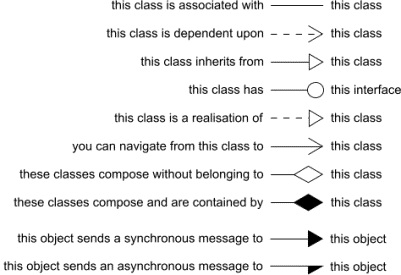
# Les 2:



Encapsulation:

Bescherming van private variabele.

Hoe stroomt data en hoe zorg je dat het systeem zichzelf niet sloopt.

Architectuur: Hoe plaats je objecten.

Ontwerpen van gedrag.

API : Application Programming Interface

[Language Features]

* Classes
* Inheritance

[Virtual Class] -> [Override Class]

[Abstract Class] (Bestaat zelf niet) (kan wel extended worden)

class Hond : Dier

virtual void Blabla() { }

override void Blabla() { }

base.functie() voert standardfunctie uit als gedefineerd in de class die je overrided.

[Property]

* Get
  + Opvragen
  + Mogelijk aanpassen en dan pas teruggeven.
  + Return
  + Access Modifiers
    - Private
    - Protected
* Set
  + Definiëren van gekregen variabele.
  + Je krijgt een keyword genaamd “value”.

private int health = 0;

public int Health {

get {

return health;

}

protected set {

health = value;

}

}

[Interface]

Een lijst van functies.

Alles binnen een interface is standard public.

public class Creature { }

public interface IAmGroot {

string TalkGroot(int count);

string TalkGroot(string grootName);

}

public class Treeman : Creature, IAmGroot {

public string TalkGroot(int count) {

if (count > 2) {

return "You are groot";

}

else {

return "I am groot";

}

}

public string TalkGroot( string grootName ) {

return "I am" + grootName;

}

}

[Recursie]

# Les 3

**Virtual** functies kunnen **Override** worden.

**Singletons** zijn een oplossing voor een probleem, niet gebruiken als het niet nodig is.

public sealed class Singleton {

private static Singleton instance = null;

private static readonly object padlock = new object();

Singleton() {

}

public static Singleton Instance {

get {

lock (padlock) {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

}

}

**Delegates** zijn variabelen die meerdere functies tegelijk kunnen uitvoeren door deze toe te voegen aan de desbetreffende delegate.

using UnityEngine;

using System.Collections;

public class DelegateScript : MonoBehaviour {

delegate void MyDelegate( int num );

MyDelegate myDelegate;

void Start() {

myDelegate = PrintNum;

myDelegate(50);

myDelegate = DoubleNum;

myDelegate(50);

}

void PrintNum( int num ) {

print("Print Num: " + num);

}

void DoubleNum( int num ) {

print("Double Num: " + num \* 2);

}

}

**Events** kunnen listeners hebben en daarmee kun je een bepaald gedrag oproepen in classes die luisteren.

using UnityEngine;

using System.Collections;

public class EventManager : MonoBehaviour {

public delegate void ClickAction();

public static event ClickAction OnClicked;

void OnGUI() {

if (GUI.Button(new Rect(Screen.width / 2 - 50, 5, 100, 30), "Click")) {

if (OnClicked != null)

OnClicked();

}

}

}

public class TeleportScript : MonoBehaviour {

void OnEnable() {

EventManager.OnClicked += Teleport;

}

void OnDisable() {

EventManager.OnClicked -= Teleport;

}

void Teleport() {

Vector3 pos = transform.position;

pos.y = Random.Range(1.0f, 3.0f);

transform.position = pos;

}

}

# Les 4

**Generics** : Een zelf definieerbare variabele, ook in **Lists** gevonden.

Array of List met vaste grootte zijn beter voor geheugen.

|  |  |  |
| --- | --- | --- |
| **Class** | **Abstract Class** | **Interface** |
| Var + Func | Var + Func en Func Proto | Func Prototype |
| Volledige Object | Niet volledige object | Object Format |

**Abstract** : Abstract functies moet je **invullen**, abstract classes met *niet* abstracte functies worden gewoon uitgevoerd. Abstract functies moet je **overriden**.

Ook normale functies kunnen **override** worden in abstract classes als ze **virtual** zijn.

**Abstract Factory :** Geeft een variabele terug, maar is niet altijd zelfde. Handig voor platform verschillen. Meerdere Instances kunnen worden opgevraagd.

**Jagged Array** is een array met meerdere parameters.

**Nested Array** is een array binnen array.

**List** kan elementen on the fly toevoegen.

Toevoegen kan met list.Add(object); Dit wordt toegevoegd aan het einde van de lijst.

Verwijderen kan met list.Remove(object);

Alle elementen vinden kan met list.FindAll(object);

**Labda** methode met list is list.FindAll((x) => {parameters vergelijken);

naam = new Type<ObjectType>();

**Queue** (**First in First out)**

**Enqueue**(object) zorgt dat je objecten achteraan in een queue stopt.

Met **Dequeue**(object) zorgt dat je objecten vooraan in de queue gebruikt en uit de queue haalt.

**Stack (Last in First out)**

**Push**(object) voegt object aan Stack toe.

**Pop**(object) haalt laatst toegevoegde object uit de stack.

**Dictionary**

**Add**(“String naam”,object)

[“String naam”] roept het op.

ContainsKey(“String naam”); alle elementen met de string.

Ook **HashSet** is vergelijkbaar.

**JSON** is trager en wordt vooral in web veel gebruikt.