# Klassen

# Klassen

- Durch die Verwendung von Typen wie string oder der Console- oder der Math-Klasse sind bereits einige Dinge zu Klassen implizit erlernt worden:
  - Methoden einer Klasse oder eines Objektes lassen sich mit dem Punkt-Operator ansprechen
  - Die Klassen Console oder auch Math bieten Methoden (Funktionen) an, die direkt über die Klasse ansprechbar sind (wie Console.WriteLine) – damit Methoden ohne ein Objekt einer Klasse ansprechbar sind, müssen diese statisch sein
  - Wenn ein Objekt einer Klasse erzeugt worden ist (in der Regel über den new-Operator, beim string auch durch Zuweisung in vereinfachter Form), können am Objekt ebenfalls Methoden aufgerufen werden, diese sind nicht statisch (z.B. Replace beim string)
  - Neben Methoden besitzt das Objekt auch Eigenschaften, die über den Punkt-Operator ebenfalls ansprechbar sind (z.B. Length am string-Objekt)
  - Jede Methode einer C#-Anwendung muß zu einer Klasse gehören, sogar die Einstiegspunkt-Funktion Main
  - Ebenso muß auch jede Variable zu einer Klasse gehören
  - Eine C#-Anwendung kann mehrere Klassen benutzen, dabei gruppieren Klassen zusammengehörige Funktionalität und machen das Suchen, Verwenden und Verstehen des Quelltextes verständlicher
  - Offensichtlich bieten Klassen auch etwas wie Nutzungsrechte, dies jedenfalls deutet das Schlüsselwort "public" vor den Klassen an, die bisher benutzt wurden

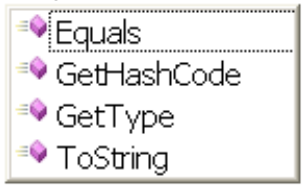
# Klassen

- Im einfachsten Fall kann eine Klasse nur aus Variablen unterschiedlichen Typs bestehen. Damit erweitert eine Klasse das Konzept der Felder, bei denen stets nur der gleiche Typ im Feld vorkommen darf.

```
class Person
{
    string m_name;
    string m_vorname;
    DateTime m_geburtstag;
    ushort m_groesse;
    double m_gewicht;
}
```

- Eine weitere Klasse im gleichen Namensraum kann die Klasse Person nutzen:

```
class PersonenTest
{
    static void main()
    {
        Person mm_paule;
        mm_paule.
    }
}
```




Hier werden über den Punktoperator aber nicht die Eigenschaften der Klasse Person angeboten.

Warum könnte das so sein?

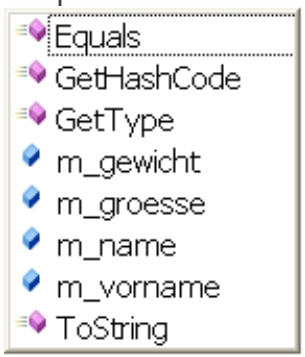
# Klassen

- Im einfachsten Fall kann eine Klasse nur aus Variablen unterschiedlichen Typs bestehen. Damit erweitert eine Klasse das Konzept der Felder, bei denen stets nur der gleiche Typ im Feld vorkommen darf.

```
class Person
{
    public string m_name;
    public string m_vorname;
    DateTime m_geburtstag;
    public ushort m_groesse;
    public double m_gewicht;
}
```

- Eine weitere Klasse im gleichen Namensraum kann die Klasse Person nutzen:

```
class PersonenTest
{
    static void main()
    {
        Person mm_paule;
        mm_paule.
    }
}
```




mit dem Schlüsselwort public werden die damit gekennzeichneten Eigenschaften nach "außen" hin sicht- und nutzbar. D.h. das Standardverhalten in Klassen ist private, was dementsprechend nicht extra gekennzeichnet werden muß, aber zur besseren Lesbarkeit doch sollte

# Klassen

- Im einfachsten Fall kann eine Klasse nur aus Variablen unterschiedlichen Typs bestehen. Damit erweitert eine Klasse das Konzept der Felder, bei denen stets nur der gleiche Typ im Feld vorkommen darf.

```
class Person
{
    public string m_name;
    public string m_vorname;
    DateTime m_geburtstag;
    public ushort m_groesse;
    public double m_gewicht;
}
```

- Eine weitere Klasse im gleichen Namensraum kann die Klasse Person nutzen:

```
class PersonenTest
{
    static void main()
    {
        Person mm_paule;
        mm_paule.m_name = "Schmidt";
    }
}
```

Hier bringt der Compiler folgenden Fehler.

Warum könnte dieser Fehler entstehen?

