# XML und Hello C#


# Übersicht

- XML
  - What? / Why?
  - Konzepte und Theorie
  - Syntax
  - Schemas
  - Code
- Einführung in das .NET
  - Konzepte / Architektur
  - Einführung in C#, Unterschiede zu Java


# was ist XML?

- XML ist eine markup Sprache, wie HTML
- XML ist entworfen um Data zu speichern und zu transportieren
  - Data Encoding
- XML ist selbstbeschreibend
- XML ist eine W3C Empfehlung
  - https://www.w3.org/

# XML vs HTML

- HTML =  HyperText Markup Language
- Entworfen für eine bestimmte Anwendung und zwar die Anzeige und Verlinkung von hypertext Dokumenten
- XML beschreibt Struktur und Inhalt
- die Präsentation ist getrennt von Struktur und Inhalt

# XML Beispiel

```
<addresses>
    <person>
        <name> Bob </name>
        <tel> 0740123456 </tel>
        <email> bob@email.com </email>
    </person>
    <person>
        <name> Dob </name>
        <tel> 0740123457 </tel>
        <email> dob@email.com </email>
    </person>
</addresses>
```

# XML

- die Menge von Tags ist nicht fest
- zusätzlichen Tags können definiert werden
- eine solche Menge kann in vielen Anwendungen verwendet werden
- Namespaces ermöglichen eine einheitliche und kohärente Beschreibung von Daten
- Zum Beispiel ein Namespace AddressBook legt fest, ob `<tel>` oder `<phone>` verwendet werden soll


# XML

- XML hat ein Schema Konzept
  - DTD und XML Schema
- XML ist ein Datenmodell
  - Ähnlich wie Semistrukturierten Daten
- XML unterstützt Internationalisierung (Unicode)
- XML ist Plattformunabhängig
  - eine XML Datei ist eine ganz normale Textdatei

# XML Beispiel

**STUDENTS**

| SID | FIRST | LAST | EMAIL |
|---|---|---|---|
| 101 | Ann | Smith | ... |
| 102 | Michael | Jones | (null) |
| 103 | Richard | Turner | ... |
| 104 | Maria | Brown | ... |

**EXERCISES**

| CAT | ENO | TOPIC | MAXPT |
|---|---|---|---|
| H | 1 | Rel. Algeb. | 10 |
| H | 2 | SQL | 10 |
| M | 1 | SQL | 14 |

**RESULTS**

| SID | CAT | ENO | POINTS |
|---|---|---|---|
| 101 | H | 1 | 10 |
| 101 | H | 2 | 8 |
| 101 | M | 1 | 12 |
| 102 | H | 1 | 9 |
| 102 | H | 2 | 9 |
| 102 | M | 1 | 10 |
| 103 | H | 1 | 5 |
| 103 | M | 1 | 7 |

# XML Beipspiel

```
<GRADES-DB>
    <STUDENT SID=’101’ FIRST=’Ann’ LAST=’Smith’/>
    <STUDENT SID=’102’ FIRST=’Michael’ LAST=’Jones’/>
    …
    <EXERCISE CAT=’H’ ENO=’1’ TOPIC=’Rel. Algeb.’/>
    …
    <RESULT SID=’101’ CAT=’H’ ENO=’1’ POINTS=’10’/>
    ...
</GRADES-DB>
```

```
<GRADES-DB>
    <STUDENTS>
        <STUDENT>
            <SID>101</SID>
            <FIRST>Ann</FIRST>
            <LAST>Smith</LAST>
        </STUDENT>
    …
    </STUDENTS>
    ...
</GRADES-DB>
```

# XML ist selbstbeschreibend

- Normalerweise ist eine Datei nur ein Bitstrom
- Nur ein Programm, das diese Datei verarbeitet, enthält Details zu
  - Aufteilen des Bitstroms in Objekten
  - Aufteilen jedes Objekt in Felder
  - Der Typ jedes Datenfelds
- Problem: man kann diese Programme verlieren
  - im dem Fall sind die Daten praktisch verloren
- Bei XML ist dies nicht der Fall

# XML für Data Exchange


- Webservices (z. B. E-Commerce) erfordern den Datenaustausch zwischen verschiedenen Anwendungen
  - die auf verschiedenen Plattformen ausgeführt werden
- XML (erweitert um Namespaces) ist ein guter Kandidat für den Datenaustausch im Web
  - nicht der Einzige

# XML and Friends


- `XML-Schemas` stärken die Datenmodellierung Funktionen von XML
  - im Vergleich zu XML mit nur DTDs
- `XPath` ist eine Sprache für den Zugriff auf Teile von XML-Dokumenten
- `XLink` und `XPointer` unterstützen Cross-Referencing
- `XSLT` ist eine Sprache zum Transformieren von XML-Dokumenten in andere XML-Dokumente
  - Styling von XML kann nur mit CSS erfolgen
- `XQuery` ist eine Sprache zum Abfragen von XML-Dokumenten

# XML Struktur


- XML besteht aus Text und Tags
- Tags schreibt man in Paaren
  `<date> ... </date>`
- Tags muss man richtig verschalten
  `<date> ... <day> ... </day> ... </date>`

# Abkürzung

- Abkürzung von leeren Elementen

  `<br></br>` → `<br/>`

  `<hr width="10"></hr>` → `<hr width="10"/>`

- Beispiel

```
<family>
   <person id = "bob">
      <name> Bob </name>
      <mother idref = "dob"/>
      <father idref = "lob"/>
   </person>
</family>
```

# XML Text


- XML hat nur ein Typ: `Text`
- ist durch Tags begrenzt
  ```
  <title> The Big Sleep </title>
  <year> 1935 </ year>
  ```
- XML Text → PCDATA (parsed character data)
  - 16-bit encoding

# XML Struktur

- Nested Tags können verwendet werden, um verschiedene Strukturen auszudrücken, z. B. ein Tupel (Datensatz)

```
<person>
   <name> Bob </name>
   <tel> 0740123456 </tel>
   <tel> 0740123457 </tel>
   <email> bob@email.com </email>
</person>
```

# XML Struktur


- Man kann eine Liste darstellen, wenn man dasselbe Tag wiederholt verwendet:

```
<addresses>
   <person> ... </person>
   <person> ... </person>
   <person> ... </person>
   <person> ... </person>
    ...
</addresses>
```

# XML Struktur

```
<addresses>
   <person>
      <name> Bob </name>
      <tel> 0740123456 </tel>
      <email> bob@email.com </email>
   </person>
   <person>
      <name> Dob </name>
      <tel> 0740123457 </tel>
      <email> dob@email.com </email>
   </person>
</addresses>
```

# Begriffe

- der Teil eines XML-Dokuments zwischen einem öffnenden und einem entsprechenden schließenden Tag wird als Element bezeichnet

```
<person>
   <name> Bob </name>
   <tel> 0740123456 </tel>
   <tel> 0740123457 </tel>
   <email> bob@email.com </email>
</person>
```

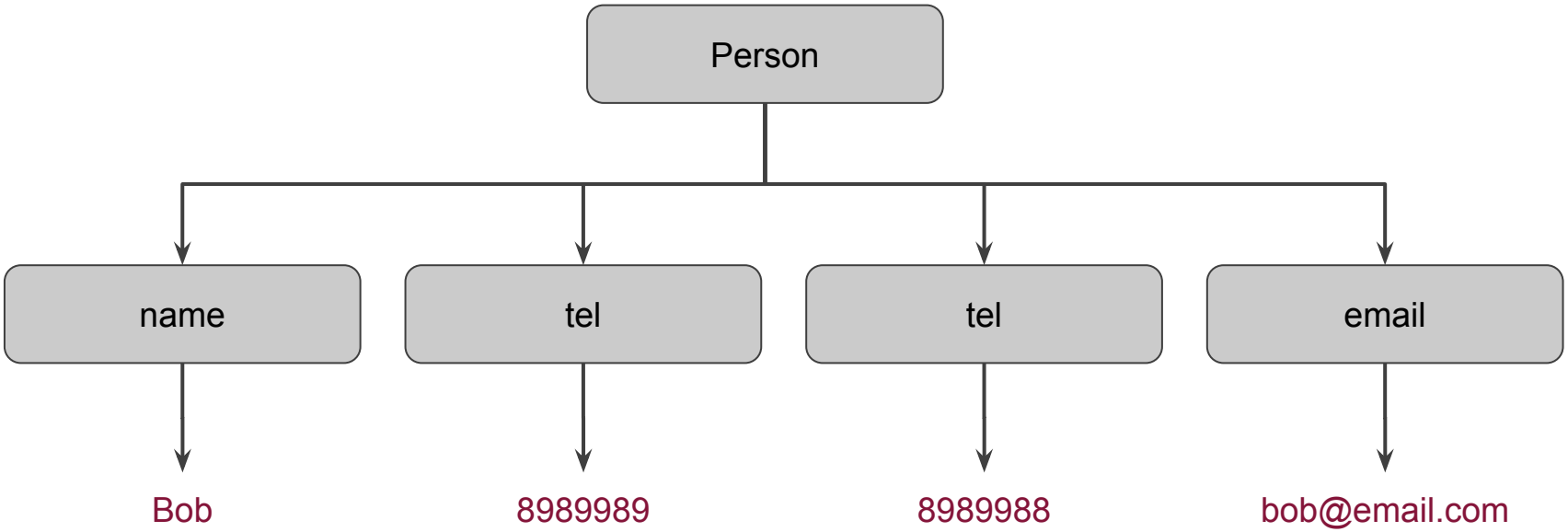
Element, sub-element of

Element

not Element

# ein XML-Dokument ist ein Baum

- Endknoten sind entweder leer oder enthalten PCDATA
- semistrukturierte Datenmodelle platzieren normalerweise die Labels an den Kanten und sind Graphs, nicht nur Bäume



