- Analoge Klassen stehen für die Ausgabe zur Verfügung.
- Die grundlegende Klasse für `byte`-Ausgabe heißt `OutputStream`.
- Auch `OutputStream` ist abstrakt :-)

Nützliche Operationen:

- `public void write(int b) throws IOException` : schreibt das unterste Byte von `b` in die Ausgabe;
- `void flush() throws IOException` : falls die Ausgabe gepuffert wurde, soll sie nun ausgegeben werden;
- `void close() throws IOException` : schließt den Ausgabe-Strom.

- Weil `OutputStream` abstrakt ist, gibt es keine Objekte der Klasse `OutputStream`, nur Objekte von Unterklassen.

Konstruktoren von Unterklassen:

- `public FileOutputStream(String path) throws IOException;`
- `public FileOutputStream(String path, boolean append) throws IOException;`
- `public DataOutputStream(OutputStream out);`
- `public PrintStream(OutputStream out)` — der Rückwärts-Kompatibilität wegen, d.h. um Ausgabe auf `System.out` und `System.err` zu machen ...

Beispiel:

```
import java.io.*;
public class File2FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream fileIn = new FileInputStream(args[0]);
   FileOutputStream fileOut = new FileOutputStream(args[1]);
   int n = fileIn.available();
   for(int i=0; i<n; ++i)
      fileOut.write(fileIn.read());
   fileIn.close(); fileOut.close();
   System.out.print("\t\tDone!!!\n");
   } // end of main
} // end of File2FileCopy
```

- Das Programm interpretiert die 1. und 2. Kommando-Zeilen-Argumente als Zugriffspfade auf eine Ein- bzw. Ausgabe-Datei.
- Die Anzahl der in der Eingabe enthaltenen Bytes wird bestimmt.
- Dann werden sukzessive die Bytes gelesen und in die Ausgabe-Datei geschrieben.

Erweiterung der Funktionalität:

Die Klasse `DataOutputStream` bietet spezielle Schreib-Methoden für verschiedene Basis-Typen an.

Beispielsweise gibt es:

- `void writeByte(int x) throws IOException;`
- `void writeChar(int x) throws IOException;`
- `void writeInt(int x) throws IOException;`
- `void writeDouble(double x) throws IOException.`

Beachte:

- `writeChar()` schreibt genau die Repräsentation eines Zeichens, die von `readChar()` verstanden wird, d.h. 2 Byte.

Beispiel:

```
import java.io.*;
public class Numbers {
public static void main(String[] args) throws IOException {
   FileOutputStream file = new FileOutputStream(args[0]);
   DataOutputStream data = new DataOutputStream(file);
   int n = Integer.parseInt(args[1]);
   for(int i=0; i<n; ++i)
      data.writeInt(i);
   data.close();
   System.out.print("\t\tDone!\n");
   } // end of main
} // end of Numbers
```

- Das Programm entnimmt der Kommando-Zeile den Datei-Pfad sowie eine Zahl `n`.
- Es wird ein `DataOutputStream` für die Datei eröffnet.
- In die Datei werden die Zahlen 0,...,`n-1` binär geschrieben.
- Das sind also 4`n` Bytes.

Achtung:

- In der Klasse `System` sind die zwei vordefinierten Ausgabe-Ströme `out` und `err` enthalten.
- `out` repräsentiert die Ausgabe auf dem Terminal.
- `err` repräsentiert die Fehler-Ausgabe für ein Programm (i.a. ebenfalls auf dem Terminal).
- Diese sollen jedes Objekt als Text auszugeben können.
- Dazu dient die Klasse `PrintStream`.

- Die `public`-Objekt-Methoden `print()` und `println()` gibt es für jeden möglichen Argument-Typ.
- Für (Programmierer-definierte) Klassen wird dabei auf die `toString()`-Methode zurückgegriffen.
- `println()` unterscheidet sich von `print`, indem nach Ausgabe des Arguments eine neue Zeile begonnen wird.

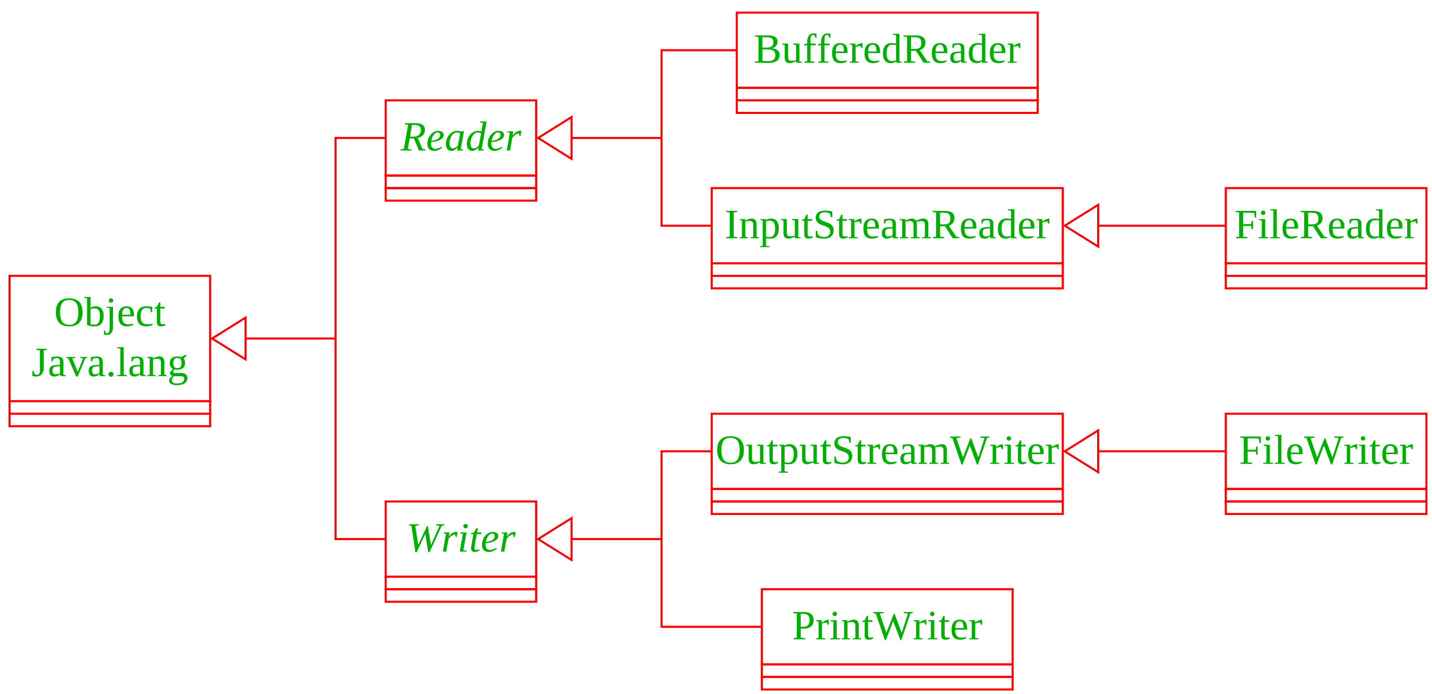
Achtung:

- `PrintStream` benutzt (neuerdings) die Standard-Codierung des Systems, d.h. bei uns Latin-1.
- Um aus einem (Unicode-)String eine Folge von sichtbaren (Latin-1-)Zeichen zu gewinnen, wird bei jedem Zeichen, das kein Latin-1-Zeichen darstellt, ein '?' gedruckt ...

$\Longrightarrow$ für echte (Unicode-) Text-Manipulation schlecht geeignet ...

16.2 Textuelle Ein- und Ausgabe

Wieder erstmal eine Übersicht über hier nützliche Klassen ...



- Die Klasse `Reader` ist abstrakt.

Wichtige Operationen:

- `public boolean ready() throws IOException :`
  liefert `true`, sofern ein Zeichen gelesen werden kann;
- `public int read() throws IOException :`
  liest ein (Unicode-)Zeichen vom Input als `int`-Wert; ist das Ende des Stroms erreicht, wird -1 geliefert;
- `void close() throws IOException :` schließt den Eingabe-Strom.

- Ein `InputStreamReader` ist ein spezieller Textleser für Eingabe-Ströme.
- Ein `FileReader` gestattet, aus einer Datei Text zu lesen.
- Ein `BufferedReader` ist ein `Reader`, der Text zeilenweise lesen kann, d.h. eine zusätzliche Methode `public String readLine() throws IOException;` zur Verfügung stellt.

Konstruktoren:

- `public InputStreamReader(InputStream in);`
- `public InputStreamReader(InputStream in, String encoding) throws IOException;`
- `public FileReader(String path) throws IOException;`
- `public BufferedReader(Reader in);`

Beispiel:

```
import java.io.*;
public class CountLines {
public static void main(String[] args) throws IOException {
   FileReader file = new FileReader(args[0]);
   BufferedReader buff = new BufferedReader(file);
   int n=0; while(null != buff.readLine()) n++;
   buff.close();
   System.out.print("Number of Lines:\t\t"+ n);
   } // end of main
} // end of CountLines
```

- Die Objekt-Methode `readLine()` liefert `null`, wenn beim Lesen das Ende der Datei erreicht wurde.
- Das Programm zählt die Anzahl der Zeilen einer Datei :-)

- Wieder stehen analoge Klassen für die Ausgabe zur Verfügung.
- Die grundlegende Klasse für textuelle Ausgabe heißt `Writer`.

Nützliche Operationen:

- `public void write(` *type* `x) throws IOException` :
  gibt es für die Typen `int` (dann wird ein einzelnes Zeichen geschrieben), `char[]` sowie `String`.
- `void flush() throws IOException` :
  falls die Ausgabe gepuffert wurde, soll sie nun tatsächlich ausgegeben werden;
- `void close() throws IOException` : schließt den Ausgabe-Strom.

- Weil `Writer` abstrakt ist, gibt keine Objekte der Klasse `Writer`, nur Objekte von Unterklassen.

Konstruktoren:

- `public OutputStreamWriter(OutputStream str);`
- `public OutputStreamWriter(OutputStream str, String encoding);`
- `public FileWriter(String path) throws IOException;`
- `public FileWriter(String path, boolean append) throws IOException;`
- `public PrintWriter(OutputStream out);`
- `public PrintWriter(Writer out);`

Beispiel:

```
import java.io.*;
public class Text2TextCopy {
public static void main(String[] args) throws IOException {
   FileReader fileIn = new FileReader(args[0]);
   FileWriter fileOut = new FileWriter(args[1]);
   int c = fileIn.read();
   for(; c!=-1; c=fileIn.read())
      fileOut.write(c);
   fileIn.close(); fileOut.close();
   System.out.print("\t\tDone!!!\n");
   } // end of main
} // end of Text2TextCopy
```

Wichtig:

- Ohne einen Aufruf von `flush()` oder `close()` kann es passieren, dass das Programm beendet wird, bevor die Ausgabe in die Datei geschrieben wurde :-(
- Zum Vergleich: in der Klasse `OutputStream` wird `flush()` automatisch nach jedem Zeichen aufgerufen, das ein Zeilenende markiert.

Bleibt, die zusätzlichen Objekt-Methoden für einen `PrintWriter` aufzulisten...

- Analog zum `PrintStream` sind dies
  `public void print(` *type* `x);` und
  `public void println(` *type* `x);`

- ... die es gestatten, Werte jeglichen Typs (aber nun evt. in geeigneter Codierung) auszudrucken.

Text-Ein- und Ausgabe arbeitet mit (Folgen von) Zeichen.

Nicht schlecht, wenn man dafür die Klasse `String` etwas näher kennen würde ...

