

# Modul 24

# Objektorientierte Programmierung

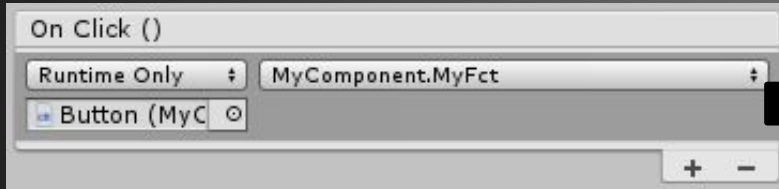
Oliver Ziegler

# Überblick

- (Kurzerklärung) UI-Buttons
- Funktionen
- Variablen
- Debugging
- Datentypen
  - Simpel
  - Komplex
- Unity-Dokumentation Research
- (Funktionsparameter)

# Button

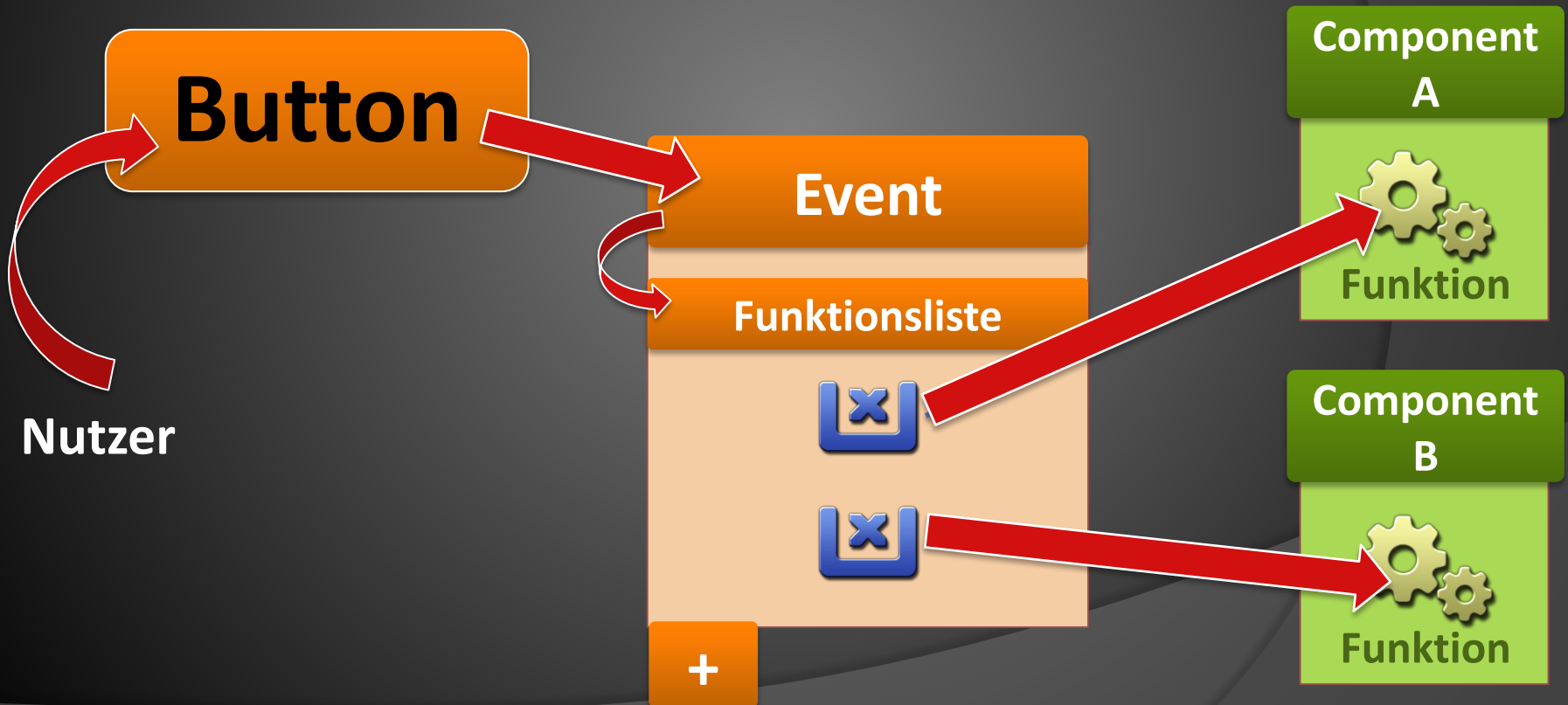
- **OnClickEvent**



```
Public class MyComponent()  
{  
    public void MyFct1()  
    {  
    }  
}
```

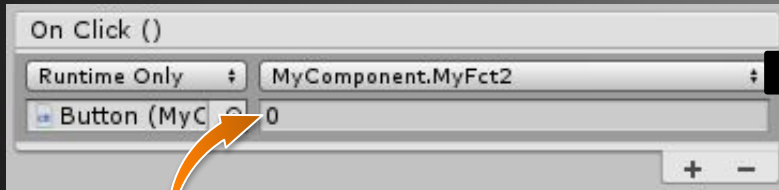
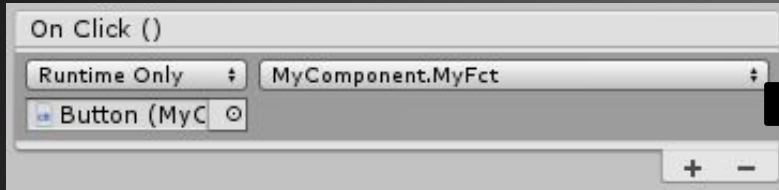
# Button -> Funktion

