technische universität
dortmund

Fakultät für Informatik
Lehrstuhl 4

Egor Kudrjaschow, Henning Brümmer, Andreas Blume
Nils Dunker, Felix Homa, Benedikt Maus, Dennis Ziebart, Bastian Korte

Wintersemester 2019/20

Rechnernetze und verteilte Systeme

Hilfsdokument

Inhalt

1 Vorwort

Dieses Dokument enthält eine Ansammlung von Hilfestellungen für die Programmieraufgabe. Dieses Dokument wird jedes Jahr mit weiteren Hilfestellungen erweitert, entsprechend sind nicht alle Hilfestellungen in diesem Dokument für die diesjährige Programmieraufgabe von Relevanz. Dieses Dokument wird ggf. während der Bearbeitungszeit aktualisiert.

Letzte Änderung: 2019-12-09 10:10:27

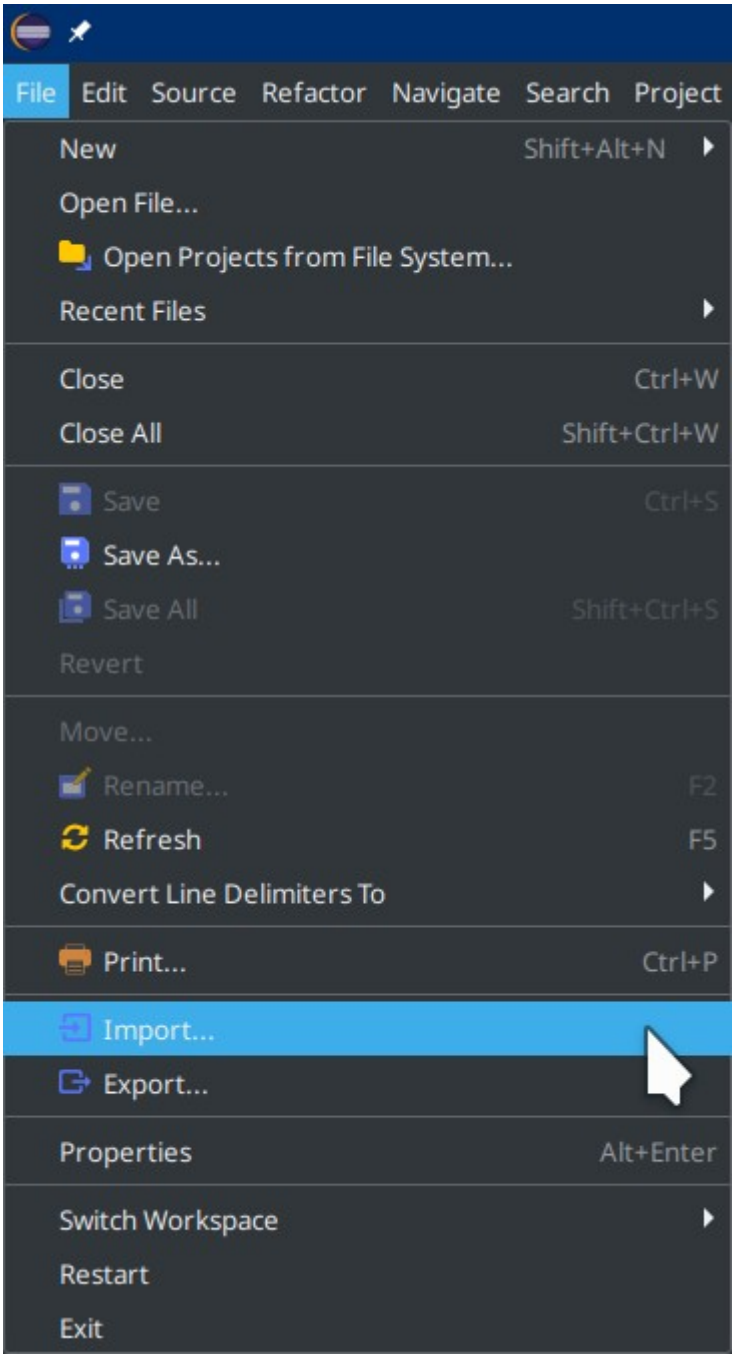
2 Entwicklungsumgebung wählen

2.1 Eclipse

Projekt importieren:

