Java™ Platform, Micro Edition

Part 7

Generic Connection Framework, HTTP and Sockets

Disclaimer

- **These slides are provided free of charge at http://www.symbianresources.com and are used during Java ME courses at the University of Applied Sciences in Hagenberg, Austria at the Mobile Computing department (http://www.fh-ooe.at/mc)**
- **Respecting the copyright laws, you are allowed to use them:**
  - **for your own, personal, non-commercial use**
  - **in the academic environment**
- **In all other cases (e.g. for commercial training), please contact andreas.jakl@fh-hagenberg.at**
- The correctness of the contents of these materials cannot be guaranteed. Andreas Jakl is not liable for incorrect information or damage that may arise from using the materials.
- This document contains copyright materials which are proprietary to Sun or various mobile device manufacturers, including Nokia, SonyEricsson and Motorola. Sun, Sun Microsystems, the Sun Logo and the Java™ Platform, Micro Edition are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Contents

- Generic Connection Framework
- HTTP Connections
- Sockets

Generic Connection Framework

- **Java SE:** 150+ classes and interfaces (java.io, java.net)
  - **Bytecode-size:** java.net 200kB+
  - **Consistency:** Different objects for protocols
- **CLDC: Generic Connection Framework** (GCF)
  - Unified API for **various protocols**
  - **Profiles define protocols** (MIDP, additional packages)
    - **Mandatory for MIDP 2.0**:
      Hypertext Transfer Protocol (HTTP)
      Hypertext Transfer Protocol over TLS/SSL (HTTPS)
- **Today:** GCF made its way into Java SE (JSR 197)

Generic Connection Framework

- **Hierarchy of interfaces and classes** to:
  - create connections (HTTP, datagrams, ...)
  - perform I/O
- GCF used by **optional packages**
  - Bluetooth, Files, SmartCards, Messaging, …
- … might look difficult at first, but is very easy to use!

Connector

- Single **Factory-class**, creates any type of connection

```
Connector.open("protocol:address;parameters");
```

- **Examples:**
  - Connector.open("http://www.mopius.com/");
  - Connector.open("socket://realreplay.mopius.com:5412");
  - Connector.open("file://data.txt");

Connector – Protocols

- Protocol is chosen automatically, based on URL
- **GCF searches** for class that implements **protocol**
  - **Success:** Returns class that implements Connection-interface
  - **Failure:** ConnectionNotFoundException
- HTTP is supported for sure, Sockets might be unavailable on phone / through operator

Example:

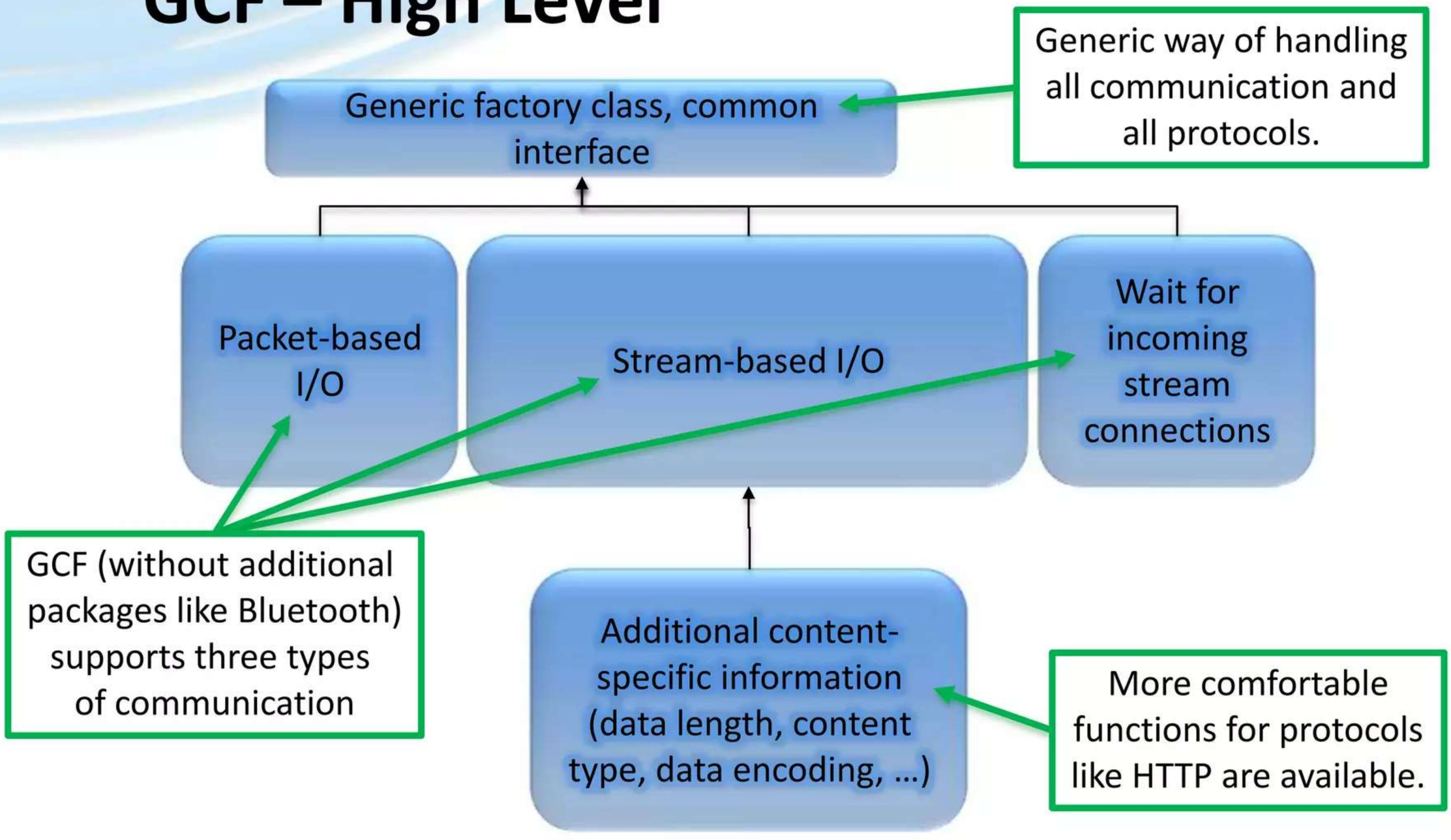
```
HttpConnection con = (HttpConnection) Connector.open (“http://www.mopius.com/”);
```

Connect and Transfer

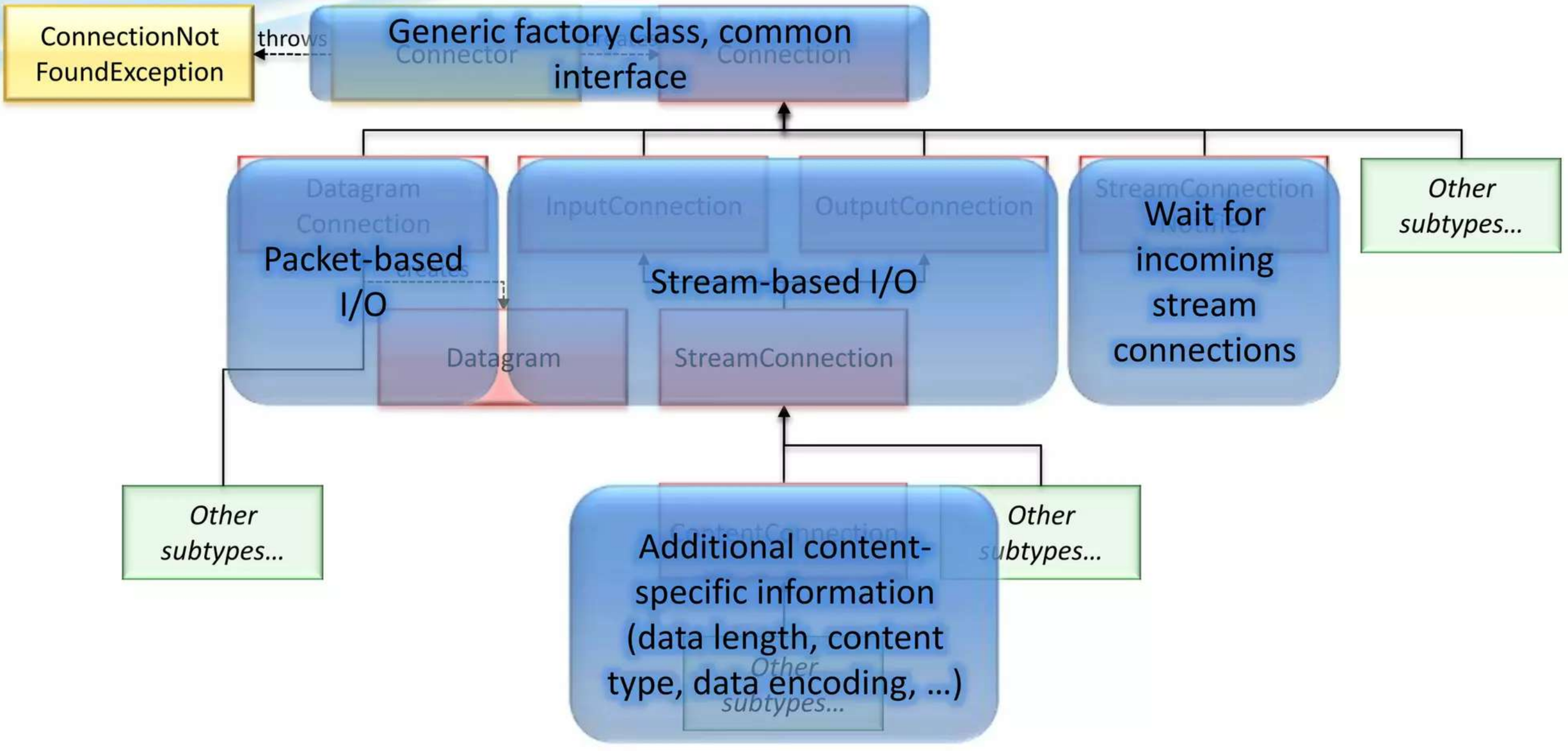
- Short example for sockets (omits error handling):

```
// Open the connection to a URL
SocketConnection con = (SocketConnection) Connector.open("socket://127.0.0.1:5000");
// Send request
DataOutputStream dos = con.openDataOutputStream();
dos.writeShort(12);
dos.writeUTF("Hello World");
dos.close();
// Receive answer
DataInputStream dis = con.openDataInputStream();
short msgId = dis.readShort();
dis.close();
// Cleanup
con.close();
```