17 Hashing und die Klasse `String`

- Die Klasse `String` stellt Wörter von (Unicode-) Zeichen dar.
- Objekte dieser Klasse sind stets konstant, d.h. können nicht verändert werden.
- Veränderbare Wörter stellt die Klasse ↑`StringBuffer` zur Verfügung.

Beispiel:

```
String str = "abc";
```

... ist äquivalent zu:

```
char[] data = new char[] {'a', 'b', 'c'};
String str = new String(data);
```

Weitere Beispiele:

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc"+cde);
String c = "abc".substring(2,3);
String d = cde.substring(1,2);
```

- Die Klasse `String` stellt Methoden zur Verfügung, um
  - einzelne Zeichen oder Teilfolgen zu untersuchen,
  - Wörter zu vergleichen,
  - neue Kopien von Wörtern zu erzeugen, die etwa nur aus Klein- (oder Groß-) Buchstaben bestehen.
- Für jede Klasse gibt es eine Methode `String toString()`, die eine `String`-Darstellung liefert.
- Der Konkatenations-Operator “+” ist mithilfe der Methode `append()` der Klasse `StringBuffer` implementiert.

Einige Konstruktoren:

- `String();`
- `String(char[] value);`
- `String(String s);`
- `String(StringBuffer buffer);`

Einige Objekt-Methoden:

- `char charAt(int index);`
- `int compareTo(String anotherString);`
- `boolean equals(Object obj);`
- `String intern();`
- `int indexOf(int chr);`
- `int indexOf(int chr, int fromIndex);`
- `int lastIndexOf(int chr);`
- `int lastIndexOf(int chr, int fromIndex);`

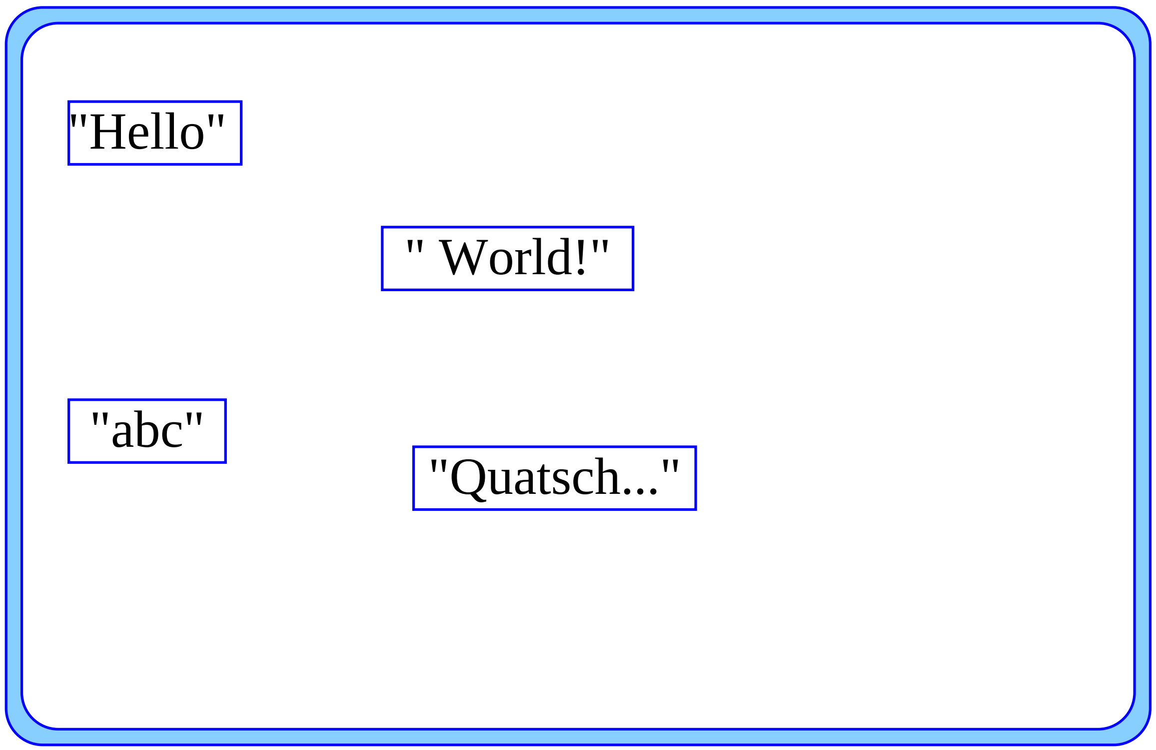
... weitere Objekt-Methoden:

- `int length();`
- `String replace(char oldChar, char newChar);`
- `String substring(int beginIndex);`
- `String substring(int beginIndex, int endIndex);`
- `char[] toCharArray();`
- `String toLowerCase();`
