.NET:
Base Class Library (BCL)

Version 5.5

Framework Class Library (FCL) – Überblick (1)

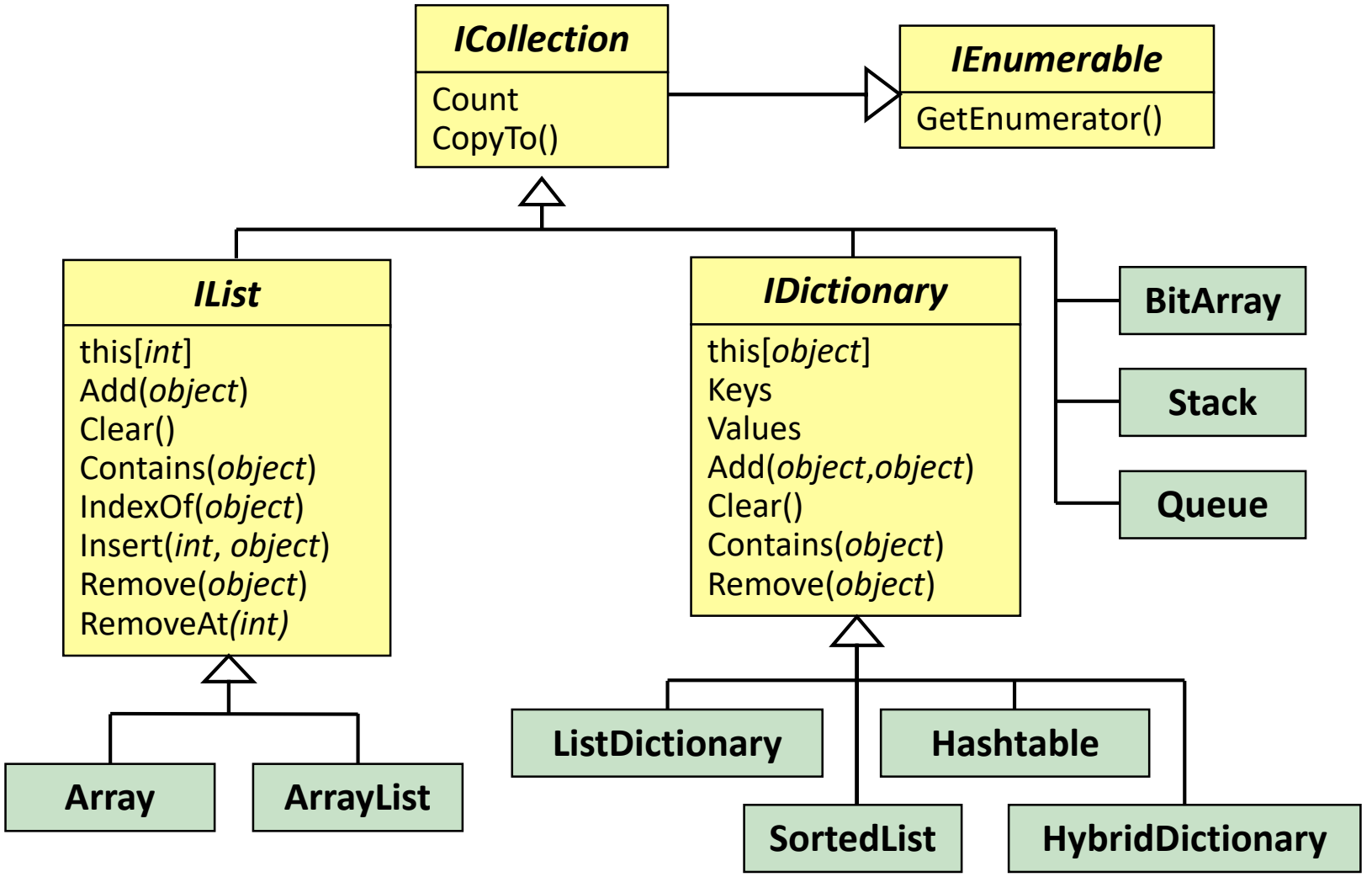
Base Class Library

System	Object, Array, String, Math, ...
Collections	Listen und Dictionaries
Generic	Generische Listen und Dictionaries
Concurrent	Thread-sichere Behälterklassen
ComponentModel	Design- und Laufzeitverhalten von Komp.
Data	Datenbankzugriff mit ADO.NET
