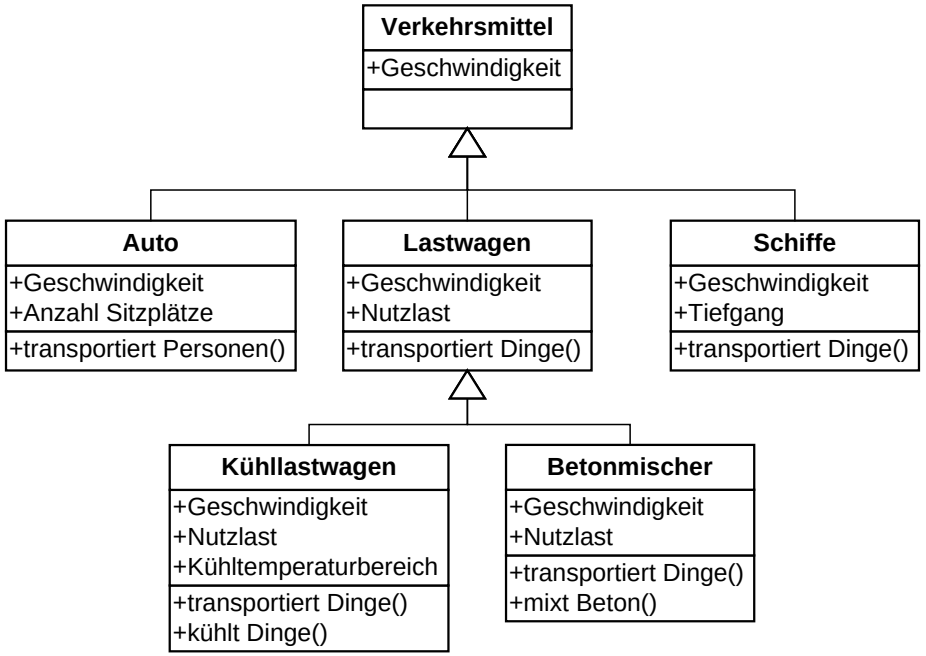
Klassen, Methoden, Eigenschaften und Objekte

Vererbung

- Klassen in der objektorientierten Programmierung stellen zunächst einmal nur Baumuster für die Erzeugung von Instanzen/Objekten dar
- Baumuster können – ähnlichen den Begrifflichkeiten der realen Welt – generalisiert bzw. abstrahiert werden. Dabei werden die Objekte durch die Relation "ist ein" in Beziehung gesetzt – wie im Skript "Einleitung OOP" schon vorgestellt
- Dieses Konzept wird in der OOP in Form von abgeleiteten Klassen aufgegriffen. Eine abgeleitete Klasse "erbt" von der Basis- oder Elternklasse alle Methoden und Eigenschaften in Abhängigkeit von den Zugriffsrechten



Vererbung

- Eine Klasse kann von einer anderen Klasse erben, wenn folgende Syntax verwendet wird:

```
class abgeleiteteklasse : Basisklasse
```

- Dabei ist Basisklasse eine bereits bestehende Klasse, die im aktuellen Namespace bzw. durch die using-Direktive verfügbar ist. In C++ wird auch eine Mehrfachvererbung angeboten, dies ist aber auf Klassenebene in C# nicht möglich
- Die Zugriffe auf Klassenelemente (Eigenschaften und Methoden) der Basisklasse werden durch die Schlüsselwörter public, private und protected geregelt:

	private (Basis)	protected (Basis)	public (Basis)
private (abgeleitet)	nicht verfügbar	verfügbar und als private einzuschränken	verfügbar und als private einzuschränken
protected (abgeleitet)	nicht verfügbar	verfügbar und als protected bereitzustellen	verfügbar und als protected einzuschränken
public (abgeleitet)	nicht verfügbar	verfügbar, aber nicht public bereitzustellen	verfügbar und auch bereitstellbar

Konvertierung von Instanzen zwischen Klassenhierarchien

- Das Objekt mm_obj wird vom Typ abgeleiteteKlasse erstellt
- Da mm_obj von Basisklasse erbt, kann die Variable mm_obj auf eine Variable vom Typ Basisklasse konvertiert werden (vgl. zu int auf double)
- Bei der Erzeugung der Variablen wird der Konstruktor der abgeleiteten Klasse gerufen, der im Beispiel eine Membervariable setzt
- Wird nun eine Methode der Basisklasse gerufen, bleiben die Werte der Membervariablen erhalten
- Bei der Konstruktion der Objekte kann sogar gleich schon auf eine Variable vom Typ Basisklasse zugewiesen werden, ohne daß sich das Ergebnis hier ändern würde

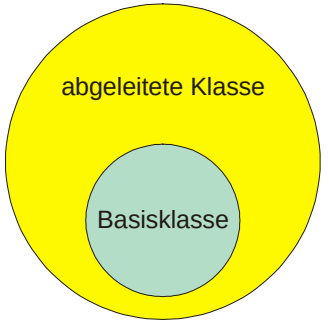
```
class Basisklasse
{
    public string MeinName;
    public Basisklasse()
    {
        MeinName = "Ich bin die Basisklasse!";
    }

    public void sagHallo()
    {
        Console.WriteLine(MeinName);
    }
}
class abgeleiteteklasse : Basisklasse
{
    public abgeleiteteklasse()
    {
        MeinName = "Ich bin die abgeleitete Klasse!";
    }

    static void Main(string[] args)
    {
        abgeleiteteklasse mm_obj = new abgeleiteteklasse();
        //ein abgeleitetes Objekt kann immer in die
        //Basisklasse konvertiert werden...
        Basisklasse mm_base = mm_obj;
        mm_base.sagHallo();
    }
}
```