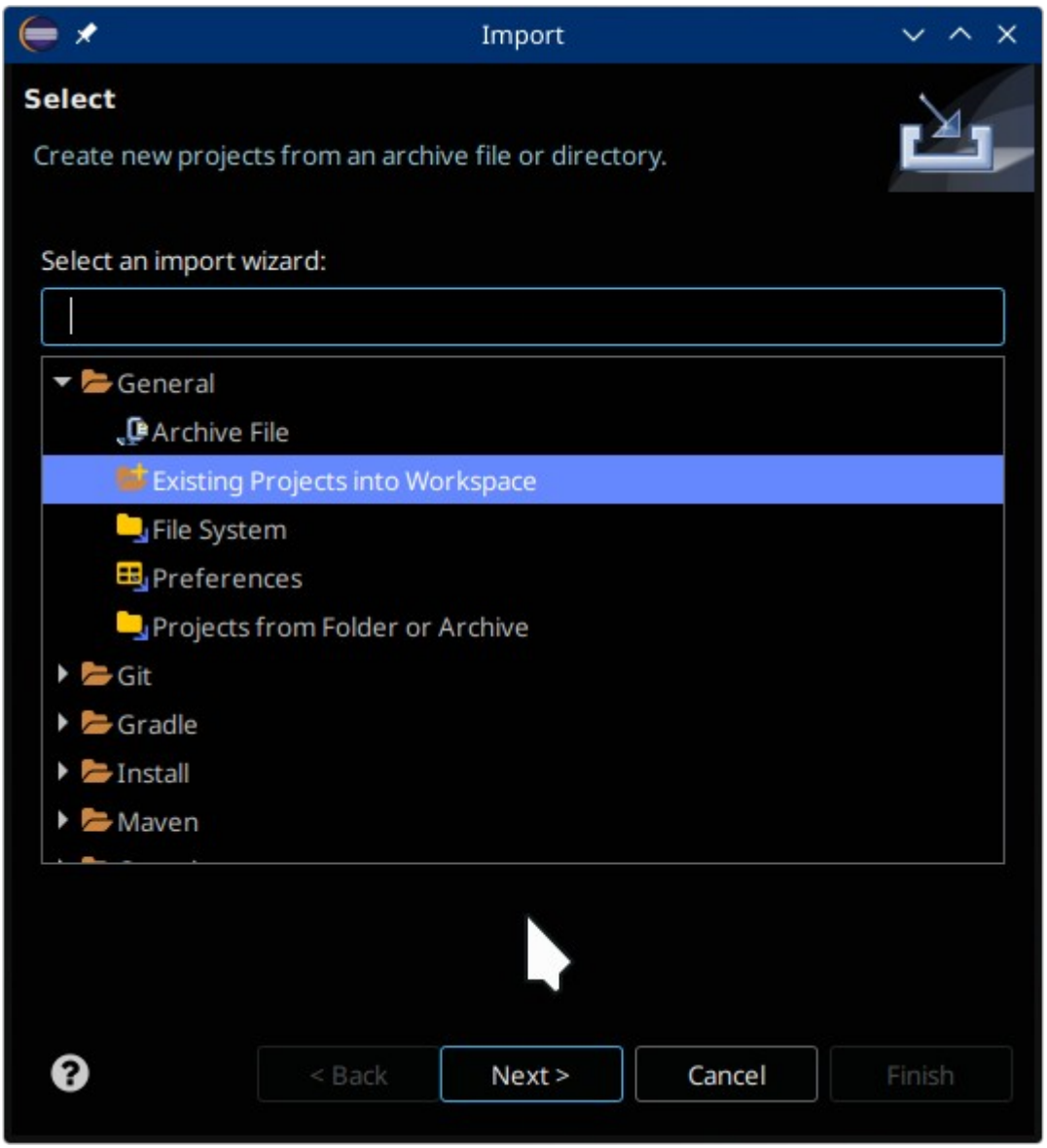
- Menu „File“ -> „Import“ (EC 1)
- „General“ -> „Existing Projects into workspace“ (EC 2)
- Entweder der Ordner der entpackten Vorgabe wählen, oder das Archiv wählen (EC 3)
- Das Projekt anhaken und mit Finish bestätigen (EC 3)