1 Verwendung der nicht zugewiesenen lokalen Variablen "mm_paule"

# Klassen

- Im einfachsten Fall kann eine Klasse nur aus Variablen unterschiedlichen Typs bestehen. Damit erweitert eine Klasse das Konzept der Felder, bei denen stets nur der gleiche Typ im Feld vorkommen darf.

```
class Person
{
    public string m_name;
    public string m_vorname;
    DateTime m_geburtstag;
    public ushort m_groesse;
    public double m_gewicht;
}
```

- Eine weitere Klasse im gleichen Namensraum kann die Klasse Person nutzen:

```
class PersonenTest
{
    static void Main()
    {
        Person mm_paule = new Person();
        mm_paule.m_name = "Schmidt";
    }
}
```

# Klassen

- Das Leben eines Objektes oder einer Instanz
  - Wenn eine Variable vom Typ einer Klasse definiert wird, ist dies immer eine Variable eines Verweistyps
  - Eine definierte Variable vom Typ einer Klasse ist zunächst einmal nicht initialisiert, der Verweis auf den Heap existiert noch nicht, der Heap-Speicher selber wurde noch nicht allokiert, das mit der Variable gemeinte Objekt oder die Instanz "lebt" noch nicht, es wird lediglich definiert, daß die Variable einmal auf ein derartiges Objekt verweisen wird
  - Eine Variable vom Typ einer Klasse wird erst durch den new-Operator zum gültigen Objekt, erst durch den new-Operator wird der Verweis gültig (Ausnahme bspw. beim Zuweisen eines String-Literals auf eine String-Variable, da wird ohne ein explizites new das Objekt von C# selbst automatisch erstellt)
- Public und Private
  - Mit diesen Schlüsselwörtern kann die Sichtbarkeit der Elemente einer Klasse nach außen hin gesteuert werden
  - Das ist einerseits aus Gründen des Zugriffsschutzes sinnvoll, aber es macht auch Sinn, bestimmte, nur intern zweckmäßig einzusetzende Funktionen oder Variablen, zur besseren Übersicht nicht nach außen offen zu legen
  - Public und Private können nicht nur auf die Elemente einer Klasse, sondern auch auf Klassen selbst angewendet werden. Dann gelten diese Schlüsselwörter mit Bezug auf den übergeordneten Namensraum

# Methoden

- Neben den Variablen, die in einer Klasse definiert werden können, kann die Klasse auch Funktionen enthalten.
- Bei den Elementen einer Klasse spricht man im Falle von Variablen häufig von Member-Variablen und bei den Funktionen von Methoden
- Methoden stellen – wie Funktionen in der prozeduralen Programmierung – eine Liste von Anweisungen dar, wie es in der Main-Methode als Programmeinstiegspunkt in bekannter Form der Fall ist
- Kennzeichen einer Methode sind
  - ein Name
  - der Zugriffsmodus (z.B. public)
  - die Definition, ob die Methode ohne gültiges Objekt benutzbar sein soll (static) oder nicht
  - einen Rückgabetyp (bei Main ist dass void, hierbei gibt es keine Rückgabe, bei der statischen Methode Math.sin wird hingegen der berechnete Sinus-Wert zurückgegeben), der Rückgabewert einer Methode kann als Operand weiter verwendet werden
  - eine Liste mit Übergabeparametern, die in der Methode benötigt werden (z.B. bei Math.Sin der Winkel, dessen Sinus-Wert berechnet werden soll), als Übergabeparameter können alle Datentypen verwendet werden

# Methoden

- Ein Beispiel für eine einfache, aber wohl überflüssige, Methode ist

```
class myMath
{
    public double Multipliziere(double a, double b)
    {
        return a * b;
    }
}
```

- Diese Methode heißt "Multipliziere", sie wird mit zwei Übergabeparametern a und b vom Typ double gerufen und gibt auch einen Wert vom Typ double zurück.
- Die Methode ist public, aber nicht static kann also nur von einem gültigen Objekt gerufen werden, ein Verwendungsbeispiel wäre:

```
class myMathTest
{
    static void Main()
    {
        myMath mm_obj = new myMath();
        double mm_Ergebnis = mm_obj.Multipliziere(3, 4);
    }
}
```

- Was wäre zu tun, um mit `myMath.Multipliziere(3, 4)` arbeiten zu können?
- Was würde `mm_Ergebnis = mm_obj.Multipliziere(mm_Ergebnis, mm_Ergebnis);` liefern?

# Kapselung

- Ein Punkt soll folgende Eigenschaften besitzen

```
class Punkt
{
    public double X;        //x-Koordinate
    public double Z;        //y-Koordinate
    public double Radius;   //Radius für Polardarstellung
    public double Winkel;   //Winkel für Polardarstellung
}
```

- Es soll möglich sein, alle Eigenschaften zu ändern – welche Problematik tut sich auf?
- Durch die Möglichkeit der gesteuerten Offenlegung bestimmter Elemente einer Klasse und das Verstecken anderer kann die Datenkonsistenz gesteigert werden.
  - Für die obige Aufgabe sollen die Eigenschaften nicht direkt geändert werden können, da hierbei in der Klasse keinerlei Kontrolle über deren Konsistenz mehr möglich ist
  - Der Zugriff auf die Eigenschaften wird über Zugriffsmethoden gesteuert, die die Konsistenz der Objektdaten sicherstellen, z.B. bei Änderung des Winkels oder des Radius wird X und Y intern aktualisert, bei Änderung von X oder Y entsprechend der Winkel und der Radius.
  - Dieses Prinzip der OOP wird als Kapselung bezeichnet, ein Beispiel für den Punkt könnte so aussehen:

# Kapselung

```
//Interne Member-Variablen
private double m_X;         //x-Koordinate
private double m_Y;         //y-Koordinate
private double m_Radius;    //Radius für Polardarstellung
private double m_Winkel;    //Winkel für Polardarstellung

public double getX() //Abfragen von X
{
    return m_X;
}
public void  setX(double X) //Setzen von X
{
    m_X = X;
    //Radius aktualisieren
    m_Radius = Math.Sqrt(m_X * m_X + m_Y * m_Y);
    //Winkel aktualisieren
    if (m_X != 0)
    {
        m_Winkel = m_Y == 0 ? 0 : Math.Atan(m_Y / m_X);
    }
    else
    {
        m_Winkel = m_Y < 0 ? -Math.PI : (m_Y == 0 ? 0 : Math.PI);
    }
}
```

```
Punkt mm_pkt = new Punkt();
mm_pkt.
```

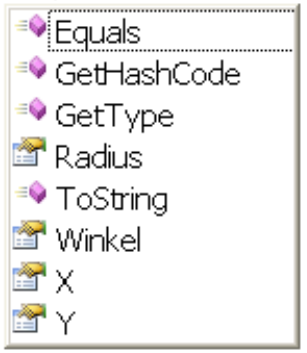

```
Punkt mm_pkt = new Punkt();
mm_pkt.setX(23.11);
double mm_test = mm_pkt.getX();
```

- Mit setX wird ein neuer X-Wert zugewiesen, mit getX abgefragt

# Kapselung bei Eigenschaften (Properties)

```
public double X
{
    get
    {
        return m_X;
    }
    set
    {
        m_X = value;
        //Radius aktualisieren
        m_Radius = Math.Sqrt(m_X * m_X + m_Y * m_Y);
        //Winkel aktualisieren
        if (m_X != 0)
        {
            m_Winkel = m_Y == 0 ? 0 : Math.Atan(m_Y / m_X);
        }
        else
        {
            m_Winkel = m_Y < 0 ? -Math.PI : (m_Y == 0 ? 0 : Math.PI);
        }
    }
}
```

```
Punkt mm_pkt = new Punkt();
mm_pkt.
```




```
Punkt mm_pkt = new Punkt();
mm_pkt.X = 23.11;
//...
double mm_test = mm_pkt.X;
```

- Die Eigenschaft kann sowohl auf der linken als auch auf der rechten Seite einer Zuweisung stehen – jeweils OHNE Klammer, die Syntax ist einfacher als bei setX und getX!

# Statische Member-Variablen

- Ebenso wie statische Methoden gibt es auch statische Member-Variablen
- Im Unterschied zu einer "normalen" Member-Variablen, bei der jede Instanz eine "eigene" Variable mit separatem Speicherbereich hat, werden statische Variablen nur einmal im Speicher gehalten
- Diese besitzen für alle Instanzen einer Klasse den gleichen Wert, sind also eine Art globale Klassenvariable. Statische Member-Variablen wie auch statische Methoden sind nur über die Klasse selbst anzusprechen, nicht aber über Instanzen
- Für den Entwurf sind statische Member-Variablen geeignet, im Klassenkontext generell gültige Daten bzw. Eigenschaften abzubilden, oder aber z.B. Zähler innerhalb der Klasse zu verwalten, um zu vermeiden, doppelte Identitäten bei laufenden Nummern in den Objekten zu bekommen
- Statisch heißt NICHT, daß der Wert nicht geändert werden kann, das Schlüsselwort für dieses Verhalten ist const, allerdings kann jede const-Variable als statisch betrachtet werden, da auch sie nur einmal im Speicher angelegt ist

# Beispiel einer einfachen Hilfsklasse

- Um eine Zufallszahl zu erzeugen, bietet C# die Klasse Random an. Es muß nun zuerst ein Objekt dieser Klasse erzeugt werden (mit dem new-Operator)
- Damit bei einem Aufruf unterschiedliche Zufallszahlen erzeugt werden, wird der Konstruktor mit einer Zahl gerufen, die die Anzahl der Millisekunden des Systemzeitgebers repräsentiert und daher in der Regel unterschiedlich ist
- Das Random-Objekt ist statisch und wird daher nur bei Erzeugen eines Objektes der Klasse Zufallszahl oder Verwenden von statischen Methoden angelegt.
- Über die statische Methode ErzeugeZahlZwischen wird vom statischen Random-Objekt immer eine weitere Zahl abgefragt und dafür gesorgt, daß eine Zahl aus dem Intervall zurückgegeben wird