- Entgegennehmen von Interaktionen
- Buttons



# Button

- **OnClickEvent**



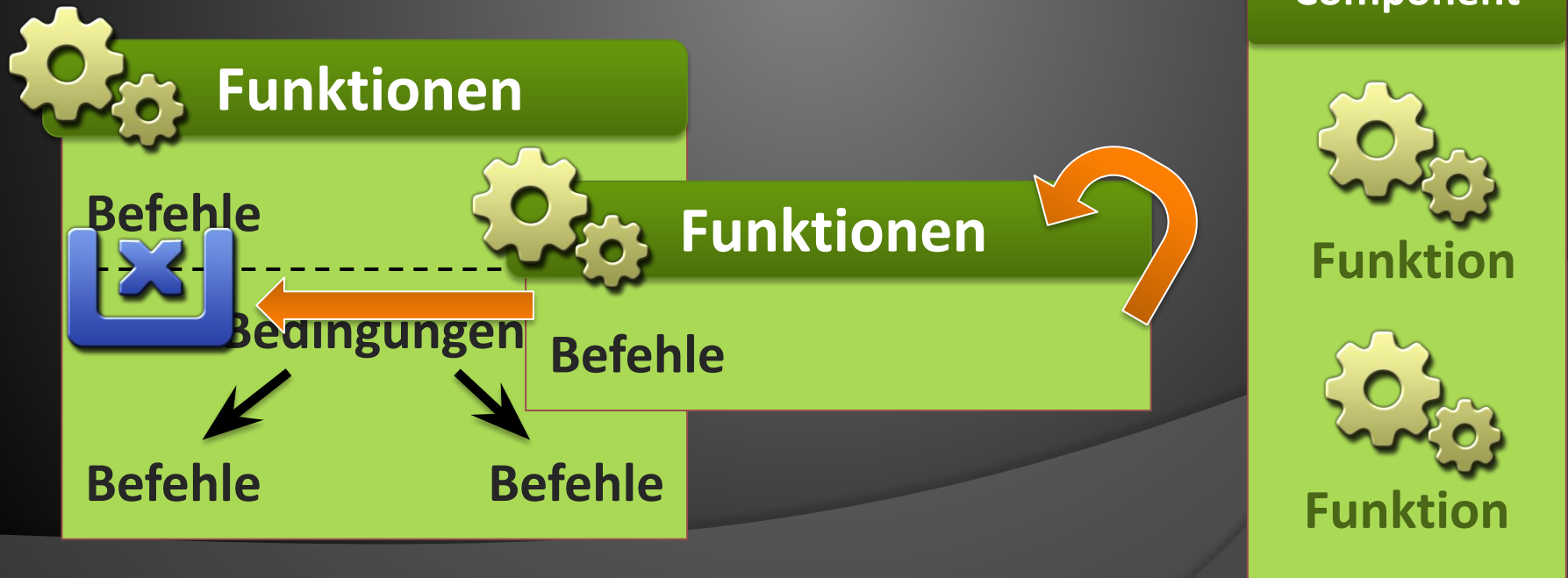
parameter



```
Public class MyComponent()  
{  
    public void MyFct1()  
    {  
    }  
}  
  
    public void MyFct2( int parameter)  
    {  
    }  
}
```

# Funktionen

- Berechnen, Entscheiden, Speichern, Laden
- Darstellen, Abspielen, Verändern
- Start durch System oder Nutzer
- Können einander aufrufen



# Funktionen

```
public class MyFirstClass: MonoBehaviour
{
    public void MyFunction()
    {
    }

    public void MyFunction2()
    {
    }
}
```

# Funktionsdefinition

Protection  
keyword

**public**

Rückgabewert  
(void, wenn  
ohne Rückgabe)

**void**

Freiwählbarer Name  
Eindeutig  
Konvention: vorn Groß

**MyFunction**

Parameter  
( ), wenn keine

**( )**



```
{
```

Inhalt/ Befehle, jeweils mit Semikolon

```
}
```

**KEIN SEMIKOLON!**

Die Funktion wird beschrieben, aber nicht ausgeführt, kein Speicher vorbereitet □ Kein Befehl



# Funktionen

```
public class MyFirstClass: MonoBehaviour
{
    public void MyFunction()
    {
    }

    public void MyFunction2()
    {
    }
}
```

- **Klassen können mehrere Funktionen haben**
- **Inhalt der Funktion durch geschweifte Klammern eingefasst**
- **public, void und () einfach akzeptieren, dazu später mehr** (Sichtbarkeit, Rückgabewert, Parameter)

# Befehle

- **Kleinste Funktionsbausteine**
- **Verändern Variablen**
- **Führen andere Funktionen aus**

```
public void MyFunction()  
{  
    MyFunction2();  
}  
  
public void MyFunction2()  
{  
    Debug.Log( „Das ist ein Test“ );  
}
```

# Befehle

- Serielle Abarbeitung
- Immer in einer Funktion (außer Definitionen)
- Immer mit SEMIKOLON beendet
- Damit der Rechner das Ende erkennt

```
public void MyFunction() {DoStuff();MyFct3();Go( „Yeah“ ); }
```

```
public void MyFunction()  
{  
    DoStuff();  
    MyFct3();  
    Go( „Yeah“ );  
}
```

Für den Rechner  
identisch



# Funktionen

```
public class MyFirstClass: MonoBehaviour
```

```
{
```

```
    MyFunction( );
```

1

Ausserhalb einer Funktion  
und keine Var.-Definition

```
    public void MyFunction()
```

```
{
```

```
        MyFunction( )
```

2

Kein Semikolon

```
}
```

```
    public void MyFunction2()
```

```
{
```

```
        MyFunction( );
```

3

Läuft

```
}
```

```
    MyFunction( )
```

4

Kein Semikolon  
und siehe 1

```
}
```

# Funktionen

```
public class MyFirstClass: MonoBehaviour
{
    public void MyFunction()
    {
        Debug.Log( „Das ist ein Test“ );
    }

    public void MyFunction2()
    {
        Debug.Log( „Now something completely different“ );
    }
}
```

- Man bräuchte eine einzelne Funktion für alles
- Das ist  $Y = 5$  nicht  $Y = 522 / X$ ;
- -> Variablen

# Funktionen

- Kurzer Ausblick auf Variablen und Parameter

```
public class Charakter: MonoBehaviour
{
    public void GreifeAn( Character target )
    {
        target.ErleideSchaden();
    }

    public void ErleideSchaden()
    {
        Debug.Log( „Autsch“ );
    }
}
```

# Variablen

Protection  
keyword

`public`

Datentyp

`string`

Freiwählbarer Name  
Eindeutig  
Konvention: vorn klein

`meineVariable`

Semikolon!