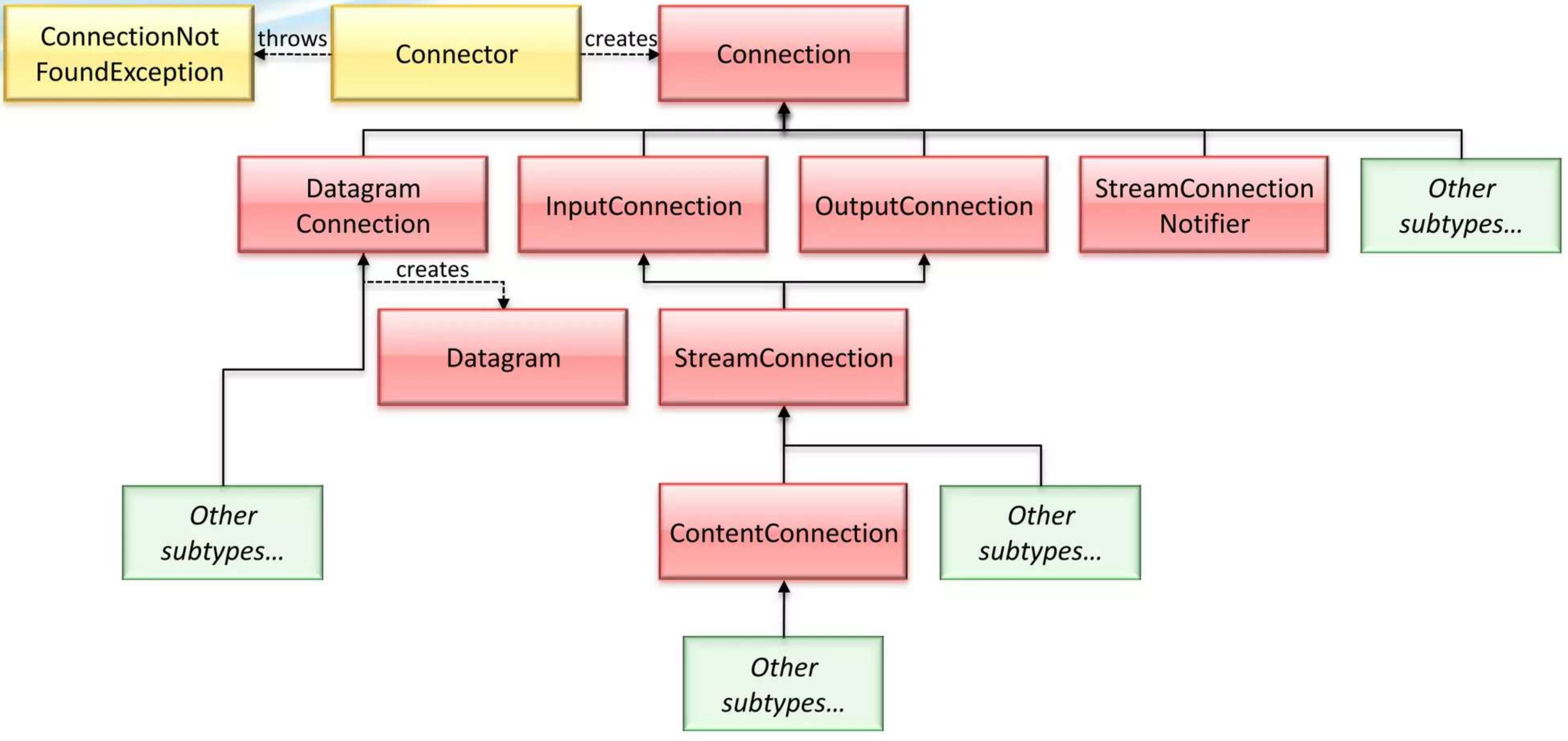
GCF – High Level



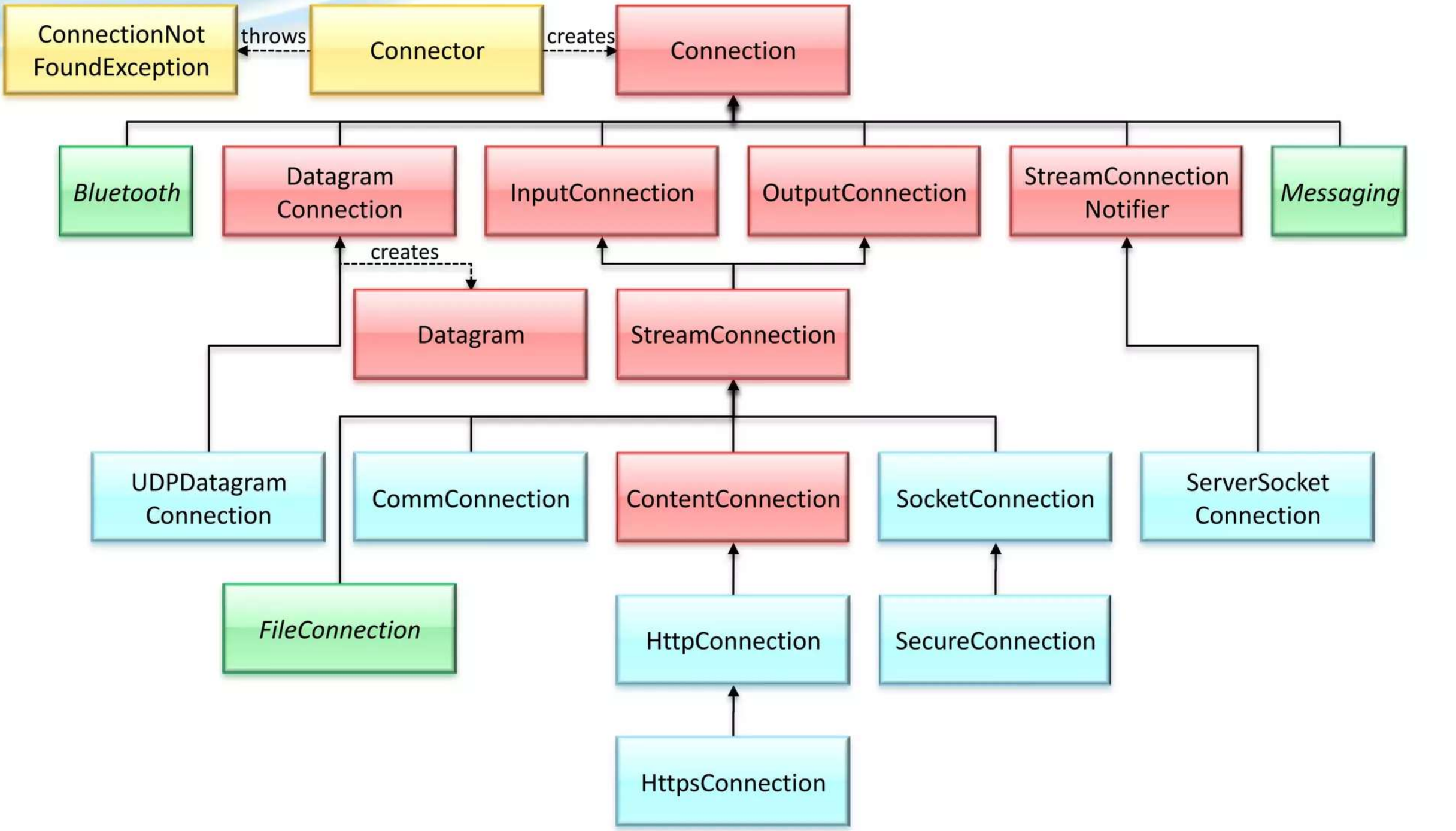
GCF – Basic Class Diagram



GCF – Basic Class Diagram



GCF – MIDP Class Diagram



MIDP 2.0 | CLDC | *Optional JSRs*

URL

- Uniform Resource Locator (URL)
- Identify connection type and endpoint

scheme://user:password@host:port/url-path;parameters

URL-part	Description
scheme	Access method / protocol = connection type to use. e.g. HTTP, FTP, …
user	User name (optional)
password	Password (optional)
host	Fully qualified name or IP address of host where resource is located
port	Port to use, interpretation depends on protocol (optional)
url-path	“Path” to the resource. Form + interpretation depend on protocol. May have optional parameters.

URL – Schemes

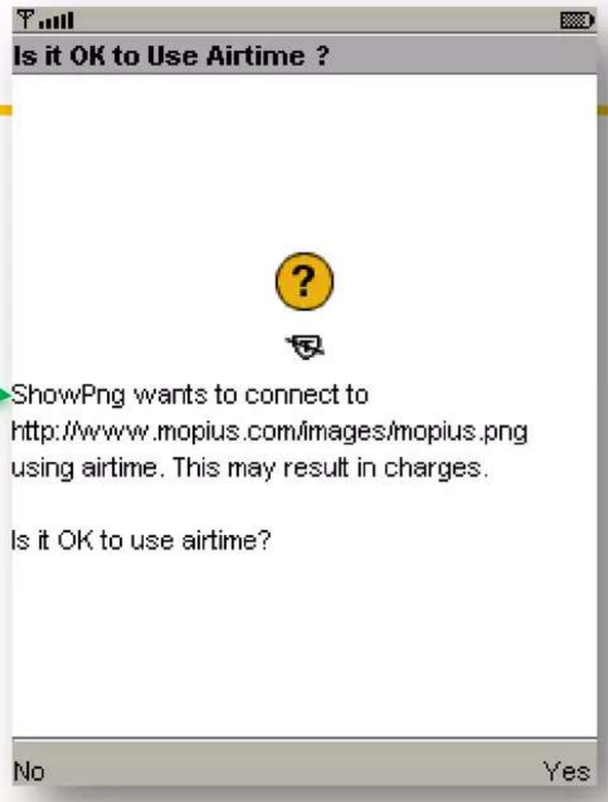
... you can only be sure that HTTP(S) is available on a device!

- Some of GCF connection types:

Scheme	Connectivity	ConnectionType	Defined by…	Optional
http	HyperText Transfer Protocol	HttpConnection	MIDP 1.0 + 2.0	
https	Secure HTTP	HttpsConnection	MIDP 2.0	
socket, serversocket	Socket	SocketConnection, ServerSocketConnection	JSR 118 (MIDP 2.0)	✔
datagram	UDP Datagram	(UDP)DatagramConnection	JSR 118 (MIDP 2.0)	✔
sms, mms	Messaging Service	MessageConnection	JSR 120, JSR 205	✔
file	File Access	FileConnection, InputConnection	JSR 75	✔
btl2cap	Bluetooth	L2CAPConnection	JSR 82	✔
comm	Serial I/O	CommConnection	MIDP 2.0	✔
adpu, jcrmi	Security Element	APDUConnetion, JavaCardRMIConnection	JSR 177	✔

Asynchronous Connection

```
public void commandAction (Command command, Displayable displayable) {
   if (command == iCmdConnect) {
      con = (HttpConnection) Connector.open("http://www.mopius.com/images/mopius.png");
   }
}
```



Establishing a connection in the command handling function leads to a deadlock!