# Gemischter Inhalt

- Ein Element kann eine Mischung aus Unterelementen und PCDATA enthalten

```
<airline>
   <name> British Airways </name>
   <motto>
      World’s <dubious> favorite</dubious> airline
   </motto>
</airline>
```

# Header


- Standalone = "No" bedeutet, dass eine externe DTD vorhanden ist
- man kann das Encoding-Attribut weglassen, und UTF-8 wird als Standard verwendet werden

```
<?xml version=”1.0” standalone=”yes/no” encoding=”UTF-8”?>
```

# Komplettes Dokument

```
<?XML version ="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE addresses SYSTEM "dbi-addresses.dtd">
<addresses>
   <person>
      <name> Bob </name>
      <tel> 0740123456 </tel>
      <tel> 0740123457 </tel>
      <email> bob@email.com </email>
   </person>
</addresses>
```

# DTD (Document Type Definition)


- DTD fügt zusätzlichen syntaktischen Anforderungen hinzu
  - well-formed
  - valid
- hilft bei der Beseitigung von Fehlern beim Erstellen oder Bearbeiten von XML-Dokumenten
- verdeutlicht die Semantik
- vereinfacht die Verarbeitung von XML-Dokumenten

# Beispiel

- Wo kann in einem Adressbuch eine Telefonnummer erscheinen?
  - Unter <Person>, unter <Name> oder unter beiden?

- Wenn man nach allen Möglichkeiten suchen muss, dauert die Verarbeitung länger und es ist möglicherweise nicht klar, wem eine Telefonnummer gehört
  - Wir möchten wissen, dass eine Telefonnummer sowohl unter einer Abteilung als auch unter dem Leiter dieser Abteilung erscheinen darf

  - Wenn wir das nicht wissen und es nur eine Telefonnummer gibt, wissen wir möglicherweise nicht, ob es sowohl der Abteilung als auch dem Manager dient oder nur einem von denen

# valide XML-Dokumenten


- Ein XML-Dokument (mit oder ohne DTD) ist wohlgeformt, wenn Tags sind syntaktisch korrekt
- Jedes Tag hat ein End-Tag
- Tags sind richtig verschachtelt
- Es gibt ein Root-Tag
- Ein Start-Tag hat nicht zwei Vorkommen desselben Attributs

# valide XML-Dokumenten


- Ein wohlgeformtes XML-Dokument ist gültig, wenn es seiner DTD entspricht
  - das Dokument entspricht der Grammatik für reguläre Ausdrücke
  - die Attributtypen sind korrekt
  - die Bedingungen für Referenzen sind erfüllt

# DTD

- DTD setzen XML-Dokumenten eine Struktur durch
- es gibt eine Beziehung zwischen einer DTD und einem Schema, die jedoch nicht eng ist
- Daher sind zusätzliche „Typisierungssysteme“ erforderlich (XML-Schemas)
- Die DTD ist eine syntaktische Spezifikation

# Address Book

```
<person>
    <name> Homer Simpson </name>
    <greet> Dr. H. Simpson </greet>
    <addr>1234 Springwater Road </addr>
    <addr> Springfield USA, 98765 </addr>
    <tel> (321) 786 2543 </tel>
    <fax> (321) 786 2544 </fax>
    <tel> (321) 786 2544 </tel>
    <email> homer@math.springfield.edu </email>
</person>
```

Exactly one name

At most one greeting

As many address lines as needed (in order)

Mixed telephones and faxes

As many as needed

# die Struktur

- name → um ein Namenselement anzugeben
- greet? → um eine optionale (0 oder 1) Begrüßung anzugeben
- name,greet? → um einen Namen gefolgt von einer optionalen Begrüßung anzugeben
- adds* → um 0 oder mehr Adresse anzugeben
- tel | fax → um ein tel oder fax Element anzugeben
- (tel|fax)* → um 0 oder mehr tel oder fax anzugeben
- (tel,fax) → tel gefolgt von fax

# die Struktur

- Die gesamte Struktur eines Personeneintrags wird also durch angegeben
  `name, greet?, addr*, (tel | fax)*, email*`

- Dies ist als regulärer Ausdruck bekannt
  - Warum ist es wichtig?

# Reguläre Ausdrücke

- A → Tag (Element) A tritt auf
- e1, e2 → Der Ausdruck e1 gefolgt von e2
- e* → 0 oder mehr Vorkommen von e
- e? → Optional: 0 oder 1 Vorkommen
- e+ → 1 oder mehr Vorkommen
- e1 | e2 → entweder e1 oder e2
- (e) → Gruppierung
  - (ab){3} → ababab

# Definition eines Elementes


- Definition eines Elements besteht aus genau einem der folgenden
- ein regulärer Ausdruck
- EMPTY bedeutet, dass das Element keinen Inhalt hat
- ANY bedeutet, dass der Inhalt eine beliebige Mischung aus #PCDATA und in der DTD definierten Elementen sein kann
- #PCDATA

# Gemischter Inhalt


- Der gemischte Inhalt wird durch eine wiederholbare Group beschrieben `(#PCDATA | Elementname | ...) *`
- Innerhalb der Gruppe keine regulären Ausdrücke - nur Elementnamen
- #PCDATA muss an erster Stelle stehen, gefolgt von 0 oder mehr Elementnamen, die durch | getrennt sind
- Die Gruppe kann 0 oder mehrmals wiederholt werden

# Address Book DTD

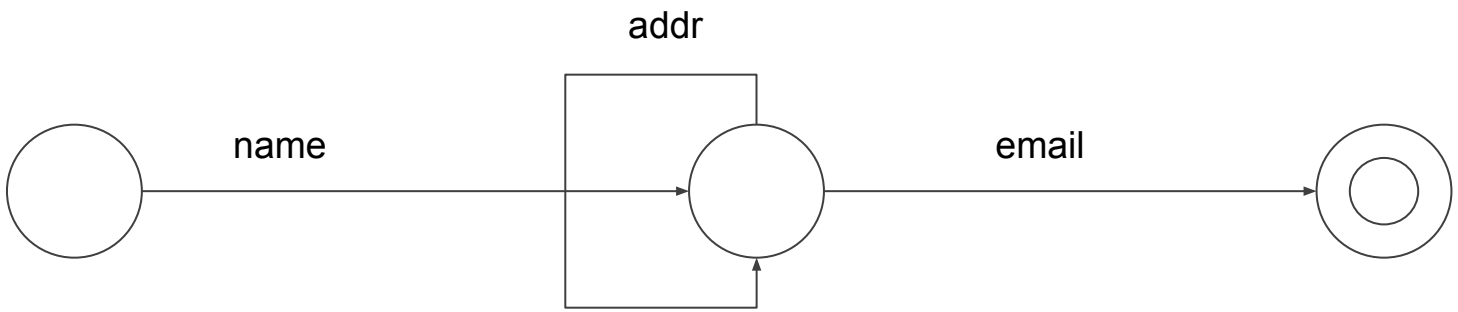
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE addressbook [
   <!ELEMENT addressbook (person*)>
   <!ELEMENT person
      (name, greet?, address*, (fax | tel)*, email*)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT greet (#PCDATA)>
   <!ELEMENT address (#PCDATA)>
   <!ELEMENT tel (#PCDATA)>
   <!ELEMENT fax (#PCDATA)>
   <!ELEMENT email (#PCDATA)>
]>
```

# Address Book XML

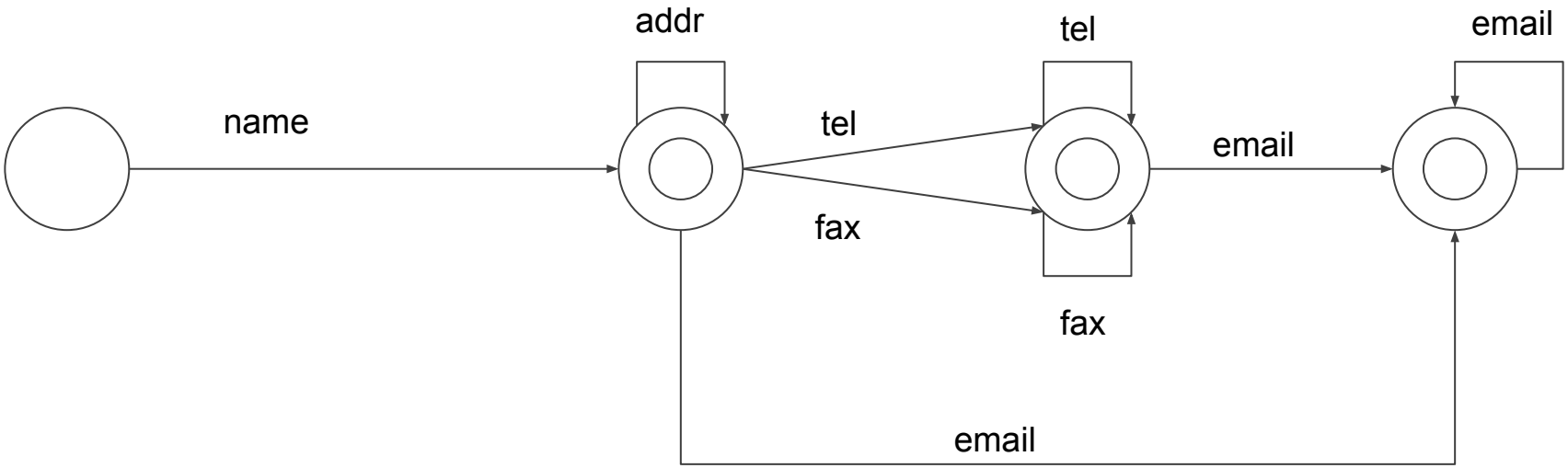
```
<addressbook>
   <person>
      <name> Bob </name>
      <greet> Mr. Bob </greet>
      <email> bob@email.com </email>
   </person>
</addressbook>
```

# Reguläre Ausdrücke

- Jeder reguläre Ausdruck bestimmt einen entsprechenden endlichen Automaten (finite-state automaton)
- einfaches Beispiel: `name, addr*, email`



# name, addr*, (tel | fax)*, email*



# Probleme

- Einige Sachen sind schwer zu spezifizieren
- Jedes Mitarbeiter-Element sollte in einer bestimmten Reihenfolge name, age und idnum Elemente enthalten

