Vergleich der objektorientierten Programmiersprachen Java und C#

Nikolaus Gebhardt

1. März 2004

Inhaltsverzeichnis

1 Einleitung

1.1 Verwendete Versionen

In dieser Arbeit werden die Sprachen Java und C# miteinander verglichen. Zum Zeitpunkt dieser Arbeit waren Java in Version 1.4.2 (beziehungsweise 1.5.0-beta, siehe unten) und C# in Version 1.2 verfügbar. Es wird in dieser Arbeit von den entsprechenden Sprachspezifikationen von Java [GJSB00] und C# [Mic02] ausgegangen und es werden die dafür erhältlichen Compiler eingesetzt, wobei allerdings an passenden Stellen auch ein Ausblick auf die bald neu erscheinenden Versionen C# 2.0 und Java 1.5 gegeben wird. Dazu wurde für C# Microsofts aktuelle Sprachspezifikation 2.0 [Mic03] verwendet, für Java ein von Sun Microsystems bereitgestellter Compiler mit Generizitätsunterstützung, der in [BCK+01] beschrieben wird und die wichtigsten Draft Specifications, die in Java 1.5 '*Tiger*' implementiert werden: [Sun01], [Sun03], [Sun04d], [Sun04c], [Sun04a], [Sun04e], und [Sun04b]. Außerdem wurde im Laufe der Erstellung dieser Arbeit die erste Betaversion der neuen Javaversion, Java 1.5.0-beta, veröffentlicht. Fehlende Stellen der Arbeit wurden danach noch hinzugefügt, und wo notwendig auch einige bestehende Stellen überarbeitet und angepasst.

1.2 Grober Überblick

Java und C# sind in vielerlei Hinsicht durchaus ähnliche Sprachen und lassen sich dadurch sicherlich leichter vergleichen als ein anderes Paar objektorientierter Sprachen, die wenig Gemeinsamkeiten haben. Dies ist unter anderem auch dem Umstand zu verdanken, dass die Microsoft Corporation C# als eigene plattformunabhängige Sprache vermutlich mit Seitenblicken auf Sun Microsystems' Java als Konkurrenz entwickelt hat. [Bay02]

Offensichtliche Gemeinsamkeiten der beiden Sprachen bemerkt man bereits, wenn man sich nur kurz mit den beiden beschäftigt: C# und Java haben eine sehr ähnliche Syntax mit vielen gleich lautenden Schlüsselwörtern, eine vergleichbare Speicherverwaltung (Garbage Collection), ähnlich umfangreiche Standardbibliotheken (die in dieser Arbeit außer Acht gelassen werden), ein gleichartiges Vererbungsystem, ähnliche primitive Typen, usw. Auf die einzelnen Details wird nun im Laufe dieser Arbeit genauer eingegangen.

2 Vergleich

2.1 Typen

In Java wird genauso wie in C# zwischen zwei Arten von Typen unterschieden, die man in Variablen speichern und als Parameter in Methoden übergeben kann, und auf denen verschiedene Operationen ausgeführt werden können.

In Java werden diese beiden Typen als *reference* (Referenz-) und *primitive types* (primitive oder auch elementare) Typen [Haw02] bezeichnet. Primitive Typen enthalten ihre Daten direkt, während Referenztypen eine Referenz auf Ihre Daten, die die Form eines Objektes haben, abspeichern. Die Typen `boolean`, `byte`, `short`, `int`, `long`, `char`, `float` und `double` sind primitive Typen. Auf alle diese definiert Java bestimmte Operationen, wie `+`, `*` und `-`, die unter Umständen auch Exceptions werfen können. Klassen, Interfaces und Arrays sind Referenztypen.

C# verwendet zwei entsprechende Typen, die als *reference* (Referenz-) und *value types* (Werttypen) bezeichnet werden, wobei diese allerdings noch in weitere Typen unterteilt werden können, und es wesentlich mehr Arten von Typen gibt. Werttypen wie `short`, `int` oder `long` sind *simple types*, die den *primitive types* in Java entsprechen. C# unterstützt allerdings noch einige andere, die es in Java nicht als primitive Typen gibt, wie `decimal`, `sbyte`, `byte`, `ushort`, `uint` und `ulong`. Außerdem gibt es noch zwei weitere benutzerdefinierte Werttypen: *enum types* für Enumerationen und *struct types* für selbst zusammengesetzte Werttypen, die Daten und Funktionen kapseln, als Werttyp jedoch keine benutzerdefinierte Vererbung unterstützen. Typen für Enumerationen wird es auch in Java 1.5 geben, allerdings sind diese in Java Referenztypen. In Abschnitt 2.18 wird näher darauf eingegangen.

Genau wie in Java sind Klassen, Interfaces und Arrays Referenztypen. Es gibt jedoch noch weitere Typen, die keine Entsprechung in Java haben: *delegates*. Es handelt sich dabei um Referenzen auf eine Methoden mit bestimmten Parametern und Rückgabetypen, also eine Art typsicherer Funktionszeiger. (Mehr dazu in Abschnitt 2.15.)

Wenn man die primitive types aus Java den simple types aus C# gleichsetzt, kann man eine Gegenüberstellung wie in Tabelle 1 vornehmen. Hierbei sieht man leicht, dass Java und C# starke Ähnlichkeiten aufweisen, und C# Java einfach um einige Features erweitert.

Kategorie		Java	C#
Value Types	primitive / simple types	`int`, `long`, `char`, `float`, `double`, `boolean`	`byte`, `sbyte`, `ushort`, `short`, `uint`, `int`, `ulong`, `long`, `char`, `float`, `double`, `decimal`, `bool`
	enum types	-	`enum e {...}`
	struct types	-	`struct s {...}`
Reference Types	class types	`class c {...}`	`class c {...}`
	interface types	`interface i {...}`	`interface i {...}`
	array types	z.B. `int []`	z.B. `int []`
	delegate types	-	`deletgate T d(...)`

Tabelle 1: Gegenüberstellung der vorhandenen Typen in C# und Java bis Version 1.4. Ab Version 1.5 gibt es auch einen Typ für Enumerationen, aber im Gegensatz zu C# nicht als Wert- sondern als Referenztyp.

2.2 Boxing

In beiden Sprachen gibt es einen impliziten Obertypen für alle Typen, der die Bezeichnung `object` (C#) beziehungsweise `Object` (Java) hat. (In C# kann man aber nach Belieben genauso auch `Object` schreiben, was aber nicht bedeutet, dass dies auch für andere Klassen gilt. C# ist genau wie Java *case sensitive*.) In Java gilt dies allerdings nur für Referenztypen, C# verwendet die als *type system unification* (Typsystemvereinheitlichung) benannte Methode, bei der auch Werttypen `object` indirekt Obertypen haben. Dies wird in C# durch automatisches Boxing/Unboxing erreicht. Folgendes Beispiel veranschaulicht dies:

```
class BoxingTest
{
    static void Main()
    {
        int i = 1;
        object o = i;   // Boxing
        int j = (int)o; // Unboxing
    }
}
```

In Java musste bis inklusive zur Version 1.4.2 das Boxing manuell durchgeführt werden. Dafür existiert für jeden primitiven Typen eine eigene Wrapperklasse, zum Beispiel gibt es für `int` einen Referenztyp `Integer`, der eine Variable `int` enthält. Das gleiche Beispiel würde also in Java 1.4.2 so aussehen:

```
public class BoxingTest // Java 1.4.2 oder älter
{
  public static void main(String[] args)
  {
    int i = 1;
    Object o = new Integer(i);   // Boxing
    int j = ((Integer)o).intValue(); // Unboxing
  }
}
```

In Java 1.5 gibt es einen ähnlichen automatischen Boxing / Unboxing Mechanismus [Sun04d] wie in C#, der zwar noch nicht vom Prototypcompiler [BCK+03], sehr wohl aber in der ersten Betaversion von Java 1.5 unterstützt wird. Dabei werden alle primitiven Typen bei Bedarf automatisch in Ihre Wrapperklassen umgewandelt, und umgekehrt. Das dem oben angeführten C#-Beispiel entsprechende Java 1.5-Programm sieht so aus:

```
public class BoxingTest // ab Java 1.5
{
  public static void main(String[] args)
  {
    int i = 1;
    Object o = i; // Boxing
    int j = (Integer)o; // Unboxing
  }
}
```

Der Vorteil des automatischen Boxing ist leicht ersichtlich: Man erspart sich viel unnötige Tipparbeit und der Code wird wesentlich übersichtlicher und verständlicher. Vor allem bei der wiederkehrenden Arbeit mit Kollektionen wie Vektoren oder Listen wird das deutlich. Statt einer unübersichtlichen Zeile wie in Java 1.4.2 notwendig:

```
vector.add(new Integer(42));
int i = ((Integer)vector.get(0)).intValue();
```

schreibt man nun in Java 1.5 (kompilierbar nur mit zusätzlichen switches: `javac -source 1.5 -Xlint BoxingTest.java`):

```
vector.add(42);
int i = (Integer)vector.get(0);
```

Einem Programmierer, der nicht weiss, was bei automatischem Boxing genau passiert, kann jedoch leicht ein Fehler unterlaufen. Es wird leicht vergessen, dass trotz der Typsystemvereinheitlichung der Unterschied zwischen Referenztypen und Werttypen nicht ignoriert werden kann:

```
using System;

class BoxingTest
{
    static void Main()
    {
        int i = 1;
        object o = i;
        i = 5;
        Console.Write((int)o);
    }
}
```

Die Ausgabe dieses Programmes ist entgegen der Erwartung mancher Programmierer `1` und nicht `5`. (In C# genauso wie in Java 1.5). Das liegt einfach daran, dass durch das implizite Boxing, das beim Zuweisen der Variable `i` and `o` geschieht, die Variable kopiert wird.