Konvertierung zwischen Klassen

- Eine Instanz einer abgeleiteten Klasse besitzt alle Methoden und Eigenschaften der Basisklasse, die Basisklasse selbst kann durch weitere Methoden und Eigenschaften erweitert werden
- Wird ein Objekt der abgeleiteten Klasse auf eine Variable vom Typ der Basisklasse konvertiert, können auch nur mehr die Methoden und Eigenschaften der Basisklasse angesprochen werden, da die Basisklasse die abgeleitete Klasse nicht kennt
- Liegt eine Variable vom Typ der Basisklasse vor, so kann diese wieder zurück in die abgeleitete Klasse verwandelt werden, an dieser Stelle muß aber der cast-Operator eingesetzt werden



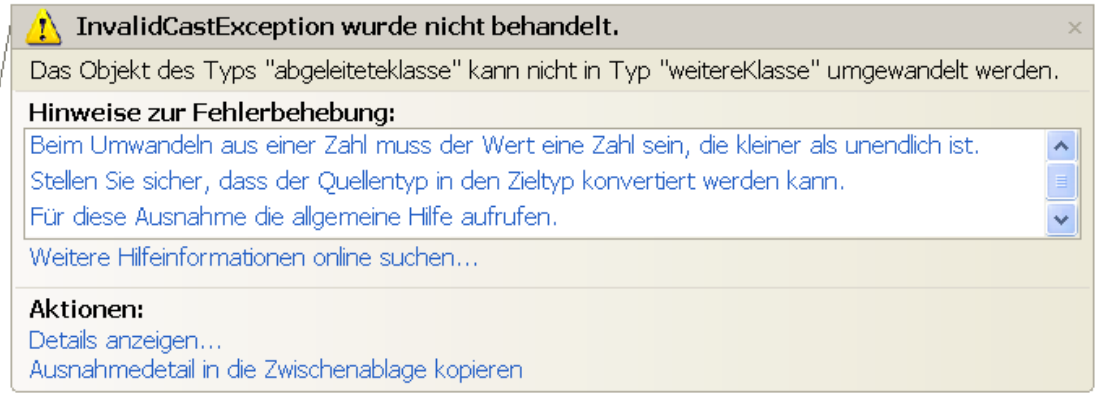

```
abgeleiteteklasse mm_obj = new abgeleiteteklasse();
//ein abgeleitetes Objekt kann immer in die
//Basisklasse konvertiert werden...
Basisklasse mm_base = mm_obj;
//und wieder zurück in die abgeleitete Klasse
abgeleiteteklasse mm_obj2 = (abgeleiteteklasse) mm_base;
```

Konvertierung zwischen Klassen

- Sind die Klassen nicht ineinander überführbar, wird ein Laufzeitfehler geworfen:

```
public abgeleiteteklasse()
{
    MeinName = "Ich bin die abgeleitete Klasse!";
}

static void Main(string[] args)
{
    abgeleiteteklasse mm_obj = new abgeleiteteklasse();
    //ein abgeleitetes Objekt kann immer in die
    //Basisklasse konvertiert werden...
    Basisklasse mm_base = mm_obj;
    //und nun in eine andere Klasse...
    //das wirft eine Exception, da diese Konvertierung nicht möglich ist
    weitereKlasse mm_obj3 = (weitereKlasse)mm_base;
```



- Dies kann durch eine Sicherheitsmaßnahme abgefangen werden
  - über IsSubclassOf kann geprüft werden, ob ein Objekt einer abgeleiteten Klasse entstammt

```
//Ist Objekt-Klasse von mm_base eine Unterklasse von weitereKlasse?
if (mm_base.GetType().IsSubclassOf(typeof(weitereKlasse)))

//Ist weitereKlasse eine Unterklasse der Objekt-Klasse von mm_base?
if (typeof(weitereKlasse).IsSubclassOf(mm_base.GetType()))

//ist Objekt mm_base vom Typ abgeleiteteKlasse??
if (mm_base.GetType() == typeof(abgeleiteteklasse))
```

Vererbung von Methoden (Polymorphie)

