Textadventure Code

Dit document legt de Zuul code uit.
Je leert hoe de applicatie werkt.

Inhoudsopgave

Code draaien

Voordat je iets aanpast, test je eerst of de code werkt.

Stap 1: Unzip het bestand
Unzip de zipfile eerst.
Belangrijk: Werk nooit binnen een zipfile!

Stap 2: Open een terminal
Open een terminal in de folder waar de .csproj file staat.

Stap 3: Run de applicatie
Geef dit commando:

```
dotnet run
```

Foutmelding: verkeerde dotnet versie?

Pas de .csproj file aan.
Verander de versie naar een versie die je hebt.

```
<TargetFramework>net8.0</TargetFramework>
```

Welke versies heb ik?

Gebruik dit commando:

```
dotnet --list-runtimes
```

Pas de .csproj file aan naar een van deze versies.

Of installeer de laatste versie:
Installeer dotnet

Test het spel
Als er geen errors meer zijn, speel dan de textadventure.
Bezoek alle kamers.

Zuul Code

Open de directory in VS Code.
De directory bevat je .csproj file en de code.

Program.cs

De code begint in Program.cs.
In de Main method staat:

```
Game game = new Game();
game.Play();
```

Regel 1: Maak een Game object

- Roep de constructor aan van Game
- Sla het object op in variabele `game`

Regel 2: Start het spel

- Roep de method Play() aan op `game`

Game.cs

De class Game heeft twee fields:

```
class Game
{
  private Parser parser;
  private Room currentRoom;

  // ... de rest van de code
}
```

Fields:

- `parser` - type Parser
- `currentRoom` - type Room

Deze worden verderop uitgelegd.

Game Constructor

In de constructor worden de fields geïnitialiseerd:

```
class Game
{
  // Constructor
  public Game ()
  {
    parser = new Parser();
    CreateRooms();
  }
}
```

Wat gebeurt hier:

1. Maak een nieuwe Parser
2. Roep CreateRooms() aan

Game.CreateRooms()

In CreateRooms() worden alle Rooms aangemaakt.
Ook worden de uitgangen tussen Rooms gemaakt.

Later maak je hier ook je Items aan.

```
private void CreateRooms()
{
  // Maak de rooms
  Room kitchen = new Room("in the kitchen");
  Room cellar = new Room("in the cellar");

  // Maak de exits
  kitchen.AddExit("down", cellar);
  cellar.AddExit("up", kitchen);

  // Start in de kitchen
  currentRoom = kitchen;
}
```

Wat gebeurt hier:

Stap 1: Maak twee Rooms

- `kitchen` (de keuken)
- `cellar` (de kelder)

Stap 2: Verbind de Rooms

- Vanuit `kitchen` ga je "down" naar `cellar`
- Vanuit `cellar` ga je "up" naar `kitchen`

Stap 3: Start positie

- Het veld `currentRoom` wijst nu naar `kitchen`
- De speler start in de keuken

Let op: `currentRoom` is een soort pointer.
Deze wijst telkens naar een andere kamer.

Game.Play()

In de method Play() loopt het spel.
Telkens wordt een nieuw Command opgehaald van de speler.

```
public void Play()
{
    PrintWelcome();

    // Hoofdloop: herhaal tot de speler wil stoppen
    bool finished = false;
    while (!finished)
    {
        Command command = parser.GetCommand();
        finished = ProcessCommand(command);
    }
    Console.WriteLine("Thank you for playing.");
}
```

Wat gebeurt hier:

Stap 1: Print welkomstbericht

Stap 2: Start de loop

- Zolang `finished` false is, blijft de loop draaien

Stap 3: Haal een commando op

- `parser.GetCommand()` print een `>` teken
- De speler kan nu een commando tikken
- GetCommand() maakt een Command object
- Het Command wordt gereturnd

Stap 4: Verwerk het commando

- ProcessCommand() verwerkt het commando
- Is het commando "quit"? Dan wordt true gereturnd

- Anders wordt false gereturnd

Stap 5: Check of het spel klaar is

- `finished = true` betekent: stop de loop
- `while (!finished)` betekent: while NOT finished
- NOT false = true, dus de loop draait door
- NOT true = false, dus de loop stopt

Stap 6: Bedank de speler

Zie ook: Parser, Command, CommandLibrary

Game.ProcessCommand(command)

Een Command heeft twee delen:

- `commandWord` (het eerste woord, bijv. "go")
- `secondWord` (het tweede woord, bijv. "north")

Beide zijn strings.

