

In diesem Kapitel:

- *Generelles*
- *Die Architektur des Java Communications API*
- *Identifikation von Ports*
- *Kommunikation mit einem Device an einem Port*
- *Serielle Ports*
- *Parallele Ports*

Java Communications API

1.1. Generelles

Das Java Communications API ist eine Standard Erweiterung von Java, mit dem Java Applikationen (ohne Applets) Daten von und zu seriellen und parallelen Ports senden und empfangen können. Das Java Communications API gestattet es dem Programmierer mit Geräten zu kommunizieren, welche an parallelen oder seriellen Ports des Rechners angeschlossen sind: Scanner, Printer, Modem und ähnlichen Geräten. Das Comm API operiert auf einem sehr tiefen Level. Es versteht, wie man Bytes an diese Ports sendet oder von ihnen empfangen kann. Aber das API hat keine Ahnung, was es konkret sendet oder empfängt: die Bedeutung der Bytes ist nicht Thema der Kommunikation. Damit wir sinnvoll mit einem Gerät kommunizieren können, müssen wir das Kommunikationsprotokoll des Gerätes verstehen.

1.2. Die Architektur des APIs

Weil das Java Communications API eine Standard Extension ist, wird es nicht standardmässig mit dem JDK installiert. Sie müssen es also vom Java Site herunterladen:

<http://java.sun.com/products/javacomm>.

Das Java Communications API umfasst ein einziges Package, `javax.comm`, welches eine Unmenge Klassen, Exceptions und Interfaces enthält. Weil das API eine Standard Extension ist, hat es den Präfix `javax`. Das API umfasst (für Windows) auch ein DLL, welches die low level Funktionen, der Zugriff auf die Ports, kapselt. Zusätzlich werden einige Driver als Klassen, im Package `com.sun.comm` mitgeliefert, also Sun eigene Implementationen, die eventuell einmal ins `javax` übernommen werden. Wir befassen uns jedoch ausschliesslich mit dem `javax` Teil.

`javax.comm` besteht aus high-level und low-level Klassen.

- Die High-level Klassen sind für die Zugriffskontrolle und Zugriffsrechtverwaltung der Kommunikationsports zuständig. Sie sind auch für die Kommunikation zuständig.

Die `CommPortIdentifier` Klasse hilft Ihnen die Ports zu finden und zu öffnen.

Die `CommPort` Klasse stellt Ihnen Eingabe- und Ausgabe-Ströme zur Verfügung.

- Die Low-Level Klassen - `javax.comm.SerialPort` und `javax.comm.ParallelPort` sind für das Managen der Ports zuständig und das Lesen und Schreiben der Kontrollinformationen. Zudem werden mit diesen Klassen die grundlegenden Events weitergeleitet.

JAVA COMMUNICATIONS API

Die Java Comm Klassen verstehen RS232 serielle Ports und IEEE 1284 parallele Ports. Aber es ist denkbar, dass später auch Universal Serial Bus (USB), FireWire oder SCSI hinzukommen.

1.3. Das Identifizieren von Ports

Die `javax.comm.CommPortIdentifier` Klasse kontrolliert die Ports des Systems. Die Methoden dieser Klasse listen verfügbare Ports auf, zeigen, welche Programme die Kontrolle über diese Ports haben und öffnen die Ports, damit über diese kommuniziert werden kann.

Die aktuelle Eingabe und Ausgabe geschieht mittels einer Instanz der Klasse `javax.comm.CommPort`.

Die Aufgabe der `javax.comm.CommPortIdentifier` Klasse ist es, dafür zu sorgen, dass die einzelnen Programme, welche Zugriff auf einen Port erlangen möchten, fair miteinander umgehen.

1.3.1. Finden der Ports

bevor ein Port benutzt werden kann, muss der Port eindeutig identifiziert werden. Sie können aber nicht einfach eine Port Instanz kreieren, da Ports eng mit den physikalischen Ports korrespondieren müssen. Ein MacIntosh besitzt beispielsweise ganz andere Ports als eine Windows Maschine.

Daher verwendet man im Java Comm API statische Methoden mit nicht öffentlichen Konstruktoren:

```
public static Enumeration getPortIdentifiers()
public static CommPortIdentifier getPortIdentifier(String portName)
    throws NoSuchPortException
public static CommPortIdentifier getPortIdentifier(CommPort port)
    throws NoSuchPortException
```

Die allgemeinste dieser Methoden ist `CommPortIdentifier.getPortIdentifiers()`, welche ein `java.util.Enumeration` liefert, welches je einen `CommPortIdentifier` für jeden verfügbaren Port des Systems liefert.