Egor Kudrjaschow, Henning Brümmer, Andreas Blume
Nils Dunker, Felix Homa, Benedikt Maus, Dennis Ziebart, Bastian Korte



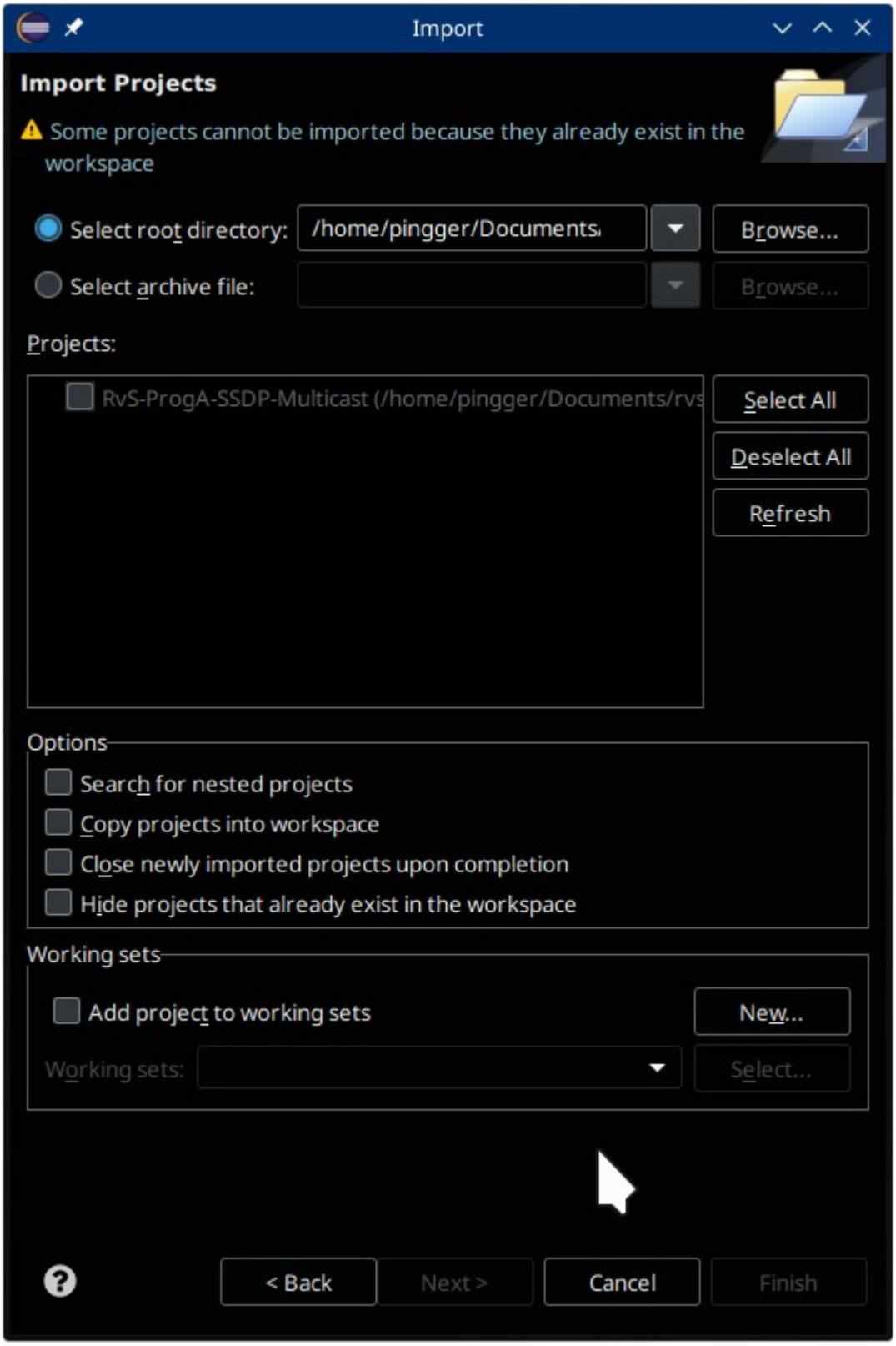
EC 1

Egor Kudrjaschow, Henning Brümmer, Andreas Blume
Nils Dunker, Felix Homa, Benedikt Maus, Dennis Ziebart, Bastian Korte