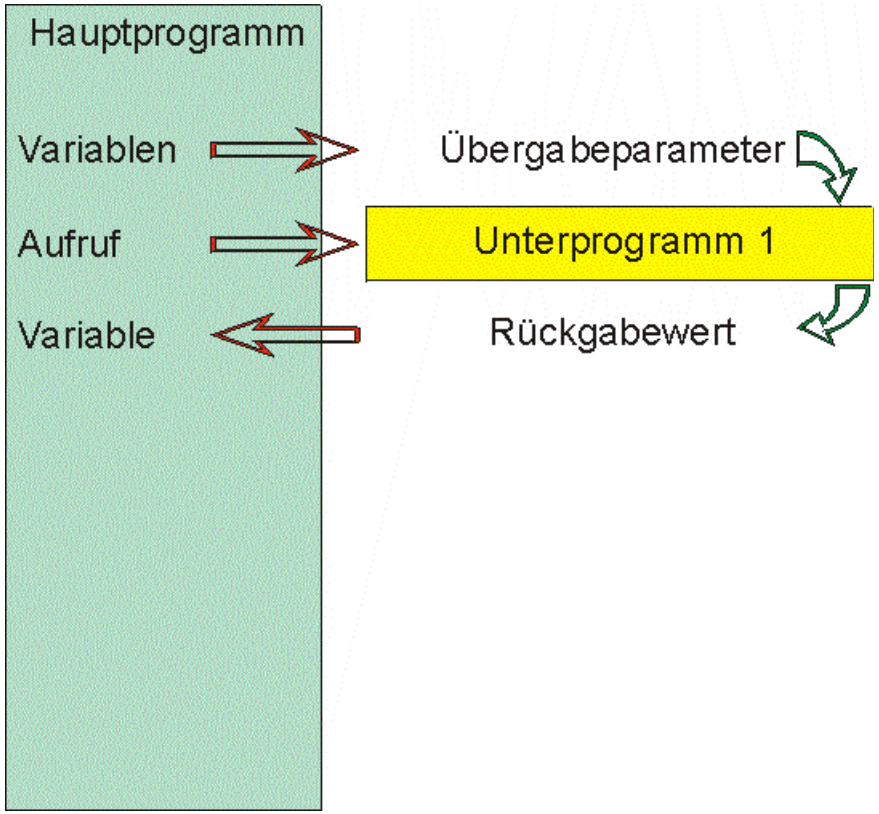
```
class Zufallszahl
{
    //Anfangspunkt für Pseudo-Zahl aus Ticks seit Mitternacht setzen...
    //da Objekt statisch, ruft Next... immer eine andere Zufallszahl ab
    static Random m_zufzahl = new Random(System.Environment.TickCount);

    //Statische Methode ohne Objekt rufbar
    public static int ErzeugeZahlZwischen(int von, int bis)
    {
        //Erzeugt double-Zufallszahl von 0,0 bis 1,0
        double mm_zz = m_zufzahl.NextDouble();
        //Zur Startzahl Differenz*Zufallszahl addieren... und zurückgeben
        return (int) (Math.Min(von,bis) + Math.Round(Math.Abs(bis - von) * mm_zz, 0));
    }
}
```

# Funktionen/Methoden - vertieft

# Prozedurale Strukturierung

- Bei allen höheren Programmiersprachen kann ein komplexes Programm in viele kleine Einheiten aufgeteilt werden. Das gängige Prinzip dazu ist Teile-und-Herrsche, bei dem kleine Funktionen zur Lösung eines Gesamtproblems kombiniert werden
- Die Modularisierung bei den prozeduralen Sprachen basiert auf Funktionen, die bestimmte Aufgaben übernehmen und die sich gegenseitig aufrufen




Beim Aufruf einer Funktion/eines Unterprogramms werden der Funktion Übergabemeter übergeben, die für die Verarbeitung in der Funktion relevant sind. Diese Parameter werden kopiert und stehen in der Funktion wie Variable zur Verfügung, die aber nicht im Funktionsblock, sondern in der Parameterliste deklariert sind und deren Initialisierung vom Hauptprogramm aus erfolgt, entweder über Literale oder Variablen

# Rückgabewert

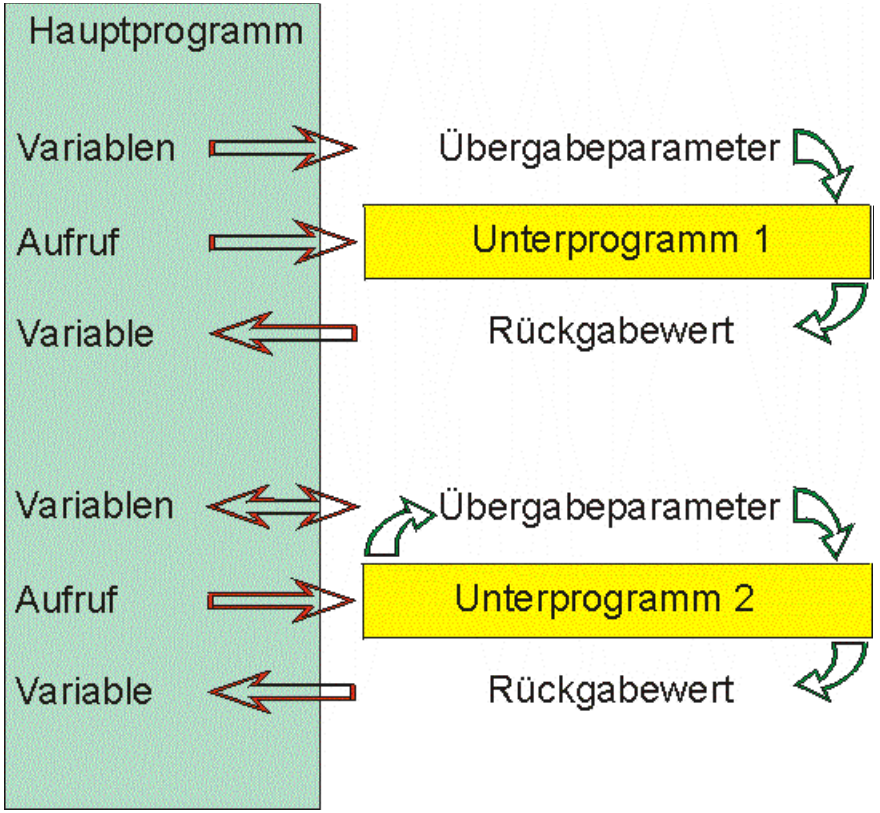
- Für den Rückgabewert einer Funktion können alle Datentypen verwendet werden – in der Funktion Multipliziere unten ist dies ein double

```
class myMath
{
    public double Multipliziere(double a, double b)
    {
        return a * b;
    }
}
```