Hier ein Beispiel:

```
package portlistener;

import javax.comm.*;
import java.util.*;
/**
 * Title:          Einfacher Port Listener
 * Description:    Beispiel für den Einsatz des Java Communication APIs
 * Copyright:      Copyright (c) J.M.Joller
 * Company:        Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class PortListener {

    public static void main(String[] args) {
        System.out.println("[PortListener]Start");
    }
}
```

JAVA COMMUNICATIONS API

```
System.out.println("[PortListener]CommPortIdentifiers
getPortIdentifiers");
Enumeration e = CommPortIdentifier.getPortIdentifiers();
System.out.println("[PortListener]CommPortIdentifiers  Auflisten");
while (e.hasMoreElements()) {
    System.out.println((CommPortIdentifier) e.nextElement());
}
System.out.println("[PortListener]Ende");
}
}
```

mit folgender Ausgabe:

```
[PortListener]Start
[PortListener]CommPortIdentifiers  getPortIdentifiers
[PortListener]CommPortIdentifiers  Auflisten
javax.comm.CommPortIdentifier@13dee9
javax.comm.CommPortIdentifier@fabe9
javax.comm.CommPortIdentifier@5f6ccd
javax.comm.CommPortIdentifier@601bb1
[PortListener]Ende
```

Mein Windows Rechner besitzt also vier Ports. Aber ich weiss nicht wie diese echt heissen. Mit den Objekt-Ids kann ich wenig anfangen.

Wir brauchen also eine bessere `toString()` Methode.

Wir können einen `CommPortIdentifier` mit Hilfe der statischen Methode `getPortIdentifier()` erhalten. Diese Methode liefert die Angaben entweder als Zeichenkette oder als Portobjekt. Als Zeichenkette können wir COM1, ... und LPT1, ... verwenden bzw. erhalten, für Windows! Auf Unix sieht das Ganze anders aus!

Hier das modifizierte Beispiel:

```
package benannterportlistener;

import javax.comm.*;

/**
 * Title:      Port Listener mit lesbaren Portnamen
 * Description: Einfaches Anwendungsbeispiel einiger Basisklassen vom Java
Communication API
 * Copyright:  Copyright (c) J.M.Joller
 * Company:    Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class NamedPortListener {
    public static void main(String[] args) {

        // COM Ports Serielle Ports
        try {
            int portNumber = 1;
            while (true) {
                CommPortIdentifier.getPortIdentifier("COM"+portNumber);
                System.out.println("COM"+portNumber);
                portNumber++;
            }
        }
    }
}
```

JAVA COMMUNICATIONS API

```
} catch(NoSuchPortException e) {
    // Ende der Schleife
}

// LPT Ports Parallele Ports
try {
    int portNumber = 1;
    while (true) {
        CommPortIdentifier.getPortIdentifier("LPT"+portNumber);
        System.out.println("LPT"+portNumber);
        portNumber++;
    }
} catch(NoSuchPortException e) {
    // Ende der Schleife
}
}
```

mit derlesbaren Ausgabe:

```
COM1
COM2
LPT1
LPT2
```

Nun wissen wir also, welche Ports (parallel und seriell) vorhanden sind. Das Programm funktioniert aber lediglich auf Windows.

1.3.2. Bestimmen von Informationen über den Port

Nachdem wir ein `CommPortIdentifier` Objekt haben, welches einen bestimmten Port identifiziert, können wir Informationen über den Port abfragen. Dazu stehen uns verschiedene Methoden zur Verfügung. Zum Beispiel:

```
public String getName()
public int getPortType()
public String getCurrentOwner()
public boolean isCurrentOwner()
```

Die `getName()` Methode liefert einen plattform-abhängigen Namen des Ports, beispielsweise "COM1" (Windows), "Serial A" (Solaris) oder "Modem" (Mac, falls das Comm API je für Mac's zur Verfügung steht).

Die `getPortType()` Methode liefert eine von zwei möglichen Konstanten:

- `CommPortIdentifier.PORT_SERIAL`
(`public static final int PORT_SERIAL=1;`) oder
- `CommPortIdentifier.PORT_PARALLEL`
(`public static final int PORT_PARALLEL=2;`)

Die `currentlyOwned()` Methode liefert `true`, falls der Prozess oder Thread oder die Applikation diesen Port zur Zeit kontrolliert; sonst wird `false` zurück geliefert.

Die `getCurrentOwner()` Methode liefert den Namen des Programms, welches den Port besetzt, sonst `null`. Allerdings werden nur Java Programme erkannt. Die Methode ist also im wesentlichen sinnlos, da beispielsweise ein Modem Prozess damit nicht erkannt wird. Im

JAVA COMMUNICATIONS API

Source Code der Klasse wird versprochen, dass dieser Fehler noch behoben werden soll.
Wann ist allerdings unklar (es wurde kein release fixiert).

