Java TCP programming

https://github.com/heig-vd-dai-course

Web · PDF

L. Delafontaine and H. Louis, with the help of GitHub Copilot.

Based on the original course by O. Liechti and J. Ehrensberger.

Objectives

- Program your own TCP client/server applications in Java with the Socket API
- Understand how to process data from streams
- Make usage of a REPL

Your applications will be able to communicate over the network!

Explore the code examples

More details for this section in the course material. You can find other resources and alternatives as well.

Explore the code examples

Individually, or in pair/group, **take 10 minutes to explore and discuss the code examples**.

Answer the questions available in the course material:

- How do the code examples work?
- What are the main takeaways of the code examples?
- What are the main differences between the code examples?

If needed, use the theoretical content to help you.

TCP

More details for this section in the course material. You can find other resources and alternatives as well.

TCP

TCP is a transport protocol that is similar to a phone call:

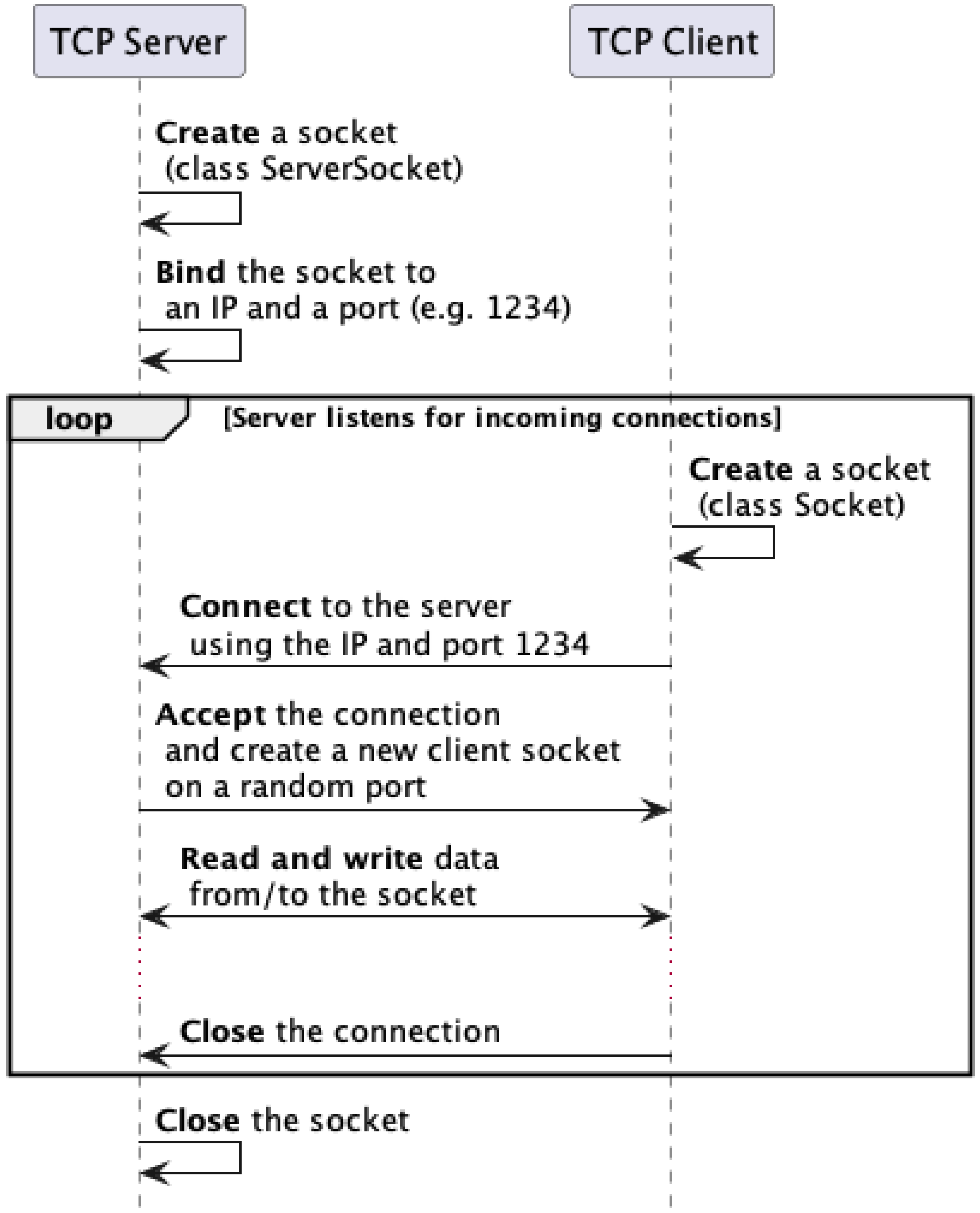
1. A connection is established between two parties (unicast)
2. Data sent is guaranteed to arrive in the same order
3. Data can be sent again if needed (retransmission)