2.3 Parameterübergabe an Methoden

Die als Argumente übergebenen Variablen werden in Java bei Aufruf einer Methode immer *by value* übergeben. Das heißt sie werden beim Aufruf kopiert. Referenztypen zeigen dann natürlich immer noch auf das gleiche Objekt, primitive Typen werden tatsächlich (falls man das so sagen kann) kopiert. Will man zum Beispiel eine Funktion schreiben, die zwei Strings miteinander vertauscht, und vielleicht zusätzlich die Strings aneinanderhängt stößt man schnell auf Probleme. Der Versuch einer Lösung wie im folgendem Beispiel scheitert, da die Parameter *by value* übergeben werden:

```
public class RefTest
{
```

```
  static void swapcat(String a, String b, String c)
  {
    c = a + b;    // aneinanderhängen
    String t = a; // tauschen
    a = b;
    b = t;
  }

  public static void main(String[] args)
  {
    String a="A", b="B", c="";
    swapcat(a, b, c);
    System.out.print(a + " " + b + " " + c);
  }
}
```

Die Ausgabe ist hier wie erwartet `A B` und nicht `B A AB`, die in diesem Beispiel aber wünschenswert gewesen wäre. Die Einschränkung auf *by value* Parameterübergabe ist hier hinderlich, und müsste irgendwie umgangen werden. Zum Beispiel wäre es möglich, die Funktion zu entfernen, und die Variablen *inline* auszutauschen. Der zusammengehängte String hätte auch einfach als Ergebnis der Methode zurückgeliefert werden können, und hätte damit kein Problem verursacht.

C# ist hier nicht so stark eingeschränkt, es gibt sowohl Wertparameter als auch Referenzparameter und Ausgabeparameter. Weil verschiedene Arten der Parameterübergabe manchmal leichter zu Programmierfehlern führen können, muss der Programmierer bei Aufruf einer Methode mit anderen als `by value` Parametern explizit angeben, dass das Argument als Ausgabe- oder Referenzparameter übergeben werden soll. Das in Java nicht funktionierende Beispiel würde in C# das korrekte Ergebnis liefern und nun so aussehen:

```
class RefTest
{
    static void swapcat(ref String a, ref String b, out String c)
    {
        c = a + b;    // aneinanderhängen
        String t = a; // tauschen
        a = b;
        b = t;
    }
```

```
    static void Main()
    {
        String a="A", b="B", c;
        swapcat(ref a, ref b, out c);
        Console.Write(a + " " + b + " " + c);
    }
}
```

Eine weitere Art der Parameterübergabe, die in C# funktioniert und auch in Java 1.5 implementiert sein wird [Sun04b], ist die Möglichkeit, eine variable Anzahl unbestimmter Parameter zu übergeben. Damit lassen sich Methoden in der Art der aus C bekannten Funktion `printf()` realisieren.

In Java funktioniert dies, indem man nach dem Parametertyp drei Punkte angibt. Mit dem Prototypcompiler [BCK+03] kann das bereits ausprobiert werden. Das folgende Beispielprogramm hat eine Funktion `print()`, die einen String ausgibt, und danach alle weiteren übergebenen Parameter.

```
public class VarArgTest
{
  static void print(String text, Object... args)
  {
    System.out.print(text);
    for (int i=0; i<args.length; ++i)
        System.out.print(args[i] + " ");
  }

  public static void main(String args[])
  {
    print("Objekte: ", new Integer(24), "hallo", new Float(66.06f));
  }
}
```

In der Ausgabe sieht man erwartungsgemäß, dass alle Objekte korrekt ausgegeben werden: `Objekte: 24 hallo 66.06`. In C# werden die variablen Parameter als ein mit dem Schlüsselwort `params` Array gekennzeichnet. Das entsprechende C#-Programm sieht folgendermaßen aus:

```
using System;

class VarArgTest
{
```

```
  static void print(string text, params object[] args)
  {
      Console.Write(text);
      for (int i=0; i<args.Length; ++i)
        Console.Write(args[i].ToString() + " ");
  }

  static void Main()
  {
    print("Objekte: ", 24, "hallo", 66.06f);
    Console.Read();
  }
}
```

2.4 Vererbung und Polymorphie

In Java und C# funktioniert Vererbung auf sehr ähnliche Weise: Eine Klasse kann immer nur von genau einer anderen Klasse erben, es gibt keine Mehrfachvererbung. Um den Nachteil dieser Beschränkung auf Einfachvererbung abzuschwächen, gibt es so genannte Interfaces, die so etwas wie abstrakte Klassen mit einigen Einschränkungen sind, auf die aber Mehrfachvererbung unterstützt wird. Bis auf kleine syntaktische Unterschiede gibt es hierbei noch einige andere Abweichungen in beiden Sprachen.

2.4.1 Interfaces

In Java ist es ohne weiteres möglich, dass eine Klasse mehrere Interfaces implementiert, die jeweils Methoden mit gleicher Signatur definieren:

```
interface Interface1
{
  void foo();
}

interface Interface2
{
  void foo();
}

class C implements Interface1, Interface2
{
```

```
  public void foo()
  {
    // Implementierung beider Interfaces
  }

  public static void main(String args[])
  {
    C c = new C();
    c.foo(); // Aufruf
  }
}
```

Dies kann aber Probleme verursachen: Beide Interfaces müssen in Java die gleiche Implementierung haben. Es gibt keine Möglichkeit, in der Klasse `C` die Methode `foo()` für `Interface1` und `Interface1` unterschiedlich zu implementieren. Und aus diesem Grund gibt es auch beim Aufruf `c.foo()` nicht die Möglichkeit, zu bestimmen, welche Methode gemeint ist, da es ohnehin nur eine Implementierung gibt. In C# hätte man das Beispiel genauso schreiben können. Allerdings hat C# eine Lösung für das genannte Problem, es ist nämlich möglich, eine Implementierung explizit für jedes Interface anzugeben:

```
interface Interface1
{
  void foo();
}

interface Interface2
{
  void foo();
}

class C : Interface1, Interface2
{
  void Interface1.foo()
  {
    // Implementierung von Interface1
  }

  void Interface2.foo()
  {
```

```
    // Implementierung von Interface2
  }

  static void Main()
  {
    C c = new C();
    ((Interface1)c).foo(); // Aufruf auch explizit
    ((Interface2)c).foo();
  }
}
```

2.4.2 Überschreiben von Methoden

Überschreibt man in Java Methoden von Oberklassen, wird beim Aufruf dieser Methode in der Oberklasse durch dynamisches Binden die überschreibende Methode der Unterklasse aufgerufen. Ein einfaches Beispiel demonstriert dies:

```
class A
{
  public void foo()
  {
    System.out.print("A.foo");
  }
}

class B extends A
{
  public void foo()
  {
    System.out.print("B.foo");
  }

  public static void main(String args[])
  {
    A a = new B();
    a.foo();
  }
}
```

Die Ausgabe des Programms ist erwartungsgemäß `B.foo`. Schreibt man das gleiche Programm jedoch in C# indem man lediglich das Schlüsselwort

`extends` durch einen Doppelpunkt austauscht, und statt `System.out.print` `System.Console.Write` einsetzt, ist die Ausgabe `A.foo`. Der Grund hierfür ist, dass C# einen Unterschied zwischen *virtuellen* und *nicht-virtuellen* Methoden macht. Per Default sind alle Methoden nicht virtuell, und es muss explizit angegeben werden, dass eine Methode virtuell ist, beziehungsweise, das sie eine virtuelle Methode überschreibt. Das korrekte Beispiel sieht also so aus:

```
class A
{
  public virtual void foo()
  {
  System.Console.Write("A.foo");
  }
}

class B : A
{
  public override void foo()
  {
    System.Console.Write("B.foo");
  }

  static void Main()
  {
    A a = new B();
    a.foo();
  }
}
```

Vergisst der Programmierer das `override`, oder überhaupt beide Schlüsselwörter, so gibt der Compiler eine Warnung aus, dass die neue Methode `foo` die alte verdeckt, und dass die Methode nicht überschrieben wurde. Will man diesen Effekt manuell tatsächlich erreichen, kann man die Methode dafür mit dem Schlüsselwort `new` markieren. Es ist nicht möglich, eine Methode zu überschreiben, die nicht als `virtual` definiert ist, ein Versuch scheitert an einem Compilerfehler.