`;`

**SEMIKOLON!**

**Das Anlegen einer Variable reserviert Speicher  
Ist für den Computer also ein Befehl**

# Datentypen

- **Simple und komplexe Datentypen**
- **Simple sind Daten wie**
  - **Zeichen**
  - **Texte**
  - **Ganze Zahlen**
  - **Gebrochene Zahlen**
  - **Wahrheitswerte**
  -



# Datentypen

- Größe des benötigten Speichers
- Hilfe für die Zusammensetzung von rohen Daten
- Erleichtert das Verwenden von Funktionen, des jeweiligen Datentyps

Datentyp	Art der Daten
string	Text (immer mit " ")
int	Ganze Zahl
bool	Wahrheitswert (true / false)
float	Kommazahl ( # . ## f ) ( 3.141f )
char	Ein Zeichen (mehrere sind ein string) (immer mit ' ')

# Variablen

```
public class MyFirstClass: MonoBehaviour
{
    public string meinText;
    public void MyFunction()
    {
        Debug.Log( „Das ist ein Test“ );
        Debug.Log( meinText );
    }
}
```

- Können feste Werte im Code ersetzen
- An ihre Stelle tritt dann ihr Inhalt  
**Zu DIESER Zeit im Programm -> dynamisch**

# Variablen

- Können auch zur Laufzeit verändert werden

```
public string meinText;  
public int y;  
public int x;
```

```
// Ausgabe: Was im Inspector stand
```

```
public void Function1()  
{
```

```
// Ausgabe: You are not prepared!
```

```
    Debug.Log(meinText);  
    meinText = „You are not prepared!“;  
    Debug.Log(meinText);  
}
```

```
public void Function2()  
{
```

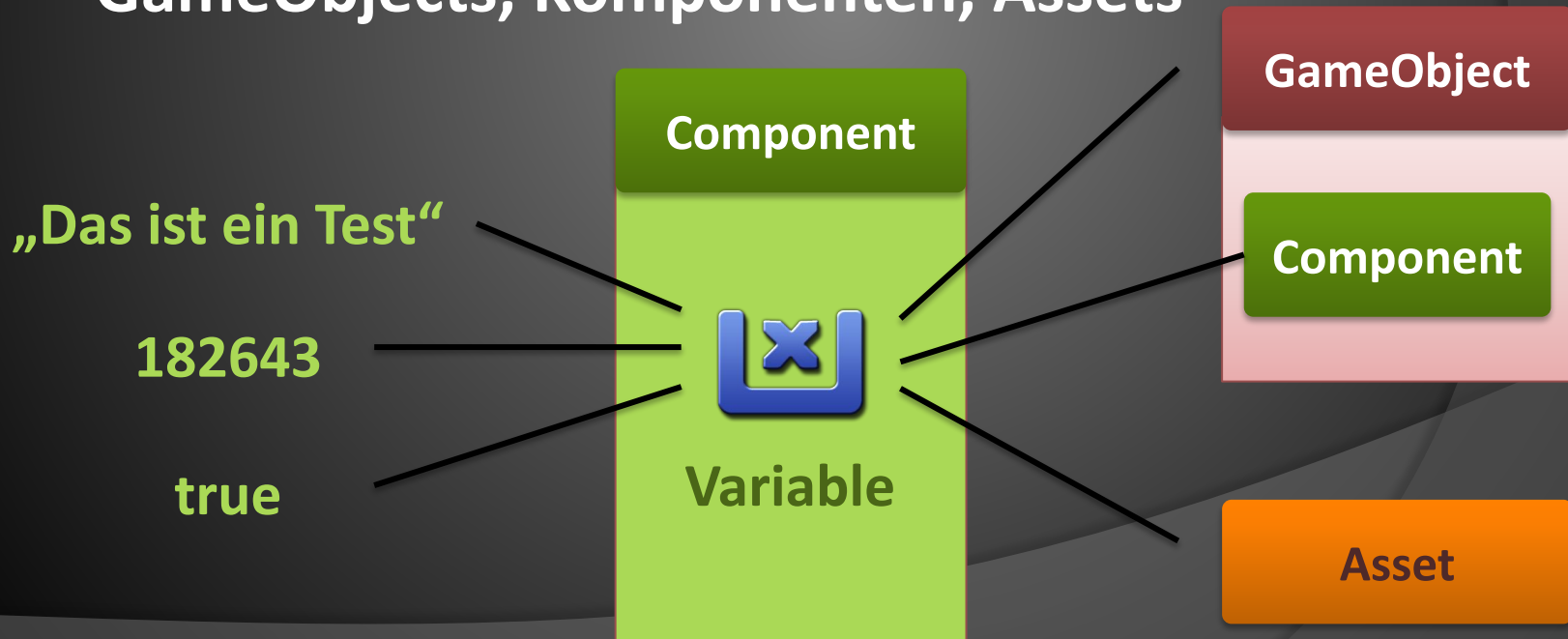
```
    y = 6584 * ( 543 / x );  
    y = y + y + x;  
}
```

# Variablen

- **Concatinieren**  
string + string  
string + „/“ + string  
string + variable
- **Rechnen**
- **Probleme**  
(Beispiel int / int  Keine float Zahl)

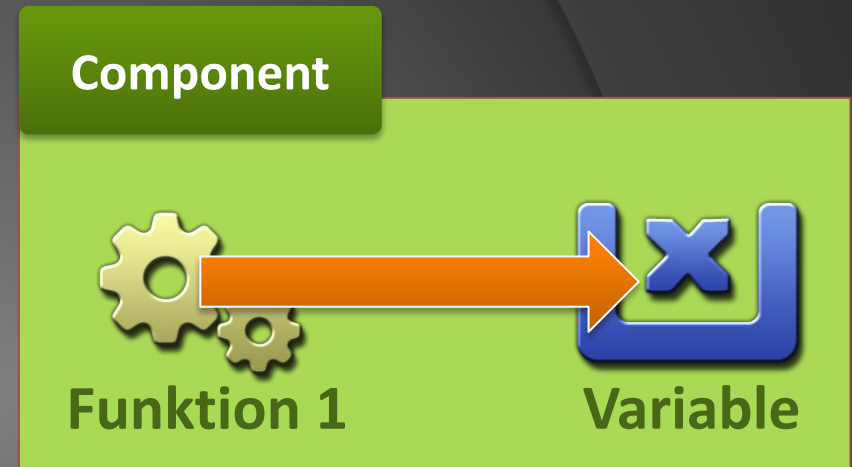
# Variablen

- Variablen können grob unterschieden werden
- Klare Daten  
Texte, Zahlen, Bool, Vektoren (also auch komplexe Datentypen)
- Verknüpfungen zu Daten  
GameObjects, Komponenten, Assets



# Verknüpfung/Interne Änderung

- Einfache Daten-Zuweisung
- Mittels „=“



```
public int dieAntwortAufDasLebenUndAlles;  
public void Funktion1()  
{  
    dieAntwortAufDasLebenUndAlles = 42;  
}
```

- ◉ Aufgabe 1    □ AddMultiplyScene
- ◉ Button „Add“ : Addiert zwei Variablen und gibt das Ergebnis in der Konsole aus.
- ◉ Button „Multiply“ : Multipliziert dieselben Variablen und gibt das Ergebnis in der Konsole aus.

