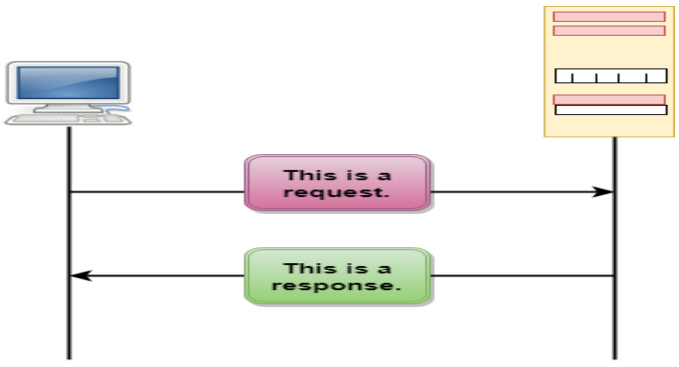
HTTP

- HTTP stands for **HyperText Transfer Protocol**.
- It is a protocol used to access the data on the World Wide Web (www).
- The HTTP protocol can be used to transfer the data in the form of plain text, hypertext, audio, video, and so on.
- This protocol is known as HyperText Transfer Protocol because of its efficiency that allows us to use in a hypertext environment where there are rapid jumps from one document to another document.
- HTTP is similar to the FTP as it also transfers the files from one host to another host. But, HTTP is simpler than FTP as HTTP uses only one connection, i.e., no control connection to transfer the files.
- HTTP is used to carry the data in the form of MIME-like format.
- HTTP is similar to SMTP as the data is transferred between client and server. The HTTP differs from the SMTP in the way the messages are sent from the client to the server and from server to the client. SMTP messages are stored and forwarded while HTTP messages are delivered immediately.

Features of HTTP:

- **Connectionless protocol:** HTTP is a connectionless protocol. HTTP client initiates a request and waits for a response from the server. When the server receives the request, the server processes the request and sends back the response to the HTTP client after which the client disconnects the connection. The connection between client and server exist only during the current request and response time only.
- **Media independent:** HTTP protocol is a media independent as data can be sent as long as both the client and server know how to handle the data content. It is required for both the client and server to specify the content type in MIME-type header.
- **Stateless:** HTTP is a stateless protocol as both the client and server know each other only during the current request. Due to this nature of the protocol, both the client and server do not retain the information between various requests of the web pages.

HTTP Transactions



The above figure shows the HTTP transaction between client and server. The client initiates a transaction by sending a request message to the server. The server replies to the request message by sending a response message.

The Protocol

HTTP (Hypertext Transfer Protocol) is the standard protocol used to communicate between web browsers (client) and web servers. It enables the exchange of various types of data and resources, such as HTML pages, images, videos, and other multimedia content. Here are the steps involved in the communication process between a web browser and a web server using the HTTP protocol:

1. The client (web browser) initiates a request to the server by sending an HTTP request message. This request message contains a URL (Uniform Resource Locator) that identifies the web page or resource that the client wants to access.
2. The server receives the request message and processes it to identify the requested resource. If the resource is available, the server generates a response message containing the requested resource data.
3. The server sends the response message back to the client, which receives and processes it to display the resource data on the screen.
4. If the client wants to access additional resources on the same server, it can reuse the existing TCP/IP connection instead of opening a new one. This reduces the overhead associated with establishing a new connection for every request.
5. If the requested resource is not available on the server or if there is an error during the communication process, the server generates an error response message and sends it back to the client.
6. The client can also send additional HTTP requests to the server, such as requests to submit data or perform other actions. These requests are also processed by the server, which generates response messages as necessary.

```
import java.net.*;
import java.io.*;
public class HTTPDemo {
  public static void main(String[] args) throws Exception {
    // Create a URL object to represent the endpoint we want to communicate with
    URL url = new URL("http://example.com");
    // Open a connection to the URL
    HttpURLConnection con = (HttpURLConnection) url.openConnection();
    // Set the request method to GET
    con.setRequestMethod("GET");
    // Send the request and retrieve the response code
    int responseCode = con.getResponseCode();
    // Print the response code to the console
    System.out.println("Response code: " + responseCode);
    // Open an input stream to read the response from the server
    InputStream inputStream = con.getInputStream();
    // Create a BufferedReader to read the response as text
    BufferedReader in = new BufferedReader(
      new InputStreamReader(inputStream));
    // Read the response line by line and print it to the console
    String inputLine;
    while ((inputLine = in.readLine()) != null) {
      System.out.println(inputLine);
    }
    // Close the input stream and disconnect from the server
    in.close();
    con.disconnect();
  }
}
```