Will man verhindern, dass eine Methode überschrieben wird, so verwendet man in Java das Schlüsselwort `final`. In C# markiert man die Methode einfach nicht als `virtual`, oder man benutzt, wenn es sich bereits um eine Methode handelt, die eine andere überschreibt, das Schlüsselwort `sealed`.

Der Vorteil der Herangehensweise von C# ist, dass der Programmierer immer genau wissen muss, welche Methoden er überschreibt, welche verdeckt, und bei welchen er von Anfang an erlaubt, dass sie überschrieben werden. Außerdem kann durch gezielten Einsatz von nicht-virtuellen Methoden die Effizienz von kritischem Code erhöht werden. Das dadurch bei den meisten Methoden notwendige vorangestellte `overrride` kann dem dem Java-gewöhntem Programmierer allerdings auch schnell lästig werden und ist mehr Tipparbeit. Ein nachträgliches Ändern der Klassenhierarchie zum Beispiel im Rahmen einer Refaktorisierung wird dadurch ebenfalls erschwert. Wenn nicht geplant war, dass bestimmte Methoden später überschrieben werden könnten, oder ehemalige Basisklassen nun selbst von einer neuen Klasse erben sollen, müssen sehr viele Schlüsselwörter geändert werden.

2.4.3 Überladen von Methoden

Das Überladen von Methoden funktioniert in Java und C# nahezu gleich. Methoden mit gleichen Namen, aber unterschiedlichen Typen und/oder Anzahl der Parameter können in der gleichen Klasse verwendet werden. Bei der Kompilierung wird die zu den angegebenen Argumenten am besten passende Methode gesucht. Bei Spezialfällen verhält sich die Suche nach der passendsten Methode in Java und C# aber unterschiedlich, wie im folgendem Beispiel:

```
class A
{
  public virtual void foo(int i)
  {
    System.Console.Write("A.foo  ");
  }
}

class B : A
{
  public override void foo(int i)
  {
    System.Console.Write("B.foo.int  ");
  }

  public virtual void foo(object o)
  {
    System.Console.Write("B.foo.object  ");
  }
```

```
  static void Main()
  {
    B b = new B();
    b.foo(1);
    b.foo("abc");
  }
}
```

Die Ausgabe eines dem obigem C# Programm entsprechenden Java-Programms würde `B.foo.int B.foo.object` lauten, da die beste Methode für den Aufruf `b.foo(1)` die ist, die einen `int` als Parameter akzeptiert, und für den Aufruf `b.foo('abc')` die Methode, die ein `Object` akzeptiert. C# geht bei der Wahl der besten Methode aber offensichtlich anders vor: Die Ausgabe des C# Programmes ist `B.foo.object B.foo.object`, was vielleicht ein wenig unverständlich anmuten mag (zumindest für mich persönlich): Obwohl es eine Methode gibt, die einen `int` akteptiert, wurde die Methode mit `object` als Parameter gewählt. Der Grund für das andere Ergebnis ist, dass Methoden, die mit `override` markiert sind, in C# nicht als Kandidaten für die Auswahl beachtet werden, wenn es andere passende Methoden gibt, die nicht überschreibend sind [Mic02].

2.5 Überladen von Operatoren

In C# ist es im Gegensatz zu Java möglich, Operatoren zu überladen. Der Vorteil davon ist, dass man dadurch in der Lage ist, in bestimmten Situationen intuitiveren Code zu schreiben. Wenn man beispielsweise eine Klasse schreibt, mit der man eine bestimmte Art von Zahlen manipulieren kann, ist Code wie `c = a + (b * a)` in C# natürlich wesentlich schneller zu lesen und zu verstehen als der in Java entsprechende Code `c = a.add(b.multiply(a))`. Hier ein kurzer Beispielcode, in dem eigene Operatoren + und * wie in diesem Beispiel eingesetzt werden:

```
class MyFloat
{
  private float f;

  public MyFloat(float f)
  {
    this.f = f;
  }
```

```
  public override string ToString()
  {
    return f.ToString();
  }

  public static MyFloat operator +(MyFloat f1, MyFloat f2)
  {
    return new MyFloat(f1.f + f2.f);
  }

  public static MyFloat operator *(MyFloat f1, MyFloat f2)
  {
    return new MyFloat(f1.f * f2.f);
  }

  static void Main()
  {
    MyFloat a = new MyFloat(2.0f);
    MyFloat b = new MyFloat(3.0f);
    System.Console.Out.Write( a + (b * a) );
  }
}
```

Die Ausgabe des Programmes ist dabei wie erwartet `8`. Um die Ausgabe mittels `System.Console.Out.Write` zu ermöglichen, wurde zusätzlich die Methode `Object.toString()` überschrieben.

In C# können die unären Operatoren `+ - ! ~ ++ -` und die binären Operatoren `+ - * / % & | ^ << >> == != > < >= <=` überladen werden. Zu binären Operatoren passende Zuweisungsoperatoren wie `*=` zu `*` können nicht selbst überladen werden, dies geschieht implizit. Die aus anderen Sprachen wie C++ bekannte Möglichkeit, die Operatoren `new, (), ||, &&` oder `=` zu überladen besteht nicht. Es gibt auch die Möglichkeit, `[ ]` zu überladen, dies geschieht über sogenannte *Indexer*, die ähnlich wie `Properties` funktionieren, auf die noch näher in Abschnitt 2.6 eingegangen wird.

```
class MyArray
{
  private float[] f = new float[3];

  public float this[int idx]
```

```
  {
    get{ return f[idx]; }
    set{ f[idx] = value; }
  }

  static void Main()
  {
    MyArray a = new MyArray();
    a[1] = 99.1f; // schreiben
    System.Console.Write( a[1] ); // lesen
  }
}
```

Die Klasse `MyArray` kann damit wie ein Array verwendet werden. Es gibt auch die Möglichkeit, mehrdimensionale Arrays zu realisieren, indem man einfach einen weiteren Index als Parameter der Methode verlangt.

Das in Java fehlende Operator-Overloading ist ein der in der Java-Community sehr oft diskutiertes Thema. In der kommenden Version 1.5 von Java wird es offensichtlich nicht unterstützt werden, und auch für spätere Versionen ist es nicht geplant. Der oft dafür genannte Grund ist, dass dadurch die Übersicht zu leicht verloren geht [Int03].

2.6 Properties

Um in Java bestimmte Eigenschaften oder Werte eines Objektes festzulegen, verwendet man meist `get()`- und `set()`-Methoden. Um zum Beispiel die private Variable `foo` in einer Klasse setzen und lesen zu können benötigt man zwei separate Methoden:

```
class GetSetTest
{
  private int foo = 0;

  public int getFoo()
  {
    return foo;
  }

  public void setFoo (int f)
  {
    foo = f;
```

```
  }

  public static void main(String args[])
  {
    GetSetTest a = new GetSetTest();
    a.setFoo( a.getFoo() + 1 );
  }
}
```

In der `main()`-Methode wird die Variable `foo` um eins erhöht, dazu sind ein Aufruf der `get()`- und der `set()`-Methode notwendig.

In C# gibt es *Properties* (Eigenschaften), die dies ein wenig vereinfachen und übersichtlicher machen. Mit Einsatz dieser Properties sieht das Beispiel in C# so aus:

```
class GetSetTest
{
  private int foo = 0;

  public int Foo
  {
    set{ foo = value; }
    get{ return foo; }
  }

  static void Main()
  {
    GetSetTest a = new GetSetTest();
    a.Foo += 1;
  }
}
```

Dabei wurde in der Property `Foo` eine Methode zum Setzen (`set`) und eine zum Lesen (`get`) des Wertes definiert. Auf den Wert kann nun syntaktisch so zugegriffen werden, als wäre die Property eine öffentliche Instanzvariable. Der Ausdruck `a.Foo += 1` ist wesentlich klarer als `a.setFoo( a.getFoo() + 1 )`. Je nach Geschmack hätte man stattdessen z.B. auch `a.Foo++` oder `a.Foo = a.Foo + 1` schreiben können. Bei komplizierteren Verschachtelungen von Klassen wird noch deutlicher, wie sinnvoll Properties sind: Statt zum Beispiel `node.getParent().setVisible(false)` ist es einfacher `node.Parent.Visible = false` zu schreiben.

Es ist auch möglich nur lesenden oder nur schreibenden Zugriff auf Properties zuzulassen. Dabei kann die unerwünschte Methode in der Property einfach weggelassen werden.