- In einer Klassenstruktur können Methoden in Unterklassen anders implementiert werden als in der Basisklasse. Ein typischer Vertreter dieser Methoden ist die Methode ToString(), die in Abhängigkeit von der Klasse verschiedenartig implementiert werden kann bzw. bei den C#-Basisklassen schon ist.
- Durch die Vererbung werden Methoden der Basisklasse an die abgeleitete Klasse vererbt und müssen nicht erneut implementiert werden. Ist dies jedoch erforderlich, weil die Methode in der abgeleiteten Klasse ein anderes Verhalten besitzen soll, so muß die Methode überschrieben werden
- Bei der Vererbung von Methoden können zwei verschiedene Möglichkeiten der Überschreibung realisiert werden:
  - die Überlagerung – hierbei wird die Methode in der Basisklasse als virtual deklariert, in abgeleiteten Klassen wird an Stelle von virtual das Schlüsselwort override verwendet. Wird ein Objekt der abgeleiteten Klasse in ein Basisklassen-Objekt konvertiert, wird trotzdem die Methode aufgerufen, die der "wirklichen" Klasse des Objektes entspricht. Dies wird als polymorphes (vielgestaltiges) Verhalten bezeichnet – die Zugriffsrechte müssen gleich sein
  - die Verdeckung – hierbei wird ebenfalls die Methode der Basisklasse in der abgeleiteten Klasse neu definiert, aber quasi überdeckt, da das virtuelle Verhalten nicht auftritt. Bei Konvertierung in die Basisklasse wird im Gegensatz zur Überlagerung die Methode der Basisklasse gerufen (übrigens auch ohne Fehler, nur mit Warnung, wenn override vergessen wird)

Verdecken und Überlagern von Methoden

```
class Basisklasse
{
    public string MeinName;
    public Basisklasse()
    {
        MeinName = "Ich bin die Basisklasse!";
    }
    public void sagHallo()
    {
        Console.WriteLine("In Basisklasse: " + MeinName);
    }
    public virtual void sagHalloVirtual()
    {
        Console.WriteLine("In Basisklasse: " + MeinName);
    }
}

class abgeleiteteklasse : Basisklasse
{
    public abgeleiteteklasse()
    {
        MeinName = "Ich bin die abgeleitete Klasse!";
    }
    new public void sagHallo() //Überdecken der Methode
    {
        Console.WriteLine("In abgeleitete Klasse: " + MeinName);
    }
    public override void sagHalloVirtual() //Überlagern der Methode
    {
        Console.WriteLine("In abgeleitete Klasse: " + MeinName);
    }
}
```

Verdecken und Überlagern von Methoden

- Im Falle einer überdeckten Methode wird die Methode des aktuellen Objekt-Datentyps gerufen
- Bei virtuellen Methoden wird zur Laufzeit die Methode der vom aktuellen Objektdatentyp am weitesten nach unten gültigen liegenden Basisklasse (bis einschließlich der eigenen Klasse) gerufen

```
static void Main(string[] args)
{
    abgeleiteteklasse mm_obj = new abgeleiteteklasse();
    Basisklasse mm_base = mm_obj;
    mm_obj.sagHallo();
    mm_base.sagHallo();
    mm_obj.sagHalloVirtual();
    mm_base.sagHalloVirtual();
}
```

Ausgabe ist hier:

```
In abgeleitete Klasse: Ich bin die abgeleitete Klasse!
In Basisklasse: Ich bin die abgeleitete Klasse!
In abgeleitete Klasse: Ich bin die abgeleitete Klasse!
In abgeleitete Klasse: Ich bin die abgeleitete Klasse!
```

Verdecken und Überlagern von Methoden

- Wird eine weitere Klasse von der abgeleiteten Klasse abgeleitet, gelten die Methoden der Elternklasse – also hier der Klasse "abgeleiteteKlasse"

```
class Enkelkind : abgeleiteteklasse
{
    public Enkelkind()
    {
        MeinName = "Ich bin Enkelkind der Basisklasse";
    }
}
```

```
static void Main(string[] args)
{
    Enkelkind mm_ek = new Enkelkind();
    mm_ek.sagHallo();
    mm_ek.sagHalloVirtual();
}
```

Ausgabe ist hier:

```
In abgeleitete Klasse: Ich bin Enkelkind der Basisklasse
In abgeleitete Klasse: Ich bin Enkelkind der Basisklasse
```

Verdecken und Überlagern von Methoden

- Mit base kann in einer Methode gezielt die Methode der Basisklasse gerufen werden, dies ist alternativ auch durch Konvertierung des this-Objektes auf den entsprechenden Typ möglich.

```
class Enkelkind : abgeleiteteklasse
{
    public Enkelkind()
    {
        MeinName = "Ich bin Enkelkind der Basisklasse";
    }
    public void baseTest()
    {
        ((Basisklasse)this).sagHallo();
        ((abgeleiteteklasse)this).sagHallo(); //oder mit base ...
        base.sagHallo();
        ((Basisklasse)this).sagHalloVirtual();
        ((abgeleiteteklasse)this).sagHalloVirtual(); //oder mit base ...
        base.sagHalloVirtual();
    }
}
```

```
static void Main(string[] args)
{
    Enkelkind mm_ek = new Enkelkind();
    mm_ek.baseTest();
}
```

Ausgabe ist hier:

```
In Basisklasse: Ich bin Enkelkind der Basisklasse
In abgeleitete Klasse: Ich bin Enkelkind der Basisklasse
In abgeleitete Klasse: Ich bin Enkelkind der Basisklasse
In abgeleitete Klasse: Ich bin Enkelkind der Basisklasse
In abgeleitete Klasse: Ich bin Enkelkind der Basisklasse
In abgeleitete Klasse: Ich bin Enkelkind der Basisklasse
```

