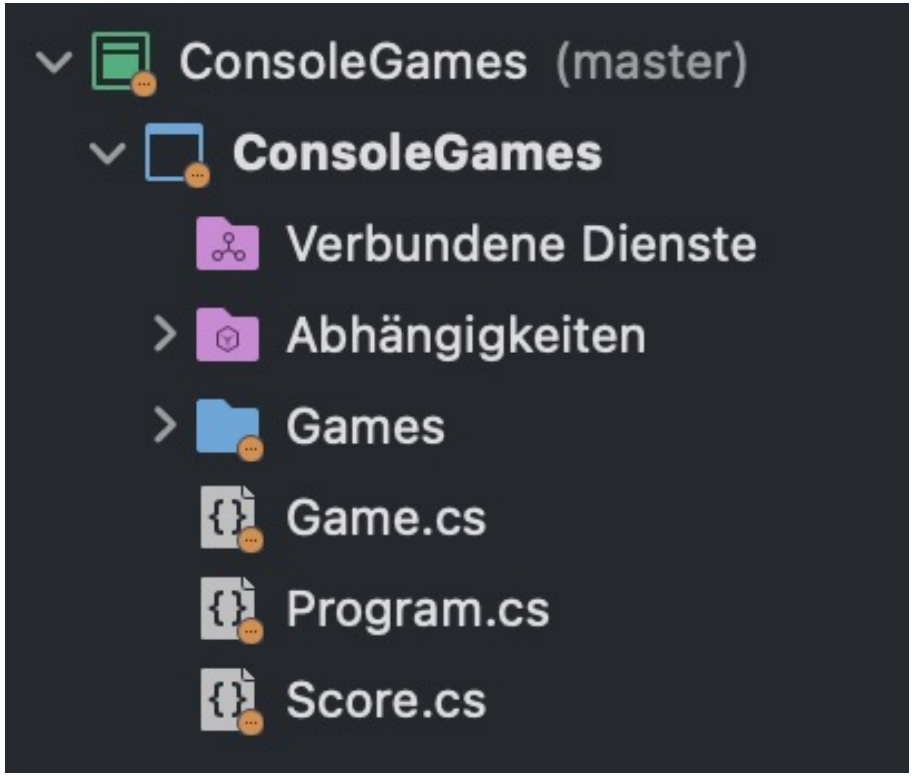


- Gibt **abstrakte Klasse** "Game":
  - Dient als Bauplan für Game, gibt vor, was ein Game alles beinhalten muss.
  - Diese ist von Framework vorgegeben.
- Erzeuge dann **Klassen** für jedes Game:
  - Jedes Game ist eine *eigene Klasse*, z.B. "GuessNumber“, …
  - … die von der abstrakten Klasse „Game“ erbt.
  - Diese gibt vor, welche Eigenschaften und Methoden zwingend implementiert werden müssen.
  - Ansonsten: Fehler!
  - So wird sichergestellt, dass alle Game-Objekte mit dem Framework kompatibel sind.
- **Objekte:** Im Hauptprogramm `Program.cs` werden dann Objekte der einzelnen Games erstellt.

ConsoleGames Demo

Framework Übersicht

- `Program.cs`: Hauptprogramm
- `Game.cs`: Klasse, die vorgibt, welche Struktur ein Game haben muss
- `Score.cs`: Klasse, die vorgibt, wie ein Score-Objetk aussieht
- Ordner `Games`: beinhaltet alle einzelnen Game-Objekte



Game-Klasse

Jedes Game muss Infos angeben wie «Name» oder «Description»: Wird vom Framework benötigt.

```
1  namespace ConsoleGames
2  {
3      public abstract class Game
4      {
5          public abstract string Name { get; } // name of game, 'get' means:
6          public abstract string Description { get; } // short descripition
7          public abstract string Rules { get; } // rules of game, how to pla
8          public abstract string Credits { get; } // name(s) of author(s) ir
9          public abstract int Year { get; } // year in which game was create
10         public abstract int LevelMax { get; } // max number of levels
11         public abstract bool TheHigherTheBetter { get; } // is a higher sc
12         public abstract Score HighScore { get; set; } // get; set means: c
13
14         public abstract Score Play(int level);
15     }
16 }
```

```
(1) Weiterspielen
(2) Level auswählen
(3) Game neu starten
(4) Game Beschreibung
(5) Game Regeln
(6) Credits
(m) Zurück zum Menu
```