2.7 Zeiger

Als C oder C++-Programmierer ist man es gewöhnt, mit Pointern, also Zeigern, auf Objekte und Speicheradressen zu arbeiten. Da man mit Hilfe von Zeigern unter anderem in der Lage ist, direkt den Inhalt des Speichers zu ändern, ist der Einsatz von Zeigern in typsicheren objektorientierten Sprachen eher unüblich. In Java gibt es daher auch keine Zeiger, in C# allerdings schon: In mit dem Schlüsselwort `unsafe` markierten Codeabschnitten oder Methoden ist es möglich, Zeiger zu verwenden. In solchen Abschnitten, die mit `unsafe` als unsicher markiert worden sind, wird keine Laufzeitüberprüfung von Typen durchgeführt. Um solchen Code kompilieren zu können, muss zusätzlich dem Compiler der Switch `/unsafe` angegeben werden. Da in C# die Position aller Objekte im Speicher vom Garbage Collector während der Laufzeit verschoben werden kann, muss bei der Zuweisung der Adresse eines so genannten `managed` Objektes zu einem Pointer innerhalb eines `fixed` Blockes geschehen. Dieser sorgt dafür, dass, während Code in diesem Block ausgeführt wird, der Garbage Collector das Objekt nicht verschieben kann. Folgendes Beispiel demonstriert die Funktionsweise aller beschriebenen Konstrukte:

```
class PointerTest
{
  unsafe public static int* find(int* start, int* end, int tofind)
  {
    for (int* p=start; p!=end; ++p)
      if (*p == tofind)
        return p;

    return null;
  }

  static void Main()
  {
    int[] array = new int[42];
    array[20] = 6;

    unsafe
```

```
    {
      fixed( int* start = &array[0])
      fixed( int * end = &array[42])
      {
        int* f = find(start, end, 6);
        if (f!=null)
           System.Console.Write("found");
      }
    }
  }
}
```

Die Funktion `find()` ist als `unsafe` markiert, weil sie als Parameter einige Zeiger erwartet, und auch mit diesen arbeitet. Sie durchsucht ein Array nach dem Vorkommen eines bestimmten Elementes. In der `Main()`-Funktion wird ein Array erzeugt, innerhalb des `unsafe`-Blocks werden zwei Zeiger auf das Array deklariert, und die `find()`-Funktion aufgerufen.

2.8 Die `main()`-Methode

Wie vielleicht schon anhand einiger Beispiele in dieser Arbeit gesehen werden konnte, ist der Einstiegspunkt in ein Programm in Java genau wie in C# eine statische `Main()`- (C#) beziehungsweise `main()`-Methode (Java). Jedoch gibt es bis auf den Unterschied in der Groß- und Kleinschreibung noch einige wenige weitere Abweichungen: In Java hat die `main()`-Methode immer genau die gleiche Signatur. Sie hat keinen Rückgabewert und akzeptiert als Parameter ein Array von Strings:

```
public static void main(String[] args) { ... }
```

C# bietet hier eine Auswahl an verschiedenen `Main()`-Methoden. Es ist jede Kombination aus mit/ohne Rückgabewert und mit/ohne Kommandozeilenparameter möglich, was vier verschiedene Kombinationen erlaubt:

```
static void Main() { ... }
static void Main(string[] args) { ... }
static int Main() {...}
static int Main(string[] args) { ... }
```

Außerdem muss der Zugriffsmodifizier in Java immer `public` sein, während er in C# ignoriert wird, er kann sowohl `public`, `protected`, `internal`, `default` oder `private` sein.

Ein weiterer Unterschied ist, dass in Java jede Klasse eine eigene `main()`-Methode haben kann. Alle Methoden werden mitkompiliert, und erst beim Start der Applikation in der VM wird als Parameter angegeben, welche Methode in welcher Klasse verwendet werden soll.

Fügt man in C# mehreren Klassen eine `Main()`-Methode hinzu, so protestiert der Compiler. Er akzeptiert nur einen Einstiegspunkt in ein Programm. Mit dem `/main`-Switch kann man dem Compiler aber veranlassen, alle `Main()`-Methoden bis auf die in einer bestimmten Klasse zu ignorieren. Ein einmal kompiliertes C#-Programm kann also nicht wie von Java-Programmen gewohnt aus verschiedenen Klassen heraus gestartet werden.

2.9 Klassen und Dateien

Java hat die Beschränkung, dass in jeder `.java`-Datei nur eine Klasse mit gleichem Namen sein kann, die als `public` deklariert ist. Es können noch weitere Klassen in dieser Datei ausprogrammiert sein, diese dürfen jedoch nicht `public` sein. C# hat diese Beschränkung nicht. Weder gibt es eine Beschränkung der Art oder Anzahl der Klassen pro `.cs` Datei, noch muss die Bennenung irgendeiner Klasse dem Dateinamen entsprechen.

Quellcodedateien in Java haben damit immer den gleichen Namen wie die darin enthaltenen öffentlichen Klassen. Genauso sind diese Dateien, wie im nächsten Abschnitt genauer beschrieben, nach Namensräumen in Verzeichnissen angeordnet. Somit ist das Projekt zwangsweise immer in gewisser Weise aufgeräumt, und fremde Programmierer werden sich leicht in der Dateistruktur von fremdem Code zurecht finden. In C# ist das nicht so, jedoch ist es dadurch in dieser Sprache ohne großen Aufwand möglich, Klassen zum Beispiel bei Refaktorisierungen umzubenennen und in andere Namensräume zu schieben, ohne auch die Dateien umzubenennen oder verschieben zu müssen, was ohne dafür gedachte Werkzeuge in Java recht umständlich sein kann.

2.10 Namensräume

Ähnlich wie bei den Dateinamen schreibt Java vor, in welchem Verzeichnis sich eine Datei befinden muss, wenn sie sich in einem bestimmten Paket befindet. Wird das nicht beachtet, lässt sich die Klasse nicht kompilieren oder verwenden. In folgendem Beispiel muss sich die Klasse `bar` in einem Ordner `foo` befinden:

```
package foo;
import java.util.*;
```

```
public class bar
{
  public void baz()
  {
    java.math.BigDecimal bd = new java.math.BigDecimal(1);
    Dictionary d = new Hashtable();
  }
}
```

Wenn eine in einem anderem Paket enthalte Klasse verwendet werden soll, müssen die Namen der Pakete, in denen sie enthalten ist, vor dem Namen der Klasse angeführt werden, wie im Beispiel `java.math.BigDecimal`. Um sich das zu ersparen, kann das Paket mit `import` importiert werden. `Dictionary` und `Hashtable` im Beispiel befinden sich im Paket `java.util`.

In C# werden die Pakete als Namespaces bezeichnet. Der verwendete Namensraum hat allerdings keine Auswirkung auf das Verzeichnis, in dem sich eine Datei befinden muss. Statt `import` verwendet man das Schlüsselwort `using`. Ein weiterer Unterschied ist, dass man in C# mehrere Namensräume pro Datei verwenden, und diese auch ineinander schachteln kann. Folgendes Beispiel zeigt ein dem obigen Java-Beispiel ähnliches C#-Programm, und fügt noch eine weitere Klasse `Quux` hinzu, die sich in der gleichen Datei, aber in einem Unternamensraum, befindet:

```
using System.Collections;

namespace foo
{
  public class bar
  {
    public void baz()
    {
      System.IO.Stream b = new System.IO.FileStream("txt", 0);
      IDictionary d = new Hashtable();
    }
  }

  namespace qux
  {
    class Quux
    {
```

```
    }
  }

}
```

2.11 Präprozessor

C und C++-Programmierer sind es gewohnt, mit einem Präprozessor zu arbeiten, durch dessen Verwendung sich für den Programmierer viele neue Möglichkeiten auftun. Mit Hilfe des Präprozessors ist es möglich, Textstellen im Code vor der wirklichen Kompilierung zu ersetzen oder wegzustreichen. Java bietet ein solches Feature nicht, in C# gibt es aber einen solchen Präprozessor. Mit seiner Hilfe ist es möglich, bedingte Kompilierungen durchzuführen, und Warnungen und Fehler bei der Kompilierung auszugeben. Textsubsitutionen wie in C oder C++ sind nicht möglich. Im folgendem Beispiel kommt bedingte Kompilierung zum Einsatz:

```
#define TEST

public class foo
{
  public static void Main()
  {
#if TEST
    System.Console.Write("Test mode.");
#else
    System.Console.Write("Release mode.");
#endif
  }
}
```

Hierbei wird nur die Zeile `System.Console.Write("Test mode.")` kompiliert, weil das Symbol `TEST` definiert ist. Kommentiert man die erste Zeile aus, wird stattdessen nur die Zeile `System.Console.Write("Release mode.")` kompiliert.

