PROGRAMMIER-GRUNDLAGEN

Variablen (Variables)

- **Variablen speichern Werte**
  - Werte können zugewiesen, abgerufen und verändert werden

- **Deklarieren (erstellen) mit [Typ] [Name];**
  - z.B.: *int zahl;*

- **Initialisieren/Zuweisen mit [Name] = [Wert];**
  - z.B.: *zahl = 5;*

- **Vergleichen mit** ==, >, <, >=, <=, *!=*

Typen (Types) - Beispiele

- **char (Character)** - ein einzelner Buchstabe/Zeichen
  - z.B.: ‘c’, ‘1’, ‘+’

- **string** - eine Reihe von Buchstaben/Zeichen
  - z.B.: *“Passwort1234”*
  - Können verknüpft werden:
    - *“Hallo ” + “Welt” + “!” == “Hallo Welt!”*
    - *“123” + “456” == “123456”*

- **bool (Boolean)** - ein Zustand, der entweder ‘wahr’ oder ‘falsch’ ist
  - *true*, *false*

- **int (Integer)** - eine natürliche Zahl
  - z.B.: *2*, *-4*, *209432*
  - Alle gängigen Rechenoperationen können ausgeführt werden
  - 123 + 456 == 579
  - 7 / 2 == 3

- **double (Double precision floating point number)** - eine reale Zahl
  - z.B.: *0.0*, *13.37*
  - Alle gängigen Rechnenoperationen können ausgeführt werden
  - 123.0 + 456.0 == 579.0
  - 7.0 / 2.0 == 3.5

- **List<Type>** - eine Liste von mehreren Elementen eines Typen
  - Initialisierung ist etwas anders als gewohnt. Wird später beim Thema ‘Objekte’ klarer werden.
  - z.B.: *List<int> zahlen = new List<int>() {2, 4, 8, 16};*

Methoden/Funktionen (Methods/Functions)

- Methoden führen einen definierten Code-Block aus

- Sie können einen optionalen Rückgabewert haben und optionale Parameter
  - [Rückgabetyp] [Methodenname]([Parametertyp] [Parametername]);

- Sie können selbst-erstellt sein…
  - z.B.: *int GetNumber(string input);*

- …oder in C# selbst, oder einer Bibliothek definiert sein
  - z.B.: *void Console.WriteLine(string s);*

- Methoden dienen dazu um das Programm zu strukturieren, lesbarer zu machen, und unnötige Code-Dopplung zu vermeiden

Zweige (Branches)

- Testen auf eine Bedingung, und führen das Programm an unterschiedlichen Stellen fort, je nachdem, ob die Bedingung zutrifft oder nicht

- *if (*[Bedingung1]*)*
  {
  // Wenn Bedingung1 zutrifft, führe diesen Code aus
  }
  else if ([Bedingung2]*)*
  {
  // Wenn Bedingung 1 nicht zutrifft, aber Bedingung2 zutrifft, führe
  // diesen Code aus
  }
  else
  {
  // Wenn weder Bedingung1, noch Bedingung2 zutreffen, führe diesen
  // Code aus
  }

Schleifen (Loops)

- Dienen zur Wiederholung des eingeschlossenen Codes
- Haben für gewöhnlich eine definierte Abbruchbedingung
- *while (*[Bedingung]*)*
  {
  // Wiederhole diesen Code solange die Bedingung zutrifft
  }

- *for (int i = 0; i < 10; i++)*
  {
  // Wiederhole diesen Code genau 10 mal. Mit der Variable i, sieht man
  // jederzeit, in welcher Wiederholung man sich befindet
  }

- *List<int> zahlen = new List<int>() {2, 4, 8, 16};*
  foreach (int zahl in zahlen)
  {
  // Führe diesen Code genau 1 mal für jeden Eintrag in der Liste aus.
  // z.B. Console.WriteLine(zahl); gibt nacheinander 2, 4, 8 und 16 aus.
  }

Klassen (Classes)

- Fassen zusammengehörende Variablen und Methoden in einem Paket zusammen

- Definition:

```
class [Name]
{
        // content
}
```

- **Member:**

- **Felder (Fields)** - zu der Klasse gehörende Variablen
  - z.B.: *public int zahl;*
  - Sollten in der Regel nicht verwendet werden -> siehe Properties