Welche Art der Methoden-Vererbung wurde hier verwendet?

```
class ElternKlasse
{
  public ElternKlasse()
  {
    Console.WriteLine("Eltern Konstruktor");
  }

  public void drucken()
  {
    Console.WriteLine("Hier: Eltern Klasse");
  }
}
```

```
class Kind01 : ElternKlasse
{
  public Kind01()
  {
    Console.WriteLine("Kind Konstruktor");
  }

  public void drucken()
  {
    Console.WriteLine("Hier: Kind Klasse");
  }

  static void Main(string [] args)
  {
    Kind01 kind = new Kind01();
    kind.drucken();
    ((ElternKlasse)kind).drucken();
  }
}
```

- Die Ausgabe dieses Programms liefert

```
Eltern Konstruktor
Kind Konstruktor
Hier: Kind Klasse
Hier: Eltern Klasse
```

Quelle: Lammarsch

Vererbung und Verdecken von Membervariablen

- Besitzt eine Klasse eine Membervariable und wird die gleiche Variable in einer abgeleiteten Klasse erneut definiert (durch `public new int mm_i`), so wird damit die Variable der Basisklasse analog zum Verhalten bei verdeckten Methoden verdeckt

```
class Basisklasse
{
    public int mm_i;
    public Basisklasse()
    {
        mm_i = 123;
    }
}

class abgeleiteteklasse : Basisklasse
{
    public new int mm_i;
    public abgeleiteteklasse()
    {
        mm_i = 321;
    }

    static void Main(string[] args)
    {
        abgeleiteteklasse mm_obj = new abgeleiteteklasse();
        Console.WriteLine("mm_i in abgeleiteter Klasse: {0}\n" +
            "und in Basisklasse: {1}", mm_obj.mm_i, ((Basisklasse)mm_obj).mm_i);
    }
}
```

abstrakte Methoden / abstrakte Klassen

- Eine abstrakte Methode wird wie folgt definiert (natürlich auch möglich mit verschiedenen Rückgabetypen und Übergabeparametern):

```
abstract protected void abstractMethod();
```

- Eine abstrakte Methode ist eine Methode ohne Implementierung – sie kann damit nicht ausgeführt werden. Eine abstrakte Methode in einer Klasse macht die Klasse, in der sie sich befindet, ebenfalls abstrakt, da die Methode den Objekten nicht zur Verfügung stehen kann. Dies ist in der Klassendefinition anzugeben:

```
abstract class Basisklasse
{
```

- Da Methoden nicht ausführbar sind und damit auch die Klasse abstrakt wird, können keine Instanzen/Objekte mit new erzeugt werden. Eine abstrakte Klasse kann damit nur
  - in die Rolle einer Basisklasse treten oder
  - statische Methoden anbieten ohne jemals instantiiert zu werden (dies ist aber auch mit einem private Konstruktor machbar)
- Wird von einer abstrakten Klasse abgeleitet, so bleiben auch die abgeleiteten Klassen abstrakt, wenn sie nicht die abstrakten Methoden implementieren. Bei der Implementierung von abstrakten Basisklassen-Methoden gelten ähnlich Regeln wie bei virtuellen Methoden (Schlüsselwort override), auch hier muß der Zugriffsmodus erhalten bleiben, darf aber nicht private sein, da sonst keine Vererbung stattfinden kann, private abstract und auch static abstract gibt es somit nicht

Versiegelte Methoden und Klassen

- Durch die virtual-Methoden findet eine dynamische Bindung der geeigneten Methoden statt, die zeitintensiver als eine statische Methodenbindung ist. (Virtual ist langsamer als verdeckte Methoden, da zur Laufzeit (dynamisch) auf die aktuelle Klasse geprüft wird)
- Alle von einer virtual Methode erstellten Implementierungen in abgeleiteten Klassen sind ebenfalls virtuell
- Soll diese Linie unterbrochen werden, können Methoden gezielt versiegelt werden. Ab dieser Klassenstufe verhalten sich die Methoden dann wieder wie "normale" Methoden, können aber weiterhin in abgeleiteten Klassen verdeckt werden

```
public sealed override void sagHalloVirtual() //Überlagern der Methode
{                                             //und weitere Überlagerungen verhindern
    Console.WriteLine("In abgeleitete Klasse: " + MeinName);
}
```

- Auch Klassen lassen sich versiegeln, ein Ableiten von versiegelten Klassen ist dann nicht mehr möglich
- Abstrakte Klassen lassen sich logischerweise nicht versiegeln (Anwendungsfall dafür wäre nur eine statische Klasse, von der nicht abgeleitet werden soll)

Eigenschaften in Feldform (Indexer)

- Ein Indexer ist eine Eigenschaft (wie im Skript Klassen und Methoden dargestellt), die ähnlich wie ein Feld agiert.
- Die Eigenschaft wird dazu definiert wie beispielsweise:

```
class Messwerte //Aufzählklasse für 100 Messwerte
{
    //Variable zum Halten des Feldes...
    private double[] m_Messwerte = new double[100];
    //Öffentliche Eigenschaft - zu beachten ist this!!
    public double this[int index]
    {
        get
        {
            //....
        }
        set
        {
            //....
        }
    }
}
```