Diagnostics	Debug und Trace-Utilities
Drawing	Grafik mit GDI+
IO	Streambasierte IO
Net	API für div. Netzwerkprotokolle (Sockets, ...)
Reflection	Manipulation von Metadaten
Runtime	
InteropServices	Native Calls, COM-Interoperabilität
Remoting	Klassen für verteilte Anwendungen
Serialization	Serialisierung und Deserialisierung

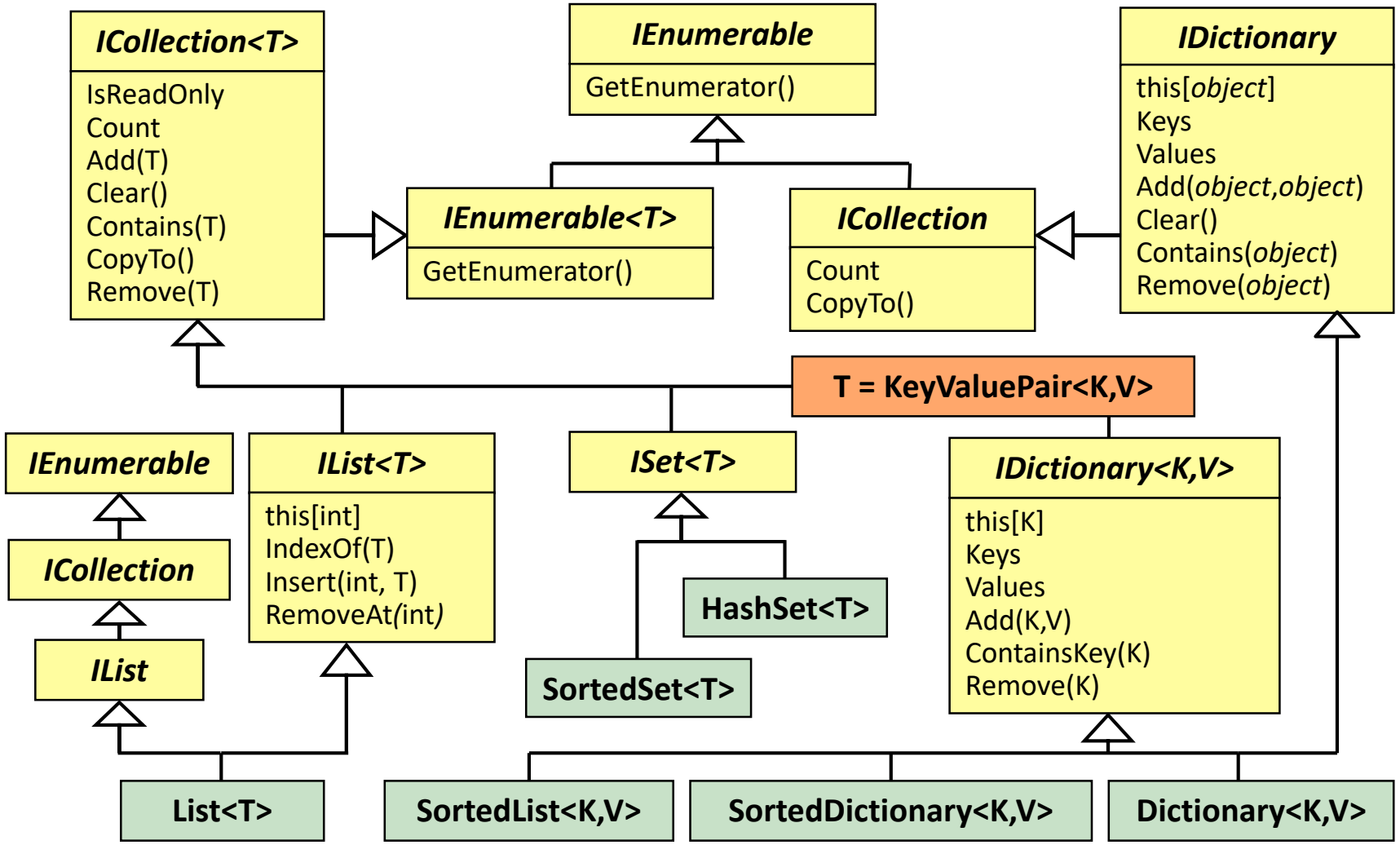
Framework Class Library (FCL) – Überblick (2)