- **Eigenschaften (Properties)** – ähnlich wie Fields, aber mit mehr Funktionalitäten
  - Sollten fast immer verwendet werden statt Fields
  - Bieten separate Funktionen für „get“ (Wert Abfragen) und „set“ (Wert setzen)
  - z.B. *public int Zahl { get; private set; }*

- **Konstruktoren (Constructors)** – Wird immer aufgerufen, wenn ein Objekt erstellt wird
  - Werden deklariert wie Methoden, aber ohne Rückgabewert
  - Tragen immer den Namen der Klasse
  - Können optional Parameter entgegennehmen
  - z.B.:

```
public [ClassName]()
{
        // content
}
```

- **Methoden (Methods)** – Zu der Klasse gehörige Funktionen
  - z.B.:

```
public void DoStuff()
{
        // content
}
```

Objekte (Objects)

- Objekte sind Instanzen einer Klasse / Klassen sind Definitionen von Objekten
- Werden in der Regel mit dem Keyword „new“ instanziiert
  - *z.B. “MyClass foo = new MyClass();”*
- Objekte enthalten dann die in der Klasse definierte Properties/Fields und Methoden
  - Auf diese kann über einen „.“ zugegriffen werden
  - z.B. *„foo.DoStuff();“*

Enumerationen (Enums)

- Stellt einen Datentyp dar, der eine fest definierte Auswahl an Optionen bietet
- z.B.:

```
public enum Colors
{
        Black,
        White
}
```

- Zugriff über [EnumName].[MemberName]
  - z.B. *color = Colors.Black;*
- Sind im Hintergrund lediglich Zahlen, denen ein Name zugewiesen wird
  - Im Beispiel: Black == 0, White == 1

Schlüsselwörter (Keywords)

- **Zugriffsmodifikatoren (Access modifiers):**
  - Definieren, wer auf Klasse / die Property / die Methode Zugriff hat
  - **Public** – Zugriff von „außen“ und „innen“ erlaubt
  - **Private** – Zugriff nur innerhalb meiner Klasse erlaubt
  - **Protected** – Zugriff nur innerhalb meiner Klasse, oder einer abgeleiteten Klasse
  - **(Internal)** – (Zugriff nur innerhalb der Solution)
  - Wenn nichts angegeben ist, wird implizit „private“ angenommen (außer bei get/set – dort gilt der Zugriffsmodifikator des Properties)

- **Static:** Markiert Klassen / Properties / Methoden, die nicht an einem Objekt (einer Instanz) existieren, sondern nur einmal global in der Anwendung.

- **Abstract:** Markiert Klassen / Properties / Methoden, die in einer abgeleiteten Klasse überschrieben werden müssen

- **Virtual:** Markiert Klassen / Properties / Methoden, die in einer abgeleiteten Klasse überschrieben werden können

- **Override:** Zeigt an, dass eine Methode / Property explizit überschrieben werden soll

Vererbung (Inheritance)

- Klassen können von genau **einer** Eltern-/Basisklasse erben

- Definition:
  class [Name] : Parent
  {
  // content
  }

- Die Kind-Klasse erhält damit automatisch **alle** Eigenschaften und Funktionen (Properties/Fields + Methods) der Elternklasse

- Die Kind-Klasse kann auf public- und protected Member der Elternklasse direkt zugreifen, nicht jedoch auf privates. (siehe Keywords)

- Virtuelle und abstrakte Funktionen können mit *override* überschrieben werden. (siehe Keywords)

- Implementierungen in der Basisklasse können innerhalb der Überschreibung mit

- Ein Objekt vom Typ der Kind-Klasse ist auch immer automatisch vom Typ der Elternklasse

Interfaces

- Definition:
  public Interface [IName]
  {
  // content
  }

- Namen starten laut Konvention immer mit großem ‚i'

- Enthalten keine Definitionen, nur Deklarationen

- Können nicht instanziiert werden.

- Klassen können beliebig viele Interfaces implementieren
  - *public class Foo : IInterface1, IInterface2*

- Wenn eine Klasse ein Interface implementiert, *müssen* dort alle zugehörigen Properties und Methoden definiert werden.

- Interfaces können als Typ verwendet werden. In diesem Falle wird ein Objekt einer beliebigen Klasse erwartet, die dieses Interface implementiert
  - Z.B. Methode *void PrintAsciiArt(IAsciiArt ascii)*; nimmt ein beliebiges Objekt entgegen, welches IAsciiArt implementiert.
    In der Methode kann dann jedoch nur auf Member zugegriffen werden, die in dem Interface deklariert sind