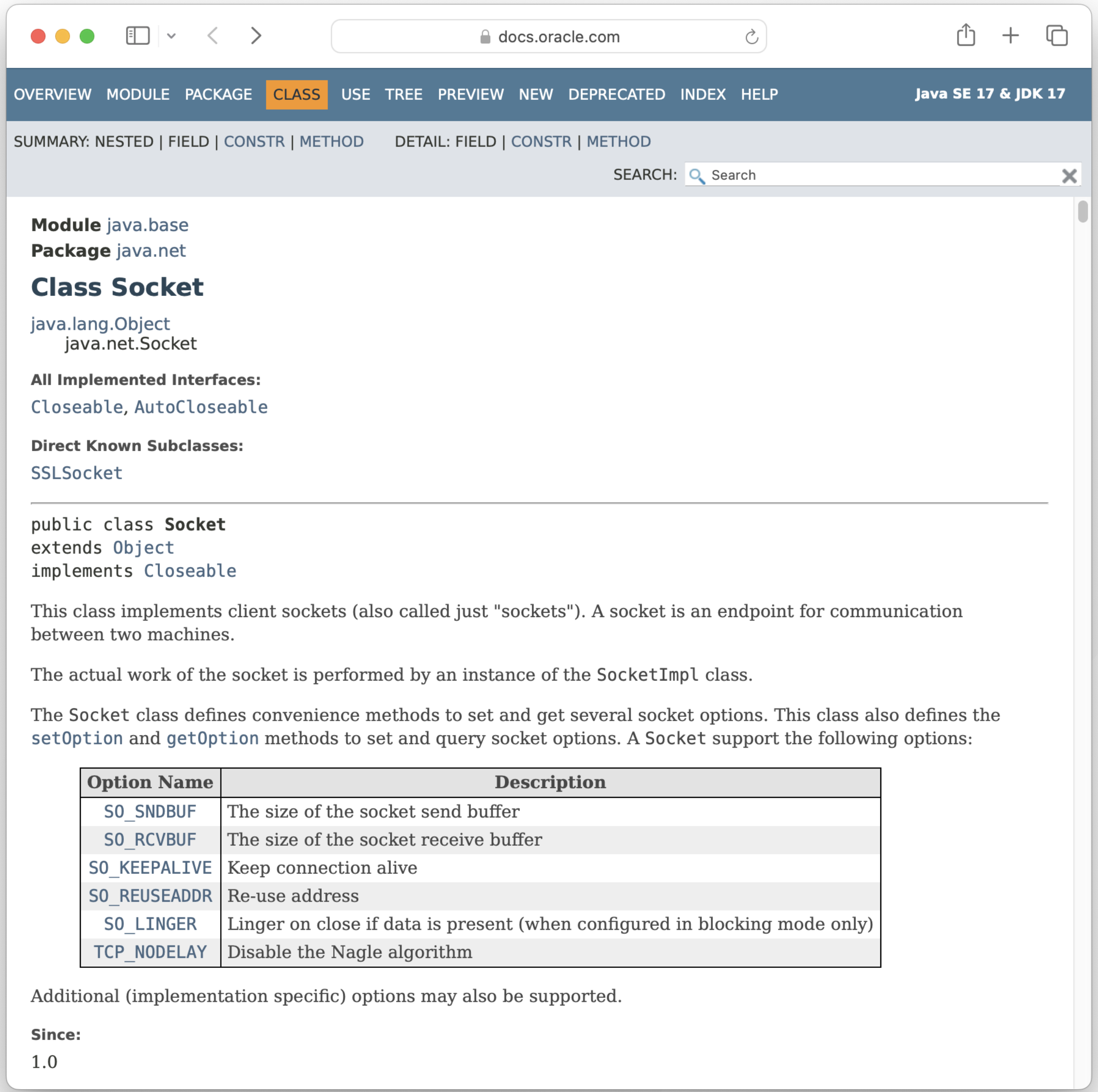
The Socket API

More details for this section in the course material. You can find other resources and alternatives as well.

The Socket API

- Originally developed by Berkeley University
- Ported to Java and many other languages
- Provides a simple API to use TCP and UDP
- A socket is a connection between two parties using a protocol and a port


TCP Server
TCP Client
Create a socket (class ServerSocket)
Bind the socket to an IP and a port (e.g. 1234)
loop
[Server listens for incoming connections]
Create a socket (class Socket)
Connect to the server using the IP and port 1234
Accept the connection and create a new client socket on a random port
Read and write data from/to the socket
Close the connection
Close the socket


docs.oracle.com

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP Java SE 17 & JDK 17

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

SEARCH: Search

Module java.base
Package java.net

Class Socket

java.lang.Object
    java.net.Socket

All Implemented Interfaces:
Closeable, AutoCloseable

Direct Known Subclasses:
SSLSocket

```
public class Socket
extends Object
implements Closeable
```

This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.