The Protocol: Keep Alive

HTTP keep-alive is a feature that allows multiple requests and responses to be sent over a single TCP connection. Without keep-alive, each HTTP request would require a separate TCP connection to be established, which can add significant overhead and latency to web page loading times. With keep-alive, however, the same TCP connection can be reused for multiple requests, reducing the amount of time needed to establish and tear down connections. This can result in faster web page loading times, especially for pages with many embedded resources, such as images, stylesheets, and JavaScript files. Keep-alive works by adding a "Connection: keep-alive" header to the HTTP request and response messages, which tells the server and client to keep the TCP connection open after the initial request/response cycle. The connection will remain open until it is explicitly closed by either the client or the server, or until a timeout occurs.

Keep-Alive

HTTP 1.0 opens a new connection for each request. In practice, the time taken to open and close all the connections in a typical web session can outweigh the time taken to transmit the data, especially for sessions with many small documents. This is even more problematic for encrypted HTTPS connections using SSL or TLS, because the hand- shake to set up a secure socket is substantially more work than setting up a regular socket.

In HTTP 1.1 and later, the server doesn't have to close the socket after it sends its response. It can leave it open and wait for a new request from the client on the same socket. Multiple requests and responses can be sent in series over a single TCP connection. However, the lockstep pattern of a client request followed by a server response remains the same.

A client indicates that it's willing to reuse a socket by including a *Connection* field in the HTTP request header with the value *Keep-Alive*:

Connection: Keep-Alive

The URL class transparently supports HTTP Keep-Alive unless explicitly turned off. That is, it will reuse a socket if you connect to the same server again before the server has closed the connection. You can control Java's use of HTTP Keep-Alive with several system properties:

- Set http.keepAlive to "true or false" to enable/disable HTTP Keep-Alive. (It is enabled by default.)
- Set http.maxConnections to the number of sockets you're willing to hold open at one time. The default is 5.
- Set http.keepAlive.remainingData to true to let Java clean up after abandoned connections (Java 6 or later). It is false by default.
- Set sun.net.http.errorstream.enableBuffering to true to attempt to buffer the relatively short error streams from 400- and 500-level responses, so the connection can be freed up for reuse sooner. It is false by default.
- Set sun.net.http.errorstream.bufferSize to the number of bytes to use for buffering error streams. The default is 4,096 bytes.
- Set sun.net.http.errorstream.timeout to the number of milliseconds before timing out a read from the error stream. It is 300 milliseconds by default.

The defaults are reasonable, except that you probably do want to set sun.net.http.errorstream. enableBuffering to true unless you want to read the error streams from failed requests. HTTP 2.0, which is mostly based on the SPDY protocol invented at Google, further optimizes HTTP transfers through header compression, pipelining requests and responses, and asynchronous connec- tion multiplexing. However, these optimizations are usually per- formed in a translation layer that shields application programmers from the details, so the code you write will still mostly follow the preceding steps 1–4. Java does not yet support HTTP 2.0; but when the capability is added, your programs shouldn't need to change to take advantage of it, as long as you access HTTP servers via the URL and URLConnection classes.