- System
  - Security — CLR Security-System, Permissions
  - Threading — Programmierung mit Threads
    - Tasks — Task Parallel Library
  - Web — Browser/Server-Programmierung (HTTP)
    - Services — XML Web Services (SOAP, WSDL)
    - UI — ASP.NET WebForms
    - MVC — ASP.NET MVC
  - Windows — Basisklassen der WPF
    - Controls — Steuerelemente der WPF
    - Forms — Windows-Forms
  - XML — Verarbeitung von XML-Dokumenten

Behälterklassen (heterogene Behälter)



Generische Behälterklassen (1)



Generische Behälterklassen (2)

System.Collections	System.Collections.Generics
IEnumerable	IEnumerable<T>
IEnumerator	IEnumerator<T>
ICollection	ICollection<T>
IList	IList<T>
IDictionary	IDictionary<K,T>
ArrayList	List<T>
HashTable	Dictionary<K,T>
SortedList	SortedList<K,T>
-	SortedDictionary<K,T>
-	ISet<T>, HashSet<T>, SortedSet<T>

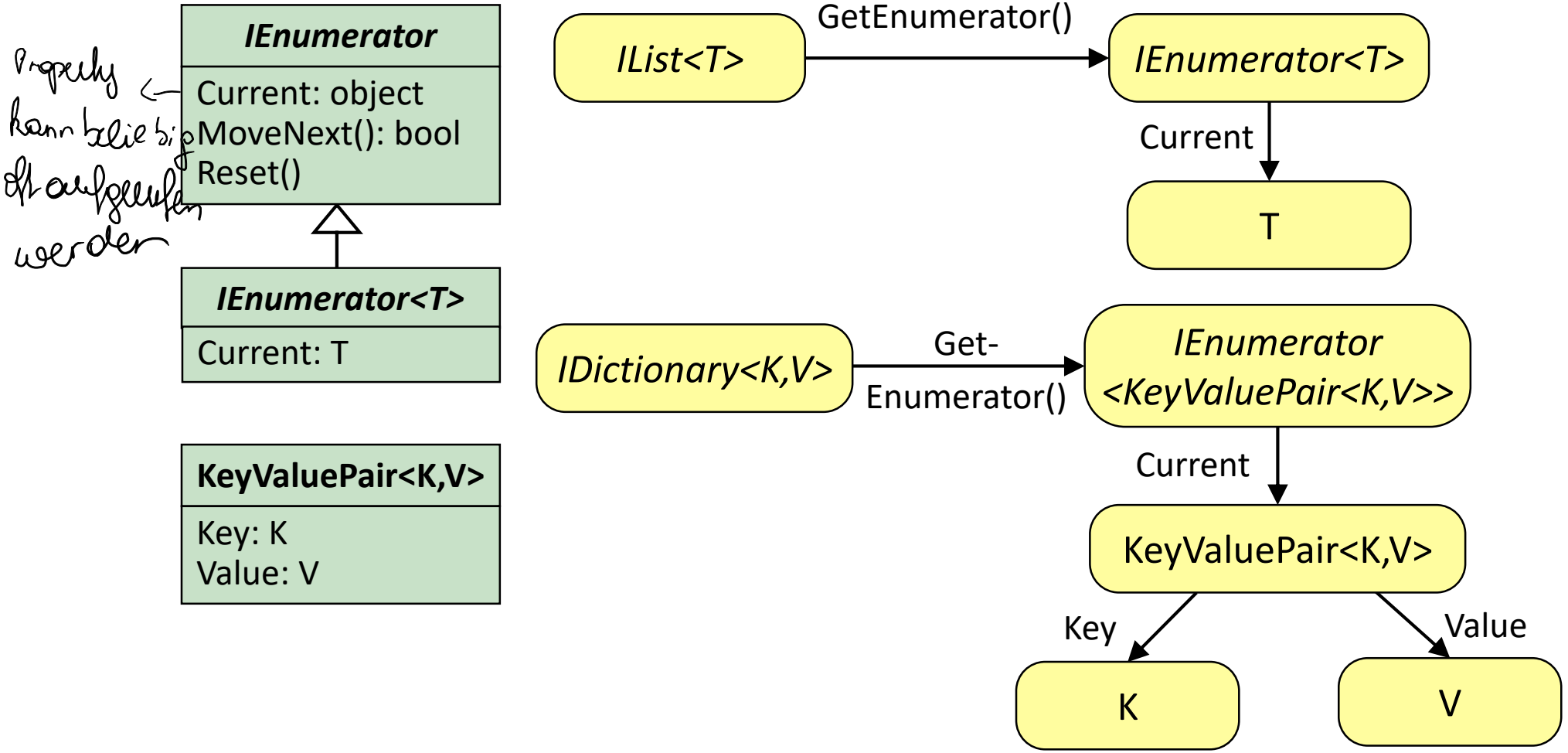
- *SortedList:* repräsentiert in Form von zwei Feldern.
- *SortedDictionary:* repräsentiert als binärer Suchbaum.
- Alternative Behälterbibliothek:
  - C5 Generic Collection Library (https://github.com/sestoft/C5)