- In dieser Form können die Klassen nicht mit foreach durchlaufen werden!
- Es gibt bessere Möglichkeiten, Messwerte, insbesondere bei veränderlichen Anzahlen von Meßwerten abzubilden (z.B. als List oder Dictionary)

Eigenschaften in Feldform (Indexer) - Beispiel

- Die private Member-Variable ist ein Integer-Wert, der 4byte hat, also 32bit. Über den Indexer, also eine Eigenschaft mit Feld-Charakter wird ein Feld von 32 Booleschen Werten adressiert, die in m_bitmuster gehalten werden
- Beim Auslesen eines bestimmten Index wird eine Maske um soviel Stellen bitverschoben, dass die entsprechende 2er Potenz entsteht.
- Diese wird anschließend binär-und mit dem gesetzten Bitmuster verglichen

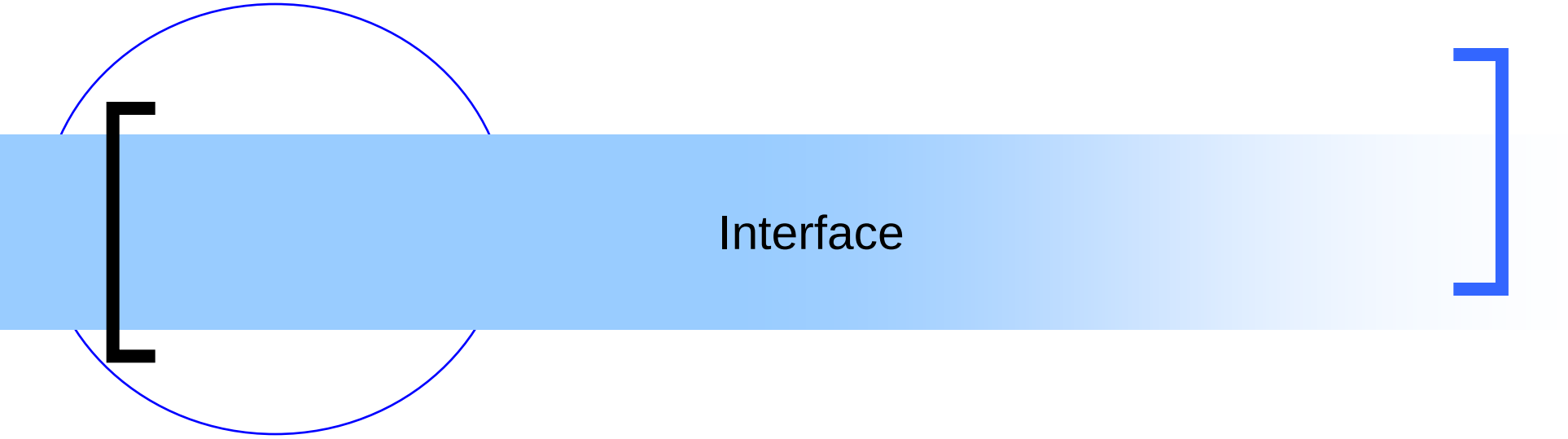
```
class Bitmuster
{
    private int m_bitmuster; //4byte = 32bit darstellbar
    public bool this[int index]
    {
        get
        {
            if ((index >= 0) && (index < 32))
            {
                int mm_maske = 1;
                //Auf Stelle des Index verschieben
                mm_maske = mm_maske << index;
                return (m_bitmuster & mm_maske) == 0 ? false : true;
            }
            else {/* Fehler werfen!! */ }
            return false;
        }
```

Eigenschaften in Feldform (Indexer) - Beispiel

- Für das Schreiben der Eigenschaft wird die set-Methode verwendet. Hier wird die gleiche 2er Potenz mittels Bit-Shift erstellt. Wenn der Wert gesetzt wird, wird mit bitweise oder verknüpft, bei Setzen auf False wird die Maske invertiert und mit bitweisem oder mit dem Bitmuster verglichen

```
set
{
    if ((index >= 0) && (index < 32))
    {
        int mm_maske = 1;
        //Auf Stelle des Index verschieben
        mm_maske = mm_maske << index;
        if (value) //Übergabeparameter true/false
        {
            //Bit setzen
            m_bitmuster = m_bitmuster | mm_maske;
        }
        else
        {
            //Bit entfernen
            mm_maske = ~mm_maske; //Maske invertieren, 0 steht an entsprechender Stelle
            m_bitmuster = m_bitmuster & mm_maske;
        }
    }
    else {/* Fehler werfen!! */ }
}
```

```
static void Main()
{
    Bitmuster mm_bm = new Bitmuster();
    mm_bm[7] = true; //set-Setzen des Wertes
    if (mm_bm[7] == true) //get-Abfrage des Wertes
    {
        Console.WriteLine("Schalter 7 ist gesetzt!");
    }
    else
    {
        Console.WriteLine("Schalter 7 ist nicht gesetzt!");
    }
}
```