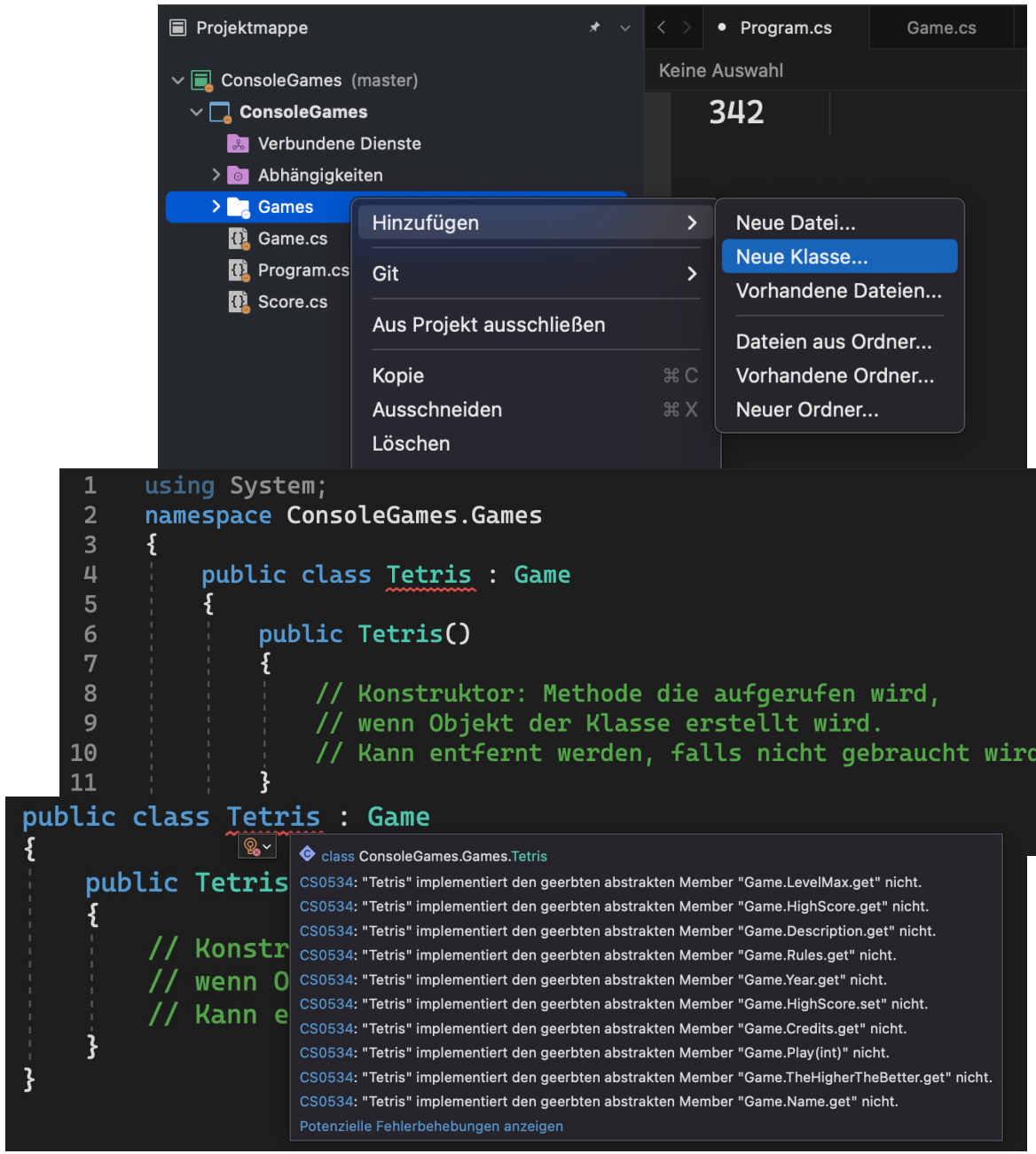
Play-Methode: Wird vom Framework aufgerufen, wenn Spiel gestartet wird. Muss Score-Objekt zurückgeben.

HighScore wird benötigt, damit überprüft werden kann, ob letzter Spielstand neuer HighScore wird. Speichern und Einlesen wird von Framework übernommen.

```
(1) Wähle Game
(2) Highscores anschauen
(3) Highscores Löschen
(q) Uf Wiederluege!
```

Neues Game erstellen

1. Erstelle neue Klasse (in neuem File) mit passendem Namen, z.B. «Tetris»
2. Jedes Spiel (selbst Klasse) muss von abstrakter Game-Klasse *erben* mit `Tetris : Game`
3. Beachte: Fehlermeldungen, weil benötigte Member (Eigenschaften und Methoden) nicht implementiert wurden.