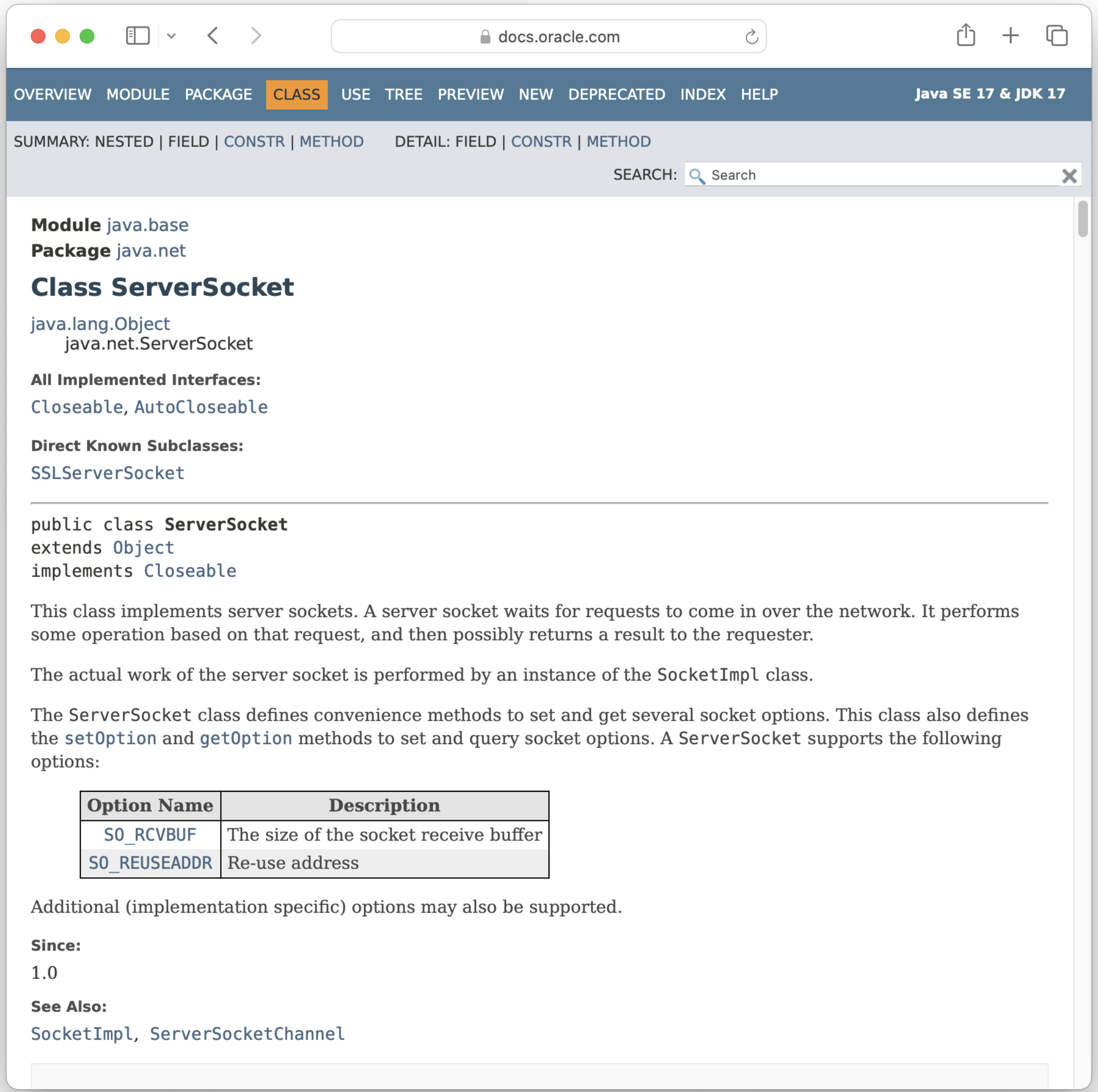
The actual work of the socket is performed by an instance of the `SocketImpl` class.

The `Socket` class defines convenience methods to set and get several socket options. This class also defines the `setOption` and `getOption` methods to set and query socket options. A `Socket` support the following options:

Option Name	Description
SO_SNDBUF	The size of the socket send buffer
SO_RCVBUF	The size of the socket receive buffer
SO_KEEPALIVE	Keep connection alive
SO_REUSEADDR	Re-use address
SO_LINGER	Linger on close if data is present (when configured in blocking mode only)
TCP_NODELAY	Disable the Nagle algorithm

Additional (implementation specific) options may also be supported.

Since:
1.0


docs.oracle.com

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP Java SE 17 & JDK 17

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

SEARCH: Search

Module java.base
Package java.net

Class ServerSocket

java.lang.Object
    java.net.ServerSocket

All Implemented Interfaces:
Closeable, AutoCloseable

Direct Known Subclasses:
SSLServerSocket

```
public class ServerSocket
extends Object
implements Closeable
```

This class implements server sockets. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester.

The actual work of the server socket is performed by an instance of the `SocketImpl` class.

The `ServerSocket` class defines convenience methods to set and get several socket options. This class also defines the `setOption` and `getOption` methods to set and query socket options. A `ServerSocket` supports the following options:

Option Name	Description
SO_RCVBUF	The size of the socket receive buffer
SO_REUSEADDR	Re-use address

Additional (implementation specific) options may also be supported.