### Hinweis: Konsolenausgabe

```
Debug.Log( „Meine Ausgabe“ );  
Debug.Log( meineZahl.ToString() );
```

# Debugging

- **Testen, Suchen und Fehlerbehebung**
- **30 – 40% der Entwicklungszeit**
- **Fehler sind Normalität!**
- **Unterscheidung**
  - „Syntax“-Fehler
  - Denkfehler
  - Implementationsfehler
- **Erfahrung -> Strategie zur Fehlerfindung**



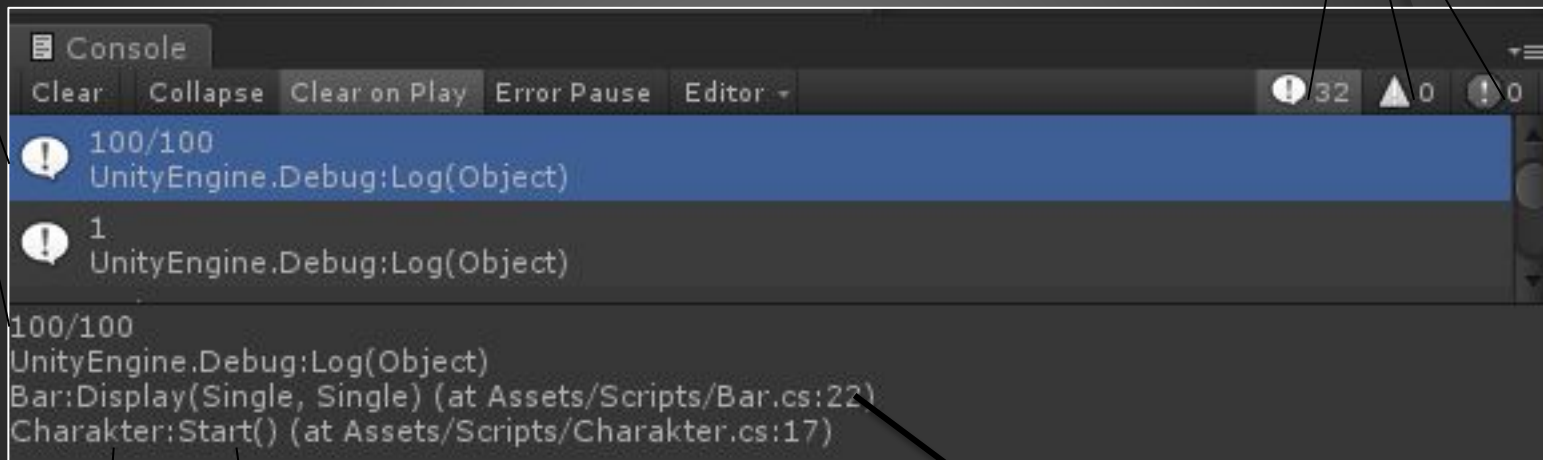
# Debugging

- **Error**
  - Vom System erkannte Fehler beim Kompilieren (vor der Laufzeit)
  - Nicht ausführbare Befehle (zur Laufzeit)
- **Warnings**
  - Hinweise auf unsaubereren Code
  - Können bei Updates zu Fehlern werden
- **Logs**
  - Vom Programmierer gesetzte Ausgabe/Hilfe

# Console (in Unity!)

Log-Text

Toogle zum Anzeigen  
der 3 Log-Arten



The screenshot shows the Unity Console window with the following content:

```
Console  
Clear Collapse Clear on Play Error Pause Editor  
32 0 0  
100/100  
UnityEngine.Debug:Log(Object)  
1  
UnityEngine.Debug:Log(Object)  
100/100  
UnityEngine.Debug:Log(Object)  
Bar:Display(Single, Single) (at Assets/Scripts/Bar.cs:22)  
Charakter:Start() (at Assets/Scripts/Charakter.cs:17)
```

1) Klasse, Funktion und Zeile des Befehl-Ursprungs

2) Nächste Station der Befehlskette

# Debugging

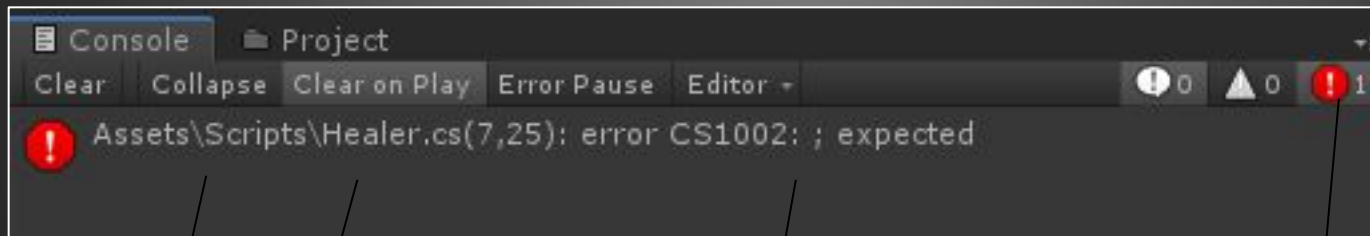
- **Kommentare**
  - Text im Code, der nicht ausgeführt wird
  - Dient der Erklärung
  - Realisiert durch „//“ vor der Zeile
- Häufig benutzt um Code-Zeilen zu deaktivieren

```
public void DoSomethingAwesome()  
{  
    // AwesomeStuff();  
    // Awesome didn't work...  
    DoSomethingNormal();  
}
```

# Console (in Unity!)

- **Errors**

**Kompilierungsfehler verhindern das Ausführen/Starten des Programms**



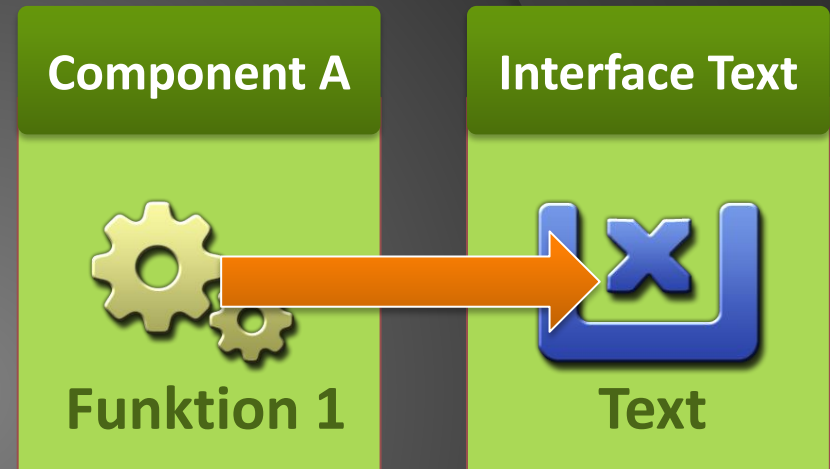
1) Pfad, (Klasse)