In ProcessCommand() worden methods aangeroepen.
Dit hangt af van het commandWord.

```
// Verwerk een commando
// Als dit commando het spel beëindigt, return true
// Anders return false
private bool ProcessCommand(Command command)
{
  bool wantToQuit = false;

  if(command.IsUnknown())
  {
    Console.WriteLine("I don't know what you mean...");
    return wantToQuit; // false
```

```
    }

    switch (command.CommandWord)
    {
      case "help":
        PrintHelp();
        break;
      case "go":
        GoRoom(command);
        break;
      case "quit":
        wantToQuit = true;
        break;
    }

    return wantToQuit;
  }
```

Wat gebeurt hier:

Stap 1: Check of het commando geldig is

- Is het commando onbekend?
- Print dan een foutmelding
- Return false (het spel gaat door)

Stap 2: Verwerk het commando

- Is het "help"? Roep PrintHelp() aan
- Is het "go"? Roep GoRoom(command) aan
- Is het "quit"? Zet wantToQuit op true

Stap 3: Return of het spel klaar is

- true = spel is klaar
- false = spel gaat door

Let op: Het command wordt meegegeven aan GoRoom().
Daar wordt gekeken naar welke richting de speler wil lopen.

Bekijk de method GoRoom() in de code.
Zie hoe er gecontroleerd wordt:

- Of er wel een deur is
- Of de speler wel een richting heeft opgegeven

Room

Een Room heeft een beschrijving en uitgangen.

```
class Room
{
  // Private fields
  private string description;
  private Dictionary<string, Room> exits;

  // Maak een room met beschrijving
  // De beschrijving is iets als "in a kitchen" of "in a
court yard"
  public Room(string desc)
  {
    description = desc;
    exits = new Dictionary<string, Room>();
  }

  // Voeg een exit toe aan deze room
  public void AddExit(string direction, Room neighbor)
  {
    exits.Add(direction, neighbor);
  }
}
```

Fields:

- `description` - de beschrijving van de kamer
- `exits` - een Dictionary met uitgangen

Constructor:

- Krijgt een beschrijving als parameter
- Maakt een lege Dictionary voor exits

Method AddExit:

- Voegt een uitgang toe
- `direction` is de richting (bijv. "north")
- `neighbor` is de kamer waarheen de uitgang gaat

Parser

De Parser zet input van de gebruiker om in een Command.

```
class Parser
{
  // Bevat alle geldige commando's
  private readonly CommandLibrary commandLibrary;

  // Constructor
  public Parser()
  {
    commandLibrary = new CommandLibrary();
  }

  // Vraag input van de gebruiker en maak een Command
  public Command GetCommand()
  {
    Console.Write("> "); // print prompt

    string word1 = null;
    string word2 = null;
```

```
        // Split de input in woorden
        string[] words = Console.ReadLine().Split(' ');
        if (words.Length > 0) { word1 = words[0]; }
        if (words.Length > 1) { word2 = words[1]; }

        // Check of dit een geldig commando is
        if (commandLibrary.IsValidCommandWord(word1)) {
          return new Command(word1, word2);
        }

        // Anders maak een "null" commando (onbekend commando)
        return new Command(null, null);
      }
    }
```

Wat gebeurt hier:

Stap 1: Print de prompt `>`

- De gebruiker ziet nu dat er input verwacht wordt

Stap 2: Lees de input

- `Console.ReadLine()` leest een regel
- `Split(' ')` splitst de regel op spaties
- Dit geeft een array van woorden

Stap 3: Pak de woorden

- Het eerste woord is word1
- Het tweede woord is word2

Stap 4: Check of het commando geldig is

- `IsValidCommandWord()` checkt dit

- Geldig? Maak een Command met word1 en word2
- Ongeldig? Maak een Command met null, null

Command

Een Command is een simpel datatype.
Het heeft alleen een CommandWord en een SecondWord.

```
class Command
{
  public string CommandWord { get; init; }
  public string SecondWord { get; init; }
}
```

Voorbeeld:

De gebruiker typt:

```
go east
```

Dan is:

- CommandWord = "go"
- SecondWord = "east"

CommandLibrary

De CommandLibrary heeft een lijst met geldige commando's.

Als je nieuwe commando's toevoegt (look, take, drop, use):
Voeg ze ook toe aan de CommandLibrary!

```
class CommandLibrary
{
  // Een List met alle geldige commando's
```

```
    private readonly List<string> validCommands;

    // Constructor - initialiseer de commando's
    public CommandLibrary()
    {
      validCommands = new List<string>();

      validCommands.Add("help");
      validCommands.Add("go");
      validCommands.Add("quit");
    }
  }
```

Wat gebeurt hier:

Constructor:

- Maak een lege List
- Voeg de geldige commando's toe

Geldige commando's:

- "help" - vraag hulp
- "go" - ga naar een andere kamer
- "quit" - stop het spel

Later voeg je toe:

- "look" - bekijk de kamer
- "take" - pak een item op
- "drop" - leg een item neer
- "use" - gebruik een item
- "status" - bekijk je status