Abstrakte Member implementieren

- Alle abstrakten Members über *implementiert* …
- … und *überschrieben* (`override`) werden.
- `Play`-Methode wird ausgeführt, wenn Game gestartet wird. Gib für Moment leeren Score zurück, damit Code ausführbar ist.
- Jetzt kann anfangen, Game zu Programmieren!

```
1   using System;
2   namespace ConsoleGames.Games
3   {
4       public class Tetris : Game
5       {
6           public override string Name => "Tetris";
7           public override string Description => "Beschreibung vom Game";
8           public override string Rules => "Erläuterung der Regeln";
9           public override string Credits => "Namen und EMail";
10          public override int Year => 2023;
11          public override int LevelMax => 42;
12          public override bool TheHigherTheBetter => true;
13          public override Score HighScore { get; set; } // no change required
14
15          public override Score Play(int level)
16          {
17              return new Score(); // for now just return empty score -> code executable
18          }
19      }
20  }
```

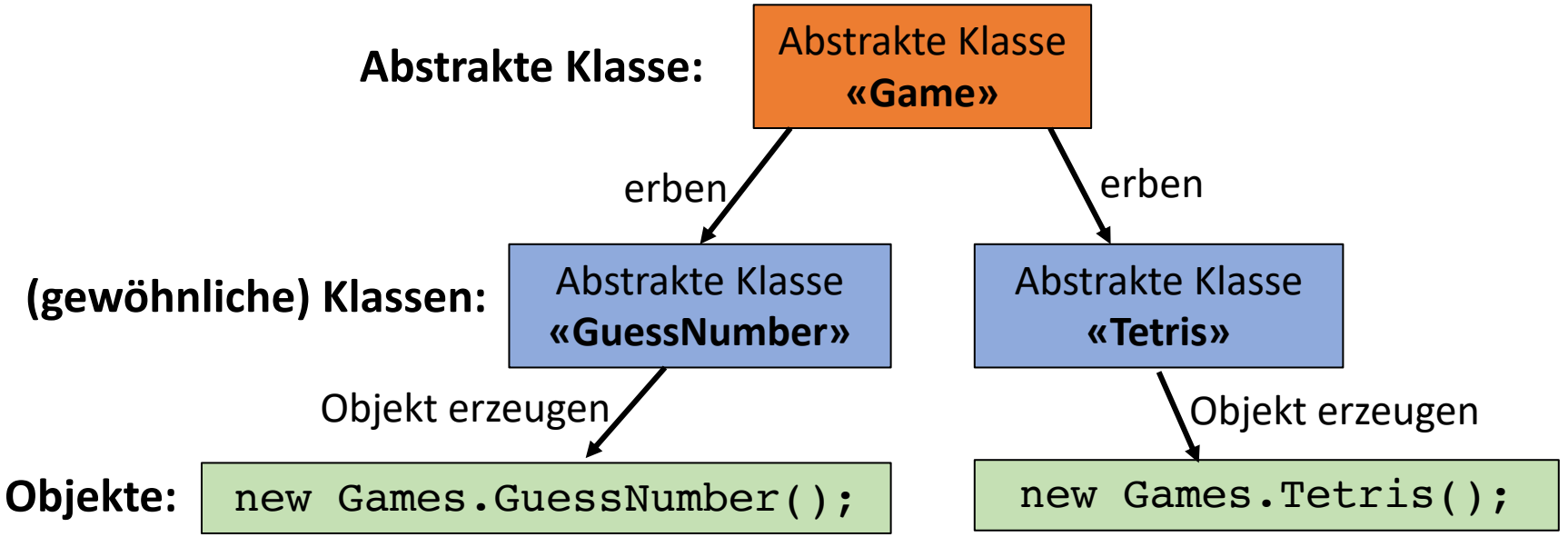
Private vs. Public

- Alle bisherigen Members deines Games sind **`public`**:
  - Sind von abstrakter Game-Klasse vorgegeben, …
  - … da sie vom Framework benötigt werden.
  - Es muss also von *aussen* auf diese Members zugegriffen werden.
- Alle weiteren Members sollen **`private`** sein:
  - Alle Felder (‘Variablen’) und Methoden (‘Funktionen’), die vom Game *intern* verwendet werden.
  - Framework kann nicht auf diese zugreifen.
- Grundsatz beim Programmieren: Zugriff auf Member soll *so restriktiv wie möglich* sein.

Game in Framework einbinden

- Erzeuge Objekt vom Typ «Tetris»: `new Games.Tetris();`
- … und füge dem `gameArray` hinzu
- Mehr muss man nicht machen, Rest läuft automatisch!

```
 6   namespace ConsoleGames
 7   {
 8       public static class Program
 9       {
10           static Game[] gameArray = new Game[] {
11               new Games.GuessNumber(),
12               new Games.Tetris()
13           };
```



Aufträge heute

- Auftrag 0:
  - Gruppen bilden (Kriterien siehe Wiki)
  - Game überlegen, mit LP besprechen …
  - … und absegnen lassen
- Auftrag I:
  1. GitHub-Repo erstellen
  2. Überblick über Framework verschaffen
  3. HangMan in Framework integrieren (copy-paste)
  4. HangMan 2.0: in Single Player Game umwandeln
- Details auf Wiki