```
<!ELEMENT employee
( (name, age, idnum) | (age, idnum, name) |
(idnum, name, age) | ... )>
```

- Nehmen wir an, dass es viel mehr Felder gab!
- d.h. n! verschiedene Möglichkeiten

# Attribute in DTD

```
<!ELEMENT height (#PCDATA)>
<!ATTLIST height
   dimension CDATA #REQUIRED “cm”
   accuracy CDATA #IMPLIED>
```

- `dimension` ist erforderlich
- `accuracy` ist optional
- CDATA ist der Typ des Attributs
  - es bedeutet "character data" und kann eine beliebige Literal als Wert erhalten

# Attribute in DTD

```
<!ATTLIST element-name
attr-name attr-type default-value
attr-name attr-type default-value
...
attr-name attr-type default-value>
```

# Attribute in DTD


- CDATA
- (value | ... | ... )
- ID, IDREF, IDREFS
- ENTITY, ENTITIES
- NMTOKEN, NMTOKENS

# Attribute in DTD

- #REQUIRED → das Attribut muss enthalten sein
- #IMPLIED → ist Optional (null default wert)
- #FIXED “value” → Der angegebene Wert (in Anführungszeichen) ist der einzig mögliche
- "Wert" → Der default Wert des Attributs, wenn keiner angegeben ist

# IDs and IDREFs


- ID steht für identifier
  - Keine zwei ID-Attribute dürfen denselben Wert (vom Typ CDATA) haben
- IDREF steht für Identifier Reference
  - Jeder einem IDREF-Attribut zugeordnete Wert muss existieren als Wert eines ID-Attributs
- IDREFS gibt mehrere (0 oder mehr) References an

# IDs and IDREFs

```
<!DOCTYPE family [
<!ELEMENT family (person*)>
<!ELEMENT person (name)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST person
          id ID #REQUIRED
          mother IDREF #IMPLIED
          father IDREF #IMPLIED
          children IDREFS #IMPLIED>
]>
```

# XML Beispiel

```
<family>
   <person id="lisa" mother="marge" father="homer">
      <name> Lisa Simpson </name> </person>
   <person id="bart" mother="marge" father="homer">
      <name> Bart Simpson </name>
   </person>
   <person id="marge" children="bart lisa">
      <name> Marge Simpson </name>
   </person>
   <person id="homer" children="bart lisa">
      <name> Homer Simpson </name>
   </person>
</family>
```

# IDs and IDREFs


- Wenn ein Attribut als ID deklariert ist
  - der zugehörige Wert muss unterschiedlich sein
  - unterschiedliche Elemente müssen unterschiedliche Werte für das ID-Attribut haben
  - auch wenn die beiden Elemente unterschiedliche Namen haben
- Wenn ein Attribut als IDREF deklariert ist
  - Der zugehörige Wert muss als Wert eines ID-Attributs vorhanden sein
- Ähnliches gilt für alle Werte eines IDREFS-Attributs
- ID, IDREF und IDREFS werden nicht typisiert

# DTD und XML

- eine DTD kann internal sein
- die DTD ist Teil der Dokumentdatei (oder extern)
- die DTD und das Dokument befinden sich in separaten Dateien
- eine externe DTD kann sich Möglicherweise befinden
  - Im lokalen Dateisystem (wo sich das Dokument befindet)
  - In einem Remote-Dateisystem

# DTD und XML


- internal DTD

```
<?xml version="1.0"?>
<!DOCTYPE db [<!ELEMENT ...> ... ]>
<db> ... </db>
```

- DTD im lokalen Dateisystem

```
<!DOCTYPE db SYSTEM "schema.dtd">
```

- DTD in einem Remote-Dateisystem

```
<!DOCTYPE db SYSTEM
"http://www.schemaauthority.com/schema.dtd">
```

# XML Namespaces

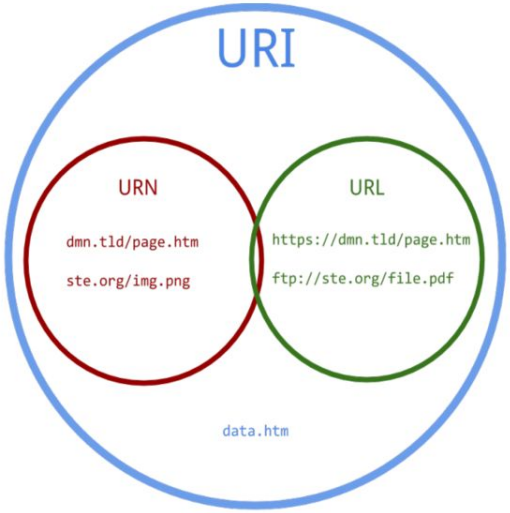
- Wenn ein Element in zwei verschiedenen XML-Dokumenten angezeigt wird, möchten wir wissen, dass er in beiden Dokumenten dieselbe Bedeutung hat
  - Wird <title> in beiden Dokumenten als <title> -Tag von XHTML verwendet?
  - Wenn zwei Dokumente für Bücher das Tag <Nummer> haben, bedeutet dies, dass sie dasselbe System zum Katalogisieren von Büchern verwenden?

# XML Namespaces


- Namespaces bieten einen Mechanismus zum Erstellen eindeutiger Namen (für Elemente und Attribute), die in XML-Dokumenten verwendet werden können
- Ein Namespace ist nur eine Sammlung von Namen, die für eine bestimmte Anwendungsdomäne erstellt wurden
- Namespaces sind keine DTDs und bieten keinen Mechanismus zur Validierung von XML-Dokumenten mit mehreren DTDs

# XML Namespaces

- Ein Namespace wird durch eine URI identifiziert
- Der URI muss auf nichts verweisen
  - Er wird als Mechanismus zum Erstellen eindeutiger Namen verwendet
- Ein Elementname oder Attributname aus einem Namespace besteht aus zwei Teilen `prefix:name`
- prefix → Namespace
- name → Name in dem Namespace



# XML Namespaces

- Ein XML Namespace wird im xmlns Attribut deklariert
- Die Verwendung von foo als Präfix anstelle der URI ist einfacher

```
<foo:book xmlns:foo="http://www.foo.org/">
   <foo:title> XML Namespaces </foo:title>
   <foo:author> John Doe </foo:author>
</foo:book>
```

# Default Namespace

- Der Standard-Namespace wird ohne Präfix deklariert
- Alle folgenden Elemente gehören zum Standard-Namespace

```
<book xmlns="http://www.foo.org/">
   <title> XML Namespaces </title>
   <author> John Doe </author>
</book>
```

# XML Namespaces


- Der Namespace-Mechanismus ist nur eine Zuordnung von Präfixen zu URIs
- `<foo: title>` wird durch `<{http://www.foo.org/} title>` ersetzt
- Dies erfolgt in einer Verarbeitungsschicht, die den aus der XML erzeugten Elementbaum bearbeitet
- Es werden eindeutige Namen erstellt


# DTDs vs. Schemas

- DTDs sind nach DB- und Programmiersprachenstandards eher schwache Spezifikationen
- Nur ein Basistyp - PCDATA
- IDREFs sind untypisiert
  - der Typ des Objekts, auf das verwiesen wird, ist nicht bekannt
- Keine Einschränkungen
- Einige XML-Erweiterungen legen ein Schema oder Typen für ein XML-Dokument fest


# XML Schema

- sind leichter zu lernen (als DTD)
- sind erweiterbar
- sind reicher und nützlicher als DTDs
- sind in XML geschrieben
- unterstützten Datentypen

# Beispiel