HTTP Methods

Communication with an HTTP server follows a request-response pattern: one stateless request followed by one stateless response. Each HTTP request has two or three parts:

- A start line containing the HTTP method and a path to the resource on which the method should be executed
- A header of name-value fields that provide meta-information such as authentication credentials and preferred formats to be used in the request
- A request body containing a representation of a resource (POST and PUT only) There are four main HTTP methods, four verbs if you will, that identify the operations
  that can be performed:
  • GET
  • POST
  • PUT
  • DELETE

1. GET: The GET method is used to retrieve a resource from the server. The client sends a GET request to the server, which responds with the requested resource. This method is commonly used for retrieving web pages, images, videos, and other static content.
2. POST: The POST method is used to submit data to the server. The client sends a POST request containing the data to be submitted, and the server processes the data and sends a response. This method is commonly used for submitting form data, uploading files, and performing other operations that require data to be sent to the server.
3. PUT: The PUT method is used to update an existing resource on the server. The client sends a PUT request containing the updated data, and the server replaces the existing resource with the new data. This method is commonly used for updating documents, images, and other resources on the server.
4. DELETE: The DELETE method is used to delete a resource from the server. The client sends a DELETE request to the server, which removes the specified resource. This method is commonly used for removing documents, images, and other resources from the server.
5. HEAD: The HEAD method is similar to the GET method, but it only retrieves the headers of a resource, not the full content. The client sends a HEAD request to the server, and the server responds with the headers of the requested resource. This method is commonly used for checking the availability and status of a resource without actually retrieving the content.
6. OPTION: The OPTIONS HTTP method is used to retrieve information about the communication options available for a particular resource on the server. When an OPTIONS request is sent to a server, the server should respond with a list of the HTTP methods that are allowed for that resource, as well as any other communication options that are available, such as supported headers and authentication methods.

The Request Body

HTTP request body is the part of an HTTP request that contains additional information sent by the client to the server. It is commonly used to send data to the server for processing, such as when submitting a form or uploading a file. The request body is typically encoded in a specific format such as JSON, XML, or form data, and is preceded by a set of HTTP headers that provide information about the request. The format of the request body depends on the content type specified in the request headers. It is an essential component of HTTP communication, enabling clients to send complex data to servers, allowing the implementation of advanced web applications and APIs. The GET method retrieves a representation of a resource identified by a URL. The exact location of the resource you want to GET from a server is specified by the various parts of the path and query string. How different paths and query strings map to different resources is determined by the server. The URL class doesn't really care about that. As long as it knows the URL, it can download from it.

POST and PUT are more complex. In these cases, the client supplies the representation of the resource, in addition to the path and the query string. The representation of the resource is sent in the body of the request, after the header. That is, it sends these four items in order:

1. A starter line including the method, path and query string, and HTTP version
2. An HTTP header
3. A blank line (two successive carriage return/linefeed pairs)
4. The body

For example, this POST request sends form data to a server:

```
POST /cgi-bin/register.pl HTTP 1.0
Date: Sun, 27 Apr 2013 12:32:36
Host: www.cafeaulait.org
Content-type: application/x-www-form-urlencoded
Content-length: 54
username=Elliotte+Harold&email=elharo%40ibiblio.org
```

In this example, the body contains an *application/x-www-form-urlencoded* data, but that’s just one possibility. In general, the body can contain arbitrary bytes. However, the HTTP header should include two fields that specify the nature of the body:

• A Content-length field that specifies how many bytes are in the body (54 in the preceding example)

• A Content-type field that specifies the MIME media type of the bytes (*application/ x-www-form-urlencoded* in the preceeding example).

The *application/x-www-form-urlencoded* MIME type used in the preceding example is common because it’s how web browsers encode most form submissions. Thus it’s used by a lot of server-side programs that talk to browsers. However, it’s hardly the only possible type you can send in the body. For example, a camera uploading a picture to a photo sharing site can send image/jpeg. A text editor might send text/html. It’s all just bytes in the end. For example, here’s a PUT request that uploads an Atom document:

```
PUT /blog/software-development/the-power-of-pomodoros/ HTTP/1.1
Host: elharo.com
User-Agent: AtomMaker/1.0
Authorization: Basic ZGFmZnk6c2VjZXJldA==
Content-Type: application/atom+xml;type=entry
Content-Length: 322
<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
 <title>The Power of Pomodoros</title>
 <id>urn:uuid:101a41a6-722b-4d9b-8afb-ccfb01d77499</id>
 <updated>2013-02-22T19:40:52Z</updated>
 <author><name>Elliotte Harold</name></author>
 <content>I hadn’t paid much attention to Pomodoro...</content>
</entry>
```

HTTP Cookies

HTTP cookies are small pieces of data that are sent from a website to a user's web browser, which are then stored on the user's computer or mobile device. Cookies are used to track user activity, personalize website content, and remember user preferences. When a user visits a website that has previously set a cookie, the website can read the cookie to gather information about the user's previous activity on the site. Cookies can also be used to remember a user's login credentials, language preference, or other settings, which can make the user's experience on the website more efficient and personalized.