EC 2

Egor Kudrjaschow, Henning Brümmer, Andreas Blume
Nils Dunker, Felix Homa, Benedikt Maus, Dennis Ziebart, Bastian Korte



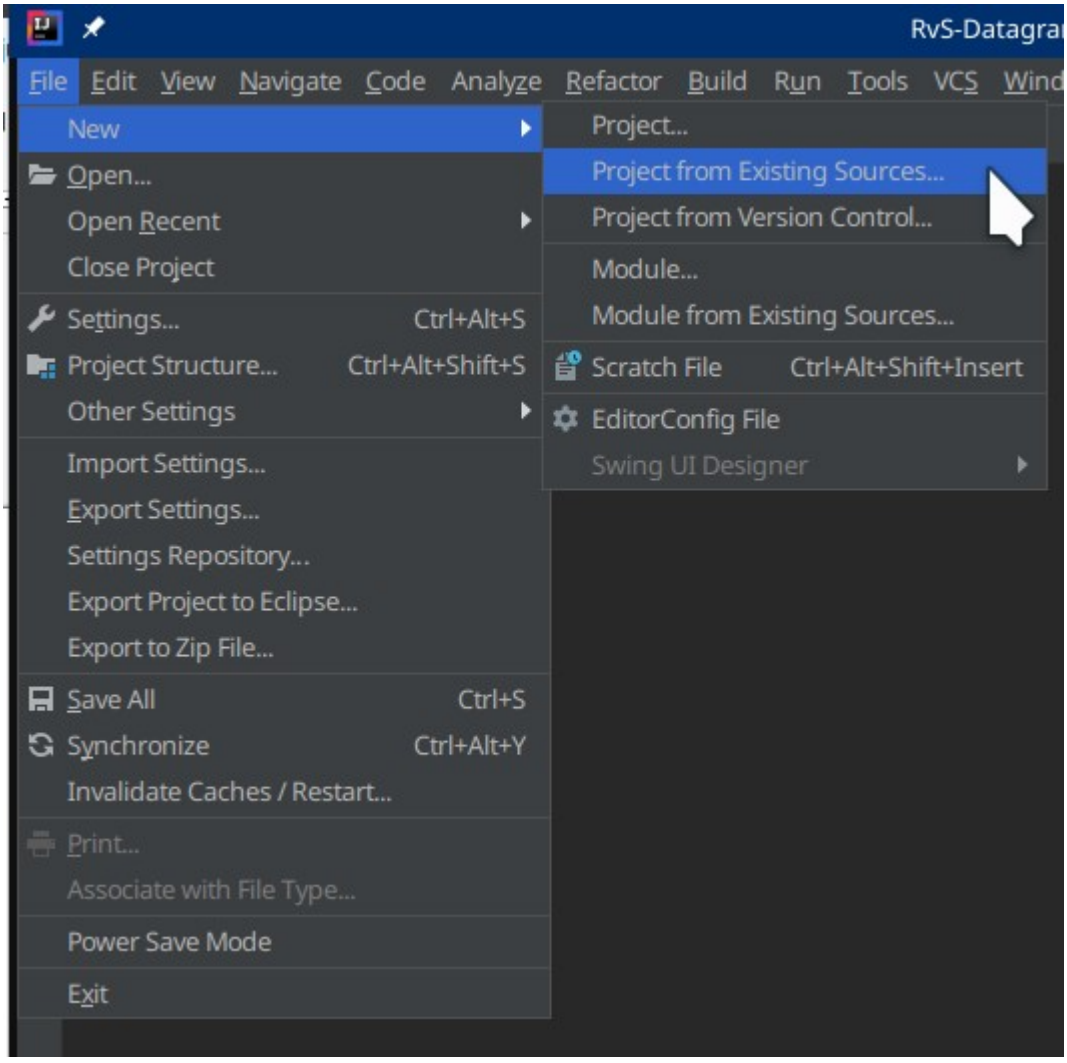
EC 3

Egor Kudrjaschow, Henning Brümmer, Andreas Blume
Nils Dunker, Felix Homa, Benedikt Maus, Dennis Ziebart, Bastian Korte