→ The user can no longer select "Yes" or "No".

Notification message provided by WTK (console):

```
Warning: To avoid potential deadlock, operations that may block, such as
 networking, should be performed in a different thread than the
 commandAction() handler.
```

Asynchronous Connection

- **Solution:** Start own thread for networking code!

```
public class MyClass extends MIDlet implements CommandListener, Runnable {
   // …
   // Process the command to connect to the network
   public void commandAction(Command command, Displayable displayable) {
      if (command == cmdConnectToWeb) {
         // Start connection in extra thread → commandAction-function is not blocked
         new Thread(this).start();
      }
   }

   // Connect to the network
   public void run() {
      // …
      HttpConnection con = (HttpConnection)Connector.open(url);
      // …
   }
}
```

Error Handling

- Connection should always be closed (→ Exception!)

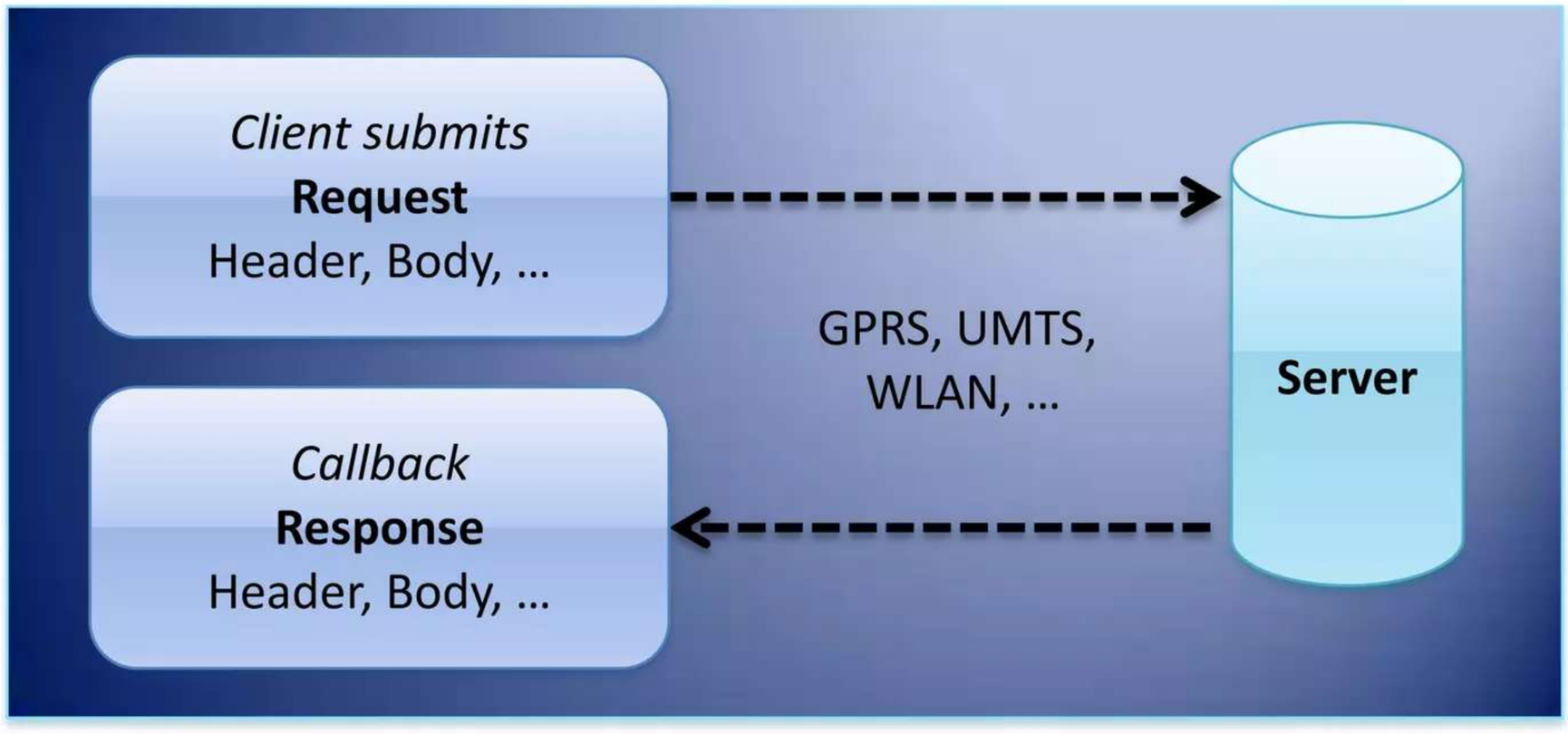
```
// Define those two variables outside the try-block, so that they are known in the finally-block!
SocketConnection con = null;
DataInputStream is = null;
try {
   con = (SocketConnection)Connector.open(url);          // Open the connection to the requested URL
   // … (optional) define the request package and/or send data …
   is = con.openDataInputStream();                       // Read from the connection
   // … read from the input stream to process data …
} catch (IOException ex) {
} finally {
   try {
      if (is != null)                // Here we make sure that everything that might still be open is closed!
         is.close();
      if (con != null)
         con.close();
   } catch (IOException ex) {    // This error can usually be ignored, as something else already went wrong…
   }
}
```