```
<?xml version="1.0"?>
<shipOrder>

<shipTo>
<name>Svendson</name>
<street>Oslo St</street>
<address>400 Main</address>
<country>Norway</country>
</shipTo>

<items>
<item>
<title>Wheel</title>
<quantity>1</quantity>
<price>10.90</price>
</item>

<item>
<title>Cam</title>
<quantity>1</quantity>
<price>9.90</price>
</item>
</items>

</shipOrder>
```

# Beispiel

```
<xsd:schema xmlns:xsd=http://www.w3.org/1999/XMLSchema>

   <xsd:element name="shipOrder" type="order"/>

   <xsd:complexType name="order">
     <xsd:element name="shipTo" type="shipAddress"/>
     <xsd:element name="items" type="cdItems"/>
   </xsd:complexType>

   <xsd:complexType name="shipAddress">
     <xsd:element name="name“ type="xsd:string"/>
     <xsd:element name="street" type="xsd:string"/>
     <xsd:element name="address" type="xsd:string"/>
     <xsd:element name="country" type="xsd:string"/>
   </xsd:complexType>
```

# Beispiel

```
  <xsd:complexType name="cdItems">
    <xsd:element name="item" minOccurs="0"
                 maxOccurs="unbounded" type="cdItem"/>
  </xsd:complexType>

  <xsd:complexType name="cdItem">
    <xsd:element name="title" type="xsd:string"/>
    <xsd:element name="quantity“
                 type="xsd:positiveInteger"/>
    <xsd:element name="price" type="xsd:decimal"/>
  </xsd:complexType>

</xsd:schema>
```

# Beispiel - Purchase Order


- Instance Document: Ein XML-Dokument, das einem XML-Schema entspricht
- Elemente, die Unterelemente enthalten oder Attribute besitzen, haben komplexe Typen
- Elemente, die Zahlen (und String, Date usw.) enthalten, aber keine Unterelemente enthalten, sind einfache Typen
- Attributes haben immer einfache Typen

# Beispiel - Purchase Order

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">

  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>

  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
```

# Beispiel - Purchase Order


```
<comment>Hurry, my lawn is going wild!</comment>
<items>

  <item partNum="872-AA">
    <productName>Lawnmower</productName>
    <quantity>1</quantity>
    <USPrice>148.95</USPrice>
    <comment>Confirm this is electric</comment>
  </item>

  <item partNum="926-AA">
    <productName>Baby Monitor</productName>
    <quantity>1</quantity>
    <USPrice>39.98</USPrice>
    <shipDate>1999-05-21</shipDate>
    </item>
  </items>

</purchaseOrder>
```

# Beispiel - Purchase Order


## PurchaseOrder Type

```
<xsd:complexType name="PurchaseOrderType">
   <xsd:sequence>
       <xsd:element name="shipTo" type="USAddress"/>
        <xsd:element name="billTo" type="USAddress"/>
        <xsd:element ref="comment" minOccurs="0"/>
        <xsd:element name="items"  type="Items"/>
   </xsd:sequence>

   <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
```

# Beispiel - Purchase Order


## USAddress Type

```
<xsd:complexType name="USAddress" >
   <xsd:sequence>
     <xsd:element name="name"   type="xsd:string"/>
     <xsd:element name="street" type="xsd:string"/>
     <xsd:element name="city"   type="xsd:string"/>
     <xsd:element name="state"  type="xsd:string"/>
     <xsd:element name="zip"    type="xsd:decimal"/>
   </xsd:sequence>
   <xsd:attribute name="country" type="xsd:string" fixed="US"/>
</xsd:complexType>
```

# Beispiel - Purchase Order

```
<xsd:complexType name="Items">
<xsd:sequence>
  <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="productName" type="xsd:string"/>
        <xsd:element name="quantity">
          <xsd:simpleType>
            <xsd:restriction base="xsd:positiveInteger">
              <xsd:maxExclusive value="100"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="USPrice"  type="xsd:decimal"/>
        <xsd:element ref="comment"   minOccurs="0"/>
        <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="partNum" type="SKU" use="required"/>
    </xsd:complexType>
  </xsd:element>
</xsd:sequence>
</xsd:complexType>
```

# Beispiel - Purchase Order

```
<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
   <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```

# Erweiterungen - Neue Typen

- Eine große Menge von integrierten Typen ist im XML-Schema verfügbar `xsd:string, xsd:integer, xsd:positiveInteger, xsd:decimal, xsd:boolean, xsd:date, xsd:NMTOKENS...`

- das folgende Beispiel definiert `myInteger`
  - Wert zwischen 10000 und 99999

```
<xsd:simpleType name="myInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

# Erweiterungen - Neue Typen


- Enumeration

```
<xsd:simpleType name="USState">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="AK"/>
    <xsd:enumeration value="AL"/>
    <xsd:enumeration value="AR"/>
    <!-- and so on ... -->
  </xsd:restriction>
</xsd:simpleType>
```

# Erweiterungen - Neue Typen

- Das XML-Schema bietet drei integrierte Listentypen
  `NMTOKENS, IDREFS, ENTITIES`

- Erstellen neuer Listentypen aus einfachen Typen:

```
<xsd:simpleType name="listOfMyIntType">
  <xsd:list itemType="myInteger"/>
</xsd:simpleType>
```

- Das folgende XML-Fragment entspricht dem obigen SimpleType:

```
<listOfMyInt>20003 15037 95977 95945</listOfMyInt>
```

# Erweiterungen - Neue Typen

- length, minLength, maxLength, enumeration

```
<xsd:simpleType name="USStateList">
  <xsd:list itemType="USState"/>
</xsd:simpleType>

<xsd:simpleType name="SixUSStates">
  <xsd:restriction base="USStateList">
    <xsd:length value="6"/>
  </xsd:restriction>
</xsd:simpleType>

<sixStates>PA NY CA NY LA AK</sixStates>
```

# XML Parsing in Java


- Serialisierer: Generiert aus bestimmter Datenstruktur ein Dokument
  - wir haben es für JSON gesehen
  - XML
- Parser: Analysiert XML-Dokument und erstellt Parse-Baum mit Tags, Text-Inhalten und Attribut-Wert-Paaren als Knoten
  - Pull- vs. Push-Parser: Wer hat die Kontrolle über das Parsen, die Anwendung oder der Parser?
  - Einschritt- vs. Mehrschritt-Parser
  - Wird das XML-Dokument in einem Schritt vollständig geparst oder Schritt für Schritt analysiert?
- Möglichst immer standardisierte APIs verwenden!


# Pull-Parser vs Push-Parser

- Objektmodell
  - Baut gesamten XML-Baum im Speicher auf
  - DOM (Document Object Model)
- Pull
  - Anwendung hat die Kontrolle über das Parsen
  - Analyse der nächsten syntaktischen Einheit muss von der Anwendung aktiv angefordert werden
  - StAX
- Push
  - Parser hat die Kontrolle über das Parsen
  - Anwendung kann für die möglichen "Events" callback Funktionen bereitstellen
  - SAX (Simple API for XML)