- `String toUpperCase();`
- `String trim();` : beseitigt White Space am Anfang und Ende des Worts.

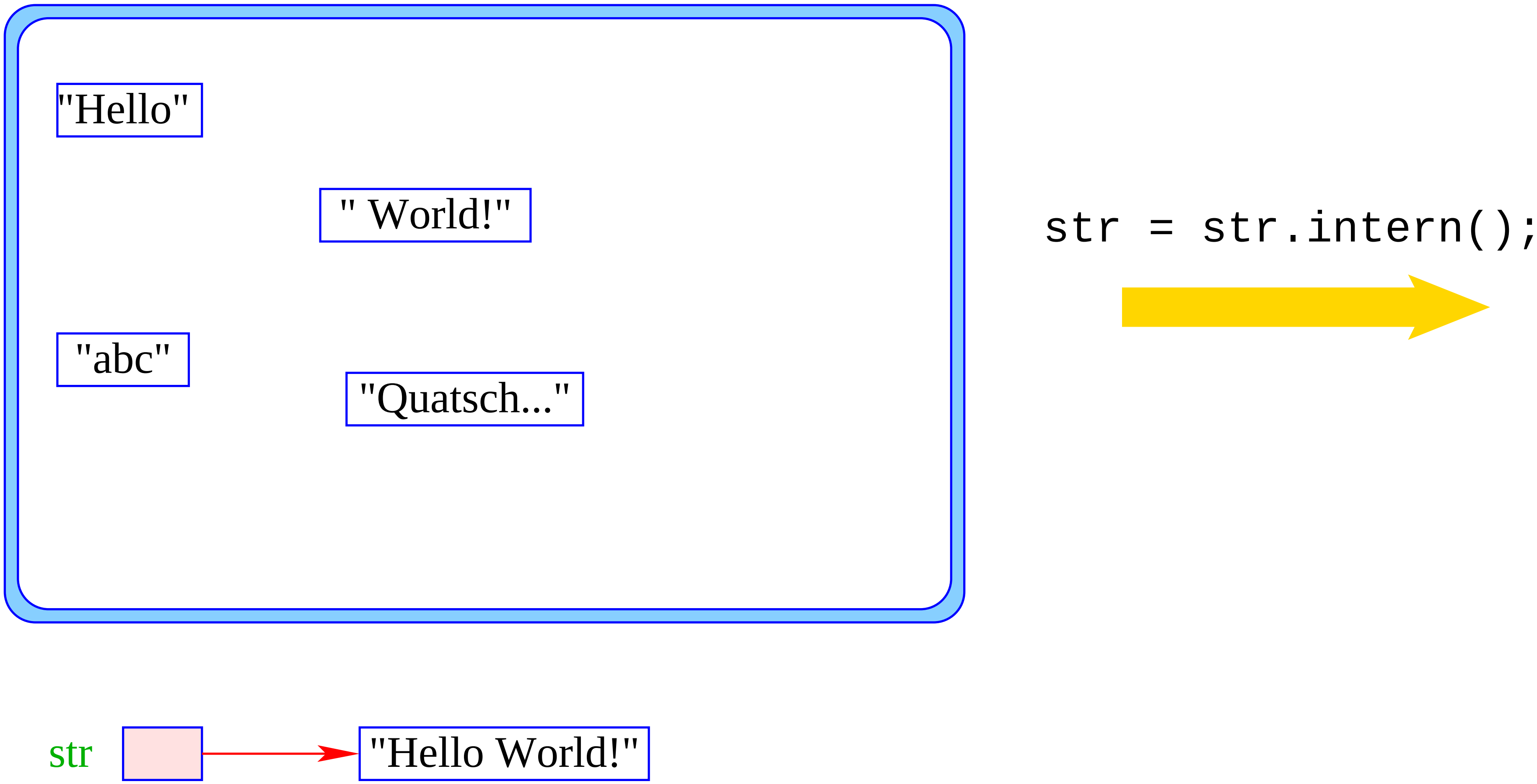
... sowie viele weitere :-)

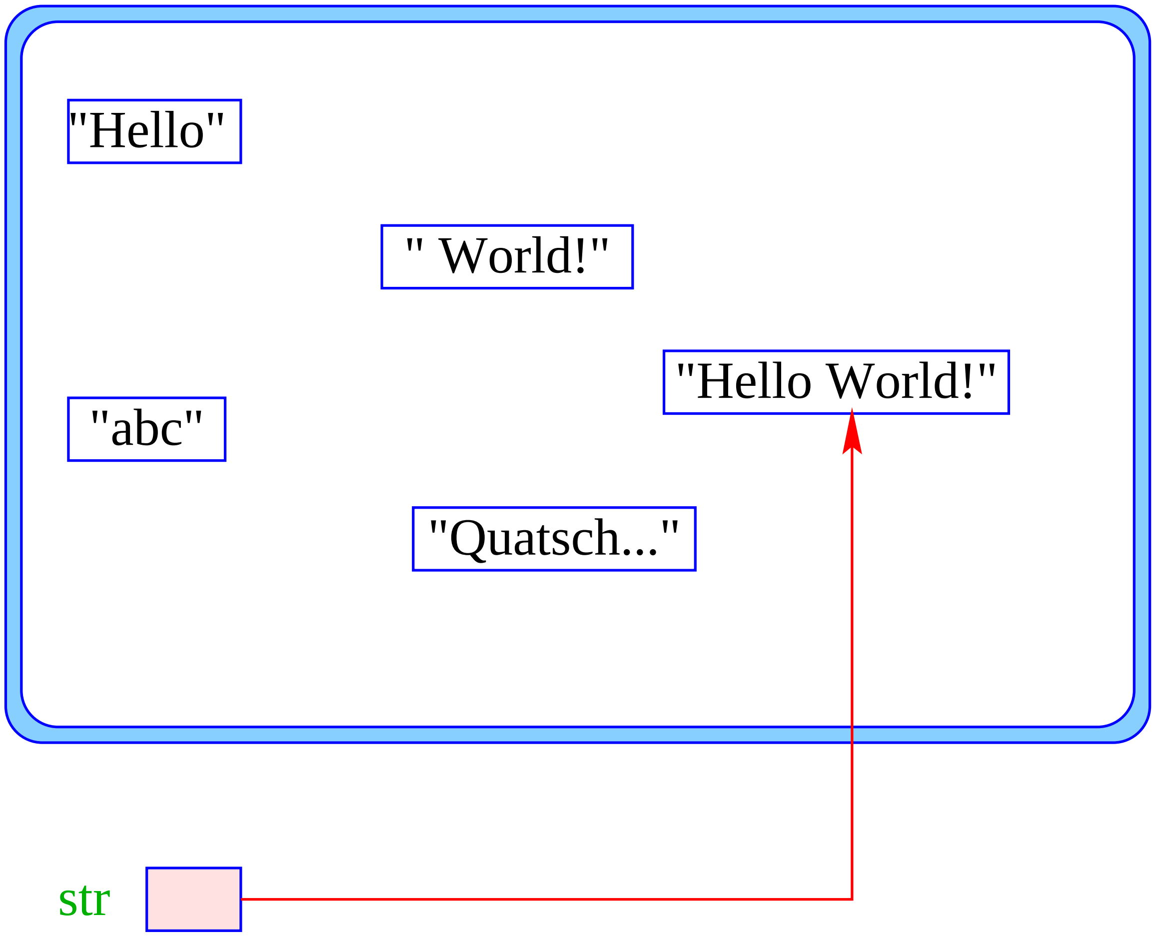
Zur Objekt-Methode `intern()`:

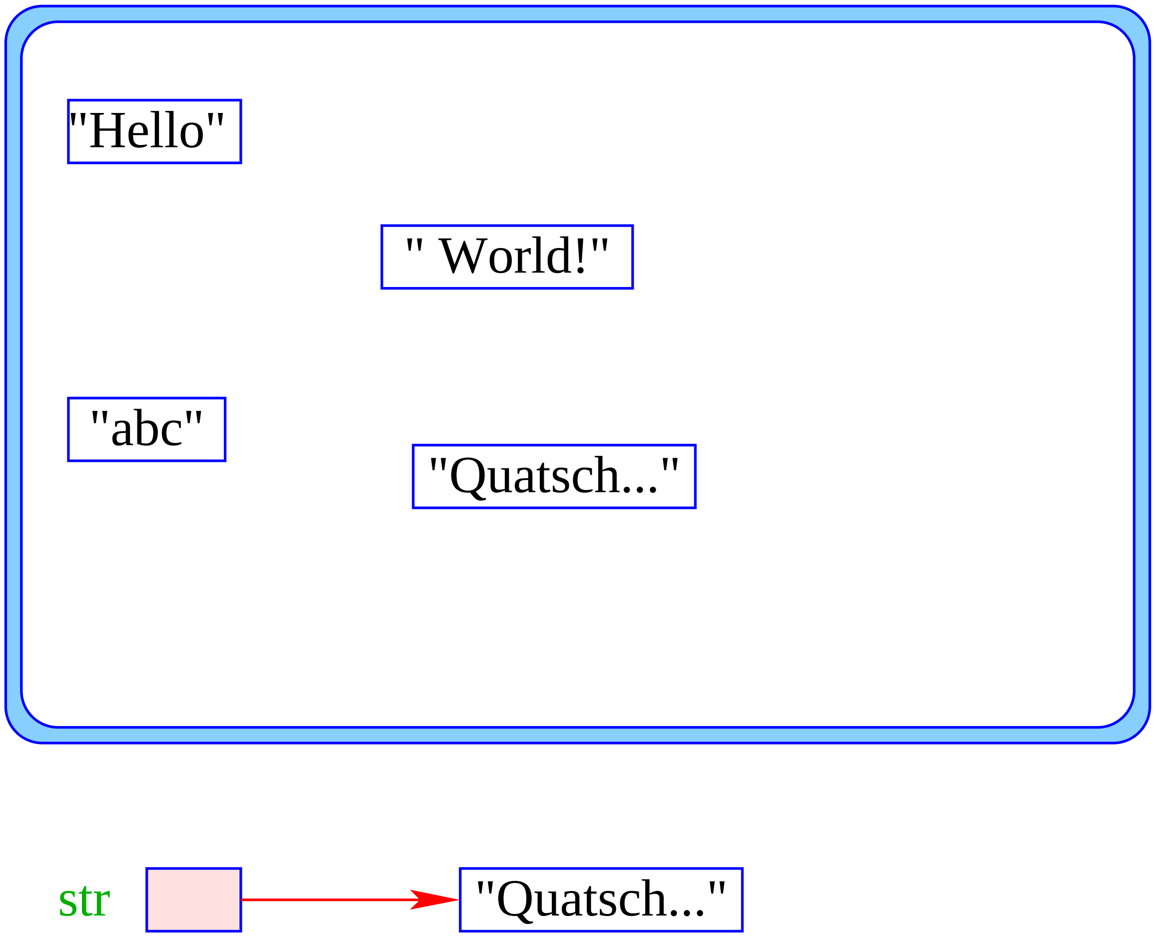
- Die Java-Klasse `String` verwaltet einen privaten String-Pool:



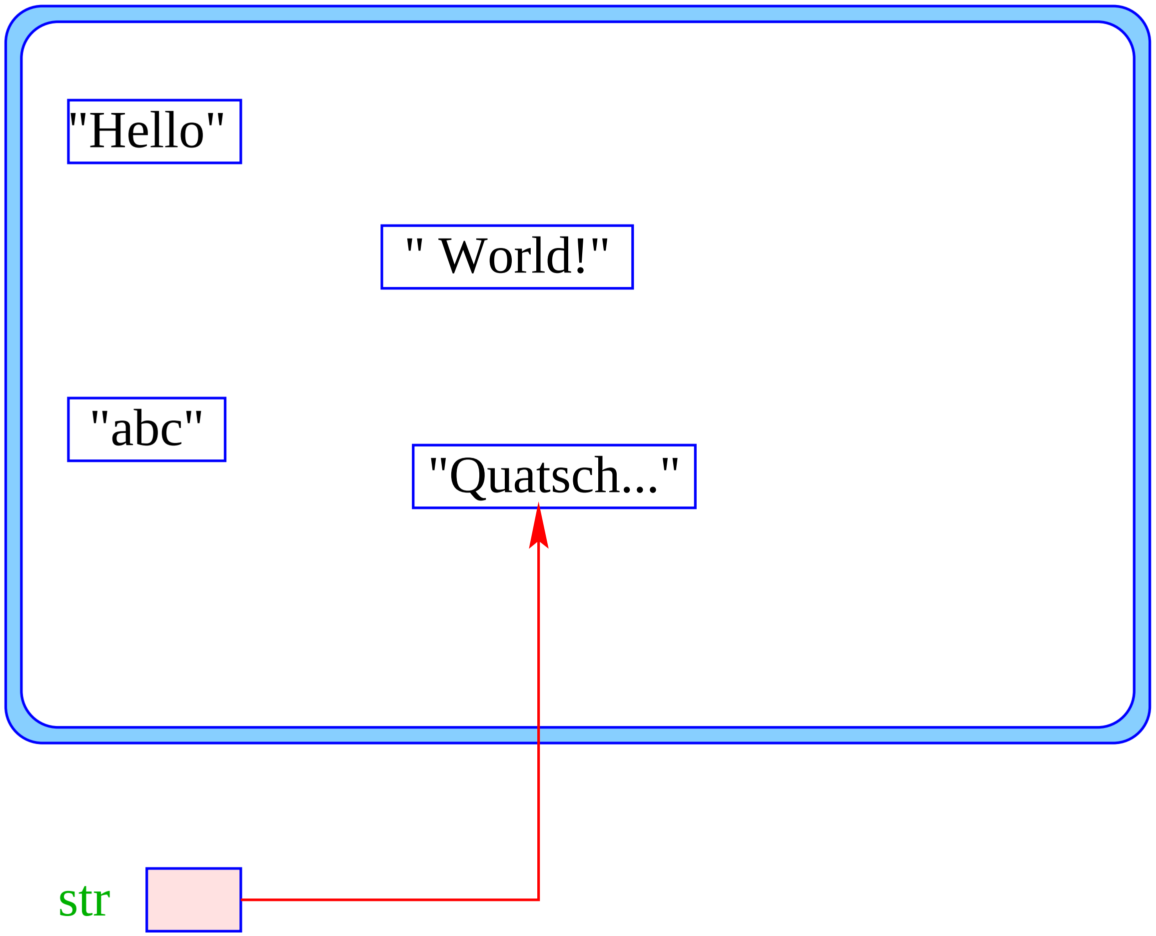
- Alle `String`-Konstanten des Programms werden automatisch in den Pool eingetragen.
- `s.intern();` überprüft, ob die gleiche Zeichenfolge wie `s` bereits im Pool ist.
- Ist dies der Fall, wird ein Verweis auf das Pool-Objekt zurück gegeben.
- Andernfalls wird `s` in den Pool eingetragen und `s` zurück geliefert.


"Hello"
" World!"
"abc"
"Quatsch..."
str = str.intern();
str
"Hello World!"


"Hello"
" World!"
"Hello World!"
"abc"
"Quatsch..."
str