In Java gibt es keine in der Sprache eingebaute Möglichkeit zu verhindern, dass bestimmter Code nicht mitkompiliert wird. Dies kann aber manchmal gewünscht sein, zum Beispiel um verschiedene Versionen von Software kompilieren zu können, wobei nicht gebrauchter Code nicht als Ballast mitgeführt werden soll, oder um Reverse Engineering von in von bestimmten Versionen nicht verwendetem Code zu verhindern.

2.12 Sichtbarkeit für Klassenelemente

Beide Sprachen haben gleiche Benennungen für Zugriffsmodifier von Klassenmembern wie Variablen und Methoden, die auch aus anderen Sprachen bekannt sind: `public`, `private` und `protected`. Diese haben in C# und Java jedoch zum Teil unterschiedliche Bedeutungen. Außerdem verfügt C# über noch zwei weitere Zugriffsmodifier: `internal` und `protected internal`. Ich habe die verschiedenen Arten der Zugriffseinschränkung und ihre Bedeutungen in beiden Sprachen in Tabelle 2 gegenübergestellt. Die Default-Sichtbarkeit ist gegeben, wenn kein Schlüsselwort für die Sichtbarkeit angegeben wird.

C#	Java	Bedeutung
`public`	`public`	Zugriff ist nicht eingeschränkt.
`private`	`private`	Zugriff ist eingeschränkt auf diese Klasse.
`internal`	`protected`	Zugriff nur im selben Paket (Java) / im selben Programm (C#).
Default	-	Wie `private`.
-	Default	Wie `protected`, allerdings ist nicht erbbar außerhalb des Paketes.
`protected`	-	Zugriff nur in der Klasse und abgeleiteten Klassen.
`protected internal`	-	Zugriff nur für abgeleitete Klassen und welche im gleichen Programm.

Tabelle 2: Gegenüberstellung der Zugriffsmodifier in Java und C#.

2.13 Die `switch`-Anweisung

Sowohl Java als auch C# verfügen über eine Konstruktion, mit der ein Ausdruck leichter als mit mehreren `if (...) else`-Abfragen auf verschiedene Werte überprüft werden kann. Diese in beiden Sprachen `switch` genannte Anweisung funktioniert in C# und Java jeweils ein wenig unterschiedlich.

Folgendes Beispiel zeigt einige Anwendungsmöglichkeiten für eine `switch`-Anweisung in Java:

```
public class switchTest
{
  public static void testSwitch(int i)
  {
```

```
  switch(i)
    {
    case 1:
      System.out.println("Eins");
      // fallthrough
    case 2:
      System.out.println("Kleiner als 3");
      break;
    case 3:
    case 4:
      System.out.println("3 oder 4");
      break;
    default:
      System.out.println("Was anderes");
      break;
      }
    }

  public static void main(String[] args)
  {
    if (args.length > 0)
      testSwitch(Integer.parseInt(args[0]));
  }
}
```

Das Programm akzeptiert einen Parameter, wandelt ihn in einen `int` um, und gibt aufgrunddessen verschiedene Texte aus. Im Fall `1` wird sowohl `Eins` als auch `Kleiner als 3` ausgegeben, da ein `break` zwischen den beiden Fällen fehlt. C# erlaubt aber genau diesen Fall nicht. Ein *fallthrough* wie bei den Fällen `3` und `4` ist aber sehr wohl erlaubt, da bei `3` keine Anweisung steht. Ein weiterer Unterschied in C# switches liegt darin, dass nicht nur primitive Typen unterstützt werden, sondern auch Strings. Daher bräuchte man in diesem speziellen Beispiel keine Umwandlung des Strings in einen Integer vorzunehmen. Folgender Code zeigt das gleiche Beispiel in C#, die Stelle mit dem illegalen *fallthrough* wurde auskommentiert:

```
public class switchTest
{
  public static void testSwitch(string i)
  {
    switch(i)
```

```
    {
    // case "1":
    // System.Console.Write("Eins.");
    // Fehler!
    case "2":
      System.Console.Write("Kleiner als 3");
      break;
    case "3":
    case "4":
      System.Console.Write("3 oder 4");
      break;
    default:
      System.Console.Write("Was anderes");
      break;
    }
  }

  static void Main(string[] args)
  {
    if (args.Length > 0)
    testSwitch(args[0]);
  }
}
```

2.14 `for`-Schleifen

Beide Sprachen haben eine `for`-Schleife, die genau gleich funktioniert. Um über Kollektionen wie Arrays oder Listen leichter iterieren zu können, bietet C# noch eine andere Schleife an: `foreach`. Damit ist es möglich über alle Kollektionen zu iterieren, die das `System.Collections.IEnumerable`-Interface implementieren. Für Arrays kann man beispielsweise statt einer normalen `for`-Schleife mit einer Zählervariable arbeiten:

```
static void Main(string[] args)
{
  for (int i=0; i<args.Length; ++i)
    System.Console.Write(args[i] + " ");
}
```

einfach folgendes schreiben:

```
static void Main(string[] args)
{
  foreach(string s in args)
    System.Console.Write(s + " ");
}
```

Java hat in der aktuellen Version 1.4.2 keine Entsprechung dafür. In der kommenden Version 1.5 wird aber ebenfalls ein *enhanced for loop* enthalten sein, der bereits mit dem Prototypcompiler [BCK+03] und mit der Betaversion nach Setzen des Switches `-source 1.5` ausprobiert werden kann. Es können damit alle Objekte durchiteriert werden, die eine `Collection`, egal ob generisch oder nicht, implementieren. Darunter fallen auch Arrays. Dabei weicht die Syntax ein wenig von der von C# ab:

```
public static void main(String[] args)
{
  for (String s : args)
    System.out.print(s + " ");
}
```

2.15 Delegates und Events

Delegates sind typsichere Funktionszeiger. Sie können nur in C# eingesetzt werden, in Java gibt es nichts Vergleichbares, und es ist auch scheinbar nicht geplant etwas wie Delegates in naher Zukunft in Java einzubauen [Int03].

Folgendes einfache Beispielprogramm zeigt den Einsatz von Delegates:

```
public delegate void Printer(string text);

public class DelegateTest
{
  public void einPrinter(string msg)
  {
    System.Console.Write(msg);
  }

  public static void log(Printer printer)
  {
    printer("test");
  }
```

```
  static void Main(string[] args)
  {
    DelegateTest d = new DelegateTest();
    log(new Printer(d.einPrinter));
  }
}
```

In der ersten Zeile wird ein Delegate deklariert, und legt damit fest wie die Signatur einer Methode aussehen muss, die aufgerufen werden kann. Der Methode `log()` wird dann die Methode `einPrinter()` als Delegate übergeben, die dann mit dem Parameter `"test"` aufgerufen wird.

Es ist außerdem möglich, einer Delegate-Variable mehr als nur eine Methode zuzuweisen. Das Hinzufügen wird mit dem Operator `+=` durchgeführt. Ein Aufruf des Delegate ruft dann alle Methoden auf. Nach Belieben kann man auch einzelne Methoden wieder wegnehmen, analog zum Hinzufügen geschieht das mit dem `-=` Operator. Folgendes Beispiel demonstriert das Aufrufen mehrerer Methoden mit Delegates. Es wird sowohl `einPrinter()` als auch `einZeilenPrinter()` mit dem String `"test"` aufgerufen:

```
public delegate void Printer(string text);

public class DelegateTest
{
  public Printer printer;

  public void einPrinter(string msg)
  {
    System.Console.Write(msg);
  }

  public void einZeilenPrinter(string msg)
  {
    System.Console.WriteLine(msg);
  }

  public void log()
  {
    printer("test");
  }

  static void Main(string[] args)
```

```
  {
    DelegateTest d = new DelegateTest();

    d.printer += new Printer(d.einPrinter);
    d.printer += new Printer(d.einZeilenPrinter);

    d.log();
  }
}
```

Mit Delegates lassen sich leicht Events realisieren, die zum Beispiel bei der Programmierung von Benutzerschnittstellen sehr verbreitet sind: Ein Objekt registriert sich bei einem anderen, um dann zum Beispiel bei einer Zustandsänderung des Objektes eine Nachricht zu erhalten und entsprechend darauf reagieren zu können. Speziell für solche Events hat C# ein eigenes Schlüsselwort: `event`. Angefügt an die Delegate-Referenz `printer` im obigen Beispiel wird daraus ein Event:

```
public class DelegateTest
{
  public event Printer printer;
  ...
```

Diese Änderung bewirkt, dass, obwohl `printer public` ist, Zugriff von außen darauf nur noch sehr eingeschränkt möglich ist. Ereignisbehandlungsmethoden können nur noch mit dem Operator `+=` hinzugefügt und mit `-=` weggenommen werden. Das Aufrufen der so registrierten Methoden ist nur mehr von innerhalb der Klasse möglich.