2.2 IntelliJ IDEA

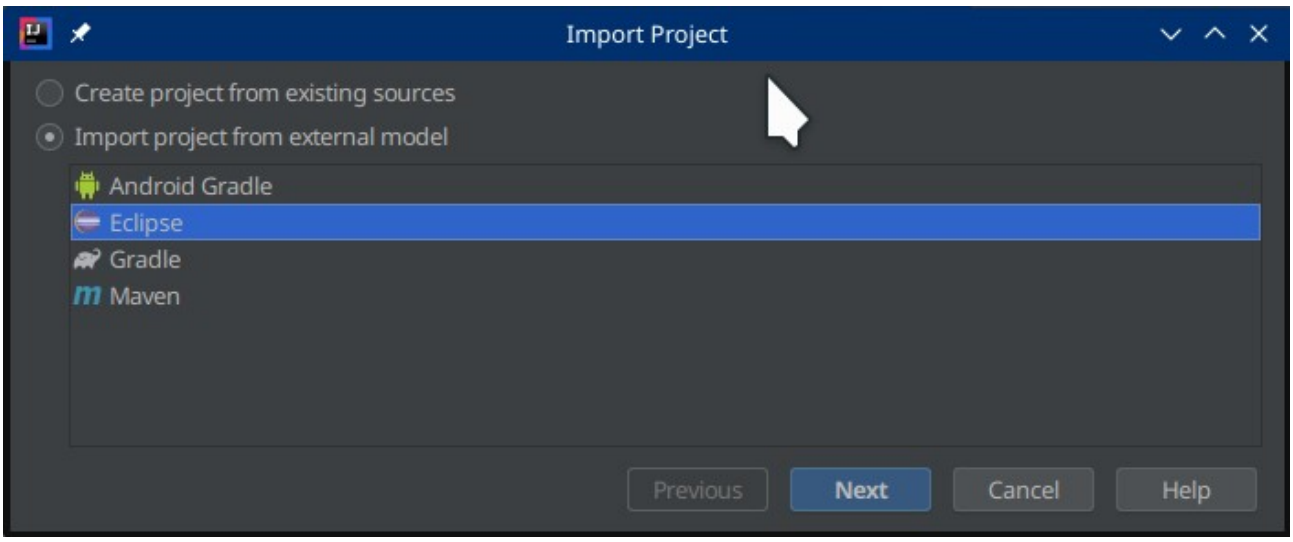
Projekt importieren:

- Menu „File“ -> „New“ -> „Project from Existing Sources“ (IJ 1)
- Den Ordner der entpackten Vorgabe auswählen.
- Als Eclipse Projekt importieren (IJ 2)
- Wie IJ 3 fortfahren
- Fortfahren bis zur Auswahl der Java Laufzeitumgebung
- Dort Java 11 oder aktueller auswählen. (Sollte dies nicht zur Verfügung stehen, so muss eine entsprechende Java Umgebung bei Oracle oder unter java.com herruntergeladen werden)