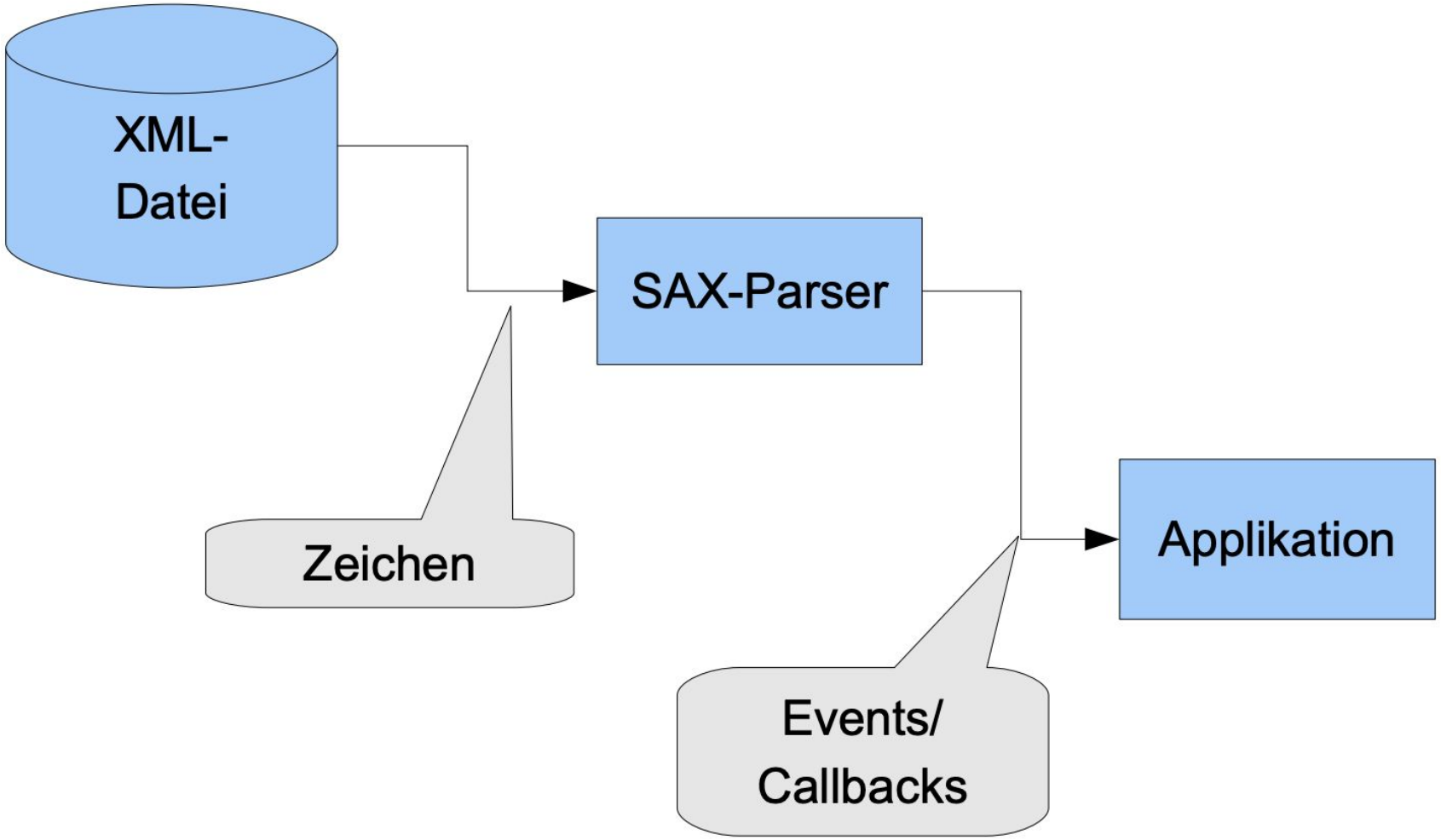
# DOM vs SAX

- Baummodell-Parser (Objektmodell/Baum)
- DOM lädt die Datei in den Speicher und parst sie dann
- hat Speicherbeschränkungen, da die gesamte XML-Datei vor dem Parsen geladen wird
- DOM unterstützt verarbeitung (man kann Knoten einfügen oder löschen)
- wenn das XML-Dokument klein ist, wär der DOM-Parser dafür geeignet
- die Suche vorwärts und rückwärts ist möglich
  - dies erleichtert die Navigation
- langsamer zur Laufzeit

# DOM vs SAX

- Ereignisbasierter Parser (Sequenz von Ereignissen)
- SAX analysiert die Datei beim Lesen, d. H. Schritt für Schritt
- Keine Speicherbeschränkungen, da der XML-Inhalt nicht im Speicher gespeichert wird
- SAX ist schreibgeschützt
- man verwendet den SAX-Parser, wenn der Speicherinhalt groß ist
- SAX liest die XML-Datei von oben nach unten und eine Rückwärtsnavigation ist nicht möglich
- Schneller zur Laufzeit

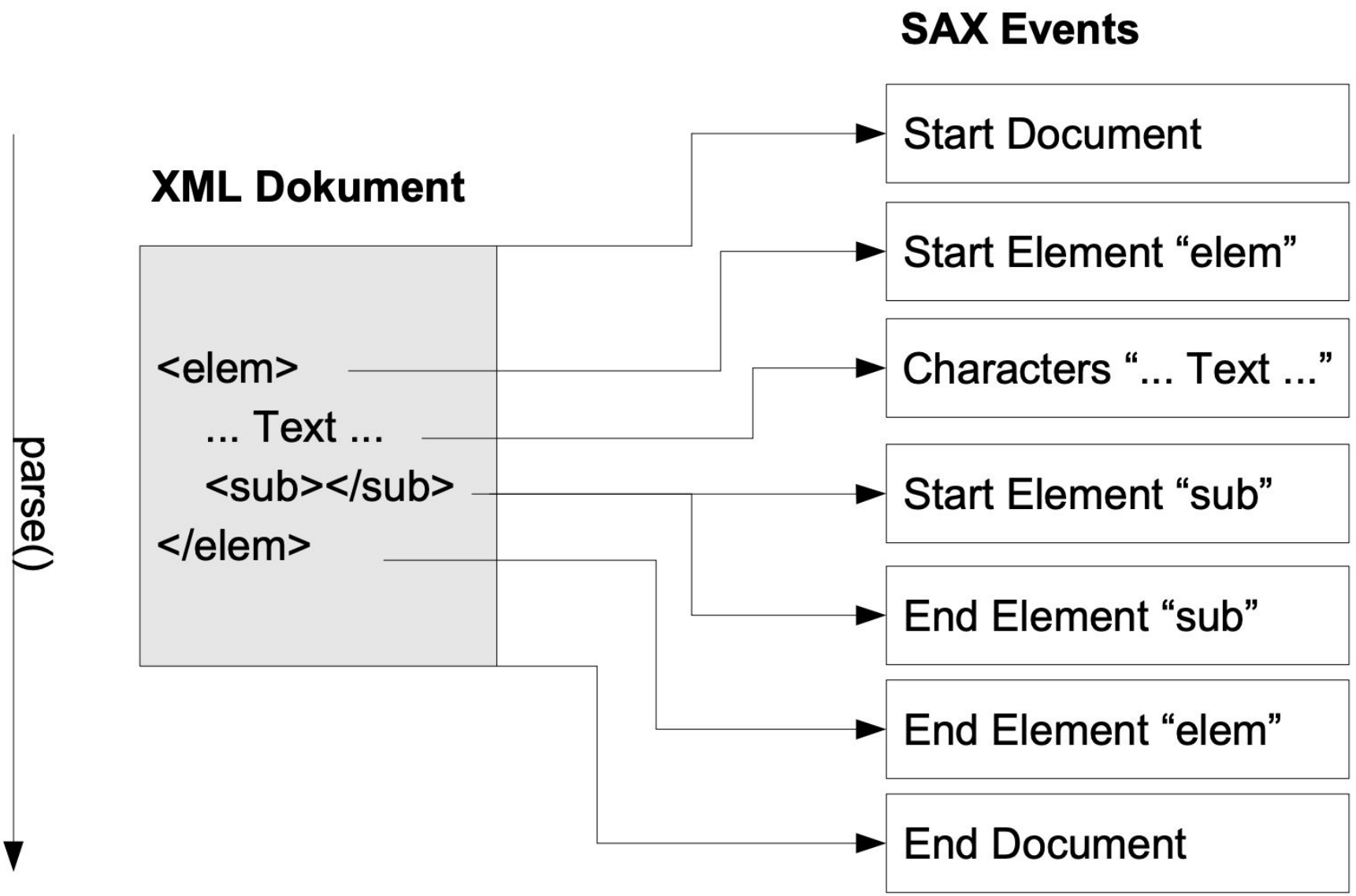
# SAX


XML-
Datei
SAX-Parser
Zeichen
Applikation
Events/
Callbacks

# SAX

- Applikation registriert callback Funktionen beim SAX-Parser
- Applikation stößt den Parser an
- SAX-Parser durchläuft das XML-Dokument einmal sequentiell
- Parser erkennt syntaktische Einheiten des XML-Dokuments
- Parser ruft für jedes Event die entsprechende callback Funktion auf