```
str = str.intern();
```


"Hello"
" World!"
"abc"
"Quatsch..."
str

Vorteil:

- Internalisierte Wörter existieren nur einmal :-)
- Test auf Gleichheit reduziert sich zu Test auf Referenz-Gleichheit, d.h. "=="

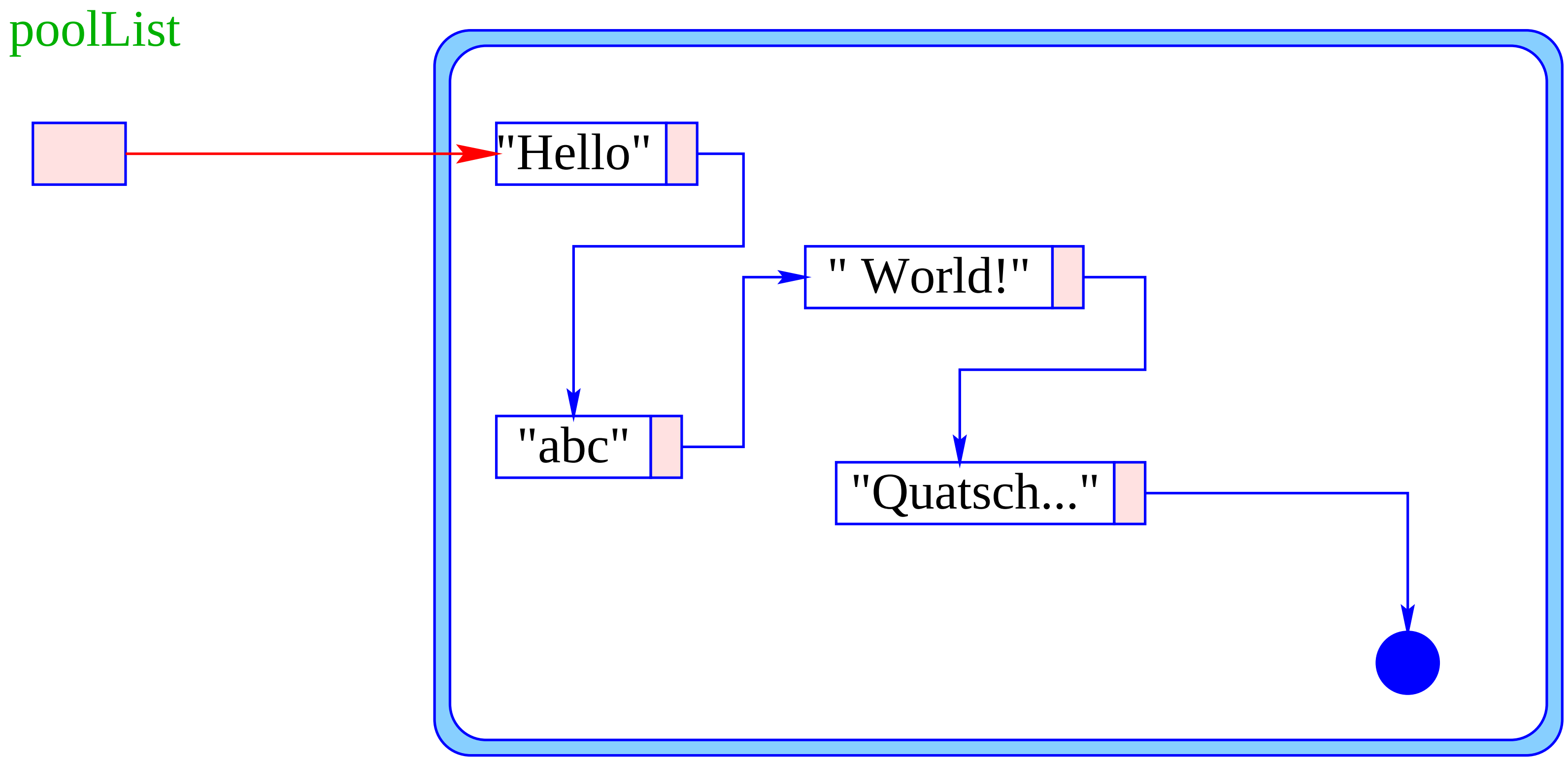
  $\Longrightarrow$ erheblich effizienter als zeichenweiser Vergleich !!!

... bleibt nur ein Problem:

- Wie findet man heraus, ob ein gleiches Wort im Pool ist ??

1. Idee:

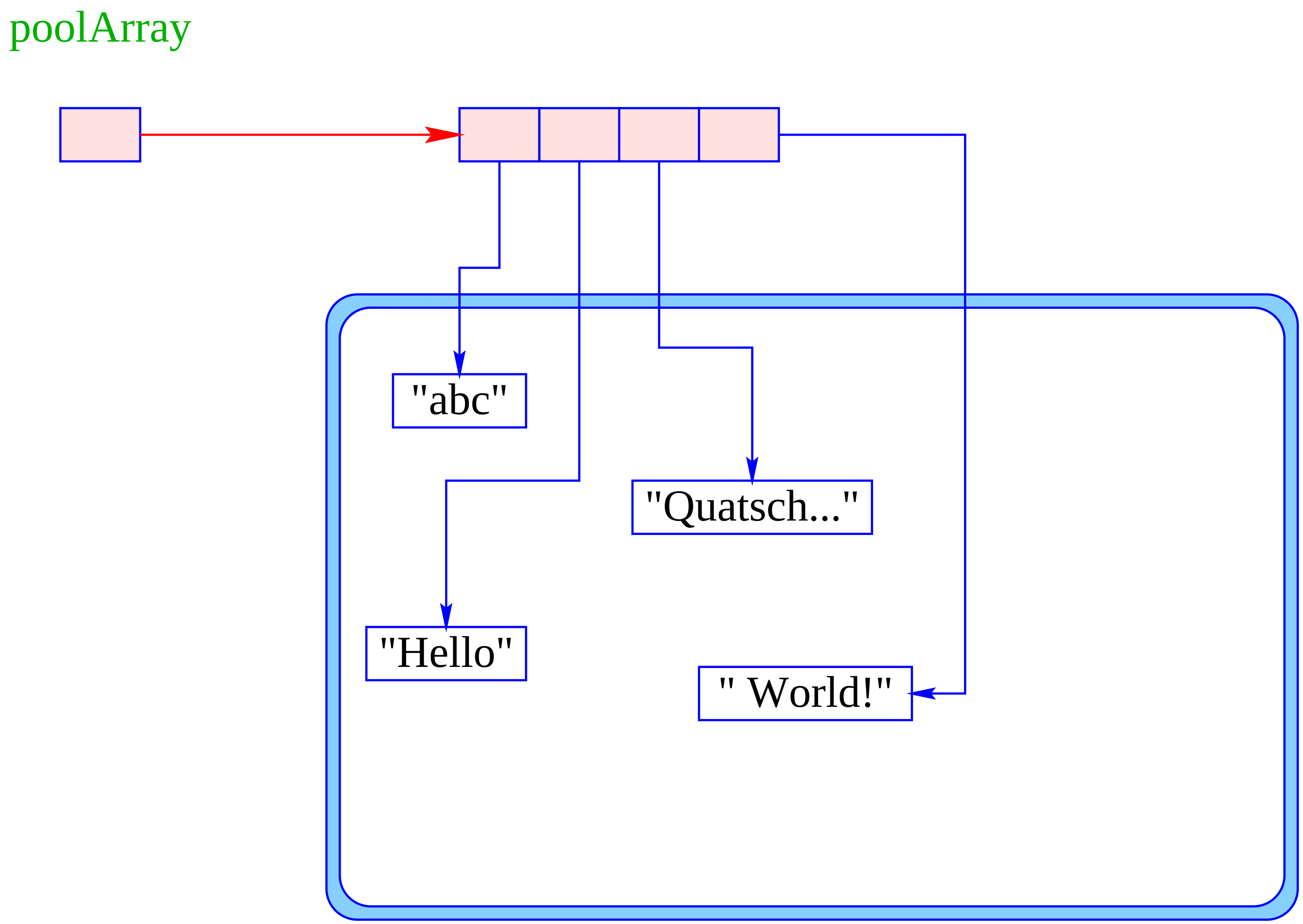
- Verwalte eine Liste der (Verweise auf die) Wörter im Pool;
- implementiere `intern()` als eine `List`-Methode, die die Liste nach dem gesuchten Wort durchsucht.
- Ist das Wort vorhanden, wird ein Verweis darauf zurückgegeben.
- Andernfalls wird das Wort (z.B. vorne) in die Liste eingefügt.


poolList
"Hello"
" World!"
"abc"
"Quatsch..."

- + Die Implementierung ist einfach.
- \- die Operation `intern()` muss das einzufügende Wort mit jedem Wort im Pool vergleichen $\Longrightarrow$ immens teuer !!!