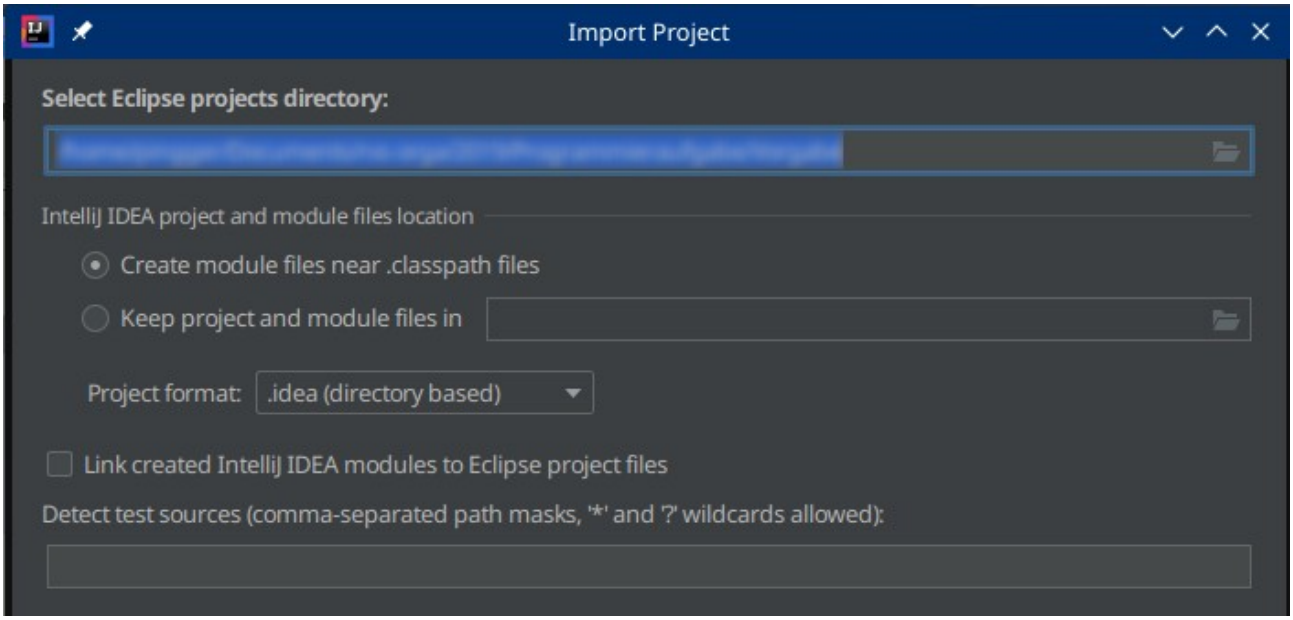


IJ 1

Egor Kudrjaschow, Henning Brümmer, Andreas Blume
Nils Dunker, Felix Homa, Benedikt Maus, Dennis Ziebart, Bastian Korte



IJ 2



IJ 3

Egor Kudrjaschow, Henning Brümmer, Andreas Blume
Nils Dunker, Felix Homa, Benedikt Maus, Dennis Ziebart, Bastian Korte

3 Implementationsdemos

DatagramSocket

DatagramSockets[1] sind Sockets die auf UDP basieren und entsprchend mit Datagrammen arbeiten. Hierfür muss eine DatagramSocket als erstes auf einen Port gebunden werden (über den Parameter im Konstruktor). Danach können über die **receive**-Methode und die **send**-Methode Paket empfangen und versendet werden. In beiden Fällen muss ein **DatagramPacket** erstellt werden, in das entweder die empfangenen Daten geschrieben werden müssen, bzw. dessen Inhalt versendet wird.

MulticastSocket

MulticastSockets[2] sind nahezu identisch zu DatagramSockets. Zusätzlich können MulticastSockets über die **joinGroup(...)**- und **leaveGroup(...)**-Methoden Multicast-Gruppen beitreten und verlassen. Unter dem angegeben Link (Fußnote 2) ist auch ein Implementierungsbeispiel zu finden.

Byte-Daten als String lesen

Um ein **byte**-Array als String zu lesen gibt es mehrere Möglichkeiten. Hier werden 2 Möglichkeiten vorgestellt.

Möglichkeit 1:

```
String text = new String(byteArray, StandardCharsets.UTF_8);
```

Hier wird lediglich der Inhalt des gesamten Arrays genommen und als UTF-8 kodierter String gelesen.

Möglichkeit 2:

Das Array wie einen Stream lesen. Hierfür bietet sich der **ByteArrayInputStream**[3] an. Dieser kann dann wie in **5.3** gelesen werden.

```
InputStream inputStream = new ByteArrayInputStream(byteArray);
[Code von 5.3]
inputStream.close();
```

Command Line Interface

Als **command line interface** bezeichnet man Anwendungen, die ohne GUI funktionieren. In Java nutzt man dafür die Streams **System.in**, **System.out** und **System.err**. Hierbei sind System.out und System.err jeweils PrintStreams[4] und System.in ein einfacher InputStream. Der System.in kann also

1 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/net/DatagramSocket.html
2 https://cr.openjdk.java.net/~iris/se/11/latestSpec/api/java.base/java/net/MulticastSocket.html
3 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/ByteArrayInputStream.html
4 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/PrintStream.html

gelesen werden, wie unter „Stream-Daten als String lesen“ erklärt wird. Aus den PrintStreams, wie unter „PrintStream“ erklärt wird.