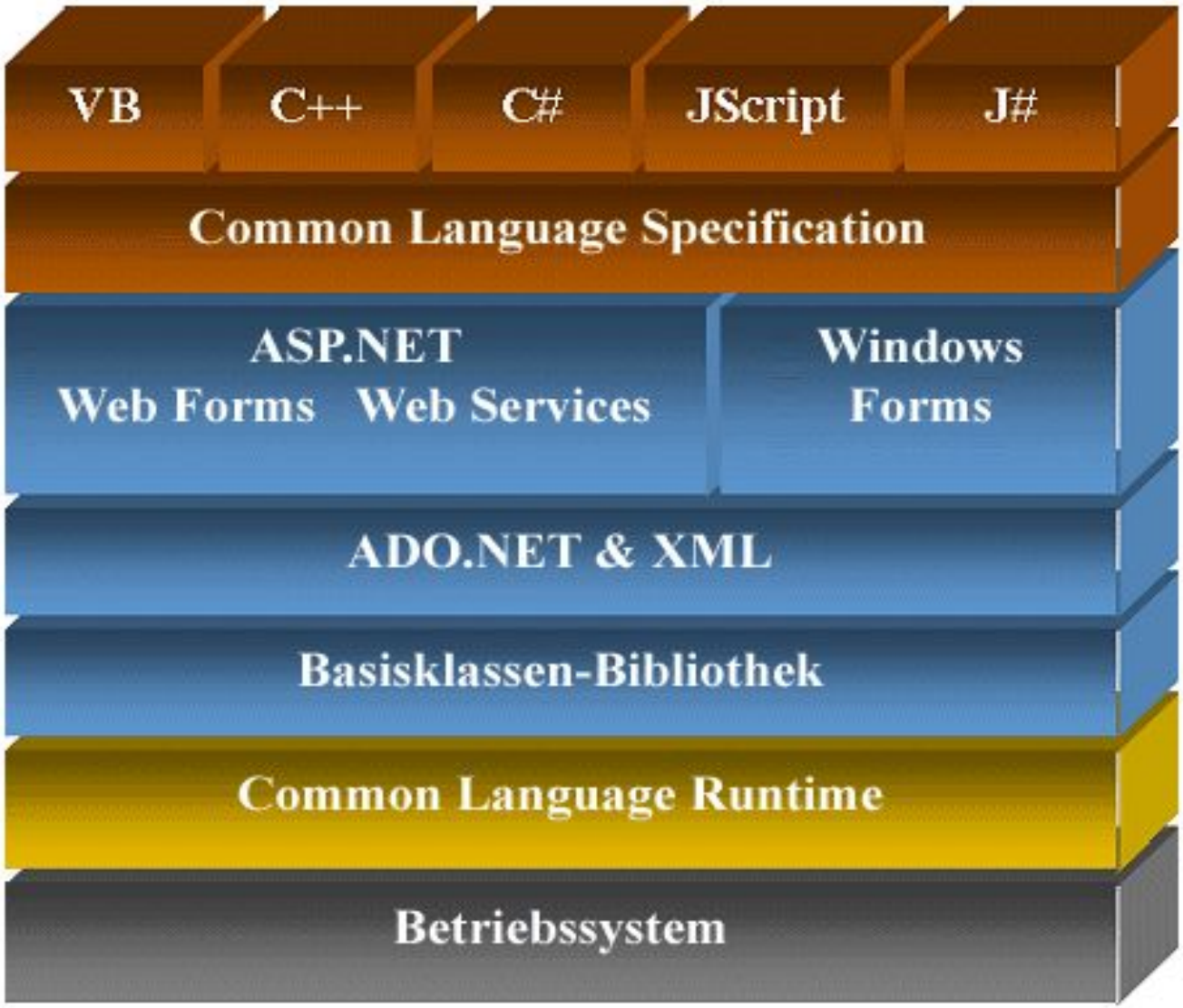
# SAX


SAX Events
XML Dokument
parse()
<elem>
... Text ...
<sub></sub>
</elem>
Start Document
Start Element “elem”
Characters “... Text ...”
Start Element “sub”
End Element “sub”
End Element “elem”
End Document

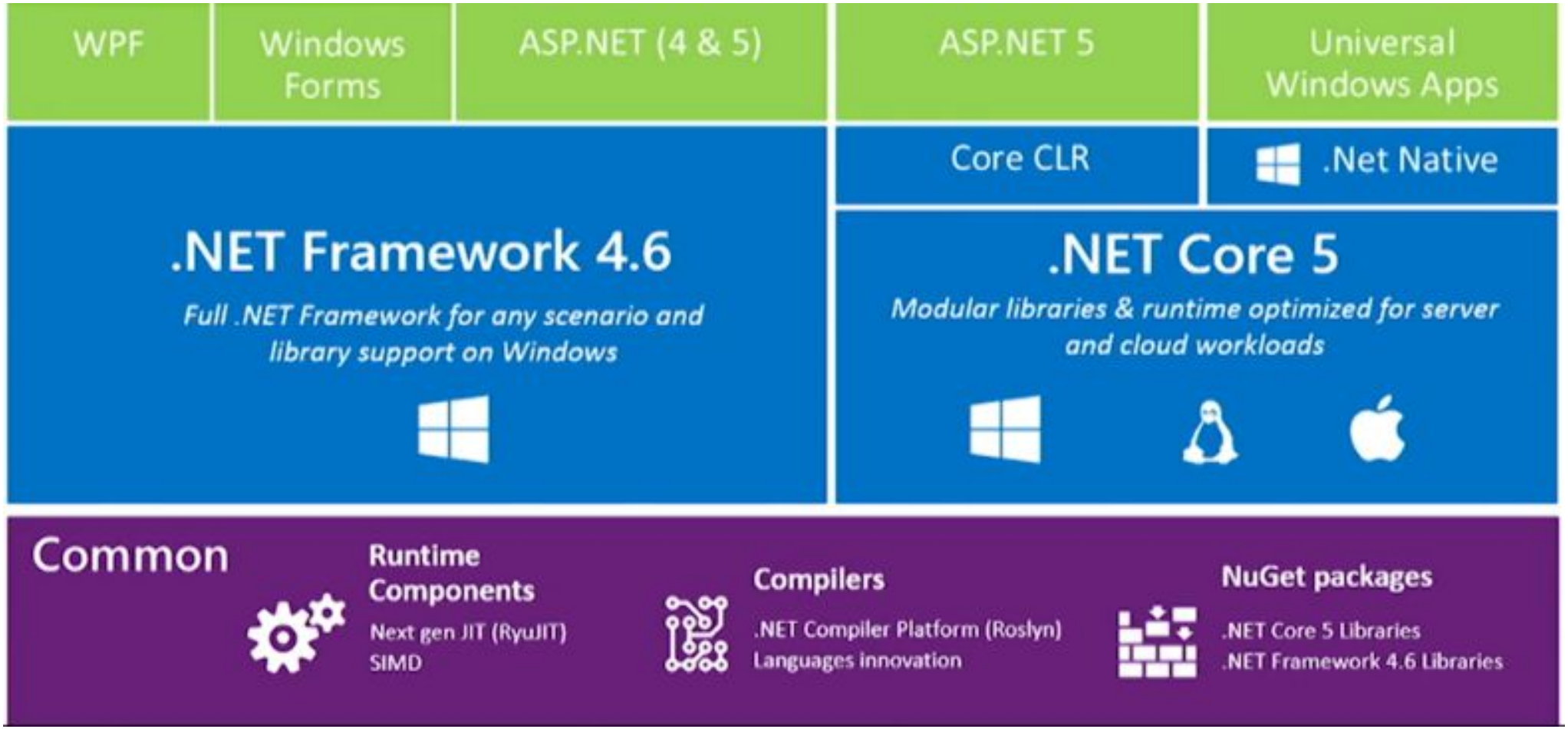
# StAX (Streaming API for XML)


- pull Modell
- zwei Interface
  - XMLEventReader
  - XMLEventWriter
- Cursor/Iterator
  - repräsentiert eine Stelle im Dokument
  - die Anwendung bewegt den Cursor vorwärts und holt die benötigte Information genau dann vom Parser, wenn sie benötigt wird
  - das Iterator-Verfahren liefert die Daten in Form von Objekten, die von der Klasse XMLEvent abgeleitet sind

# .NET: the early years

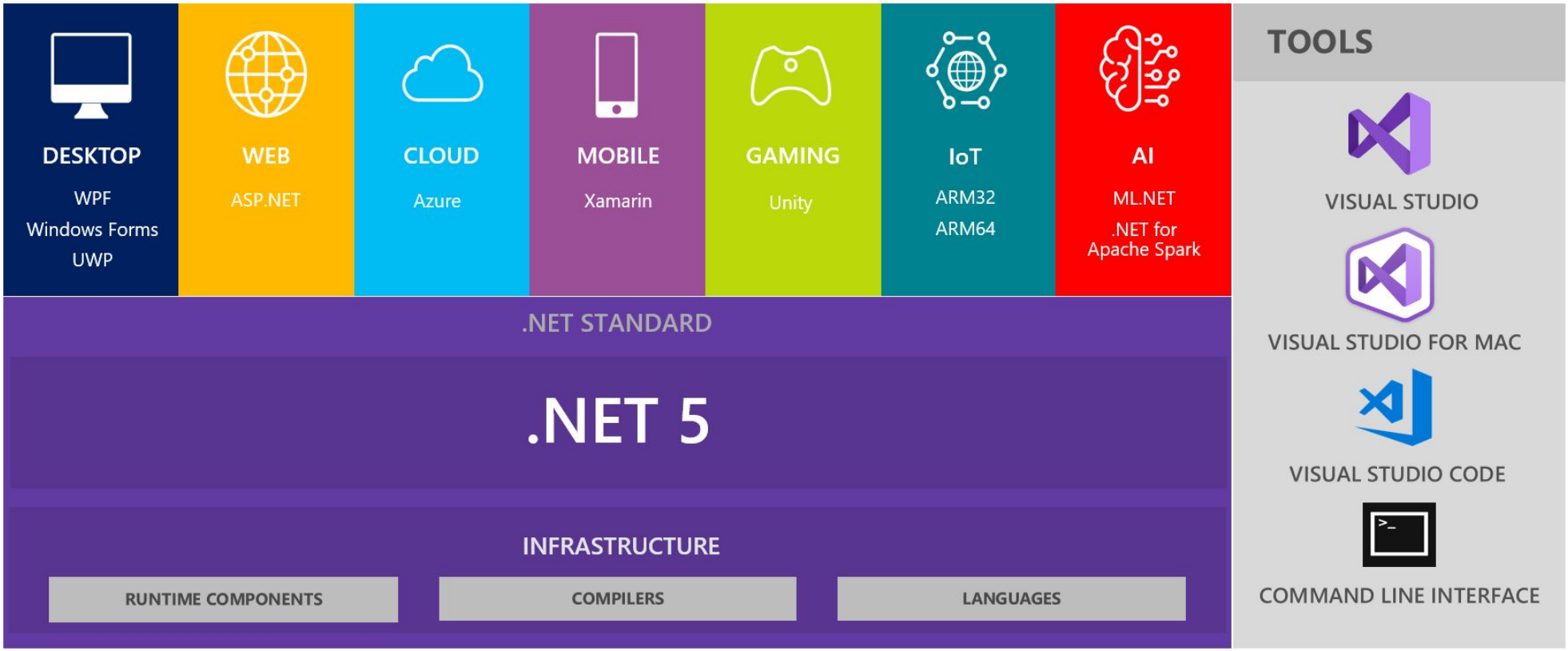


# .NET: the microsoft stack



# .NET: one framework to rule them all

## .NET – A unified platform



# .NET


- **Common Language Runtime (CLR)** mit einer für alle .NET gemeinsamen Common Intermediate Language (CIL).
  - Z.B. ist der Garbage Collector in der CLR implementiert