Interface

Was sind Interfaces

- In C# gibt es in der Klassenhierarchie keine Mehrfachvererbung. Dadurch werden Probleme wie Namensgleichheiten und damit –konflikte umgangen, auch ist die Konvertierung zwischen Objekten teilweise problematisch und wenig nachvollziehbar
- In C# wird anstelle der Mehrfachvererbung in Klassenhierarchien das Interface angeboten. Ein Interface (deutsch: Schnittstelle) definiert – ähnlich wie eine abstrakte Klasse – Methoden und Eigenschaften ohne Implementierung
- Da Interfaces keine Implementierung besitzen, spricht man beim "Ableiten von Interfaces" vom Implementieren eines Interfaces, die Klasse "füllt" quasi das Interface mit Leben durch eine eigene Implementierung
- Klassen können mehrere Interfaces implementieren, durch die Trennung innerhalb der Klasse entstehen keine Namenskonflikte, aber es kann ermöglicht werden, daß Klassen, die in keinerlei Weise voneinander abhängig sind, trotz eigener Hierarchien noch gemeinsame Funktionalität abbilden
- C# enthält einige vorgefertigte Interfaces, die bestimmte Funktionalität ermöglichen – diese Interface beginnen immer mit I…, z.B. IEnumerator

Einfaches Beispiel eines Interface

```
//Primitves Interface
interface inter1
{
    double Wert
    {
        get;
        set;
    }
}
//.. das inhaltlich gleiche mit
//anderem Namen noch einmal
interface inter2
{
    double Wert
    {
        get;
        set;
    }
}


static void Main()
{
    interfacetester mm_it = new interfacetester();
    inter1 mm_i1 = mm_it; //Holen des Interface
    inter2 mm_i2 = mm_it;
    mm_i1.Wert = 23.1;  //Zugriff auf Interface-Property
    mm_i2.Wert = 123.4;
    Console.WriteLine("Produkt ist {0}", mm_it.produkt());
}
```

```
//Klasse, die diese beiden Interfaces
//implementiert
class interfacetester : inter1, inter2
{
    double m_w1, m_w2;
    double inter1.Wert //interface 1
    {
        get
        {
            return m_w1;
        }
        set
        {
            m_w1 = value;
        }
    }
    double inter2.Wert //interface 2
    {
        get
        {
            return m_w2;
        }
        set
        {
            m_w2 = value;
        }
    }
    public double produkt()
    {
        //Zugriff auf Interface-Methoden
        return ((inter1)this).Wert  * ((inter2)this).Wert ;
    }
}
```

Beispiel: den Indexer "foreach-fähig" machen

- Klasse erbt von IEnumerator und IEnumerable
- Methoden der Schnittstelle sind in der Klasse zu implementieren
- Nun kann der Indexer auch mit foreach durchlaufen werden
- foreach immer nur mit readonly

```
class Messwerte_MitForEach : IEnumerable, IEnumerator //Aufzählklasse für 100 Messwerte
```

```
#region Methode der Schnittstelle IEnumerator
public IEnumerator GetEnumerator()
{
    Reset();
    //Den Enumerator des Feldes selber herausgeben
    return m_Messwerte.GetEnumerator();
}
#endregion

#region Methoden der Schnittstelle IEnumerable
public bool MoveNext()
{
    if (++m_pos >= 100)
    {
        return false;
    }
    return true;
}
public void Reset()
{
    m_pos = 0;
}

public object Current
{
    get
    {
        return m_Messwerte[m_pos];
    }
}
#endregion
```

```
static void Main()
{
    Messwerte_MitForEach mm_w = new Messwerte_MitForEach();
    foreach (double mm_wert in mm_w)
    {
        Console.WriteLine("Wert {0}", mm_wert);
    }
}
```

Fehlerbehandlung

Allgemeine Fehlerbehandlung

- Der Compiler entdeckt Kompilierfehler, dies sind im wesentlichen Syntaxfehler und Fehler durch fehlerhafte Verwendung von Datentypen
- Wenn Fehler während der Ausführung des Programms auftreten, weil z.B. Dateien nicht gefunden werden können, durch 0 geteilt wird oder Bereichsüberschreitungen auftreten, handelt es sich um Laufzeitfehler, die i.d.R. nicht beim Kompilieren entdeckt werden können, sondern durch fehlerhaftes Design und/oder mangelnde Absicherung des Programm verursacht werden. Derartige Fehler führen in der Regel zum Absturz
- Logische Fehler durch falsche Algorithmen lassen das Programm nicht abstürzen, verursachen aber falsche Ergebnisse
- Laufzeitfehler, die zum Absturz führen können, lassen sich durch eine Fehlerbehandlung abfangen
- Das Exception Handling besteht im wesentlichen aus vier Befehlen:
  - try
  - catch
  - finally
  - throw

```
try
{
    //Anweisungen
}
catch (Fehlertypklasse variable)
{
    //Anweisungen im Fehlerfall
    //Reaktionen auf Fehler
}
finally
{
    //Anweisungen, die in jedem Falle
    //ausgeführt werden (auch ohne catch)
}
```

Beispiele

- int.Parse(string s) kann drei Fehlertypen auslösen:
  - ArgumentNullException tritt auf, wenn s als nicht initialisierte Variable übergeben wird (also null ist)
  - FormatException tritt auf, wenn s einen Text enthält, der nicht als Integer evaluiert werden kann
  - OverflowException tritt dann auf, wenn der Zahlenwert im String zu groß für den Datentyp Integer ist

```
try
{
    int j = 0;
    int i = 25 / j; // 25/0 würde Compiler merken!
}
catch (Exception e)
{
    Console.WriteLine("Fehler:\n{0}\n{1}", e.StackTrace, e.GetType().ToString());
}
```