Thread-sichere Behälterklassen

- System.Collections (.NET 1.0)
  - Nicht Thread-sicher.
  - Property *Synchronized* liefert einen Thread-sicheren Wrapper.
  - Nachteile: ineffizient, Iteration ist nicht Thread-sicher.
- System.Collections.Generic (.NET 2.0)
  - Nicht Thread-sicher.
  - Nachteil: Kein Synchronisationsmechanismus.
- System.Collections.Concurrent (.NET 4.0)
  - Spezialisierte Thread-sichere Behälterklassen:
    - `BlockingCollection<T>`
    - `ConcurrentQueue<T>`
    - `ConcurrentStack<T>`
    - `ConcurrentBag<T>`
    - `ConcurrentDictionary<K,V>`

    } Implementierung des Producer/Consumer-Musters
  - *GetEnumerator* liefert einen Snapshot -> Thread-sichere Iteration

Enumeratoren (Iteratoren)



Enumeratoren – Anwendung

Verwendung von *MoveNext()* und *Current*

```
IDictionary dict<int,string> =
  new Dictionary<int,string> ();
…
IEnumerator<int,string> e =
   dict.GetEnumerator();
while (e.MoveNext())
  Process(e.Current.Key, e.Current.Value);
```

```
Ilist<int> list =
  new ArrayList<int>();
…
IEnumerator<int> e = list.GetEnumerator();
while (e.MoveNext())
  Process(e.Current);
```