2. Idee:

- Verwalte ein sortiertes Feld von (Verweisen auf) `String`-Objekte.
- Herausfinden, ob ein Wort bereits im Pool ist, ist dann ganz einfach ...


poolArray
"abc"
"Quatsch..."
"Hello"
" World!"

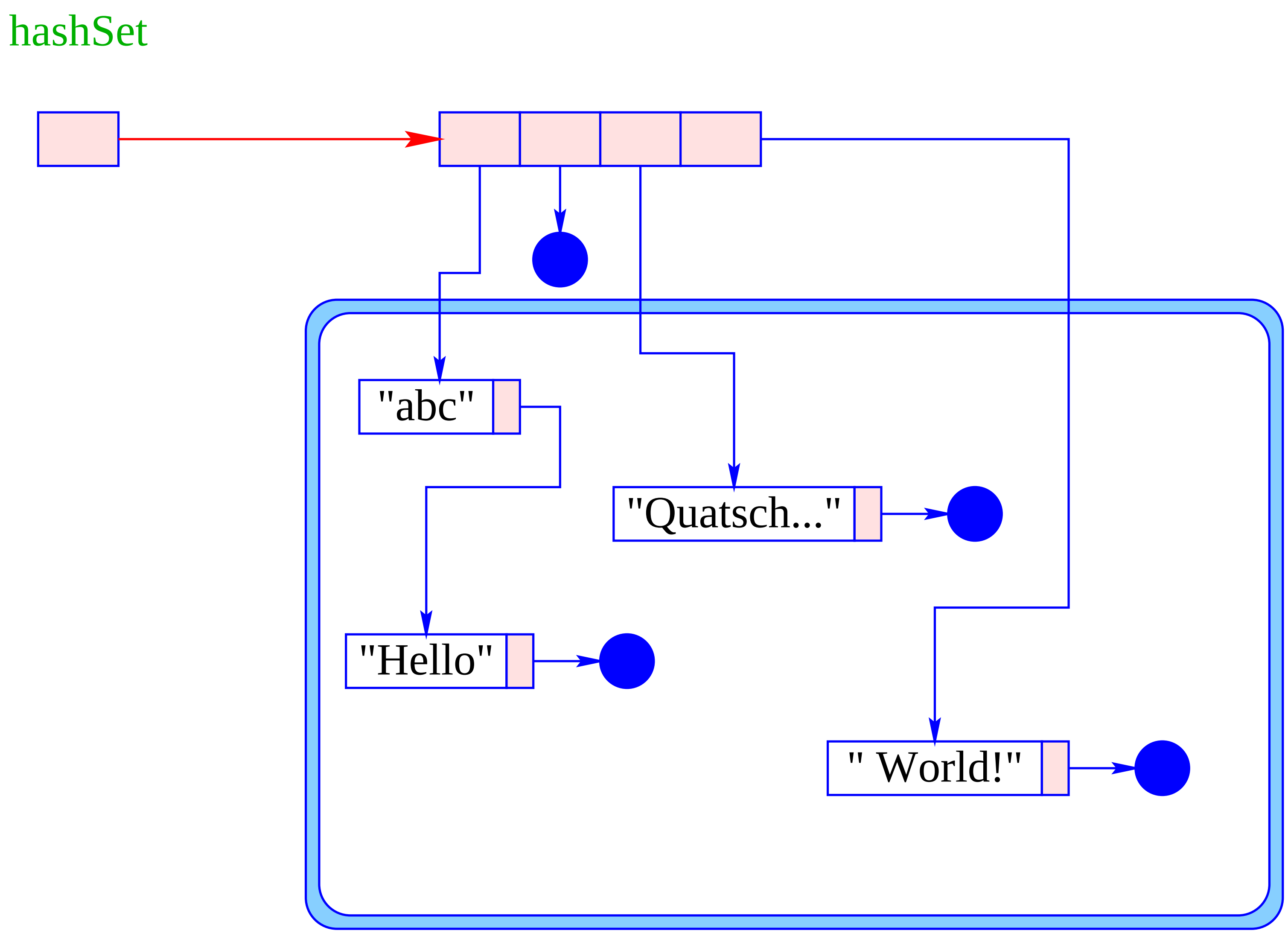
+ Auffinden eines Worts im Pool ist einfach.

- Einfügen eines neuen Worts erfordert aber evt. Kopieren aller bereits vorhandenen Verweise ...

$\Longrightarrow$ immer noch sehr teuer !!!

3. Idee: Hashing

- Verwalte nicht eine, sondern viele Listen!
- Verteile die Wörter (ungefähr) gleichmäßig über die Listen.
- Auffinden der richtigen Liste muss schnell möglich sein.
- In der richtigen Liste wird dann sequentiell gesucht.


hashSet
"abc"
"Quatsch..."
"Hello"
" World!"

Auffinden der richtigen Liste:

- Benutze eine (leicht zu berechnende :-) Funktion `hash: String -> int;`
- Eine solche Funktion heißt Hash-Funktion.
- Eine Hash-Funktion ist gut, wenn sie die Wörter (einigermaßen) gleichmäßig verteilt.