- Ist ein Rückgabedatentyp festgelegt, erwartet der Compiler auch die Rückgabe eines diesem Datentyp entsprechenden Wertes, entweder als Variable oder Literal des entsprechenden Datentyps. Erfolgt keine Rückgabe (findet sich also kein return in der Methode), wirft der Compiler einen Fehler
- Der Rückgabewert einer Methode wird return als Wert übergeben und in der Syntax hinten angehängt. Es wird also der rechtsseitige Operand ausgewertet und als Literal an das Hauptprogramm zurückgegeben, gleichzeitig wird die Methode verlassen und die lokal definierten Variablen werden freigegeben
- Soll kein Wert zurückgegeben werden, wird als Datentyp void verwendet. In diesem Fall wird mit einem return ohne Operand die Methode verlassen

# Funktionsaufrufe

- Bei den höheren Programmiersprachen können Funktionen Werte übergeben werden, mit denen in der Funktion gerechnet wird (bspw. Math.Sin(Math.Pi/3))
- Jede Methode gibt am Ende exakt einen Wert zurück, der im Hauptprogramm weiter verarbeitet werden kann




Die Übergabeparameter werden (zur Verwendung in der Funktion) kopiert. Auch wenn im Hauptprogramm eine Variable an die Funktion übergeben wird, bleibt deren Wert daher im Hauptprogramm unverändert.

Die Funktion kann nur einen Rückgabeparameter zurückgeben, das reicht in manchen Fällen nicht aus bzw. würde mehrere Methoden erforderlich machen

Um mehr als einen Wert zurückgeben zu können, besteht die Möglichkeit, Werte in den Übergabeparametern nicht zu kopieren, sondern als Referenz auf den Variableninhalt des Hauptprogramms zu übergeben. In diesem Falle kann das Unterprogramm Überparameter ändern und auch die Variableninhalte im Hauptprogramm ändern sich.

# Call by value und call by reference

- Denkbar sei folgende Forderung an das Design einer Methode zur Berechnung der Nullstellen einer quadratischen Gleichung
  - drei double-Parameter sollen in Kopie übergeben werden ($ax^2+bx+c$)
  - Rückgabewert soll ein boolscher Wert sein, der den Erfolg der Methode anzeigt (Hier: reelle Nullstellen gefunden oder nicht)
  - die beiden möglichen x-Werte sollen in Form von Übergabeparametern übergeben werden
- C# bietet zwei Möglichkeiten dieser Rückgabe an: das Kennzeichnen eines Übergabemeters mit
  - ref (call by reference) hierbei wird nicht die Variable kopiert, sondern der Verweis auf die Variable wird an die Funktion übergeben – in der Handhabung des Parameters ändert sich nichts, nur daß der Wert eben auch im Hauptprogramm geändert wird. Die ref-Parameter müssen im Hauptprogramm definiert und initialisiert werden .
  - out (call by reference) auch hier wird nicht kopiert, allerdings braucht hier keine Initialisierung im Hauptprogramm zu erfolgen, die Deklaration aber weiterhin. Bei out verlangt der Compiler eine Zuweisung auf die mit out deklarierten Übergabeparameter, kontrolliert also die Rückgabe ähnlich wie beim return-Wert
- Ohne ref oder out werden Übergabeparameter kopiert (call by value)
- Die Funktion könnte dann etwa so aussehen:

```
bool berechneNullstellen(double a, double, b, double c, ref double x1,
ref double x2)
```