Verwendung von *foreach*

```
Ilist<int> list =
  new ArrayList<int>();
…
foreach (int elem in list)
  Process(elem);
```

```
IDictionary dict<int,string> =
  new Dictionary<int,string> ();
…
foreach (KeyValuePair<int,string> p in dict)
  Process(p.Key, p.Value);
```

Automatisch generierte Iteratoren (1)

- Die Implementierung von Iteratoren kann mit dem Schlüsselwort *yield* vereinfacht werden.
- *yield* kann auch eingesetzt werden, wenn eine Methode einen Behälter vom Typ *IEnumerable* zurückgibt.

```
public class DaysOfTheWeek : IEnumerable<string> {
  string[] days = { "Sun", "Mon", "Tue", "Wed", "Thr",  "Fri", "Sat"};
  public IEnumerator<string> GetEnumerator() {
    for (int i=0; i<days.Length; i++)
      yield return days[i];
  }
  public IEnumerable<string> GetWorkingDays() {
    for (int i=1; i<days.Length-1; i++)
      yield return days[i];
  }
}
```

nur soviele Elemente zurückgeben wie MoveNext() aufgerufen wird

drüber

wird erst ausgeführt, wenn wirklich iteriert wird, im Hintergrund liegt eine State Machine

Automatisch generierte Iteratoren (2)

- Mit *yield* lassen sich Iteratoren einfach realisieren, deren Implementierung sonst aufwändig wäre.

```
public class TreeSet<T> : IEnumerable<T>
  public IEnumerator<T> GetEnumerator() {
    return EnumerateItems(root).GetEnumerator();
  }
  private IEnumerable<T> EnumerateItems(Node<T> n) {
    if (n == null) yield break;
    foreach (T val in EnumerateItems(n.Left))
      yield return val;
    yield return n.Val;
    foreach (T val in EnumerateItems(n.Right))
      yield return val;
    }
  }
}
```

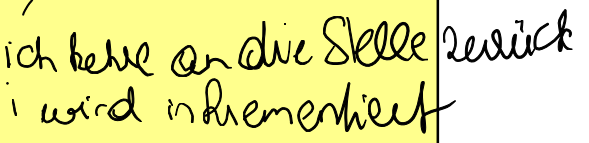
braucht keinen Stack, weil es intern über die State Machine geregelt wird.

Automatisch generierte Iteratoren (3)