- Hat das Feld `hashSet` die Größe m, und gibt es n Wörter im Pool, dann müssen pro Aufruf von `intern();` nur Listen einer Länge ca. n/m durchsucht werden !!!

Sei s das Wort $s_0 s_1 \ldots s_{k-1}$.

Beispiele für Hash-Funktionen:

- $h_0(s) = s_0 + s_{k-1}$;
- $h_1(s) = s_0 + s_1 + \ldots + s_{k-1}$;
- $h_2(s) = (\ldots((s_0 * p) + s_1) * p + \ldots) * p + s_{k-1}$ für eine krumme Zahl p.

(Die `String`-Objekt-Methode `hashCode()` entspricht der Funktion h_2 mit $p = 31$.)

String	h_0	h_1	h_2 $(p = 7)$
alloc	196	523	276109
add	197	297	5553
and	197	307	5623
const	215	551	282083
div	218	323	5753
eq	214	214	820
fjump	214	546	287868
false	203	523	284371
halt	220	425	41297
jump	218	444	42966
less	223	439	42913
leq	221	322	6112
...		...	

String	h_0	h_1	h_2
...		...	
load	208	416	43262
mod	209	320	6218
mul	217	334	6268
neq	223	324	6210
neg	213	314	6200
not	226	337	6283
or	225	225	891
read	214	412	44830
store	216	557	322241
sub	213	330	6552
true	217	448	46294
write	220	555	330879