# Aufruf einer Methode mit call by value Parametern

- Die TryParse-Methode der Zahlen-Datentypen versucht, den im ersten Parameter übergebenen String als Zahl zu interpretieren. Gelingt dies, gibt die Methode den Wert true zurück und füllt gleichzeitig den zweiten Parameter (call by reference vom out-Typ) mit dem gefundenen Wert. Ist TryParse nicht in der Lage, eine Zahl zu ermitteln, wird false zurückgegeben und der zweite Parameter wird mit dem Standardwert initialisiert – eine kleine Methode zeigt die Anwendung:

```
static double inputDouble(string Eingabeaufforderung)
{
    double mm_w;
    string mm_s;
    do
    {
        //Aufforderung
        Console.Write(Eingabeaufforderung);
        //Eingabe
        mm_s = Console.ReadLine();
        //Dezimaltrenner Punkt und Komma, Komma ist Standard
        //daher Punkt durch Komma ersetzen
        mm_s = mm_s.Replace(".", ",");
    }
    //Austrittsbedingung, Schleife nur verlassen, wenn Wert ok
    //dabei schreibt TryParse den double Wert in die Variable mm_w
    while (false == double.TryParse(mm_s, out mm_w));
    return mm_w;
}
```

# Variable Parameterliste

- Wenn einer Methode mehrere Parameterwerte gleichen Typs ohne genaue Kenntnis der Anzahl übergeben werden sollen, kann dies mit einer als Feld definierten Parameterliste erfolgen. Dabei bewirkt das Schlüsselwort params, daß die übergebenen Werte im Feld Summanden angelegt werden – params darf daher nur den letzten Parameter kennzeichnen, andere dürfen Parameter vorher gelistet sein. Params kann wegen der variablen Länge niemals call by reference unterstützen

```
class myMath
{
    public static double Summiere(params double[] Summanden)
    {
        double mm_ret = 0;
        foreach (double summand in Summanden)
        {
            mm_ret += summand;
        }
        return mm_ret;
    }
}
```

- Der Aufruf ist folgenderweise möglich (Array-Aufruf auch ohne params):

```
double mm_summe1 = myMath.Summiere(2.4, 4.1, 12.0, 13.2);
double[] mm_array = {1.2, 3.4, 15.4, 16.1, 9, 11};
double mm_summe2 = myMath.Summiere(mm_array);
```

# Überladen von Methoden

- In vielen Fällen soll eine Methode für verschiedene Argumenttypen verfügbar sein. Ohne die Möglichkeit des Überladens müßte dafür für jede Methode ein neuer Name gefunden werden, z.B. WriteDouble2Line für das Schreiben eines Double-Wertes und WriteString2Line für das Schreiben eines String-Wertes
- In vielen Programmiersprachen bietet die Möglichkeit des Überladens eine elegante Methode zur vereinfachten Handhabung: Überladen ermöglicht die Verwendung des gleichen Methodennamens für verschiedene Arten von Parameterlisten, die im konkreten Fall gerufene Methode wird anhand der Datentypen der Parameterliste bestimmt:

```
Console.WriteLine(
```

▲1 von 19▼ void **Console.WriteLine** ()
Schreibt das aktuelle Zeichen für den Zeilenabschluss in den Standardausgabestream.

```
Console.WriteLine(
```

▲2 von 19▼ void Console.WriteLine (**bool value**)
**value:** Der zu schreibende Wert.

```
Console.WriteLine(
```

▲11 von 19▼ void Console.WriteLine (**string value**)
**value:** Der zu schreibende Wert.

- Da die Unterscheidung nur auf dem Datentyp und der Reihung der Parameter beruht, müssen diese unterschiedlich sein

# Überladen von Methoden

- Wie bereits beim Konstruktor bietet C# die Möglichkeit, Methoden gleichen Namens mit unterschiedlichen Parameterlisten zu definieren
- Neben unterschiedlichen Parameterlisten kann auch ein unterschiedlicher Rückgabedatentyp verwendet werden, siehe Beispiel

```
class Hoch01
{
  static int hochDrei(int zahl)
  {
    Console.WriteLine("Datentyp: int");
    return(zahl*zahl*zahl);
  }

  static double hochDrei(double zahl)
  {
    Console.WriteLine("Datentyp: double");
    return(zahl*zahl*zahl);
  }

  static void Main(string [] args)
  {
    Console.WriteLine("hochDrei({0})={1}",10,hochDrei(10));
    Console.WriteLine("hochDrei({0})={1}",1.5,hochDrei(1.5));
  }
}
```

Quelle: Lammarsch

# Grenzen und Möglichkeiten der Überladung

- Nur der Rückgabedatentyp reicht für die Unterscheidung von Methoden nicht aus
  - Mehrdeutigkeit durch implizite Umwandlungen bei Zuweisungen (z.B. bei Zuweisung auf eine Variable vom Typ double könnten sowohl die Methode mit einem int-Rückgabetyp oder einem short-Rückgabetyp verwendet werden
- Eine Unterscheidung anhand der Spezifikation ref, out oder params ist ebenso nicht möglich, da der Compiler dies nicht zwangsläufig am Quellcode erkennen kann, hier müßte der Compiler verstehen, was inhaltlich ablaufen soll (Variable soll in Funktion kopiert oder verändert werden ist nicht aus Quellcode unmittelbar ersichtlich)
- Wird die Reihenfolge der Parameter mit unterschiedlichen Datentypen geändert, ist dies eine zulässige Unterscheidung