Schauen wir uns ein einfaches Programm an, welches Ports abfragen kann:

```
package portabfrage;

import javax.comm.*;
import java.util.*;

/**
 * Title:      Port Listener mit Verzierungen
 * Description: Abfrage der Ports und Ausgabe unter Ausnutzung der Port
Eigenschaften
 * Copyright:  Copyright (c) J.M.Joller
 * Company:    Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class PrettyPortListener {
    public static void main(String[] args) {
        System.out.println("[PrettyPortListener]Start");
        Enumeration e = CommPortIdentifier.getPortIdentifiers();
        while (e.hasMoreElements()) {
            CommPortIdentifier com = (CommPortIdentifier) e.nextElement();
            System.out.print("[PrettyPortListener]Port Name : "+com.getName() );

            switch (com.getPortType() ) {
                case CommPortIdentifier.PORT_SERIAL :
                    System.out.print(" ist ein serieller Port");
                    break;
                case CommPortIdentifier.PORT_PARALLEL :
                    System.out.print(" ist ein paralleler Port");
                    break;
                default:
                    // für Erweiterungen, wie USB, ...
                    System.out.print(" ist ein unbekannter Porttyp");
                    break;
            }
            if (com.isCurrentlyOwned() ) {
                System.out.println(", der zur Zeit von "+com.getCurrentOwner()+
                    " benutzt wird.");
            }
            else {
                System.out.println(" und wird zur Zeit nicht benutzt.");
            }
        }
        System.out.println("[PrettyPortListener]Ende");
    }
}
```

mit folgender Ausgabe:

```
PrettyPortListener]Start
[PrettyPortListener]Port Name : COM1 ist ein serieller Port und wird zur Zeit nicht benutzt.
[PrettyPortListener]Port Name : COM2 ist ein serieller Port und wird zur Zeit nicht benutzt.
[PrettyPortListener]Port Name : LPT1 ist ein paralleler Port und wird zur Zeit nicht benutzt.
[PrettyPortListener]Port Name : LPT2 ist ein paralleler Port und wird zur Zeit nicht benutzt.
[PrettyPortListener]Ende
```

1.3.3. Öffnen von Ports

Bevor man in einen Port schreiben oder aus einem Port lesen kann, muss man diesen öffnen. Das Öffnen eines Ports gibt Ihrer Applikation das exklusive Recht für diesen Port, bis Sie die Kontrolle abgeben oder das Programm beendet wird. Ob Sie einen Port tatsächlich öffnen können, ist allerdings nicht sicher! Falls ein anderes Programm den Port benutzt, wird eine `PortInUseException` geworfen. Witzigerweise ist diese `Exception` keine Unterklasse der `IOException` Klasse:

```
public class PortInUseException extends Exception
```

Der `CommPortIdentifier` besitzt zwei `open()` Methoden, welche ein `javax.comm.CommPort` Objekt zurück liefern. Mit diesem Objekt können Sie Daten aus einem Port lesen oder Daten in einen Port schreiben.

Die erste `open()` Methode besitzt zwei Argumente, den Namen und einen Time-Out Parameter:

```
public synchronized CommPort open(String name, int timeout)
    throws PortInUseException
```

Als Name geben Sie den Namen des Programms an, welches den Port benutzen möchte. Dieser Name wird auch von der `getCurrentOwner()` Methode zurückgeliefert.

Der Timeout Parameter gibt die maximale Anzahl Millisekunden an, für die die aufrufende Methode blockiert wird, falls sie auf den Port warten muss. Falls die Methode nicht innerhalb dieser Zeitdauer abgeschlossen werden kann, wird die `PortInUseException` geworfen.

Beispiel:

```
package portowner;

import javax.comm.*;
import java.util.*;

/**
 * Title:          Port Owner
 * Description:    Bestimmen des Besitzers bzw. öffnen eines Ports
 * Copyright:     Copyright (c) J.M.Joller
 * Company:      Joller-Voss
 * @author J.M.Joller
 * @version 1.0
 */

public class PortOpener {
    public static void main(String[] args) {
        System.out.println("[PortOpener]Start");
        Enumeration diePorts = CommPortIdentifier.getPortIdentifiers();
        while (diePorts.hasMoreElements()) {
            CommPortIdentifier com = (CommPortIdentifier) diePorts.nextElement();
            System.out.print("[PortOpener]" + com.getName());
            switch (com.getPortType() ) {
                case CommPortIdentifier.PORT_SERIAL :
                    System.out.print(" ist ein serieller Port");
                    break;
                case CommPortIdentifier.PORT_PARALLEL :

```

JAVA COMMUNICATIONS API

```
        System.out.print(" ist ein paralleler Port");
        break;
    default:
        // für Erweiterungen, wie USB, ...
        System.out.print(" ist ein unbekannter Porttyp");
        break;
    }
    try {
        CommPort derPort = com.open("PortOpener",10);
        System.out.println(" wird zur Zeit nicht benutzt.");
        derPort.close();
    } catch (PortInUseException e) {
        String owner = com.getCurrentOwner();
        if (owner == null) owner = "unbekannt";
        System.out.println(" wird zur Zeit von "+owner+ " benutzt.");
    }
}
System.out.println("[PortOpener]Ende");
}
```

mit folgender Ausgabe :

```
PortOpener]Start
[PortOpener]COM1 ist ein serieller Port wird zur Zeit nicht benutzt.
[PortOpener]COM2 ist ein serieller Port wird zur Zeit nicht benutzt.
[PortOpener]LPT1 ist ein paralleler Port wird zur Zeit nicht benutzt.
[PortOpener]LPT2 ist ein paralleler Port wird zur Zeit von Port currently not owned benutzt.
[PortOpener]Ende
```

Obschon ich einen Port besetzt habe, mit einem Scanner, wurde dies nicht erkannt, da der Scanner nicht ein Java Programm einsetzt, welches in der selben JVM läuft, wie dieses Programm.