JavaDoc

JavaDoc ist die von Java eingebaute Dokumentierungshilfe. An jedes Objekt, dass einen Sichbarkeitsmodifikator (public, private, ...) erhalten kann, kann ein JavaDoc-Kommentar hinzugefügt werden. Beispiel an einer Methode:

```
/**
 * Methode zum Demonstrieren von JavaDoc.
 *
 * @param demo Parameter zum demonstrieren von Parameter dokumentation
 * @throws DemoException Tritt ein wenn die Demonstration fehlschlägt.
 * @return einen Wert
 * @author Einhorn
 */
public int demoMethode(int demo) throws DemoException
{
    ...
}
```

Es gilt, dass ein JavaDoc Block mit '/**' beginnt. Als erstes wird die Erklärung eingetragen (darf mehrzeilig sein). Danach sollen die Parameter **@…** gegeben werden und ebenfalls dokumentiert werden. Sollte der JavaDoc Kommentar nicht ausreichen um eine Methode vollständig zu erklären, so können innerhalb des Codes weitere Kommentare mit '//' eingefügt werden.

PrintStream

PrintStreams[5] sind eine einfache Möglichkeit um Text-Daten in einen Stream zu schreiben. PrintStreams müssen, abgesehen von den System-Streams, erst über den Konstruktor geöffnet werden. Danach können z.B. über **print(String)** und **println(String)** Text in den darunter liegenden Stream geschrieben werden (letzteres fügt einen **platformabhängigen** (Unix \n, Windows \r\n) Zeilenumbruch ein). Bevor direkt in den darunter liegenden Stream geschrieben wird, MUSS **flush()** auf dem PrintStream aufgerufen werden. Nach der Benutzung muss der PrintStream mit **close()** geschlossen werden. **System-Streams** dürfen nicht geschlossen werden! Dies wird durch Java übernommen.

Stream-Daten als String lesen

Um möglichst einfach Stream-Daten als String lesen zu können, bietet es sich an einen **InputStreamReader**[6] in Kombination mit einem **BufferedReader**[7] zu nutzen. Dieser muss mit einem gegebenen **InputStream** initialisiert werden. Zusätzlich kann noch ein Zeichensatz oder

5 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/PrintStream.html
6 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStreamReader.html
7 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/BufferedReader.html

Zeichensatz-Decoder übergeben werden. Überlicherweise wird UTF-8 verwendet, dies kann in der Klasse **StandardCharsets**[8] gefunden werden. Das entsprechende Attribut lautet dort **UTF_8**. Beispiel:

```
InputStreamReader streamReader = new InputStreamReader(„inputStream“,
                                 StandardCharsets.UTF_8);
BufferedReader reader = new BufferedReader(streamReader);
String line = reader.readLine(); // Liest eine Zeile ohne Zeilenumbruch
[...]
reader.close(); // Schließt automatisch auch den streamReader
```

Zu beachten ist, dass die **readLine**-Methode ggf. auch **null** zurück gibt. Dies passiert, wenn kein Zeilenende innerhalb der gegeben Puffergröße gefunden werden kann, oder das Ende des Streams erreicht wurde. Die Methode **ready** gibt Auskunft darüber, ob aktuell eine vollständige Zeile gelesen werden kann. Im Rahmen der Programmieraufgabe sollte die Standard-Puffergröße ausreichend sein. Sollte dies nicht der Fall sein, lässt diese sich im Konstruktor des BufferedReaders als zweiter Parameter vom Typ **int** (die Anzahl der Bytes, die der Puffer groß sein soll) mit angeben.

String als Bytes

Auch hier gibt es mehrere Möglichkeiten. Für einzelne Textzeilen bietet sich die **getBytes(Charset)**-Methode der **String**-Klasse an. Auch hier sollte UTF-8 genutzt werden.