- Es besteht weiterhin die Möglichkeit, auch Operatoren zu überschreiben. Bei eigenen Klassen kann damit bspw. Vergleichsoperatoren implementiert werden, die es ermöglichen, Objekte der Klasse zu vergleichen, wobei die Vergleichskriterien dann individuell wählbar sind, siehe nächste Folie

# Beispiel eines überladenen Operators

```
class RechtEck
{
    public int laenge, breite;

    public RechtEck(int laenge, int breite)
    {
        this.laenge = laenge;  //this ist notwendig, um
        this.breite = breite;  //Member-Variable anzusprechen
    }

    public static bool operator !=(RechtEck erstesRechteck, RechtEck zweitesRechteck)
    {
        return !(erstesRechteck == zweitesRechteck);
    }

    public static bool operator ==(RechtEck erstesRechteck, RechtEck zweitesRechteck)
    {
        if (erstesRechteck.laenge == zweitesRechteck.laenge)
        {
            if (erstesRechteck.breite == zweitesRechteck.breite)
            {
                return true;
            }
        }
        return false;
    }

    public static void Main()
    {
        RechtEck mm_r1 = new RechtEck(2, 1);
        RechtEck mm_r2 = new RechtEck(2, 1);
        Console.WriteLine("r1 == r2 liefert {0}", (mm_r1 == mm_r2).ToString());
    }
}
```

# Objekte und Instanzen

# Die Geburt eines Objektes

- Ein Variable, die ein Objekt einer Klasse enthält, ist immer eine Verweistyp-Variable
- Wenn bsp. eine Klasse cKlasse exisitiert, wird mit

```
cKlasse mm_newObj;
```

  auf dem Stack nur ein Verweis auf einen aktuell noch nicht gültigen Heap-Speicher in Form des Null-Wertes angelegt (siehe oben am Beispiel der Person)
- Damit die Variable mm_myObj nun auf ein gültiges Objekt der Klasse zeigen kann, muß dieses zunächst erzeugt werden, dies erfolgt mit Hilfe des new-Operators:

```
mm_newObj = new cKlasse();
```

- Auffällig an der Schreibweise ist, daß dem Klassennamen nach new ein Klammernpaar folgt, das wie ein Methodenaufruf ohne Parameter aussieht
- In der Tat geschieht genau dies, denn beim Erzeugen eines Objektes wird von C# im Hintergrund ein so genannter Konstruktor ausgeführt. In den behandelten Klassen gab es noch keine Konstruktoren, hier führt C# deshalb einen Standard-Konstruktor aus
- Es ist jedoch möglich, einen eigenen Konstruktor zu definieren, der bestimmte Initialisierungen der Membervariablen oder sonstiges vornimmt

# Konstruktor

```
class cKlasse
{
    private string m_wert;      //Member
    public string Wert          //Read-Only-Property ohne Set
    {
        get
        {
            return m_wert;
        }
    }
    //Explizit definierter Konstruktor
    public cKlasse()            //Wenn dieser nicht public wäre, könnte in cKlasseTest keine
    {                           //Instanz erzeugt werden
        m_wert = "Startwert";
    }
}

class cKlasseTest
{
    static void Main()
    {
        cKlasse mm_newObj; //Deklaration der Variablen
        mm_newObj = new cKlasse(); //Erzeugen der Instanz - Konstruktor-Aufruf
        Console.WriteLine(mm_newObj.Wert); //Ausgabe von "Startwert"
    }
}
```

# Konstruktor

- Durch setzen der Zugriffsrechte für den Konstruktor kann das Erstellen von Objekten gezielt gesteuert werden
- Konstruktoren haben ähnliche Signaturen wie Methoden, nur ohne Rückgabetyp, die Rückgabe ist intern immer das neue Objekt
- Wie bei Methoden können den Konstruktoren auch Parameter übergeben werden, die zur gezielten Initialisierung der Objekte herangezogen werden können
- Wird ein Konstruktor explizit implementiert, so erstellt C# keinen Standard-Konstruktor mehr, d.h. bei Erstellen nur eines Konstruktors mit Übergabeparametern kann ein Objekt dieser Klasse nur dann noch mit `new Klasse()` erstellt werden, wenn auch dieser Konstruktor explizit implementiert wurde
- Ist das nicht der Fall, so kann ein Objekt nur noch durch den Konstruktor mit Übergabeparametern erstellt werden – auch dies kann gezielt zur Sicherung der Datenkonsistenz und der Kapselung verwendet werden

# Konstruktor

```
class cKlasse
{
    private string m_wert;      //Member
    public string Wert          //Read-Only-Property ohne Set
    {                           //Wert kann von außen nur bei Konstruktion gesetzt und
        get                     //dann nur noch abgefragt werden
        {
            return m_wert;
        }
    }
    //Explizit definierter Konstruktor - mit Parameter-Übergabe
    public cKlasse(string wert) //Weil nur ein Konstruktor mit Parameter implementiert ist,
    {                           //kann keine Instanz nur mit new cKlasse() erzeugt werden!!
        m_wert = wert;
    }
}

class cKlasseTest
{
    static void Main()
    {
        cKlasse mm_newObj;                  //Deklaration der Variablen
        mm_newObj = new cKlasse("Paule");   //Konstruktor-Aufruf mit Parameter
        Console.WriteLine(mm_newObj.Wert); //Ausgabe von "Paule"
    }
}
```