Die zweite `open()` Methode verwendet eine Dateibeschreibung als Parameter:

```
public CommPort open(FileDescriptor fd)
                    throws UnsupportedOperationException
```

Diese Methode istz beispielsweise in Unix sinnvoll, da dort alle Eingabe / Ausgabe Ports als Dateien beschrieben werden.

Falls Sie die Methode auf einem Windows Rechner einsetzen wollen, wird eine `UnsupportedOperationException` geworfen.

Das Gegenstück, die `close()` Methode zur `open()` Methode, gibt es nicht. Es liegt an Ihnen die Ports möglichst schnell zu schliessen.

JAVA COMMUNICATIONS API

1.3.4. Warten auf einen Port mit Ownership

Der `CommPortIdentifier` besitzt als Objekt zwei Methoden, mit denen man die Veränderungen der Ownership eines Ports feststellen kann:

```
public void addPortOwnershipListener(CommPortOwnershipListener listener)
public void removePortOwnershipListener(CommPortOwnershipListener listener)
```

Die Port Ownership Events werden beim öffnen, schliessen, .. eines Ports gefeuert, gesetzt. Um auf die Ereignisse reagieren zu können, muss ein `CommPortOwnershipListener` Objekt mit dem `CommPortIdentifier` verknüpft werden (der Listener muss registriert werden):

```
public void addPortOwnershipListener(CommPortOwnershipListener listener)
```

Natürlich können Sie eine Registrierung auch rückgängig machen:

```
public void removePortOwnershipListener(CommPortOwnershipListener listener)
```

Das Listener Interface `javax.comm.CommPortOwnershipListener` ist ein Subinterface von `java.util.EventListener`, mit einer einzigen Methode:

```
public abstract void ownershipChange(int type)
```

Das `CommPortOwnershipListener` Interface ist ungewöhnlich: im Gegensatz zu anderen Event Listener Interfaces übergibt die Listener Methode eine `int`, kein Objekt. Der Wert der Integer Zahl ist in der Regel einer der folgenden vordefinierten Werte:

```
CommPortOwnershipListener.PORT_OWNED
CommPortOwnershipListener.PORT_UNOWNED
CommPortOwnershipListener.PORT_OWNERSHIP_REQUESTED
```

Die Bedeutung der drei Werte ergibt sich eigentlich aus deren Bezeichnung:

`PORT_OWNED`:

eine Applikation besitzt das Zugriffsrecht auf diesen Port;

`PORT_UNOWNED`:

eine andere Applikation hat den Zugriff auf diesen Port freigegeben;

`PORT_OWNERSHIP_REQUESTED`:

eine Applikation hat den Zugriff auf diesen Port verlangt, besitzt aber noch keine Zugriffsrechte. Falls eine Applikation dieses Event abfängt, könnte sie beispielsweise den Port freigeben, damit die andere Applikation diesen benutzen kann.

Das folgende, eher unbrauchbare Programm (weil nur innerhalb der Java Programme die Events weitergeleitet werden!), zeigt diese Ereignissteuerung:

```
package portownership;

import javax.comm.*;

/**
 * Title:          Port Ownership
 * Description:    Beispiel für die Eventsteuerung mit den Port Ownership
 * Events
 */
```

JAVA COMMUNICATIONS API

```
* @author J.M.Joller
* @version 1.0
*/

public class PortWatcher implements CommPortOwnershipListener {

    String portName;

    public PortWatcher(String portName) throws NoSuchPortException {
        this.portName = portName;
        System.out.println("[PortWatcher]CommPortIdentifier
getPortIdentifier");
        CommPortIdentifier portIdentifier =
            CommPortIdentifier.getPortIdentifier(portName);
        System.out.println("[PortWatcher]CommPortIdentifier
addPortOwnershipListener");
        portIdentifier.addPortOwnershipListener(this);
    }

    public void ownershipChange(int type) {

        System.out.println("[PortWatcher]ownershipChange");
        switch (type) {

            case CommPortOwnershipListener.PORT_OWNED:
                System.out.println("[PortWatcher]" + portName + " ist nicht
verfügbar");
                break;
            case CommPortOwnershipListener.PORT_UNOWNED:
                System.out.println("[PortWatcher]" + portName + " ist nicht
verfügbar");
                break;
            case CommPortOwnershipListener.PORT_OWNERSHIP_REQUESTED:
                System.out.println("[PortWatcher]Eine Applikation verlangt den Port
"
                + portName);
                break;
            default:
                System.out.println("[PortWatcher]Unknown Port Ownership Event,
Typus: " + type);
        }
    }

    public static void main(String[] args) {
        System.out.println("[PortWatcher]Start");
        try {
            //PortWatcher pw = new PortWatcher(args[0]);
            System.out.println("[PortWatcher]COM1");
            PortWatcher pw = new PortWatcher("COM1");
        }
        catch (Exception e) {
            System.err.println("Usage: java PortWatcher port_name");
        }
    }
}
```

mit folgender Ausgabe:

```
PortWatcher]Start
[PortWatcher]CommPortIdentifier    getPortIdentifier
[PortWatcher]CommPortIdentifier    addPortOwnershipListener
```

1.4. *Kommunikation mit einem Gerät an einem Port*

Die `open()` Methode des `CommPortIdentifier` Objekts liefert ein `CommPort` Objekt zurück. Diese Klasse, `javax.comm.CommPort` besitzt Methoden, mit deren Hilfe Eingabe- und Ausgabe-Ströme von und zu einem Port bestimmt werden können. Zudem gibt es einige Methoden, mit deren Hilfe die Charakteristiken des Ports verändert werden können.