Für mehrzeiligen Text empfiehlt es sich entweder **StringBuilder**[9] oder einen **PrintStream**[10] in Kombination mit einem **ByteArrayOutputStream**[11] zu benutzen.

StringBuilder:

```
StringBuilder builder = new StringBuilder();
builder.append("Erste Zeile"+"\r\n"); // \r\n ist der Zeilenumbruch
builder.append("Zweite Zeile"+"\r\n");
[...]
byte[] data = builder.toString().toBytes(StandardCharsets.UTF_8);
```

PrintStream:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
PrintStream ps = new PrintStream(baos, true, StandardCharsets.UTF_8);
ps.println("Erste Zeile");
ps.println("Zweite Zeile");
[...]
ps.close();
byte[] data = baos.toByteArray();
```

8 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/nio/charset/StandardCharsets.html
9 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/StringBuilder.html
10 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/PrintStream.html
11 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/ByteArrayOutputStream.html

Zu Beachten ist hierbei, das println einen Zeilenumbruch nach dem lokalen System-Standard erzeugt. Für SSDP ist jedoch explizit der Zeilenumbruch „\r\n“ verlangt, daher sollte stattdessen die print-Methode wie folgt genutzt werden:

```
ps.print("Text Zeile" + "\r\n");
```

Die Konsolen-Ausgabe in Java erfolgt ebenfalls über einen **PrintStream**. Dieser ist unter **System.out** zu finden. Dieser sollte niemals geschlossen werden, da in diesem Fall die Java VM dies von selbst beim Programmende durchführt.

Threads

Threads[12] erlauben es mehrere Programmabläufe innerhalb des selben Prozesses durchzuführen.

Für Ressourcen (z.B. globale Variablen und Methoden) die von mehreren Threads benutzt werden und die Änderungen an Datenstrukturen durchführen muss eine Synchronisierung des Zugriffs durchgeführt werden, siehe **Threadsynchroniserung**.

Die gängige Methode Threads zu erstellen, ist eine Klasse das Interface **Runnable** implementieren zu lassen und dann die Methode **public void run()** zu implementiern.

```
public class MyWorkerClass implements Runnable{
	@Override
	public void run() {
		[...] // Code der im Thread ausgeführt werden soll
	}
}
```

Den Thread startet man nun, in dem man die Klasse instanziert, einen neuen Thread erstellt, ihm das Objekt zuweist und startet, also:

```
MyWorkerClass blupp = new MyWorkerClass();
Thread myWorkerThread = new Thread(blupp);
myWorkerThread.setName("Name des Threads"); // Optional
myWorkerThread.start();
```

Zu beachten ist, dass die **start**-Methode aufgerufen wird, jedoch die **run**-Methode mit Code gefüllt wird. Zu dem macht das Benennen des Threads das Debuggen einfacher, da der Standard Titel „Thread-1“ (wobei 1 dynamisch ist), nicht besonders aussagekräftig ist.

Threadsynchronisierung

Wenn Objekte aus mehreren Threads möglicherweise gleichzeitig benutzt werden könnten und diese Objekte nicht *Threadsafe* sind, so muss die Synchronisierung selbst durchgeführt werden.

Es gibt *ThreadSafe* Datenstrukturen in Java, die meisten Datenstrukturen sind dies jedoch nicht. Java stellt hierfür den **synchronized**-Befehl zur Verfügung. Beispiel:

12 https://cr.openjdk.java.net/~iris/se/11/latestSpec/api/java.base/java/lang/Thread.html

Egor Kudrjaschow, Henning Brümmer, Andreas Blume
Nils Dunker, Felix Homa, Benedikt Maus, Dennis Ziebart, Bastian Korte


```
public class XY {
      [...]
      public void myMethod() {
            [...]
            synchronized(myList) {
                  myList.add(blih);
                  myList.add(blah);
            }
            [...]
      }
}
```

In diesem Beispiel wird erst der Zugriff **myList** gesichert, danach Änderungen an der Liste durchgeführt und die Liste danach (durch schließen des **synchronized**-Blocks) wieder freigegeben. Wenn ein weiterer Thread in einen **synchronized(myList)** Block möchte, wartet dieser bis **myList** wieder freigegeben wurde.