Java hat wie gesagt keine Delegates, und es ist auch nicht geplant in Zukunft etwas Vergleichbares in diese Sprache einzubauen. Auch wenn Sun Microsystems argumentiert, dass Delegates unnötig sind [Sun], versuchen Programmierer diese in Java mit verschiedenen Methoden nachzuahmen. Dabei wird zum Beispiel probiert, die Eigenschaften von Delegates mit Hilfe von Laufzeittypinformation über Reflektion zu nachzubilden. Anders als auf diese Art ist es kaum möglich, in Java Referenzen auf statische und nicht statische Methoden, die keine Gemeinsamkeiten außer den übergebenen Parametertypen haben, zu erhalten und diese nach Bedarf aufzurufen. Nachteile davon sind, dass der Compiler solchen Code nicht auf Typsicherheit überprüfen kann, und dass aufgrund des Versuches, die Funktionalität von Delegates in einer nicht dafür entworfenen Sprache nachzuahmen, komplizierte Konstrukte mit viel Overhead entstehen.

Eine einfachere Alternative zu dieser Vorgangsweise sind Javas anonyme und innere Klassen, wie Sie beispielsweise oft in der Programmierung mit *Swing* und seiner Ereignisbehandlung eingesetzt werden. Dabei muss eine Klasse, die bestimmte Ereignisse erhalten will, das dafür vorgesehene Interface implementieren und dieses bei dem Objekt, von dem das Ereignis ausgeht, mit dem Aufruf einer Methode registrieren, vergleichbar zum `+=` Operator in C#. Wenn man für den Ereignis-Empfänger statt einer Methode in C# eine innere Klasse in Java nimmt, so kann man C#-Code mit Delegates fast eins zu eins nach Java übertragen. Statt der Delegate-Definition wird einfach ein Interface definiert, statt der Methoden werden innere Klassen verwendet. Als Beispiel hier die Übersetzung des letzten C# Beispiels nach Java:

```
import java.util.Vector;

interface Printer // entspricht der Delegate-Definition
{
  void print(String msg);
}

public class DelegateTest
{
  private class einPrinter implements Printer
  {
    public void print (String msg)
    {
      System.out.print(msg);
    }
  }

  private class einZeilenPrinter implements Printer
  {
    public void print (String msg)
    {
      System.out.println(msg);
    }
  }

  public void log()
  {
    for (int i=0; i<handler.size(); ++i)
```

```
      handler.get(i).print("test");
  }

  DelegateTest()
  {
    add(new einPrinter());
    add(new einZeilenPrinter());

    log();
  }

  public static void main(String[] args)
  {
    DelegateTest d = new DelegateTest();
  }

  private Vector<Printer> handler = new Vector<Printer>();

  public void add(Printer p) // entspricht +=
  {
   handler.add(p);
  }
}
```

Es wurde lediglich ein (generischer) Vektor zur Speicherung aller aufzurufenden Methoden, und eine dem `+=`-Operator entsprechende Methode hinzugefügt. Außerdem wurde der Testcode in den Konstruktor `DelegateTest()` verlegt, um die inneren Klassen anlegen zu können. Das Beispiel funktioniert genau wie die C#-Version. Statt innerer Klassen hätten auch anonyme Klassen verwendet werden können.

Zwischen Microsoft und Sun ist vor allem umstritten, ob es besser ist Delegates oder innere Klassen zu verwenden, und beide Firmen haben Dokumente veröffentlicht, die sich mit dieser Thematik auseinandersetzen und jeweils die eigene Technologie favorisieren. Offensichtliche Nachteile von inneren Klassen sind, dass diese ein wenig schlechter lesbar sind, und kein automatisches Konstrukt zum Hinzufügen und Entfernen von Eventhandlern bieten. Delegates in C# haben den Nachteil, dass es zum Beispiel bei Events nicht möglich ist, nach Rückgabe eines bestimmten Wertes die Weiterreichung des Events an andere Delegates zu beenden. [Sun], [Mic]

2.16 Anonyme Klassen und Methoden

In Java gibt es die Möglichkeit, anonyme Klassen zu verwenden. Es handelt sich dabei um Klassen, die keinen Namen haben (daher anonym), und genau an der Stelle instanziiert werden, wo sie deklariert werden. Häufig werden diese verwendet, um Rückruffunktionen für Ereignisse bereitzustellen (siehe Abschnitt 2.15). Folgendes Beispiel demonstriert den Einsatz einer anonymen Klasse:

```
class EventReceiver
{
  public void OnEvent(int param)
  {
    // leere implementierung
  }
}

public class AnonymTest
{
  public static void execute(EventReceiver e )
  {
    e.OnEvent(42); // rufe e auf
  }

  public static void main(String[] args)
  {
    execute( new EventReceiver()
      {
        public void OnEvent(int param)
        {
          System.out.print("Ereignis war:" + param);
        }
      });
  }
}
```

Beim Aufruf der Funktion `execute` wird die Instanz einer neuen anonymen Klasse erstellt, die die Methode `OnEvent()` von `EventReceiver` überschreibt und damit dafür sorgt, dass der übergebene Parameter zusammen mit einem kurzem Text ausgegeben wird.

Anonyme Klassen können, wie in Abschnitt 2.15 beschrieben, wie innere Klassen als Gegenstück zu Delegates in C# eingesetzt werden. C# hat der-

zeit in der aktuellen Version 1.2 keine Entsprechung zu anonymen Klassen. Mit Hinblick auf den Einsatz als Delegates werden aber in C# Version 2.0 anonyme Methoden eingeführt. Damit wird man dann das obere Beispiel folgendermaßen übertragen können:

```
public delegate void Event(int param);

public class AnonymTest
{
  public static void execute(Event e)
  {
    e(42); // rufe e auf
  }

  static void Main()
  {
    execute( delegate(int param)
      {
        System.Console.Write("Ereignis war:" + param);
      } );
  }
}
```

2.17 Generizität

Java und C# unterstützen in den aktuellen Versionen keine Generizität. Allerdings ändert sich das in der nächsten Version beider Sprachen: Sowohl in C# 2.0 als auch Java 1.5 wird es voraussichtlich volle Unterstützung für Generizität geben. Anhand des von Sun Microsystems veröffentlichten Prototypcompilers [BCK+03], der Betaversion 1.5 und der Sprachspezifikation von C# 2.0 [Mic03] ist es schon möglich, sich einen guten Überblick über die Unterschiede der beiden Sprachen beim Einsatz von Generizität zu verschaffen.

Die Syntax ist in beiden Sprachen sehr ähnlich, Typparameter bei Methoden oder Klassen werden, wie vielleicht aus anderen Sprachen wie C++ bekannt, in spitzen Klammern (z.B. `<T>`) angegeben. Um zum Beispiel einen generischen Vektor für Strings zu verwenden, schreibt man in beiden Sprachen:

```
Vector<String> v = new Vector<String>();
```

Generische Klassen und Methoden definiert man genauso mit spitzen Klammern nach ihrem Namen. Es können mehr als ein Typparameter angegeben werden, welche dann innerhalb der Klasse verwendet werden können, als ob es gewöhnliche Typen wären:

```
public class Dictionary<K, V>
{
  public void add(K k, V v) { ... }
}
```

Gebundene Generizität wird in C# und Java ebenfalls unterstützt, jedoch unterscheidet sich die Syntax hier ein wenig. Will man, dass der Typparameter `K` im obigen Beispiel zumindest das Interface `IComparable` implementiert, um dessen Methoden verwenden zu können, schreibt man das so:

```
// Java:
public class dictionary<K extends IComparable, V>  { ... }

// C#:
public class dictionary<K, V> where K: IComparable { ... }
```

In beiden Sprachen können für jeden Typparameter als Beschränkungen genau eine Klasse und mehrere Interfaces festgelegt werden.