Ausgabe:

```
Fehler:
    bei Hallo.cKlasseTest.Main(String[] args) in C:\Übungen\testprogramm
c#\hallo\program.cs:Zeile 156.
System.DivideByZeroException
```

Der try-catch-finally-Block

- Der try-Block kennzeichnet den Block, in dem C# eine Fehlerüberwachung durchführt, der Block steht wie alle Blöcke in C# in geschweiften Klammern, wobei ebenfalls die Regel gilt, daß Block-einleitende Befehle ohne Semikolon stehen
- Tritt im try-Block eine Ausnahme auf, so wird intern ein Fehler geworfen und das Programm verzeigt unmittelbar bei Auftreten des Fehlers in den catch-Block, Code im try-Block unterhalb der Fehlerauftrittsstelle wird nicht mehr ausgeführt, d.h. der Fehler unterbricht unmittelbar die aktuelle Befehlsausführung
- Ein möglicher Parameter im Catch-Statement klassifiziert den Fehlertyp, den dieser catch-Block abfangen soll. Besteht für einen geworfenen Fehler kein catch-Block, kann dieser an dieser Stelle nicht abgefangen werden, die Fehlertypen entstammen einer Klassenhierarchie, wird ein passender Fehler einer übergeordneten Fehlerklasse gefunden, wird entsprechend verzweigt
- Mehrere catch-Blöcke können einem try-Block zugeordnet werden, um individuelle Fehlerbehandlungen für entsprechende Fehlerklassen zu ermöglichen
- Durch den unmittelbaren Abbruch der Ausführung kann das Programm in einem undefinierten Zustand verbleiben – durch Verwendung des finally-Blocks kann Programmcode definiert werden, der immer, aber (ohne weiteres try) ohne Fehlerbehandlung ausgeführt wird

Fehlerklassen

- In C# sind bspw. folgende Fehlerklassen definiert:

Exception	Basisklasse aller Fehlerklassen
SystemException	Basisklasse für zur Laufzeit generierte Ausnahmen
IndexOutOfRangeException	Bereichsgrenzen eines Arrays werden verletzt (Zugriff auf nicht vohandenen Index)
NullReferenceException	Verweisvariable, die nicht initialisiert ist und auf die zugegriffen wird
ArithmeticException	Basisklasse aller arithmetischen Fehler
DivideByZeroException	Teilen einer Zahl durch 0
OverflowException	Überlauf der Wertegrenzen einer Variablen
FormatException	Format eines Arguments einer kombinierten Formatierung stimmt nicht und verursacht einen Fehler
IOException	Ein Fehler beim Schreiben oder Lesen aus einem Ein-/Ausgabestream ist aufgetreten (z.B. durch Schreibschutz)
DirectoryNotFoundException	Ein Verzeichnis ist nicht auf dem Datenträger verfügbar
FileNotFoundException	Der Datenzugriff auf eine Datei schlägt fehl, da die Datei nicht vorhanden ist
ArgumentNullException	Ein Argument einer Methode besitzt einen ungültigen Verweis

- Es ist möglich, eigene Fehlerklassen zu erstellen

Auslösen eines Fehlers

- In manchen Situationen sollte ein Fehler ausgelöst werden, wenn bspw. eine Methode nicht das gewünschte Ergebnis berechnen kann oder ein Fehler an die aufrufende Methode zurückgegeben werden soll, aber erkennbar bleiben soll, was genau den Fehler auslöste. Dies kann in C# mit dem throw-Befehl realisiert werden:

```
private static void HauptFunktion()
{
    try
    {
        aufgerufeneFunktion();
    }
    catch (Exception ex)
    {
        //aufgetretenen Fehler ex weiter werfen, aber
        //zeigen, von wo die Funktion gerufen wurde...
        throw new Exception("Fehler in HauptFunktion!", ex);
    }
}
private static void aufgerufeneFunktion()
{
    // Fehlersituation tritt auf - Fehler wird geworfen...
    throw new Exception("Fehler in aufgerufeneFunktion!");
}
```

Auslesen des Fehlerstacks

```
static void Main(string[] args)
{
    Console.Title = "Die Nachrichten einer Exception und ihrer inneren Exceptions ermitteln";

    try
    {
        Console.WriteLine("Aufruf von Wirffehler");
        HauptFunktion();
    }
    catch (Exception ex)
    {
        Console.WriteLine("Oberster Fehler im Fehlerstack:\n{0}", ex.Message);
        Console.WriteLine("Nächster Fehler im Fehlerstack:\n{0}", ex.InnerException.Message);
    }
}
```