- **Common Language Specification (CLS)** und **Common Type System (CTS)**.
  - Alle .NET-Sprachen basieren auf gemeinsamen Basis-Typen

- **Umfangreiche Klassenbibliothek**
  - grafische Oberfläche, Web-Anwendungen, Datenbank, Sockets, XML, Multi-Threading, Kryptographie usw.

# ein Sprachenmix

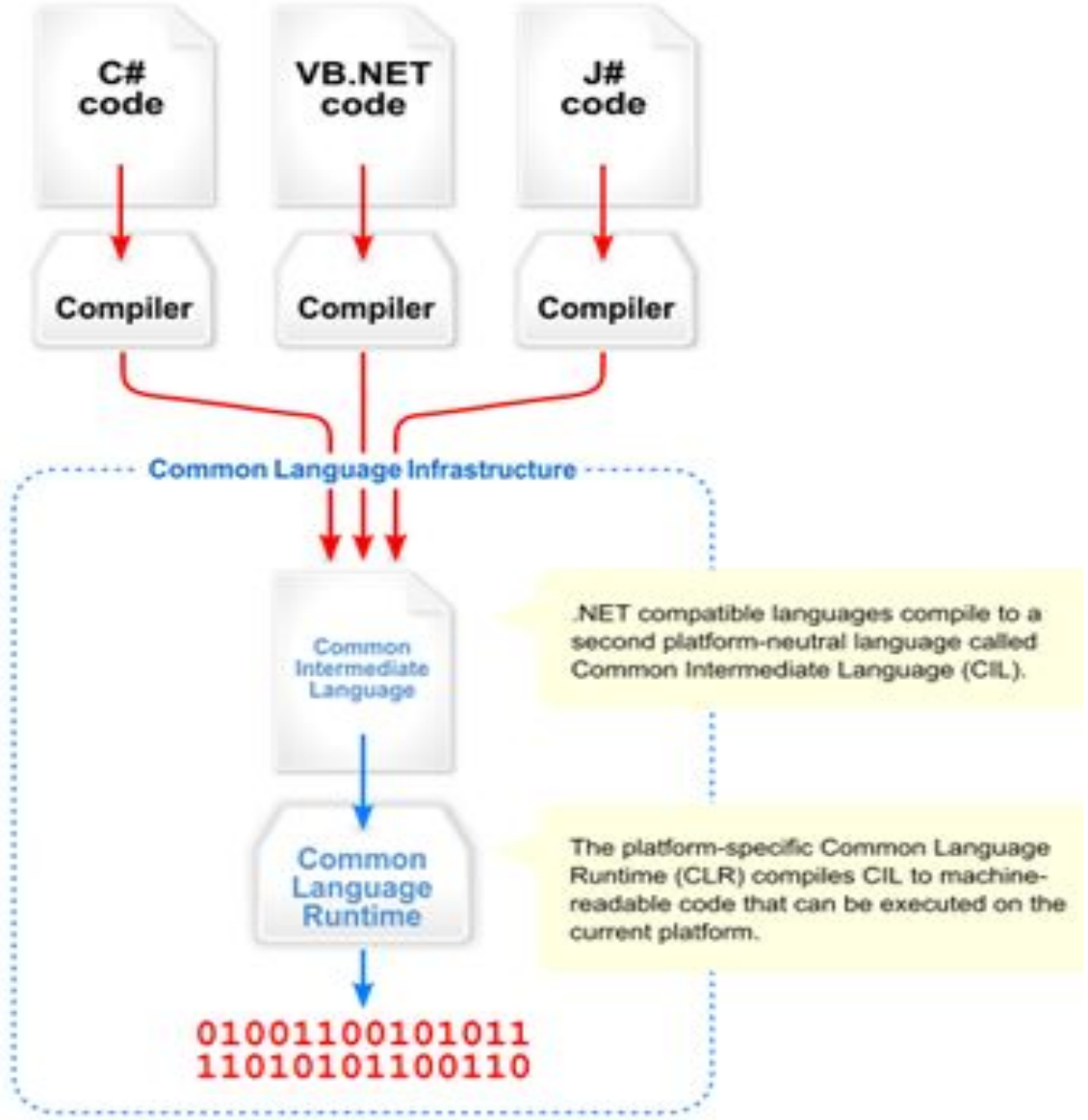
- eine .NET-Anwendung kann in unterschiedlichen Sprachen geschrieben werden (z. B. C#, J#, C++/CLI, Visual Basic .NET)
- eine Klasse, die in C# geschrieben ist, kann von einer Klasse in Visual Basic beerbt werden
- es gibt (theoretisch) keine „bevorzugte“ Programmiersprache. Vorteil: Jeder kann in der Sprache seiner Wahl programmieren
- die Klassenbibliothek, das Typsystem und die Laufzeitumgebung ist für alle .NET Sprachen gleich

# Common Intermediate Language

- Die CIL ist der „Befehlssatz“ der Virtual Machine von .NET
  - D.h. .NET-Anwendungen sind plattformunabhängig in CIL geschrieben
  - Der CIL-Code sichert die kompatibilität zwischen den verschiedenen .NET Programmiersprachen
  - CIL ist eine „objektorientierte Assemblersprache“

```
.method public static void Main() cil managed
 {
      .entrypoint
      .maxstack 1
      ldstr "Hallo Welt!"
      call void [mscorlib]System.Console::WriteLine(string)
      ret
 }
```

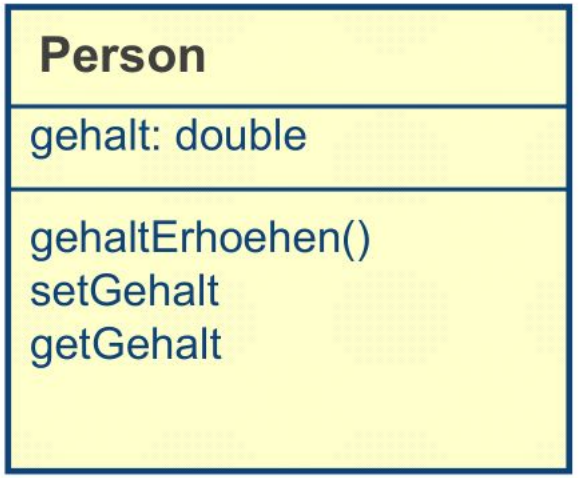
# Common Language Runtime



# Einfaches Beispiel

- C# Code: Person.cs

```
1  class Person
2  {
3      private double gehalt;
4      public double getGehalt() {
5          return gehalt;
6      }
7
8      public void setGehalt(int g) {
9          gehalt = g;
10     }
11
12     public void gehaltErhoehen() {
13         gehalt = gehalt + 100;
14     }
15 }
```



# C# vs. Java

- streng typisierte objektorientierte Programmiersprache
- wird übersetzt in Intermediate Language (IL): ähnlich Java-Bytecode
- wird ausgeführt von CLR – ähnlich JVM
- Anforderungen:
  - Architekturunabhängigkeit
  - Sprachunabhängigkeit

# C# und Java


- keine Header-Dateien
- Mehrfachvererbung von Schnittstellen (nicht von Implementierungen)
- keine globalen Funktionen oder Konstanten (alles in Klassen)
- Arrays und Strings mit festen Längen und Zugriffskontrolle
- Alle Variablen müssen vor der ersten Verwendung initialisiert werden
- alle Objekte erben von Object-Klasse
- Objekte werden auf dem Heap erzeugt (mit dem Schlüsselwort new)
- Garbage Collector, Reflection
- Thread Unterstützung, Synchronisation
- Generics

# Erste Schritte

- Hello World

```
1  using System;
2
3  public class Hello {
4      public static void Main() {
5          Console.WriteLine("Hello World");
6      }
7  }
```

- alle Klassen der .NET-Architektur im System  Namespace
  - using Anweisung

- Main-Methode als Einstiegspunkt:
  - `public static void Main(string[] args) { ... }`