Since:
1.0

See Also:
SocketImpl, ServerSocketChannel

Client/server common functions

Operation	Description
`socket()`	Creates a new socket
`getInputStream()`	Gets the input stream of a socket
`getOutputStream()`	Gets the output stream of a socket
`close()`	Closes a socket

Client structure and functions

1. Create a `Socket`
2. Connect the socket to an IP address and a port number
3. Read and write data from/to the socket
4. Flush and close the socket

Operation	Description
`connect()`	Connects a socket to an IP address and a port number

Server structure and functions

1. Create a `ServerSocket`
2. Bind the socket to an IP address and a port number
3. Listen for incoming connections
4. Loop
    1. Accept an incoming connection - creates a new `Socket` on a port
    2. Read and write data from/to the socket
    3. Flush and close the socket
5. Close the `ServerSocket`

Operation	Description
`bind()`	Binds a socket to an IP address and a port number
`listen()`	Listens for incoming connections
`accept()`	Accepts an incoming connection

To make it simple, a socket is just like a file that you can open, read from, write to and close. To exchange data, sockets on both sides must be connected.

Processing data from streams

More details for this section in the course material. You can find other resources and alternatives as well.

Processing data from streams

- Sockets use data streams to send and receive data, just like files
- Get an input stream to read data from a socket
- Get an output stream to write data to a socket

Get the streams from a socket:

```java
// Get input stream
input = socket.getInputStream();

// Get output stream
output = socket.getOutputStream();
```

Read and write data from/to the streams as text:

```java
// Get input stream as text
input = new InputStreamReader(socket.getInputStream(), StandardCharsets.UTF_8);

// Get output stream as text
output = new OutputStreamWriter(socket.getOutputStream(), StandardCharsets.UTF_8);
```

Improve performance with a buffer (with a binary stream):

```java
// Get input stream as binary with buffer
input = new BufferedReader(new InputStreamReader(socket.getInputStream());

// Get output stream as binary with buffer
output = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream());
```

Variable length data

Data sent can have a variable length. Manage this using one of the two methods:

- Use a delimiter
- Communicate a fixed length

This must be defined by your application protocol!

Using a delimiter:

```java
// End of transmission character
String EOT = "\u0004";

// Read data until the delimiter is found
String line;
while ((line = in.readLine()) != null && !line.equals(EOT)) {
  System.out.println(
    "[Server " + SERVER_ID + "] received data from client: " + line
  );
}
```

Communicating a fixed length:

```java
// Send the length of the data
out.write("DATA_LENGTH " + data.length() + "\n");

// Send the data
out.write(data);
```

```java
// Read the length of the data
String[] parts = in.readLine().split(" ");
int dataLength = Integer.parseInt(parts[1]);

// Read the data
for (int i = 0; i < dataLength; i++) {
  System.out.print((char) in.read());
}
```

Read-Eval-Print Loop (REPL)

More details for this section in the course material. You can find other resources and alternatives as well.

Read-Eval-Print Loop (REPL)

A REPL is a concept that allows you to interact with a program by sending commands to it without restarting it.

In networking, it is used to send commands to a server that will read the commands, evaluate them, and send back the result.

All of this is done in a loop until the client or the server decides to close the connection.

Useful to keep a connection open and send multiple commands!

Questions

Do you have any questions?

Practical content

What will you do?

- Update your application protocol with the new knowledge you gained
- Execute the code examples and run multiple clients at the same time
- Explore the Java TCP programming template
- Implement and Dockerize the *"Guess the number"* game (optional)

Find the practical content

You can find the practical content for this chapter on GitHub.

Finished? Was it easy? Was it hard?

Can you let us know what was easy and what was difficult for you during this chapter?

This will help us to improve the course and adapt the content to your needs. If we notice some difficulties, we will come back to you to help you.

➡️ GitHub Discussions

You can use reactions to express your opinion on a comment!

What will you do next?

In the next chapter, you will learn the following topics:

- Java UDP programming
  - How does it compare to TCP?
  - How to create efficient UDP network applications
  - Implement the *"Temperature monitoring"* application using UDP (optional)

Sources

- Main illustration by Carl Nenzen Loven on Unsplash
- Illustration by Aline de Nadai on Unsplash
- Illustration by Alexander Andrews on Unsplash
- Illustration by Oleksandra Bardash on Unsplash
- Illustration by patricia serna on Unsplash