2) Fehlermeldung

3) Fehleranzahl/-Indikator

# Verknüpfung/Externe Änderung

- Ansprechen der Variable (Komponent)
- . - Punkt!
- Auswahl der gewünschten Variable

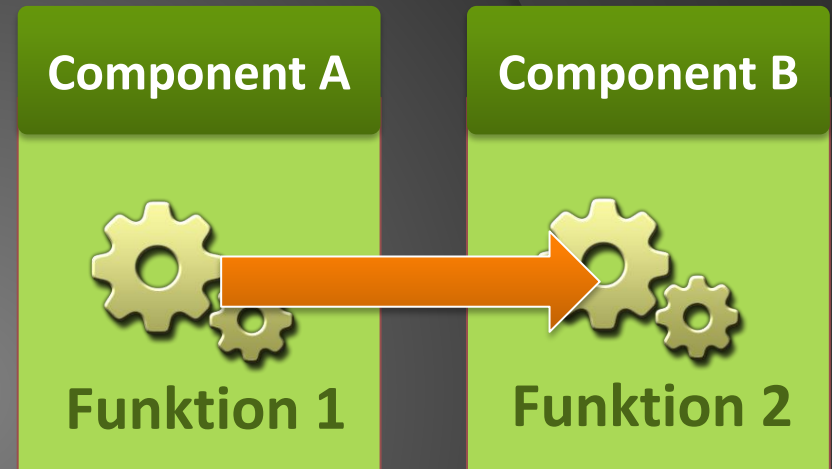


```
public Text myInterfaceText;  
  
public void Funktion1()  
{  
    myInterfaceText . text = „Das ist ein Test“;  
}
```

```
public class Text  
{  
    public string text;  
  
    ZeigMalDenTextAn();  
}
```

# Funktion anderer Objekte

- Aufruf der Funktion (Komponent)
- . - Punkt!
- Auswahl der Funktion wie sonst



```
public MyCoolComponent comp;  
  
public void Funktion1()  
{  
    comp. DoStuff();  
}
```

```
public class MyCoolComponent  
{  
    public void DoStuff()  
    {  
    }  
}
```

# Documentation Research

- Suchen nach Funktion oder Variable einer Klasse
  - Unity-Documentation
  - Internet-Research, Foren
- Beispiel: `UnityEngine.UI.Text``text`

**Achtung: Interface namespace**

**Oben hinzufügen**

```
using UnityEngine;  
using UnityEngine.UI;
```



- ⦿ Aufgabe 2.1      □ ProgressBarScene
- ⦿ Erhöhen einer int-Variable um Wert einer anderen int-Variable.  
Ausgabe im UI-Text ProgressText.

**Hinweis: Zahl zu Text machen (int zu string überführen)**

```
int myNumber = 826;  
string hallo = myNumber.ToString( );
```

- Aufgabe 2.2      □ ProgressBarScene
- Recherche zu „Image“ an Objekt ProgressBar
- Variablen-Wert als Füllung der Grafik des Objekts ProgressBar darstellen
- *(Zusatzaufgaben für gesamte Stunde:  
Auf 100 begrenzen (ohne Bedingungen: if,  
switch,..)*

```
Image progressBarGraphic;  
progressBarGraphic.fillAmount = ??? ;
```

# Klassen/Funktionsvariablen



```
public int element = 2;  
  
public void MagicFct()  
{  
    element = element + 10;  
}
```

# Klassen/Funktionsvariablen

## Klasse

```
public int speed; //Klassenvariable
```

## Funktion

```
int distance; //Funktionsvariable
```

```
public int intelligence; //Klassenvariable
```

## Funktion

```
int distance; //Neue Funktionsvariable
```

```
public string name; //Klassenvariable
```

# Funktionsvariablen

## class MyComponent

```
public int myGlobalNumber;
```

### void MyFunction()

```
public int myNumber;
```

```
myGlobalNumber = 10;  
myNumber = 55;
```

### void MyFunction2()

```
public int myNumber;
```

```
myGlobalNumber = 1000;  
myNumber = myNumber + 30;
```

## Klassenvariablen

- Werden auf derselben Ebene wie Funktionen definiert
- Erscheinen im Inspector
- Werden bis zum Szenenwechsel gespeichert

## Funktionsvariablen

- Werden innerhalb von Funktionen definiert
- Erscheinen NICHT im Inspector
- Werden nach der Funktion gelöscht

# Klassen/Funktionsvariablen

## KlassenVariablen

- werden außerhalb von Funktionen definiert
- Sie sind von allen Funktionen ansprechbar (intern und extern)
- Sie werden mit der Instanz gespeichert

## FunktionsVariablen

- werden innerhalb einer Funktion definiert
- Sie existieren nur innerhalb dieser Funktion
- Inhalt/Speicher wird nach dem letzten Befehl wieder gelöscht

# Funktionsvariablen

- Variablen können übrigens schon bei der Definition mit Daten gefüllt werden

```
class MyComponent
```

```
public int myGlobalNumber = 55;
```

```
void MyFunction()
```

```
public int myNumber = 100000;
```

```
public string meinText = „BAM“ ;
```

# Funktionsvariablen

```
class MyComponent
```

```
public int myGlobalNumber;
```

```
void SuperFunction()
```

```
public int myNumber = 5;  
myNumber = 1000;  
myGlobalNumber = myGlobalNumber +  
myNumber;
```

```
void AwesomeFunction()
```

```
public int myNumber;  
  
myGlobalNumber = myGlobalNumber +  
myNumber;
```