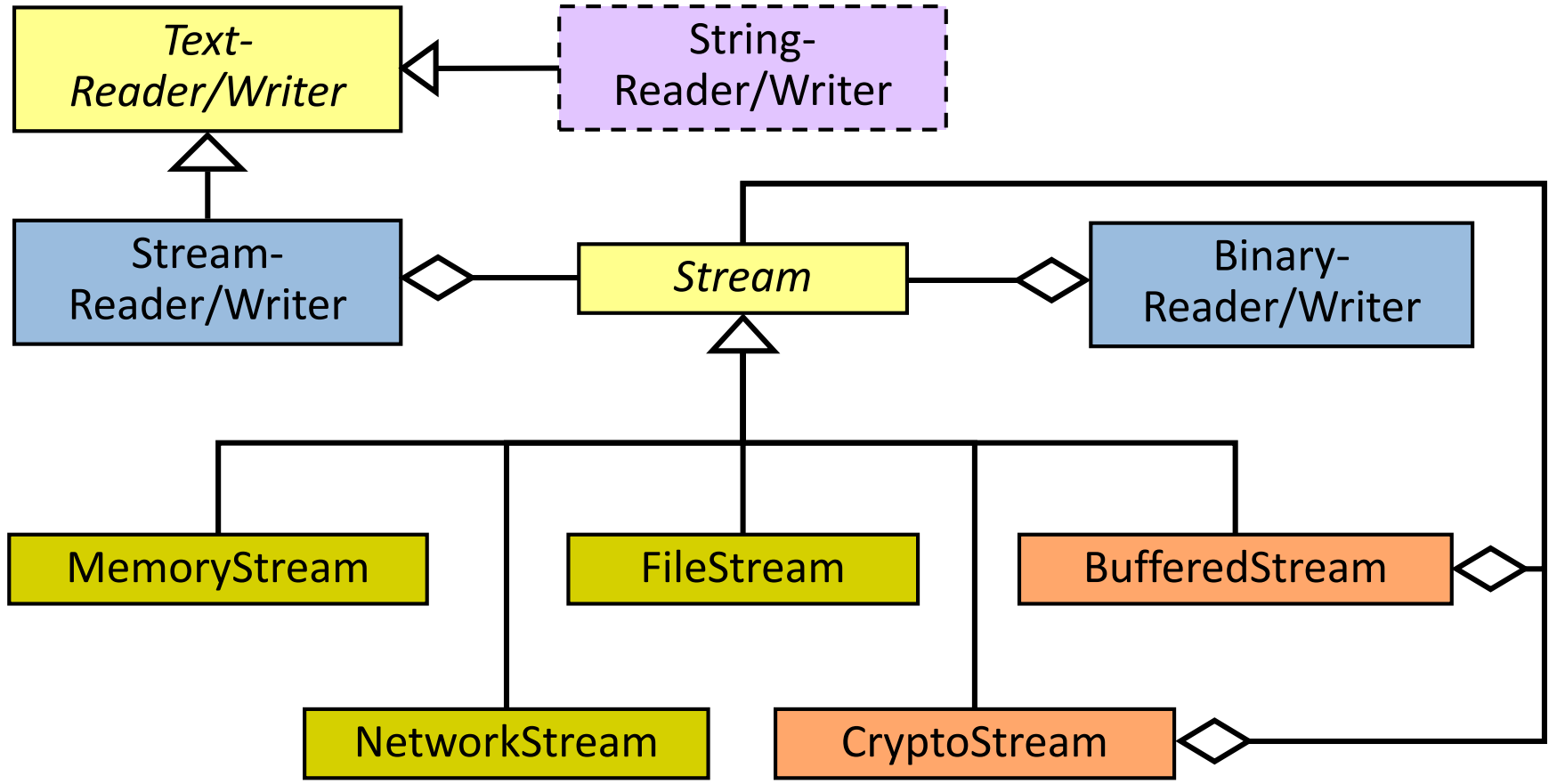
- Mit *yield return* realisierte Enumerationen werden verzögert ausgeführt *(deferred execution)*.

```
private IEnumerable<int> FindPrimes(int from, int to) {
  int i = from;
  while (i <= to) {
    while (! IsPrime(i)) i++;
    if (i <= to) yield return i++;
  }
}
```

```
IEnumerable<int> primes = primeCalc.FindPrimes(6, 11);
IEnumerator<int> pe = primes.GetEnumerator();
pe.MoveNext();                    → IsPrime(6), IsPrime(7)
Console.WriteLine(pe.Current);    → 7
pe.MoveNext();                    → IsPrime(8), ..., IsPrime(11)
Console.WriteLine(pe.Current);    → 11
```



Streams: Klassenhierarchie



Streams: Beispiel

```
Stream s = new FileStream("ascii.out", FileMode.Create);
StreamWriter sw = new StreamWriter(s);
sw.WriteLine("Current time {0}", DateTime.Now);
for (int i=0; i<10; i++)
  sw.Write("{0,4}", i);
sw.Close();
```