Java setzt homogene Übersetzung ein, um Generizität verwenden zu können. Dabei werden bei der Kompilierung alle Typparameter durch einen gemeinsamen Obertypen ersetzt. Dies ist der Typ `Object` bei ungebundenen Typparametern, ansonsten die Klasse der ersten Schranke. Da in Java primitive Typen keinen gemeinsamen Obertypen haben, können nur Referenztypen als Typparameter verwendet werden. In C# haben einfache Typen, wie in Abschnitt 2.2 beschrieben, indirekt `object` als Obertypen. Folgende Zeile ließe daher in Java nicht kompilieren, in C# allerdings schon:

```
dictionary<String, int> d = new dictionary<String, int>();
```

2.18 Enumerationen

Um einen Wert mit einer bestimmten Bedeutung aus einer Anzahl endlich verschiedener Möglichkeiten festzulegen, kann man Konstanten verwenden. Um beispielsweise einer Funktion eine von vier Arten von Speichergrößeneinheiten zu übergeben, kann für jede Art eine eigene Konstante mit einem eindeutigen Wert definiert werden. Folgendes Beispiel demonstriert dies in Java:

```
public class EnumTest
{
  public static final int MEMORY_BIT = 0;
  public static final int MEMORY_BYTE = 1;
  public static final int MEMORY_KILOBYTE = 2;
  public static final int MEMORY_MEGABYTE = 3;

  // memoryValue muss eine der oberen Konstanten sein.
  public static void foo(int memoryValue)
  {
    // tu irgendwas
  }

  public static void main(String[] args)
  {
    foo(MEMORY_BIT); // ok
    foo(42);         // Problem ?
  }
}
```

Die im Beispiel gezeigte Vorgehensweise zur Realisierung solcher Enumerationen ist die einzige Möglichkeit, die Java bis Version 1.4.2 bietet. Diese ist aber nicht sehr vorteilhaft: Enumerationen dieser Art sind nicht typsicher. Der zweite Aufruf von `foo()` in der Main-Methode mit dem Parameter `42` verdeutlicht das: `42` ist kein Wert, den `foo()` akzeptiert, der Compiler hat aber keine Möglichkeit das in irgendeiner Weise zu überprüfen. Lediglich der Kommentar über der Funktion `foo` kann dem Programmierer anzeigen, dass der Funktion nur eine der angegebenen Konstanten übergeben werden darf, und das ist recht unsicher. Wenn `foo()` mit einem anderem Wert aufgerufen wird, könnte es je nach Implementierung von `foo()` zu Programmfehlern kommen.

Enumerations-Typen vermeiden dieses Problem, diese gibt es sowohl in C#, als auch in Java ab Version 1.5, jedoch mit einigen Unterschieden. Obiges Beispiel lässt sich leicht nach Java 1.5 übertragen, in dem man statt der Konstante den neu eingeführten `enum`-Typ verwendet:

```
public class EnumTest
{
  public enum Memory { BIT, BYTE, KILOBYTE, MEGABYTE }

  public static void foo(Memory memoryValue)
```

```
  {
    // tu irgendwas
  }

  public static void main(String[] args)
  {
    foo(Memory.BIT); // ok
    // foo(42);      // Compiler protestiert
  }
}
```

Typsicherheit ist nun gewährleistet: Der Compiler verweigert zu Recht die Zeile mit `foo(42)` zu kompilieren. Zudem ist das Programm auch wesentlich übersichtlicher geworden. Die Namen aller Enumerationswerte befinden sich nun in einem eigenen Namensraum, und müssen nicht mehr mit dem Präfix `MEMORY_` angeschrieben werden.

Das Beispielprogramm läßt sich ohne großartige Änderungen (unterschiedliche Signatur der `Main()`-Methode) in C# übernehmen. Genau wie in Java weigert sich auch hier der Compiler den Aufruf `foo(42)` zu akzeptieren. In C# sind Enumerationen jedoch einfache Typen und nicht Referenztypen wie in Java. Jeder Enumerationstyp in C# hat damit einen unterliegenden Typ, der, wenn nicht anders angegeben, `int` ist. Somit kann man einfach zwischen Enumerationstypen und ihren unterliegenden Typen hin- und herkonvertieren. Ausserdem ist es damit auch möglich, die statische Typsicherheit wie im oberen Beispiel zu umgehen. Dies ist manchmal recht praktisch, wenn ein Enumerationswert aus anderen Daten errechnet werden muss, birgt aber natürlich auch Gefahren. Folgendes Programm entspricht oberem Beispiel in C#, außerdem demonstriert es die letztgenannte Möglichkeit:

```
public class EnumTest
{
  public enum Memory { BIT, BYTE, KILOBYTE, MEGABYTE }

  public static void foo(Memory memoryValue)
  {
    // tu irgendwas
  }

  static void Main()
  {
    foo(Memory.BIT); // ok
```

```
    foo(42);          // Compiler protestiert
    foo((Memory)42); // ok
  }
}
```

In beiden Sprachen ist es möglich, jedem Element der Enumeration einen Wert zuzuweisen. Die Möglichkeiten in C# sind hier jedoch gegenüber Java eingeschränkt. Der unterliegende Typ einer Enumeration kann nur ein ganzzahliger primitiver Typ sein, also entweder `byte, sbyte, short, ushort, int, uint, long`, oder `ulong`. Will man also, dass jedes Element der Memory-Enumeration aus den oberen Beispielen als Wert die Anzahl der enthalten Bits erhält, kann man das so erreichen:

```
public enum Memory : long
{
  BIT = 1,
  BYTE = 8,
  KILOBYTE = 8192,
  MEGABYTE = 8388608
}
```

Es ist in C# jedoch unmöglich, dass das Element einer Enumeration beispielsweise einen String als Wert erhält. Das ist in den meisten Fällen jedoch auch unsinnig, durch eine Enumeration will man schließlich meist durchzählen können. Will man auf Namen für Elemente aus Enumerationen nicht verzichen, ist es möglich, diese mit einem Aufruf von `Enum.getName()` zu erhalten:

```
string name = System.Enum.GetName(typeof(Memory), Memory.BIT);
```

Um bei dem problematischen Aufruf von zum Beispiel `foo((Memory)42)` in der Methode `foo()` überprüfen zu können, ob der übergebene Wert tatsächlich Element der Enumeration ist, kann man `Enum.IsDefined` verwenden:

```
Enum.IsDefined(typeof(Memory), memoryValue);
```

Da Enumerationen in Java Referenztypen sind und viel mit Klassen gemeinsam haben, können diese im Unterschied zu jenen in C# auch Konstruktoren, Instanzvariablen und Methoden besitzen. Mit Hilfe dieser ist es unter Anderem möglich, zu jedem Element einer Enumeration einen Wert abzuspeichern. Das Beispiel, in dem jedes Element der Enumeration den Wert der Anzahl der enthaltenen Bits abspeichert, würde in Java 1.5 so realisiert werden können:

```
enum Memory
{
  BIT(1), BYTE(8), KILOBYTE(8192), MEGABYTE(8388608);

  Memory(int v)
  {
    value = v;
  }

  public int value;
}
```

Das ist eine ganz andere Herangehensweise als in C#. Der abgespeicherte Wert müsste hier in der Methode `foo()` mit `memoryValue.value` abgefragt werden. Mit Sicherheit ist das ein wenig unübersichtlicher, jedoch eröffnen sich dem Programmierer hier mehr Möglichkeiten. Zum Beispiel können so auch ganz andere Werte wie Strings zusätzlich zu jedem Enumerationselement abgespeichert werden. Durch Hinzufügen eines weiteren Konstruktors, der einen anderen Typ als `int` als Parameter akzeptiert, wäre es sogar möglich je nach verwendetem Element andere zusätzliche Daten abzuspeichern.

2.19 Exceptions

Wenn in Java eine Exception in einer Methode geworfen wird, die nicht Unterklasse von `RuntimeException` ist, muss die Signatur der Methode demensprechend mit `throws` markiert werden. Das heißt, jede andere Methode, die eine solche Methode aufruft, und nicht selbst die gleiche Markierung aufweist, wird gezwungen mit Hilfe eines `try-catch`-Blockes Vorkehrungen dafür treffen, dass eine solche Exception behandelt wird. Folgendes Beispiel demonstriert das:

```
public class ExceptionTest
{
  public static void foo() throws Exception
  {
    throw new Exception("Es ist etwas böses passiert.");
  }

  public static void main(String[] args)
  {
```

```
    try
    {
      foo();
    }
    catch(Exception e)
    {
      System.out.print(e);
    }
  }
}
```

Würde man versuchen, den `try-catch`-Block in der `main()`-Methode zu entfernen, würde der Compiler sich weigern den Code zu kompilieren. Möglichkeiten um zu vermeiden, die Exception abfangen zu müssen, wären, aus der Exception eine RuntimeException (oder eine Unterklasse davon) zu machen, oder die `main()`-Methode selbst mit `throws Exception` zu markieren.

Mit diesem Konstrukt können Benutzer von fremden Methoden gezwungen werden, bestimmte Ausnahmesituationen zu behandeln, was an vielen Stellen in Programmen äußerst sinnvoll ist. In C# gibt es diese Möglichkeit jedoch nicht. Programmierer dürfen nach Belieben Exceptions ignorieren:

```
public class ExceptionTest
{
  public static void foo()
  {
    throw new System.Exception("Es ist etwas böses passiert.");
  }

  public static void Main()
  {
    foo();
  }
}
```

Das Beispiel kompiliert ohne Probleme in C#. Man könnte also sagen, dass in C# alle Exceptions RuntimeExceptions sind. Es ist mit Sicherheit bequemer, bei Aufruf einer Methode, die unter Umständen eine Ausnahme werfen könnte, nicht unbedingt einen `try-catch`-Block schreiben zu müssen, allerdings können dadurch auch leicht Probleme übersehen werden, die nur durch Studium der Dokumentation oder Implementierung der jeweiligen Methode, und ansonsten erst während der Laufzeit, schmerzhaft sichtbar werden.