Information about the

HyperText Transfer Protocol

HTTP

- HTTP = Hypertext Transfer Protocol
- **Request/Response-protocol**



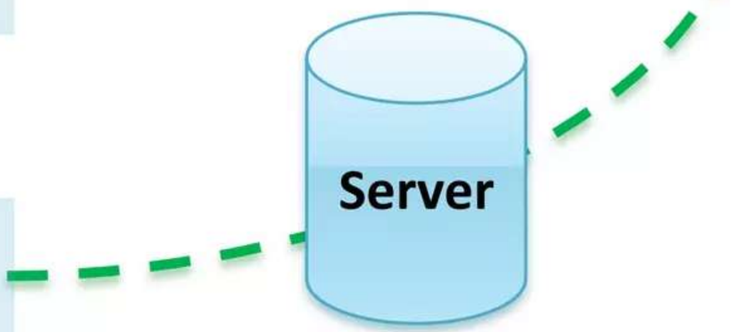
HTTP in JavaME

- **Similar to the logical structure:**
  1. Open a connection
  2. Prepare the request message
  3. Send request + body (optional, POST only)
  4. Retrieve response
  5. Close connection (can’t be reused!)

HTTP Messages

Request	
Method	Action that the server should execute
URL	Location of requested resource
Version	HTTP/1.1 or HTTP/1.0
Header	Additional information (Languages, cookies, …)
Body	To send additional data (POST-request)

Response	
Code	Status of request. e.g. 200, 404, …
Message	Textual version of code. e.g. OK, Not Found, …
Version	HTTP/1.1 or HTTP/1.0
Header	Additional information (Content length, type, …)
Body	If successful: requested content



Request Method

- 3 request methods supported:

Request	Description
GET (Default)	Request information – data is sent as part of the URL (→ Limits max. amount of data). e.g.: http://www.mopius.com/test.php?user=me&pwd=xxx
POST	Request information – data is sent as a separate stream (message body)
HEAD	Request only meta-information about the resource

HTTP Get

JavaME-code:

```
String url = "http://www.mopius.com/test.php?user=me&pwd=xxx";
HttpConnection con = (HttpConnection)Connector.open(url);
con.setRequestMethod(HttpConnection.GET);     // Define method as GET
conn.setRequestProperty("User-Agent", "Mozilla/4.0");
int rc = con.getResponseCode();                          // Request is sent with first read-function
                                                         // No write required!
```

HTTP-Request received by the web-server:

```
GET /test.php?user=me&pwd=xxx HTTP/1.1
Host: www.mopius.com
User-Agent: Mozilla/4.0
```

HTTP Post – Form

JavaME-code:

```
String url = "http://www.mopius.com/test.php";
String data = "user=me&pwd=xxx";            // Data to send, without &
// Special characters have to be URL-encoded manually (e.g. " " -> "%20")
HttpConnection con = (HttpConnection)Connector.open(url);
con.setRequestMethod(HttpConnection.POST);          // Define method as POST
con.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
con.setRequestProperty("User-Agent", "Mozilla/4.0");
con.setRequestProperty("Content-Length", Integer.toString(data.length()));
OutputStream os = con.openOutputStream();
os.write(data.getBytes());
os.close();                                 // Request is sent when output-stream is closed
int rc = con.getResponseCode();
```