Wie groß ist  
„myGlobalNumber“  
am Ende?

```
myGlobalNumber = 20;
```

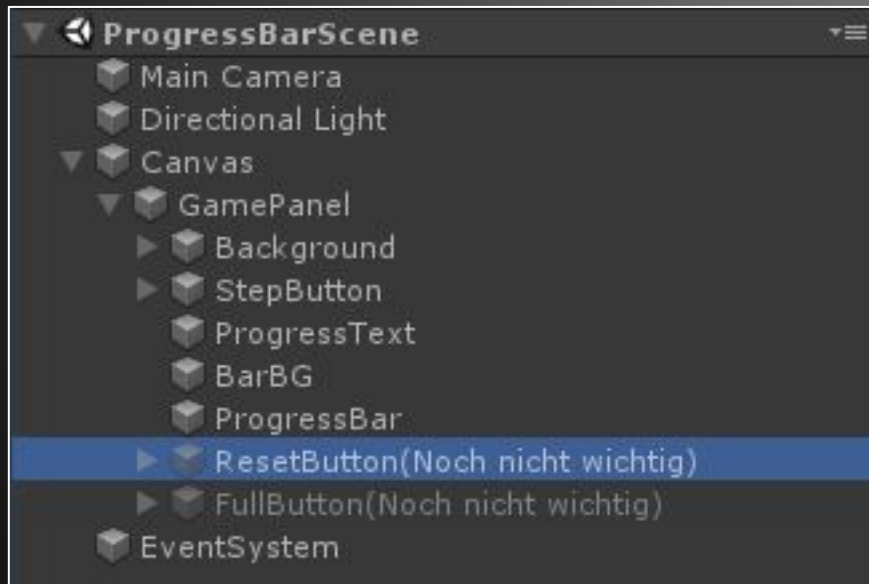
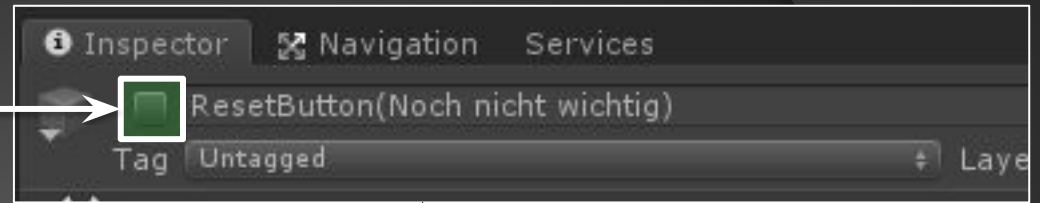
```
SuperFunction();
```

```
AwesomeFunction();
```

```
SuperFunction();
```



# Aktivieren/Deaktivieren von GameObjecten



Nutzen der  
Buttons  
„Reset“- und „Full“

- ⦿ Aufgabe 2.3      □ ProgressBarScene
- ⦿ Nutze die Buttons „Reset“- und „Full“-Button
- ⦿ **Button „Reset“:**  
Variablen-Wert zurücksetzen (0) und Darstellung anpassen.
- ⦿ **Button „Voll“:**  
Variablen-Wert auf 100 setzen und Darstellung anpassen.

# Variablen

- Klassen / Funktion – Variablen

```
public int element = 2;
```

```
public void Function1() {  
    int index = 10;  
    element = element + index;  
    Function2();  
}
```

```
public void Function2() {  
    int index = 34;  
    element = element + index;  
}
```

```
public void Function3() {  
    element = element + index;  
}
```

# Funktion: Parameter

- Erlauben es, Daten als Funktionsvariablen in die Funktion mitzugeben

```
public int ergebnis = 55;
```

```
void MyFunction ( )
```

```
AddiereZuErgebnis( 526 );
```

```
void AddiereZuErgebnis( int myParameter )
```

```
ergebnis = ergebnis + myParameter ;
```



The diagram illustrates the flow of data from a function call to a function definition. It features two code blocks on a light green background. The top block shows a function call: `void MyFunction ( )` followed by `AddiereZuErgebnis( 526 );`. The number `526` is highlighted in yellow. A red arrow points from this `526` down to the parameter `int myParameter` in the function definition below. The function definition is `void AddiereZuErgebnis( int myParameter )` followed by `ergebnis = ergebnis + myParameter ;`. The parameter `myParameter` is highlighted in red. A second red arrow points from `myParameter` down to the `myParameter` in the assignment statement.

# Parameter

- Funktionsvariablen, die von außen gesetzt werden

```
Debug.Log( „Das ist ein Test“ );
```

```
public class Debug
{
    public void Log( string text )
    {
        DisplayInConsole( text );
    }
}
```