3 Schlussbemerkungen

Während der Erstellung dieser Arbeit habe ich neben zahlreichen kleinen Testprogrammen auch ein im Verhältnis dazu großes Programm implementiert, einmal in Java und einmal in C#. Es handelt sich dabei um ein einfaches Spiel im Stil von Snakes, mit Hauptmenü, Highscore-Tablle und Spielmodus in verschiedenen Schwierigkeitsstufen und mehreren Levels. Die Abbildungen 1 und 2 zeigen Screenshots der beiden Programme, die sich bis auf einige Kleinigkeiten fast identisch verhalten.

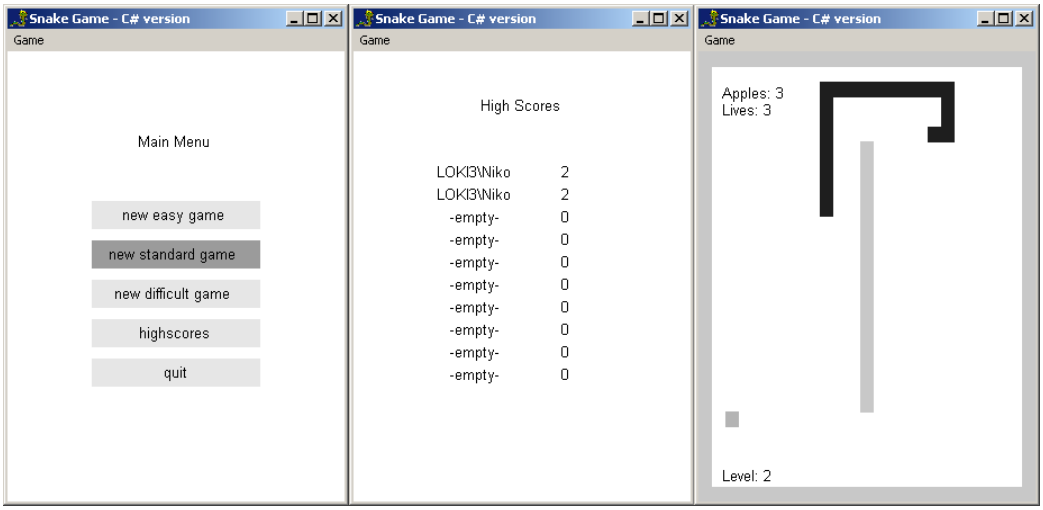

Abbildung 1: Die C#-Version des Spieles. Da das Programm ohne eine VM läuft, ist die Bildwiedergabe wesentlich schneller als bei der Java-Version (Abb. 2), durch die frameunabhängige Implementierung der Spiellogik merkt das der Benutzer jedoch nicht.

Bei der Implementierung der Spiele und der anderen Testprogramme sind mir außer den in dieser Arbeit angeführten Unterschieden in den Sprachen noch viele andere interessante Dinge aufgefallen, die aber für diese Arbeit nicht relevant waren, es geht hier schließlich wirklich nur um die Sprachen selbst. Beispielsweise bemerkt man schnell Unterschiede in der Performance beider Sprachen, nicht nur beim produzierten Code, sondern auch bei den Compilern selbst. Auch haben zum Beispiel die Bibliotheken von C# und Java völlig unterschiedliche Stärken und Schwächen.

3.1 Fazit und Ausblick

Java und C# sind zwei sehr ähnliche Sprachen, die aber, wie in dieser Arbeit gezeigt wurde, im Detail viele Unterschiede aufweisen. C# ist eine neuere

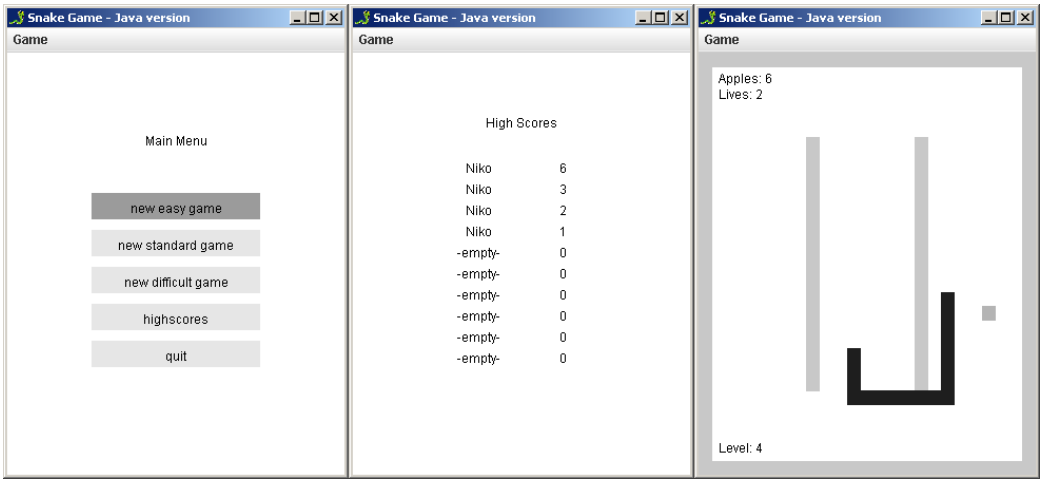

Abbildung 2: Das in Java ausprogrammierte und in der VM 1.5.0-beta laufende Spiel. Die neue VM scheint ein Windows-XP Theme zu verwenden, daher ist der Menübalken silbern eingefärbt, ansonsten sind die Programme fast identisch.

Sprache als Java, verfügt über fast alle Features des Konkurrenten, und erweitert diese in den meisten Fällen. Durch die vielen möglichen neuen Features in C# hat der Programmierer mehr Einfluss auf die Generierung und Verwendung von Code, beispielsweise durch die Einführung des Schlüsselwortes `virtual`, mit dem Methoden markiert werden müssen, wenn sie virtuell sein sollen. Dies kann am Anfang wie andere Dinge auch gewöhnungsbedürftig sein, vor allem in diesem Zusammenhang das bei überschreibenden Methoden notwendige `overrride` wird dem Java gewöhnten Programmierer am Anfang ein wenig lästig.

Sun Micosystems Java und Microsofts C# sind direkte Konkurrenten, wovon der Endbenutzer, also der Programmierer, deutlich profitiert. Mit den auch schon in dieser Arbeit mehrmals angeschnittenen voraussichtlichen Änderungen und Erweiterungen der beiden Sprachen versuchen Sun und Microsoft auf die Bedürfnisse der Benutzer einzugehen. Es ist sicher interessant zu erleben, wie sich die Sprachen in Zukunft weiterentwickeln werden.

Literatur

[Bay02] Jürgen Bayer. *C#*. Addison-Wesley, Martin-Kollar-Straße 10-12, D-81829 München/Germany, 2002.

[BCK+01] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. *Adding Generics to the Java Programming Language: Participant Draft Specification*, April 2001.

[BCK+03] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. *Adding Generics to the Java Programming Language: Participant Draft Specification, Version 2.0*, June 2003.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, June 2000.

[Haw02] Florian Hawlitzek. *Java 2*. Addison-Wesley, Martin-Kollar-Straße 10-12, D-81829 München/Germany, 2002.

[Int03] New Java Language Features in J2SE 1.5, interview with Joshua Bloch and Neal Gafter. `http://java.sun.com/developer/community/chat/JavaLive/2003/jl0729.html`, July 2003.

[Mic] Article: The Truth About Delegates. `http://msdn.microsoft.com/vjsharp/productinfo/visualj/visualj6/technica%l/articles/general/truth/default.aspx`.

[Mic02] Microsoft Corporation. *C# Language Specification Version 1.2*, December 2002.

[Mic03] Microsoft Corporation. *C# Version 2.0 Specification*, July 2003.

[Sun] White Paper: About Microsoft's "Delegates". `http://java.sun.com/docs/white/delegates.html`.

[Sun01] Sun Microsystems, Inc. *Java Object Serialization Specification Revision 1.5.0*, 2001.

[Sun03] Sun Microsystems, Inc. *A Program Annotation Facility for the Java Programming Language, JSR-175 Public Draft Specification*, 2003.

[Sun04a] Sun Microsystems, Inc. *A Typesafe Enum Facility for the Java Programming Language, Public Review Draft*, 2004.

[Sun04b] Sun Microsystems, Inc. *Adding Variable-Arity Methods (Varargs) to the Java Programming Language, Public Review Draft*, 2004.

[Sun04c] Sun Microsystems, Inc. *An enhanced for loop for the Java Programming Language, Public Review Draft*, 2004.

[Sun04d] Sun Microsystems, Inc. *Autoboxing and Auto-Unboxing support for the Java Programming Language, Public Review Draft*, 2004.

[Sun04e] Sun Microsystems, Inc. *Importing Static Members in the Java Programming Language, Public Review Draft*, 2004.