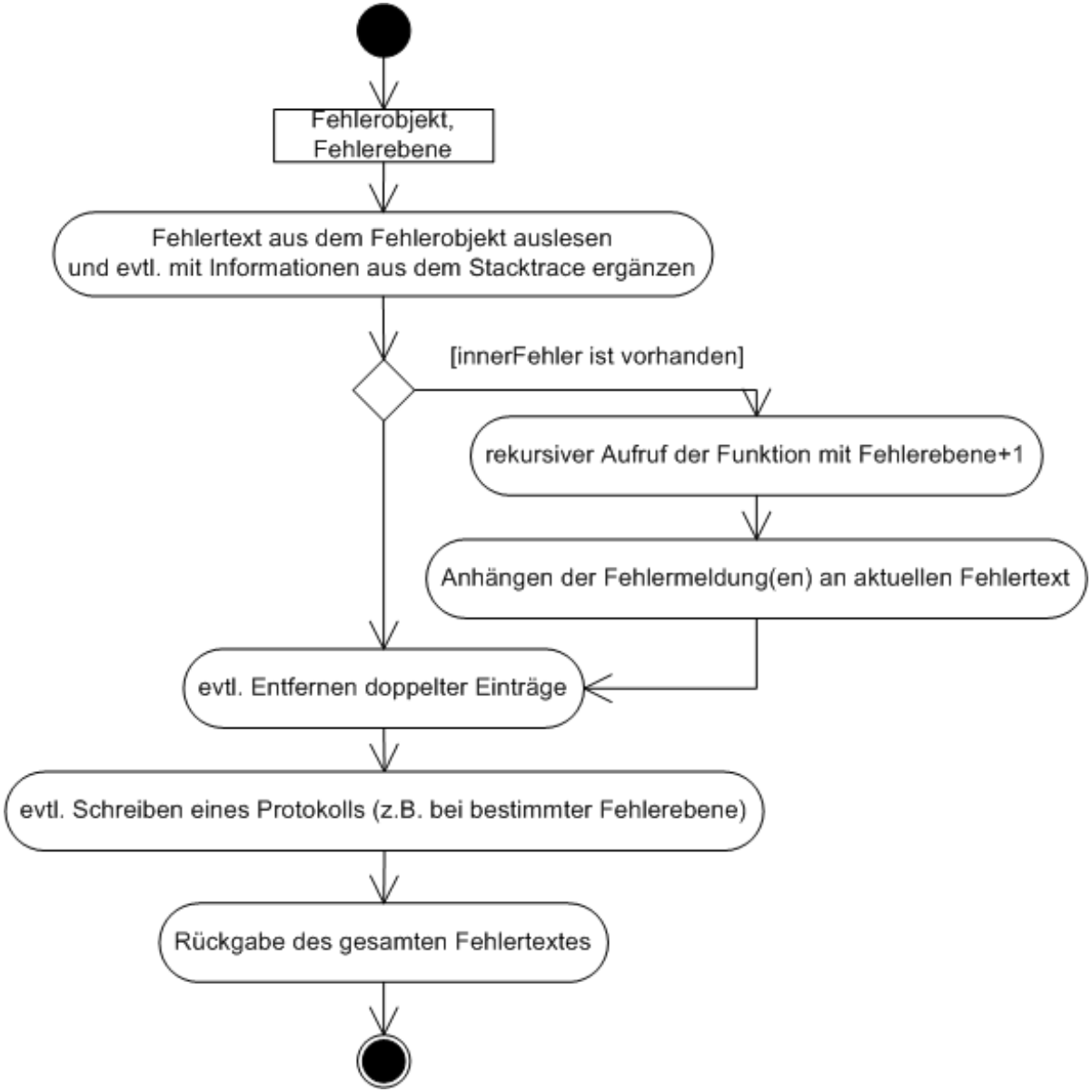
Wenn an dieser Stelle keine InnerException vorhanden wäre, würde ein weiterer Fehler auftreten.

Sinnvoll ist die Verwendung einer eigenen Fehlerausgabeklasse, die Fehler bspw. protokolliert und den Fehlerstapel automatisch ausliest

Ausgabe:

```
Aufruf von Wirffehler
Oberster Fehler im Fehlerstack:
Fehler in HauptFunktion!
Nächster Fehler im Fehlerstack:
Fehler in aufgerufeneFunktion!
```

Aktivitäten-Diagramm einer Fehlerausgabeklassen-Methode



Early und Late Binding

- Bisher wurden alle Objektttypen bei ihrer Deklaration auf den expliziten Klassentyp oder eine Basisklasse definiert.
- Dadurch kann der Compiler prüfen, ob beispielsweise Methodenaufrufe oder Eigenschaftszugriffe korrekt und zulässig sind
- Insbesondere, wenn Bibliotheken, die in unterschiedlichen Versionen vorhanden sein können, verwendet werden, kann es zu Problemen kommen, da die Klassendefinitionen Unterschiede aufweisen und die Zuweisung fehlschlägt.
- Damit ein Methodenaufruf kompilierbar ist, muß die Methode vorher bekannt sein – dies ist das sog. Early Binding – alle Objekte werden mit zugehörigem Klassentyp definiert.
- Beim Late Binding wird diese Zuordnung erst zur Laufzeit vorgenommen, der Compiler prüft nicht mehr die Korrektheit von Methodenschreibweise oder Aufruf, es wird erst zur Laufzeit untersucht, ob das aktuelle Objekt eine Methode mit Aufrufparametern wie im Quelltext vorgegeben hat. Das schafft eine größere Flexibilität, ist aber weitaus fehleranfälliger und tendenziell etwas langsamer
- Der Typ für Late Binding heißt dynamic und wird wie ein Datentyp verwendet