# Parameter

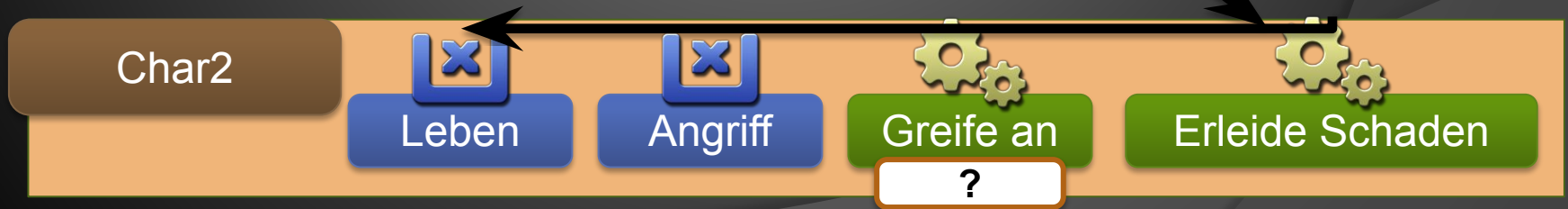
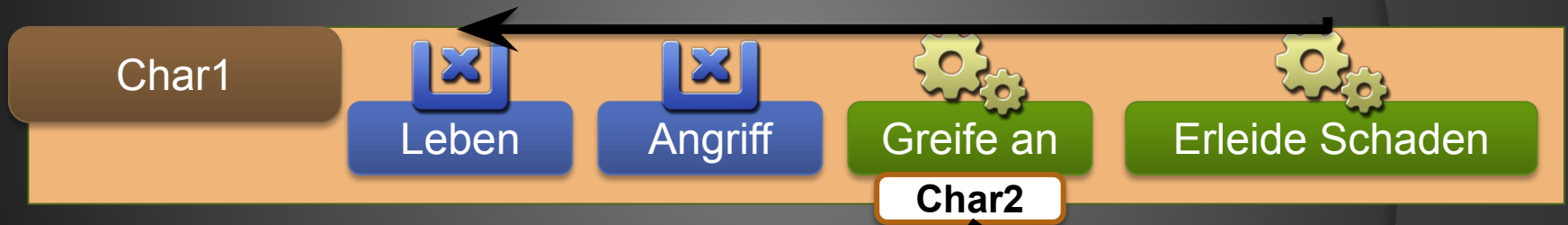
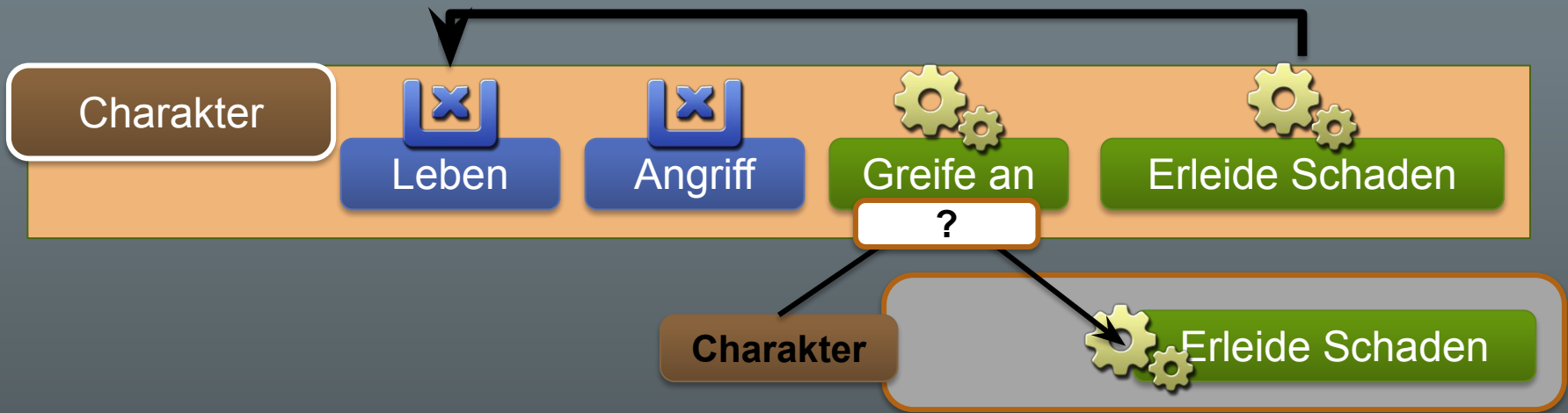
- Funktionsvariablen, die von außen gesetzt werden
- Ermöglichen einer Funktion unendliche Probleme gleicher Art zu lösen
- Verallgemeinern mehrerer Funktionen zu einer kann Fehler und Arbeitsaufwand verringern

```
public void ZumQuadrat( int parameter1 )
{
    int quadrat = parameter1 * parameter1 ;
    Debug.Log( quadrat );
}
public int zahl;
public void MyButtonFunction( )
{
    ZumQuadrat ( zahl );
}
```

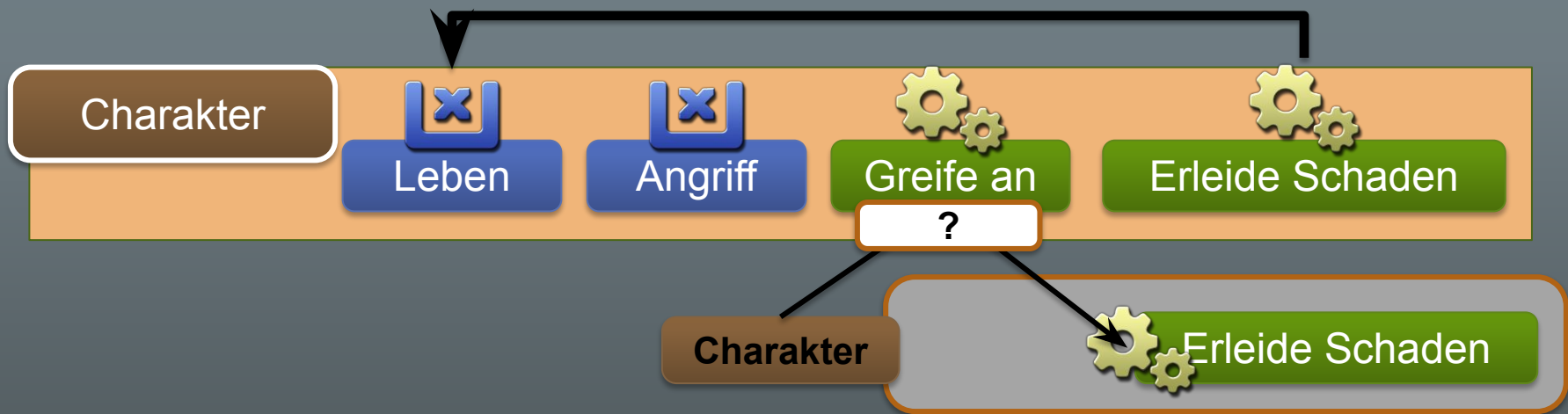
# Parameter

- Funktionsvariablen, die von außen gesetzt werden
- Ermöglichen einer Funktion unendliche Probleme gleicher Art zu lösen
- Verallgemeinern mehrerer Funktionen zu einer kann Fehler und Arbeitsaufwand verringern

```
public void ZumQuadrat( int parameter1 )
{
    int quadrat = parameter1 * parameter1 ;
    Debug.Log( quadrat );
}
public int zahl;
public void MyButtonFunction( )
{
    ZumQuadrat ( zahl );
}
```