# Das Leben einer Instanz

- Wenn eine Instanz einer Klasse vorhanden ist, können alle Eigenschaften der Klasse in dieser Instanz individuelle Werte annehmen – d.h. jede Instanz kann mit eigenen Eigenschaften versehen werden, nur statische Eigenschaften gelten für alle Instanzen, sind daher auch nur über Klasse.Eigenschaft erreichbar
- Beim Aufruf von Methoden stehen in den Methoden die Eigenschaften der Instanzen direkt zur Verfügung, d.h. beim Aufruf innerhalb der Klasse müssen keine Eigenschaften per Parameterliste weiter gegeben werden, sondern stehen unmittelbar am Objekt zur Verfügung
- Ist in einer Methode der Übergabeparameter so benannt wie eine Membervariable, muß bei einer Zuweisung auf die Membervariable das Schlüsselwort this als Platzhalter für das aktuelle Objekt / die aktuelle Instanz verwendet werden, um ein Verwechseln zu verhindern

```
private int Zaehler;
void setZaehler(int Zaehler)
{
    this.Zaehler = Zaehler;
}
```

# Verwendung von this

- this wird verwendet, um einen bestimmten Konstruktor zusätzlich vor Ausführung des aktuellen zu rufen oder um Member-Variablen zu setzen, die den gleichen Namen wie Übergabeparameter haben

```
class RechtEck
{
    int laenge, breite;

    public RechtEck(int laenge, int breite)
    {
        this.laenge = laenge;  //this ist notwendig, um
        this.breite = breite;  //Member-Variable anzusprechen
    }

    public RechtEck()
        : this(2, 1)     //Aufruf des oberen Konstruktors mit
                         //Standard-Initialisierung
    {
    }

    public void Ausgabe()
    {
        Console.WriteLine("Die Länge ist:   {0,9} \n" +
                          "Der Breite ist:  {1,9} \n", laenge, breite);
    }
}
```

# Das Lebensende einer Instanz

- Wird eine Variable am Ende eines Blocks zerstört, so trifft dies auch die Instanz, d.h. eine Variable, die auf ein Objekt einer Klasse zeigt
- Beim Zerstören einer Verweisvariablen wird der Stackspeicher wieder freigegeben, der Heap bleibt zunächst erhalten – für die Verweistypen bedeutet dies, daß eine Variable, die auf den Heap verweist, nicht mehr vorhanden ist und sofern keine andere Variable auf den gleichen Heap zeigt, ein Speicherleck entstanden ist. Im Gegensatz zu C++ durchsucht in C# ein Programm den Heap nach derartigen unreferenzierten Bereichen und räumt diese auf, dieses Programm heißt Garbage Collector (Müllsammler), es kann über die Klasse GC.Collect() kontrolliert gestartet werden
- Beim Zerstören einer Instanz läuft – ähnlich wie beim Erzeugen – im Hintergrund automatisch eine Methode ab, die u.a. die Member-Variablen der Instanz aufräumt – der Destruktor. Der Destruktor sieht wie ein Konstruktor aus, nur beginnt er mit einer Tilde

```
~cKlasse() //Destruktor der Klasse
{
    Console.WriteLine("Ich scheide nun dahin....");
}
```

# Destruktor

- Analog zum Konstruktor kann auch ein Destruktor implementiert werden, um notwendige Aufräumarbeiten selber zu organisieren und nicht dem Standard-Destruktor von C# das Feld zu überlassen
- Im Unterschied zum Konstruktor gibt es aber nur einen Destruktor. Der Destruktor kann nur vom GC und nicht explizit wie ein Konstruktor aufgerufen werden, daher gibt es keine Zugriffsrechte und auch keine Möglichkeit zur Parameterübergabe
- In Sprachen wie C++ wird der Destruktor häufig für das Aufräumen des Speichers bei Verweistypen-Membervariablen verwendet, da es keinen Garbage Collector gibt. In C# braucht man sich um derartige Speicherlecks nicht zu kümmern, da verwaister Speicher durch den GC freigegeben wird, daher werden Destruktoren in C# für diese Aufgabe nicht herangezogen und sind insgesamt eher selten anzutreffen
- Denkbar sind Destruktoren bei Klassen, bei denen sichergestellt werden soll, daß bestimmte Zustände wieder hergestellt werden, z.B. eine Datenbank oder eine Datei soll bei Lebensende des Objektes wieder geschlossen werden. Hier greift häufig das Kapselungs-Prinzip, bei dem Datenbank- oder Dateiverbindung selber über Klassen realisiert sind und deren Objekte sich bereits um diese Aufgaben kümmern, d.h. bei Zerstören des Hauptobjektes werden auch die Membervariablen zerstört und das sorgt dann für entsprechende Wirkung