# BankLibrary

```
1  namespace BankLibrary {
2      public class Bank {
3          public static int deposit(Account account,
4              int amount){
5              return account.deposit(amount);
6          }
7      }
8
9      public class Account {
10         private int balance = 0;
11
12         public int deposit(int amount) {
13             balance += amount; return balance;
14         }
15
16         public int withdraw(int amount) {
17             balance -= amount; return balance;
18         }
19     }
20 }
```

```
> mcs -target:library Bank.cs
```

# BankClient

```
1  using System;
2
3  namespace Customer {
4      class Customer {
5          public static void Main(string[] args) {
6              BankLibrary.Account account =
7                  new BankLibrary.Account();
8              Console.WriteLine("deposit 3 -> account = {0}",
9                  BankLibrary.Bank.deposit(account, 3));
10             Console.WriteLine("deposit 5 -> account = {0}",
11                 BankLibrary.Bank.deposit(account, 5));
12         }
13     }
14 }
```

```
> mcs -r:Bank.dll Customer.cs
```

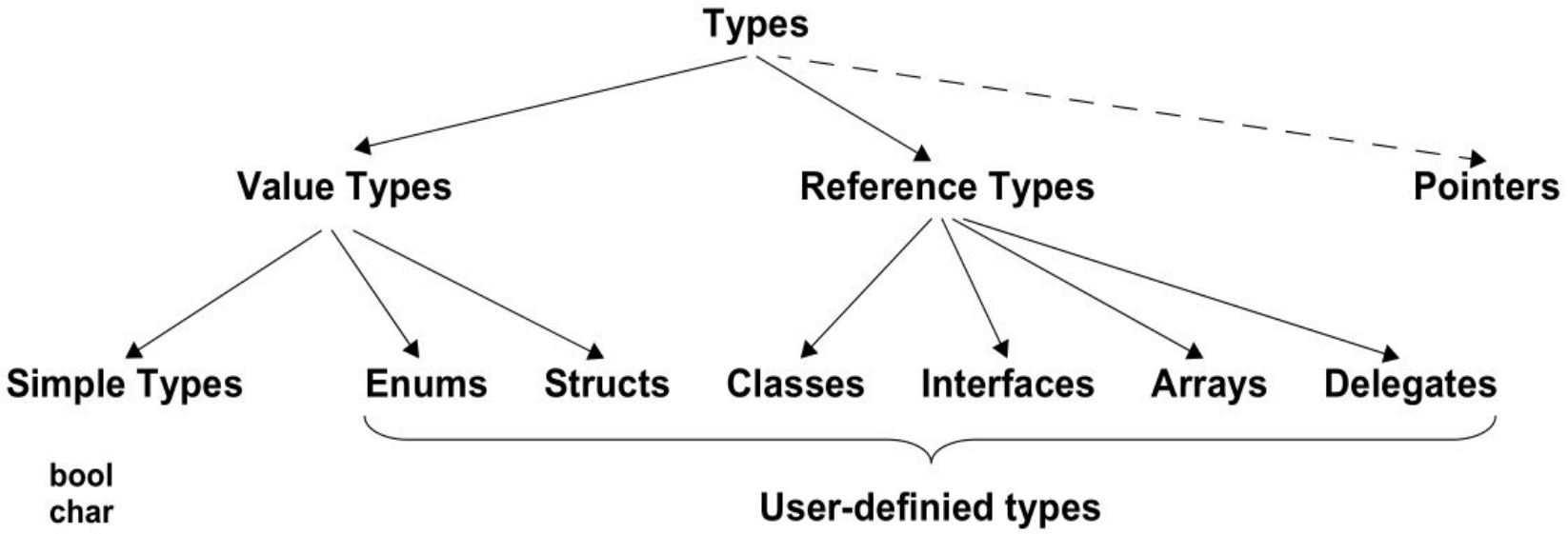
```
> mono Customer.exe
```

# Kommentare

- Kommentierter Bereich
  - /* Dies ist ein Kommentar,
  - der über mehrere
  - Zeilen verläuft */

- Zeilenkommentar
  - int x = 1; // Anfangswert
  - // ab jetzt folgen Initialisierungen

# Typsystem




bool
char
sbyte
byte
short
ushort
int
uint
long
ulong
float
double
decimal


# Primitive Typen

- byte , short , int , long , float , double , bool , char (Unicode)
  - wie in Java
- vorzeichenlose Typen: byte , ushort , uint , ulong
- decimal : 128 Bit Zahl
- Konstanten
  - wie in C/C++
  - const int i = 3;

# Typen

- Enum
  - Liste von Konstanten mit Namen

```
public enum Farbe {Gelb = 1, Rot = 2};

Farbe farbe = Farbe.Gelb
```

- Array

```
1  int[] array = new int[3];
2  int[] array = new int[] { 1, 2, 3 };
3  int[] array = { 1, 2, 3 };
4  int len = array.Length
5
6  int[][] 2D_array = new int[2][];
7  a[0] = new int[3]; a[1] = new int[4];
```

# Strings

- Objekte der Klasse string sind konstante, invariante Objekte
  - bei Modifikationen wird neues string-Objekt als Rückgabewert geliefert
  - Aneinanderhängen mit "+"
  - Indizierung: s[i], Länge der Zeichenkette: s.Length

- Vergleich von string-Objekten

```
1  string s1 = "Hello";
2  string s2 = string.Copy(s1);
3
4  if (s1 == s2) { ... }                  // true (1)
5  if ((Object)s1 == (Object)s2) { ... }   // false (2)
6  if (s1.Equals(s2)) { ... }             // true (3)
```

# Strings

- Strings parsen

```
1  char[] trennzeichen = {' ', ',', ':'};
2  // oder:
3  string s_trennzeichen = " ,:";
4  char[] trennzeichen = s_trennzeichen.ToCharArray();
5
6  string test = "Vorname,Nachnahme Strasse:PLZ,Ort";
7  string[] parts = test.Split(trennzeichen);
```

- Ein-/Ausgabe (Console)

```
1  using System;
2
3  Console.Write("kein Newline am Ende");
4  Console.WriteLine("diesmal mit Newline");
5  Console.WriteLine("Heute ist der {0}. {1}", 22, "Januar");
6
7  string input = Console.ReadLine();
```

# Strukturen

- C# unterstützt Strukturen:
  - ähnlich C/C++

- Strukturen werden in C# auf dem Stack angelegt ( value type )
  - Klassen (class) werden immer auf dem Heap erzeugt

- Keine Vererbung möglich, Strukturen können aber Interfaces implementieren

```
1  public struct Account {
2      public int balance;
3      public Account(int amount) { balance = amount; }
4      public void Withdraw(int amount) { balance -= amount; }
5  }
```

# Boxing

- Value types (primitive Type, Strukturen, Enums) können in ein Objekt ( Boxing ) und wieder zurück gewandelt werden ( Unboxing )

- Nachteile:
  - Wrapper-Objekt muss (auf dem Heap) erzeugt werden

  - Man erkennt nicht auf Anhieb, dass das Verfahren teuer ist

```
1  Object obj = 333;                    // boxing (explizit)
2  int i = (int) obj;                       // unboxing
3
4  Stack stack = new Stack();               // Stack enthält Objekte
5  stack.Push(i);                       // boxing (implizit)
6  int j = (int) stack.Pop();                              // unboxing
```

# Switch-Anweisung

- Kontrollfluss muss explizit festgelegt werden
  - break (oder return, goto, throw) muss am Ende jeder case-Anweisung stehen
  - falls kein case zutrifft: default-Label

- als Switch-Typ ist auch ein String erlaubt:

```
1  switch(name) {
2      case "Zaphod Beeblebrox":
3          Console.WriteLine( "Hello Zaphod" );
4      break;
5      case "Ford Prefect":
6          Console.WriteLine( "Hi" );
7      break;
8  }
```

# foreach-Anweisung

- foreach-Anweisung arbeitet auf allen Objekten, die das Interface System.Collections.IEnumerable implementieren

- Auch Arrays implementieren dieses Interface

```
1  foreach (Object o in collection) { ... }
2
3  foreach (int i in array) { ... }
```

# Assemblies, Namespaces

- Namensräume (namespaces) wie in Java: Trennung mit “ . ”
  - “syntactic sugar” für lange Klassennamen
  - Namensräume importieren mit using-Schlüsselwort
- Eine Assembly besteht aus mehreren Dateien (einem Projekt), die zu einer .exe- (Executable) oder .dll-Datei (Bibliothek) kompiliert werden
  - definieren einen eigenen Namensraum
  - verschiedene Versionen einer Assembly können parallel existieren

# Zugriffslevel

- private (Zugriff nur innerhalb der Klasse, wie in Java)
- internal (Default - Zugriff innerhalb der Assembly)
- protected (Zugriff innerhalb der Klasse und abgeleiteter Klassen)
- internal protected (wie protected, zusätzlich im Assembly)
- public (Zugriff immer erlaubt)

# Klassen, Interfaces, Vererbung

- Übernommen von C++ ; Syntax identisch für Klassen und Interfaces
- Jedoch wie Java keine Vererbung unter Angabe von Zugriffsrechten
- Eine Klasse kann max. von einer anderen Klasse erben, aber mehrere Interfaces implementieren