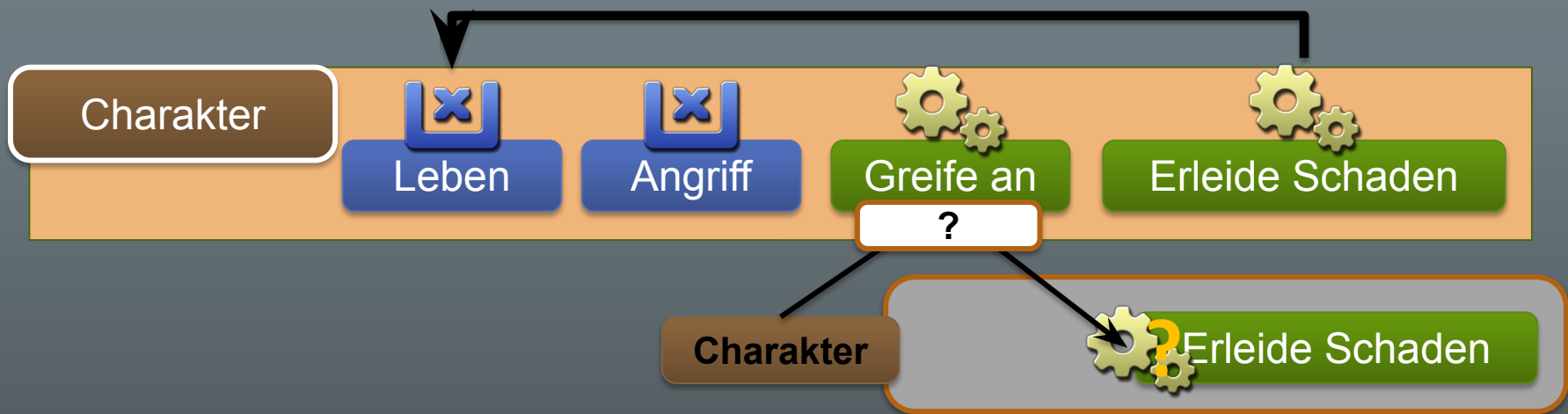


```
public int leben;  
public int angriff;
```

```
public void GreifeAn( Character targetChar )  
{  
    targetChar.ErleideSchaden( angriff );  
}
```

Objekt-Wechsel

```
public void ErleideSchaden( int schaden )  
{  
    leben = leben - schaden;  
}
```



```
public int leben;
public int angriff;

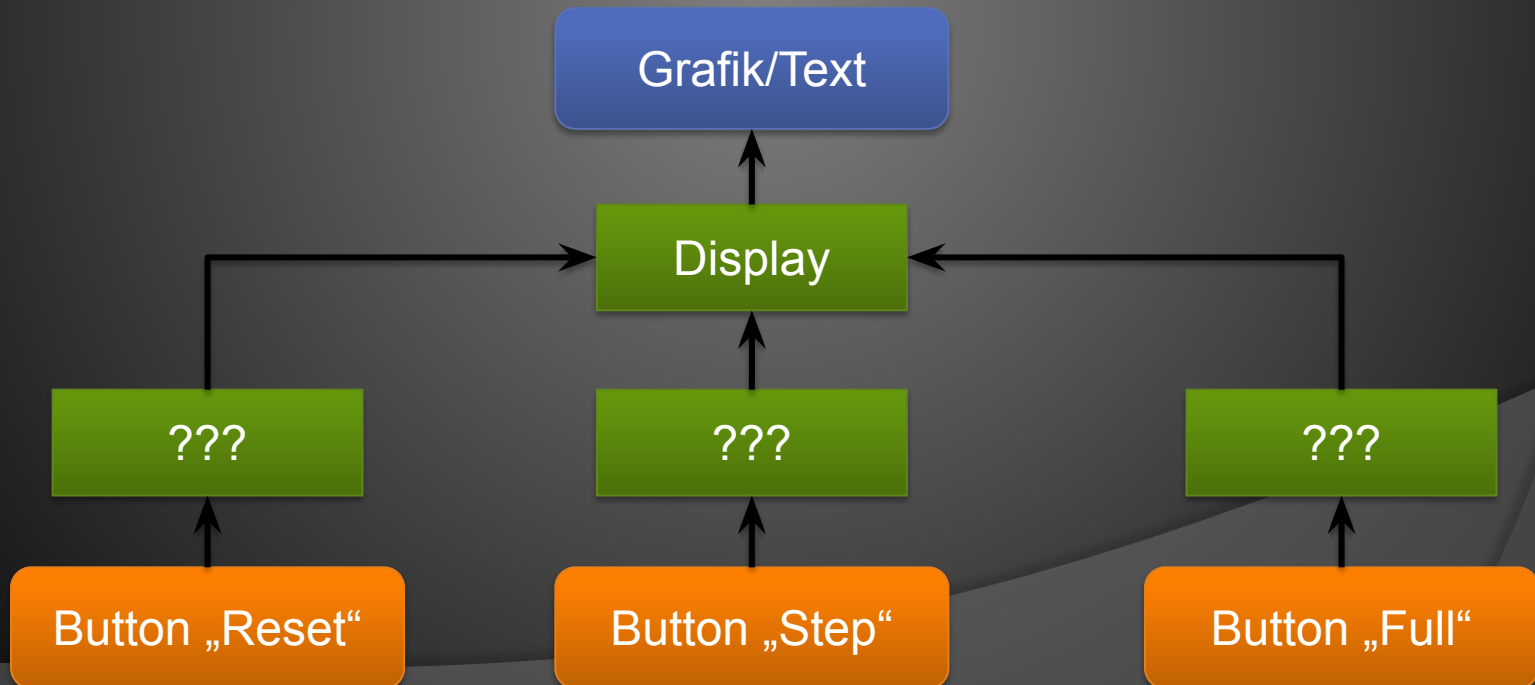
public int level;

public void GreifeAn( Character targetChar )
{
    int schaden = ( angriff * 4 ) * ( level / 2 );
    targetChar.ErleideSchaden( schaden );
}
```

## ⦿ Aufgabe 2.4    □ ProgressBarScene

Nutzen von Parametern zur besseren Funktionsnutzung

- ⦿ **Alle drei Buttons sollen am Ende dieselbe Funktion aufrufen, die dann den entsprechenden Parameter „in die Anzeige einträgt“**



# Komplexe / Referenz-Datentypen

- Datentypen sind generell definierte Klassen mit speziellen Funktionen.
- Die häufigen/einfachen Datentypen wurden dabei in den meisten höheren Prog.-Sprachen so vereinfacht, dass viel Schreibarbeit erspart bleibt.
- Normalerweise müsste jede Variable folgenderweise definiert werden

```
public void MyFct()  
{  
  
    int myNumber = new int();  
  
}
```

- Erst das Schlüsselwort „new“ veranlasst den Computer Speicher für diese Variable zu reservieren

# NULL

- Erfolgt dieser Aufruf nicht, gibt es zwar die Variable, aber sie hat keinen Wert.

Dieser nicht vorhandene Wert wird als **NULL** beschrieben

```
public UnityEngine.UI.Text myText;
public void MyFct()
{
    myText.text = „Hallo“;
    // Ist im Inspector für „myText“ kein entsprechendes Objekt
    // zugewiesen, wird der Fehler „NullReference“ ausgegeben.
    // Eine Referenz zu etwas nicht vorhandenem
}
```

# Vektoren

- Verknüpfter, Komplexer Datentyp