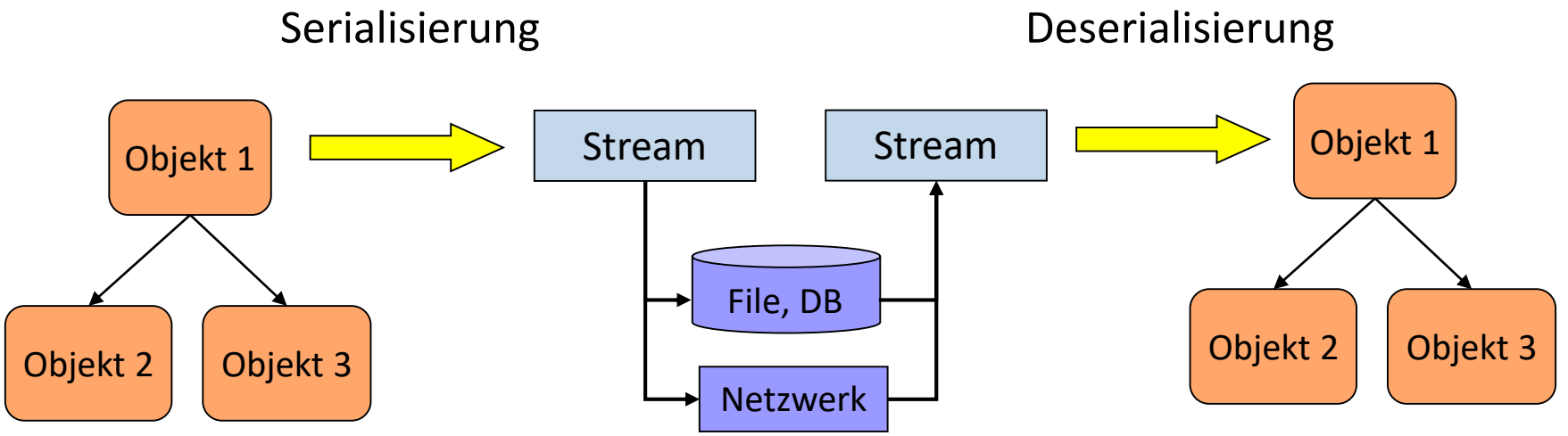
ascii.out

```
Current time 19.07.2003 12:08:08
   0   1   2   3   4   5   6   7   8   9
```

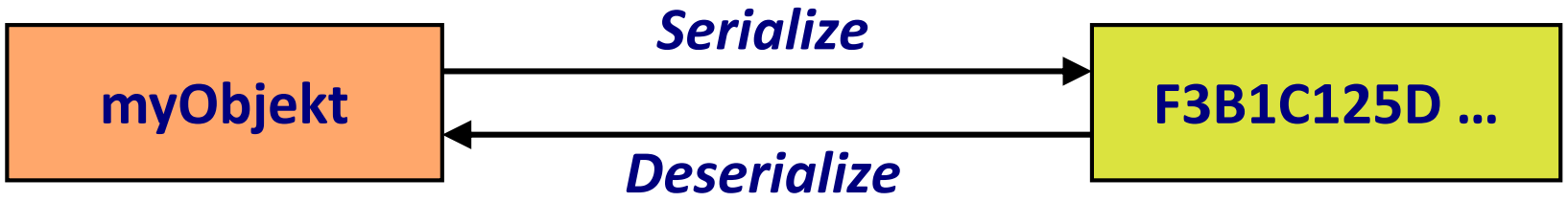
```
Stream s = new FileStream("ascii.out", FileMode.Open);
StreamReader sr = new StreamReader(s);
Console.WriteLine(sr.ReadToEnd());
sr.Close();
```

Serialisierung

- Serialisierung ist der Prozess, mit dem Objekte in eine Form gebracht werden, in der sie
  - auf einem persistenten Medium gespeichert bzw.
  - zu einem anderen Prozess transportiert werden können.