HTTP-Request received by the web-server:

```
POST /test.php HTTP/1.1
Host: www.mopius.com
User-Agent: Mozilla/4.0
Content-Length: 15
Content-Type: application/x-www-form-urlencoded

user=me&pwd=xxx
```

HTTP Post – Raw Data

- For **raw client-server** communication:
- **Set content type**

```
con.setRequestProperty("Content-Type", "application/octet-stream");
```

- **Create byte array**, e.g. using DataOutputStream

```
DataOutputStream dos = new DataOutputStream( con.openOutputStream() );
dos.writeBoolean(true);
dos.writeUTF("Hello World");
dos.close();
```

Server Response

- Status line contains numeric value + text
- **Query response code:**

```
int rc = con.getResponseCode();
```

Overview:
HTTP status categories

Range	Category
1xx	Information
2xx	Success
3xx	Redirection
4xx	Client errors
5xx	Server errors

Most important status codes and the respective **constants in HttpConnection**

Code	HttpConnection.
200	HTTP_OK
403	HTTP_FORBIDDEN
404	HTTP_NOT_FOUND
500	HTTP_INTERNAL_ERROR

Retrieving a HTTP Response

- **Simple version:** (creates an image based on stream)

```
// ... send request ...
int rc = con.getResponseCode();                 // Retrieve HTTP response code
if (rc == HttpConnection.HTTP_OK) {             // Response in most cases only available for OK
   is = con.openDataInputStream();              // Get the data input stream
   Image img = Image.createImage(is);           // Read image
}
```

- **Next slide:**
  - More complex version
  - Reads response differently, depending on if the length is known

Retrieving a HTTP Response

```
// ... send request ...
int rc = con.getResponseCode();                 // Retrieve HTTP response code
if (rc == HttpConnection.HTTP_OK) {             // Response in most cases only available for OK
   is = con.openDataInputStream();              // Get the data input stream
   int length = (int)con.getLength();           // Get length of the response - int is enough
   byte data[];
   if (length > 0) {                            // If the server set the content length...
      data = new byte[length];                // Do a direct, more efficient read using a fixed byte array
      int totalReadBytes = 0;                  // is.read() might not read everything at once
      while (totalReadBytes < length) {        // → continue reading until expected length is received
         int curReadBytes = is.read(data, totalReadBytes, length - totalReadBytes);
         if (curReadBytes == -1) break;        // Reached the end before the expected length?
         totalReadBytes += curReadBytes;
      }
   } else {                                     // If the length is unknown, however, use a dynamic stream
      ByteArrayOutputStream bos = new ByteArrayOutputStream();
      int ch;
      while ((ch = is.read()) != -1) {          // Read each byte until a the end is reached
         bos.write(ch);                         // Append the new character to the byte array output stream
      }
      bos.flush();
      data = bos.toByteArray();                 // Create a byte array based on the stream
      bos.close();
   }
}
```

Low-Level Communication

Sockets

Socket Communication

- **Multiplayer games / applications – HTTP?**
  - Request / response might not be useful
  - Large overhead through headers
- **Sockets:**
  - Server listens at specific port
  - Client connects to server
  - Bidirectional communication possible

Streams / Datagrams

- **Stream socket:**
  - Uses **TCP** (Connection-oriented protocol)
  - Open – send/receive ... – Close
  - Delivery & order of data are guaranteed
- **Datagram socket:**
  - Uses **UDP** (Record-oriented system)
  - Datagram = chunk of data, no stream
  - Delivery not guaranteed
  - Faster than TCP
- **Mobile phone:**
  - Performance issues in any case (network speed)!

Sockets and Java ME

- Similar to HTTP:

TCP-Connection:

```
SocketConnection con = (SocketConnection) Connector.open("socket://127.0.0.1:5000");
```

UDP-Connection:

```
UDPDatagramConnection con = (UDPDatagramConnection)
                            Connector.open("datagram://127.0.0.1:5000");
```

- Writing and Reading:

```
InputStream is = con.openInputStream();
OutputStream os = con.openOutputStream();
// ... write and read to/from the streams ...
is.close();
os.close();
con.close();
```

That’s it!

Thanks for your attention