1.4.1. Kommunikation mit einem Port

Grundsätzlich geschieht die Kommunikation mit einem Port in fünf Schritten:

1. öffnen des Ports mit der `open()` Methode des `CommPortIdentifier`s.
Falls der Port verfügbar ist, wird ein `CommPort` Objekt zurück geliefert.
Sonst wird eine `PortInUseException` geworfen.
2. bestimmen des Ausgabestroms des Ports mit der `getOutputStream()` Methode des `CommPort` Objekts.
3. bestimmen des Eingabestroms des Ports mit der `getInputStream()` Methode des `CommPort` Objekts.
4. lesen und schreiben der Daten in und aus diesen Strömen.
5. schliessen des Ports mit der `close()` Methode des `CommPort` Objekts.

Das Bestimmen der Eingabe- und Ausgabe-Ströme erinnert an die in der Netzwerk-Programmierung verwendeten Methoden, in `java.net.URL`. Im Unterschied zu einer Netzwerkklassse versteht ein Port aber kein Protokoll und auch der Content ist im egal! Wenn Sie beispielsweise ein Modem ansteuern wollen, müssen Sie die Befehlsfolgen kennen; das System hilft Ihnen nicht im geringsten!

Die obigen zwei Methoden für die Ströme sind abstrakt definiert:

```
public abstract InputStream getInputStream() throws IOException
public abstract OutputStream getOutputStream() throws IOException
```

Aber obschon die Methoden abstrakt deklariert sind, erhält jede Instanz von `CommPort` eine konkrete Unterklasse von `CommPort` und eine Implementation dieser Methoden.

Einige Ports sind unidirektional, Sie können also lediglich schreiben oder lesen. Falls ein Port kein Lesen erlaubt, liefert die Methode `getInputStream()` `null` als Rückgabewert (`InputStream`).

Das folgende Beispiel zeigt, wie sehr rudimentär mit einem Modem kommuniziert werden könnte. Damit die Eingabe und Ausgabe besser funktionieren, werden je ein Thread für beide Richtungen kreiert:

```
package porttyper;

/**
 * Title:          Einfacher Port Typer
 * Description:    Beispiel, wie mit einem Modem kommuniziert werden kann
 * Copyright:      Copyright (c) J.M.Joller
```

JAVA COMMUNICATIONS API

```
* Company:      Joller-Voss
* @author J.M.Joller
* @version 1.0
*/

import javax.comm.*;
import java.util.*;
import java.io.*;

public class PortTyper {

    public static void main(String[] args) {
        System.out.println("[PortTyper]Start");
        if (args.length < 1) {
            System.out.println("Usage: java PortTyper portName");
            //return;
        }

        try {
            System.out.println("[PortTyper]CommPortIdentifier
getPortIdentifier");
            //CommPortIdentifier com =
CommPortIdentifier.getPortIdentifier(args[0]);
            CommPortIdentifier com =
CommPortIdentifier.getPortIdentifier("COM1");
            System.out.println("[PortTyper]CommPortIdentifier open");
            CommPort derPort = com.open("PortOpener", 10);
            System.out.println("[PortTyper]CopyThread  input");
            CopyThread input = new CopyThread(System.in,
derPort.getOutputStream(), "Eingabe");
            System.out.println("[PortTyper]CopyThread  output");
            CopyThread output = new CopyThread(derPort.getInputStream(),
System.out, "Ausgabe");
            input.start();
            output.start();
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}

class CopyThread extends Thread {

    InputStream dieEingabe;
    OutputStream dieAusgabe;
    String strRichtung;

    CopyThread(InputStream in) {
        this(in, System.out);
    }

    CopyThread(OutputStream out) {
        this(System.in, out);
    }

    CopyThread(InputStream in, OutputStream out, String richtung) {
        dieEingabe = in;
    }
}
```

JAVA COMMUNICATIONS API

```
        dieAusgabe = out;
        strRichtung = richtung;
    }

    CopyThread(InputStream in, OutputStream out) {
        dieEingabe = in;
        dieAusgabe = out;
    }

    public void run() {
        System.out.println("[CopyThread]run() "+strRichtung);
        try {
            byte[] buffer = new byte[256];
            while (true) {
                int bytesRead = dieEingabe.read(buffer);
                if (bytesRead == -1) break;
                System.out.println("[CopyThread]Echo");
                for (int ix=0; ix< bytesRead; ix++)
                    System.out.print((char)buffer[ix]);
                dieAusgabe.write(buffer, 0, bytesRead);
            }
        }
        catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

mit der Ausgabe:

```
[PortTyper]Start
Usage: java PortTyper portName
[PortTyper]CommPortIdentifier getPortIdentifier
[PortTyper]CommPortIdentifier open
[PortTyper]CopyThread input
[PortTyper]CopyThread output
[CopyThread]run() Eingabe
[CopyThread]run() Ausgabe
Hallo
[CopyThread]Echo
Hallo
```

Da ich kein Modem angeschlossen habe, kann ich die AT Befehle nicht auswendig und auch nicht über's Modem kommunizieren, daher die banale Übungseingabe.

1.4.2. Port Eigenschaften

Die Klasse `javax.comm.CommPort` besitzt einige Driver-abhängige Methoden mit denen die Port Eigenschaften angepasst werden können. Diese Grössen sind aber überwiegend generische Grössen, wie etwa die Buffergrösse. Um beispielsweise die Bausrate einer Verbindung zu setzen, müssen Sie speziellere Unterklassen einsetzen, beispielsweise `javax.comm.SerialPort` oder `javax.comm.ParallelPort`.

Fünf generische Eigenschaften sind Receive Treshold, Receive Timeout, Receive Framing, Input Buffer Size und Output Buffer Size. Vier dieser Eigenschaften bestimmen wie und wann der Eingabestrom geblockt wird.

Die Receive Threshold bestimmt die Anzahl Bytes, welche vorhanden sein müssen, bevor ein `read()` Befehl abgeschlossen wird.

JAVA COMMUNICATIONS API

Die Receive Timeout gibt an, wie lange der `read()` Befehl warten muss (in Millisekunden) bevor er abgeschlossen wird.

Die Grösse des Eingabepuffers gibt an, wieviele Bytes für den seriellen Port zur Verfügung stehen. Falls der Puffer voll ist, wird der `read()` Befehl abgeschlossen und der Inhalt des Puffers gelesen und zurückgesetzt.

Beispiel:

- falls die Threshold auf 5 gesetzt wird, wartet der `read()` Befehl, bis 5 Bytes vorhanden sind.
- falls die Timeout auf 10 (Millisekunden) gesetzt wird, wartet `read()` 10 Millisekunden bevor der Befehl abbricht. Falls früher Daten vorhanden sind, verfährt die Methode wie beim Threshold.
- falls der Eingabepuffer und die Receive Threshold gesetzt sind, muss der kleinere der beiden Werte erreicht sein, bevor der `read()` Befehl abgeschlossen werden kann.
- falls Receive Framing eingeschaltet ist, wird der `read()` Befehl immer sofort abgeschlossen.

Was liefert `read()` wann?

Receive Threshold	Receive Timeout	Receive Framing	Input Buffer Size	<code>read()</code> wird beendet falls
disabled	disabled	disabled	b Bytes	wird abgeschlossen, sobald irgendwelche Daten vorliegen
n Bytes	disabled	disabled	b Bytes	entweder n oder b Bytes sind verfügbar
disabled	t (ms)	disabled	b Bytes	liefert nach t ms Daten oder sobald welche vorhanden sind
n Bytes	t ms	disabled	b Bytes	liefert nach t Millisekunden oder falls entweder n oder b Bytes vorliegen die Daten
disabled	disabled	enabled	b Bytes	liefert die Daten sofort
n Bytes	disabled	enabled	b Bytes	liefert die Daten sofort
disabled	t ms	enabled	b Bytes	liefert die Daten sofort
n Bytes	t ms	enabled	b Bytes	liefert die Daten sofort

Der Output Buffer Size ist die Anzahl Bytes, die der Driver für den Ausgabestrom speichern kann, bevor die Daten in den Port geschrieben werden. Der Puffer ist wichtig, weil ein Programm sehr leicht schneller Daten schreiben kann, als der Port senden. Pufferüberläufe sind ein Standardproblem, speziell bei älteren PCs.

Jede Eigenschaft besitzt vier zugeordnete Methoden, zum Enablen, zum Disablen, zum Liefern des aktuellen Wertes, zum Prüfen, ob die Eigenschaft überhaupt vorhanden ist.

```
public abstract void enableReceiveThreshold(int size)
    throws UnsupportedOperationException
public abstract void disableReceiveThreshold()
public abstract boolean isReceiveThresholdEnabled()
public abstract int getReceiveThreshold()
```

Für die anderen Eigenschaften existieren die entsprechenden Methoden.

JAVA COMMUNICATIONS API

Hier die vier Methoden für den Parameter Receive Timeout:

```
public abstract void enableReceiveTimeout(int size)
    throws UnsupportedOperationException
public abstract void disableReceiveTimeout()
public abstract boolean isReceiveTimeoutEnabled()
public abstract int getReceiveTimeout()
```

Die vier Methoden zum Anpassen bzw. Abfragen der Framing Eigenschaft:

```
public abstract void enableReceiveFraming(int size)
    throws UnsupportedOperationException
public abstract void disableReceiveFraming()
public abstract boolean isReceiveFramingEnabled()
public abstract int getReceiveFramingByte()
```

Die vier Methoden zum Anpassen bzw. Abfragen der Puffergrösse:

```
public abstract void setInputBufferSize(int size)
public abstract int getInputBufferSize()
public abstract void setOutputBufferSize(int size)
public abstract int getOutputBufferSize()
```

Alle Treiber müssen Eingabe- und Ausgabe-Puffer unterstützen. Daher gibt es keine Abfrage Methoden, ob der Puffer überhaupt existiert oder nicht.

Immer wenn ein Treiber eine Eigenschaft nicht unterstützt, wird bei der unerlaubten Abfrage oder dem unerlaubten Setzen einer Eigenschaft eine `UnsupportedOperationException` geworfen. Sie können damit natürlich auch Tests aufbauen! Das folgende Beispiel zeigt, wie damit der Port getestet wird:

```
package porttester;

/**
 * Title:          Port Tester
 * Description:    Einfaches Beispiel zum Testen der Ports
 * Copyright:      Copyright (c) 2000
 * Company:        Joller-Voss GmbH
 * @author J.M.Joller
 * @version 1.0
 */

import javax.comm.*;
import java.util.*;

public class PortTester {

    public static void main(String[] args) {
        System.out.println("[PortTester]Start");

        System.out.println("[PortTester]CommPortIdentifiers.getPortIdentifiers()");
        Enumeration thePorts = CommPortIdentifier.getPortIdentifiers();
        while (thePorts.hasMoreElements()) {
            CommPortIdentifier com = (CommPortIdentifier) thePorts.nextElement();
            System.out.print("[PortTester]" + com.getName());
        }
    }
}
```

JAVA COMMUNICATIONS API

```
switch(com.getPortType()) {
    case CommPortIdentifier.PORT_SERIAL:
        System.out.println(", ein serieller Port: ");
        break;
    case CommPortIdentifier.PORT_PARALLEL:
        System.out.println(", ein paralleler Port: ");
        break;
    default:
        // falls weitere Ports dazukommen
        // USB, FireWire oder was auch immer
        System.out.println(" , unbekannter Port-Typ: ");
        break;
}

try {
    System.out.println("[PortTester]Testen der Properties");
    CommPort thePort = com.open("Port Tester", 20);
    testProperties(thePort);
    thePort.close();
}
catch (PortInUseException e) {
    System.out.println("[PortTester]Der Port wird benutzt. Die
Eigenschaften können nicht ermittelt werden");
}
System.out.println("[PortTester]Ende");
}
}

public static void testProperties(CommPort thePort) {

    try {
        thePort.enableReceiveThreshold(10);
        System.out.println("[PortTester]Receive Threshold wird unterstützt");
    }
    catch (UnsupportedCommOperationException e) {
        System.out.println("[PortTester]Receive Threshold wird nicht
unterstützt");
    }

    try {
        thePort.enableReceiveTimeout(10);
        System.out.println("[PortTester]Receive Timeout wird unterstützt");
    }
    catch (UnsupportedCommOperationException e) {
        System.out.println("[PortTester]Receive Threshold wird nicht
unterstützt");
    }

    try {
        thePort.enableReceiveFraming(10);
        System.out.println("[PortTester]Receive Framing wird unterstützt");
    }
    catch (UnsupportedCommOperationException e) {
        System.out.println("[PortTester]Receive Threshold wird nicht
unterstützt");
    }
}
}
```

Die Ausgabe zeigt, dass ich gerade am Drucken war:

```
[PortTester]Start
[PortTester]CommPortIdentifiers.getPortIdentifiers()
[PortTester]COM1, ein serieller Port:
[PortTester]Testen der Properties
[PortTester]Receive Threshold wird unterstützt
[PortTester]Receive Timeout wird unterstützt
[PortTester]Receive Framing wird unterstützt
[PortTester]Ende
[PortTester]COM2, ein serieller Port:
[PortTester]Testen der Properties
[PortTester]Receive Threshold wird unterstützt
[PortTester]Receive Timeout wird unterstützt
[PortTester]Receive Framing wird unterstützt
[PortTester]Ende
[PortTester]LPT1, ein paralleler Port:
[PortTester]Testen der Properties
[PortTester]Receive Threshold wird unterstützt
[PortTester]Receive Timeout wird unterstützt
[PortTester]Receive Framing wird unterstützt
[PortTester]Ende
[PortTester]LPT2, ein paralleler Port:
[PortTester]Testen der Properties
[PortTester]Der Port wird benutzt. Die Eigenschaften können nicht ermittelt werden
[PortTester]Ende
```

1.5. *Serielle Ports*

Die `javax.comm.SerialPort` Klasse ist eine abstrakte Unterklasse von `CommPort`, welche einige Methoden und Konstanten enthält / definiert, die beim Einsatz von RS232 seriellen Ports oder Geräten eingesetzt werden könnten. Ziel der Klasse ist es also, dass der Programmierer serielle Ports anschauen und Parameter setzen oder abfragen kann. Die Oberklasse `CommPort` liefert die einfachen Eingabe- und Ausgabefunktionen. `SerialPort` besitzt einen öffentlichen Konstruktor, den man in Applikationen nicht einsetzen sollte!

Die bessere Technik besteht darin,

- zuerst die `CommPortIdentifier` zu bestimmen
- dann deren `open()` Methode einzusetzen
- und schliesslich das Ergebnis auf einen `SerialPort` zu kassen.

Hier ein Beispiel:

```
CommPortIdentifier cpi = CommPortIdentifier.getPortIdentifier("COM2");
if (cpi.getType() == CommPortIdentifier.PORT_SPECIAL) {
    try {
        SerialPort modem = (SerialPort) cpi.open(); // kassen
    } catch (PortInUseException e) {
        // ...
    }
}
```

Die `SerialPort` Klasse besitzt sehr viele Methoden. Diese lassen sich grob in drei Kategorien einteilen:

- Methoden, welche den Zustand des Ports liefern
- Methoden, welche den Zustand des Ports setzen
- Methoden, welche Zustandsänderungen des Ports beobachten

1.5.1. Kontrollfunktionen

Die Daten können nicht einfach über eine Leitung versandt werden. Man muss sich um jede Menge kümmern, bevor die Daten auch korrekt ankommen. Einer der Gründe liegt in der analog / digital Umwandlung. Daher benötigen wir eine ganze Menge Protokolle und Ebenen, bevor die Kommunikation sauber funktioniert.

Serielle Kommunikation benutzt grundsätzlich einfache Protokolle. Ein Bit entspricht einem Voltstoss von 3 bis 25 Volt über ein serielles Kabel, während einer Zeitdauer, die invers proportional zur Baud Rate ist. -3 bis -25 Volt wird als 0 Bit interpretiert.

Bits werden zu seriellen Dateneinheiten, serial data units, SDAs zusammengefasst. Eine gängige Länge für SDUs ist 8 Bit (für binäre Daten) oder 7 Bit (für ASCII Text). Die meisten Geräte verwenden heute 8 Bit per SDU. Die einzelnen SDUs folgen mit einem Abstand aufeinander.

Eines der Probleme der asynchronen seriellen Kommunikation ist das Bestimmen der SDU Grenzen (Anfang und Ende). Damit diese Aufgabe leichter gelöst werden kann, wird der SDU ein Startbit (0) vorangestellt und am Ende folgt ein oder zwei Stop Bits. Stop Bits dauern immer länger als Datenbits. Damit kann das Ende einer SDU bestimmt werden.

Neben Daten und Start / Stop Bits kann eine SDU auch noch ein Paritätsbit haben, also eine einfache Fehlerprüfung (nicht Korrektur!). Dabei werden zwei Schemas verwendet: gerade und ungerade Parität. Das Paritätsbit wird aufgrund der vorhandenen Bits berechnet. Falls das Paritätsbit nach der Übertragung gleich wie vor der Übertragung ist, kann man davon ausgehen, dass nicht viel schief lief (oder alles).

Die Notation 8N1 bedeutet, dass man 8Bit no parity und ein Stopbit verwendet.
Die Notation 7E1 besagt: 7 Bit SDUs, even Parity, 1 Stopbit.
Üblich ist 8N1!

Unter der Baus Rate versteht man die Anzahl Zustandswechsel pro Sekunde. Baud und Bit / sec sind nicht identisch! Moderne Modems können mehrere Bits pro Sekunde senden.

Das Java COMM API gestattet es Ihnen all diese Parameter zu setzen, inklusive Baudrate, Datenbits, Stopbits und Parität. Die folgenden vier Methoden der `SerialPort` Klasse liefern die Werte der Einstellungen:

```
public abstract int getBausRate()  
public abstract int getDataBits()  
public abstract int getStopBits()  
public abstract int getParity()
```

Setzen können Sie die Werte nur alle zusammen:

```
public abstract void setSerialPortParams(int baud, int dataBits,  
                                         int stopBits, int parity)  
    throws UnsupportedOperationException
```

JAVA COMMUNICATIONS API

Falls der Treiber den entsprechenden Wert nicht unterstützt, wird eine `UnsupportedCommOperationException` geworfen.

Folgende Werte sind erlaubt (die Baudrate ist allgemeiner definierbar):

```
SerialPort.DATABITS_5 // 5 Datenbits pro Byte
SerialPort.DATABITS_6 // 6 Datenbits pro Byte
SerialPort.DATABITS_7 // 7 Datenbits pro Byte
SerialPort.DATABITS_8 // 8 Datenbits pro Byte
SerialPort.STOPBITS_1 // 1 Stopbit
SerialPort.STOPBITS_2 // 2 Stopbits
SerialPort.STOPBITS_1_5 // 1.5 Stopbits (nicht darüber nachdenken!)
SerialPort.PARITY_NONE // keine Parität
SerialPort.PARITY_ODD // ungerade Parität
SerialPort.PARITY_EVEN // gerade Parität
```