Binäre Serialisierung



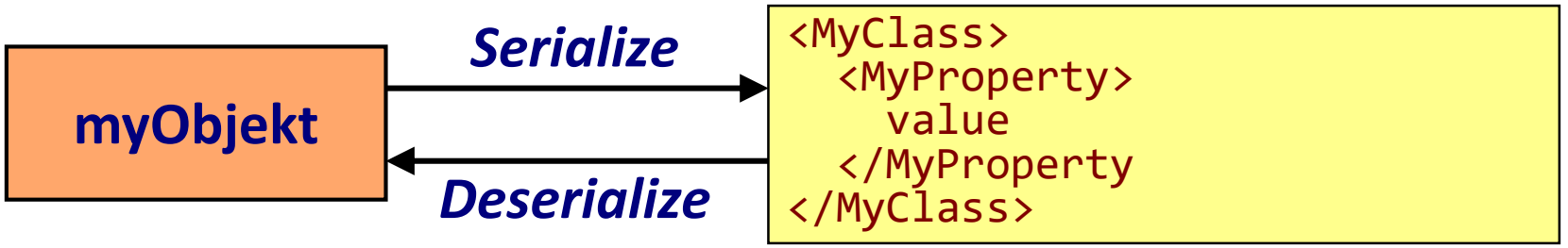

```
[Serializable]
public class MyClass { … }
```

```
IFormatter formatter = new BinaryFormatter();
Stream stream = new SomeStream(…);

formatter.Serialize(stream, new MyClass());

MyClass myObject = (MyClass)formatter.Deserialize(stream);
```

XML-basierte Serialisierung




```
public class MyClass {
  [XmlElement]
  public object MyProperty { get; set; }
}
```

```
Stream stream = new SomeStream(…);
XmlSerializer ser = new XmlSerializer(typeof(MyClass));

ser.Serialize(stream, new MyClass());

MyClass myObject = (MyClass)ser.Deserialize(stream);
```

XmlTextReader und XmlTextWriter (1)

- Mit XmlTextWriter/XmlTextReader können Xml-Dokumente auf Ebene der XML-Tokens verarbeitet werden.
- Dieser Ansatz entspricht konzeptionell der StaX-API von Java.
- Man hat volle Kontrolle über das XML-Dokument.
- Aufwändigste Methode zum Verarbeiten von XML-Dokumenten.

```
Stream stream = new SomeStream(…)
using (XmlTextWriter xmlw =
        new XmlTextWriter(stream, null)) {
  xmlw.WriteStartDocument();
  xmlw.WriteStartElement("elem");
  xmlw.WriteElementString("subElem",
                          "body");
  xmlw.WriteEndElement();
  xmlw.WriteEndDocument();
}
```

```
<elem>
  <subElem>
    body
  </subElem>
</elem>
```

XmlTextReader und XmlTextWriter (2)

```
Stream stream = new SomeStream(…)
using (XmlTextReader xmlr =
       new XmlTextReader(stream)) {
  xmlr.Read();
  Debug.Assert(
    xmlr.NodeType == XmlNodeType.Element &&
    xmlr.Name == "elem");
  xmlr.Read();
  Debug.Assert(
    xmlr.NodeType == XmlNodeType.Element &&
    xmlr.Name == "subElem");
  xmlr.Read();
  Debug.Assert(
    xmlr.NodeType == XmlNodeType.Text &&
    xmlr.Value == "body"
  xmlr.Read();
  Debug.Assert(
    xmlr.NodeType == XmlNodeType.EndElement)
  …
}
```

```
<elem>
  <subElem>
    body
  </subElem>
</elem>
```

DOM-basierte XML-Verarbeitung

- XmlDocument: Repräsentation eines XML-Dokuments im Speicher (DOM).
- Dokument kann mit einer Operation gelesen bzw. geschrieben werden. Im Speicher kann das XML-Dokument verändert werden.

```
Stream stream = new SomeStream(…)

XmlDocument dom = new XmlDocument();
dom.Load(stream);
XmlNodeList elems = dom.DocumentElement.ChildNodes;
foreach (XmlNode elem in elems) {
  Process(elem.InnerText);

XmlNode newElem = dom.CreateElement("elem");
newElem.InnerText = "…";
elems.AppendChild(newElem);

dom.Save(stream);
```