CookieHandler Class

The CookiesHandler class is a programming class used to manage HTTP cookies in a web application. It provides methods for retrieving, setting, updating, and deleting cookies on a user's computer. The class also offers options to ensure the security of cookies, such as setting the expiration time, domain, and security level. Overall, the CookiesHandler class is an essential tool for web developers to manage cookies and provide a personalized and secure user experience on their web applications. Here are some of the key uses of the CookiesHandler class:

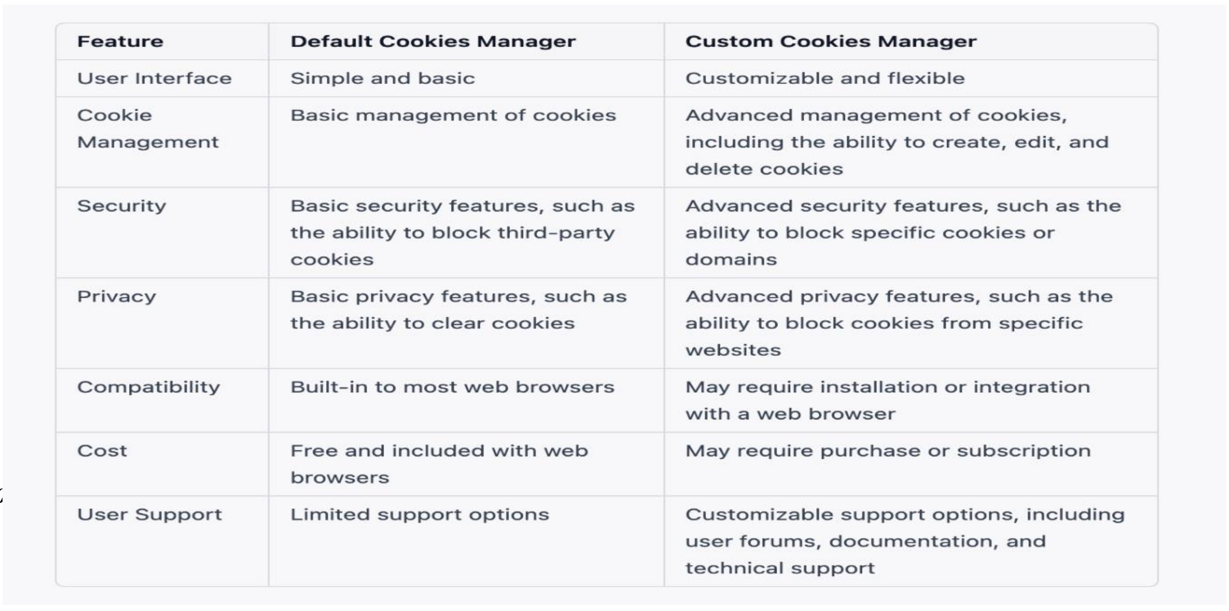
1. Retrieving Cookies: The CookiesHandler class can retrieve cookies that have been set on a user's computer by a web server. This can be useful for accessing information such as user preferences or login credentials.
2. Setting Cookies: The CookiesHandler class can also set cookies on a user's computer that can be read by a web server. This can be useful for storing user preferences or login credentials for future use.
3. Updating Cookies: The CookiesHandler class can update the value of an existing cookie on a user's computer. This can be useful for keeping track of user activity or preferences over time.
4. Deleting Cookies: The CookiesHandler class can also delete cookies that have been set on a user's computer. This can be useful for removing sensitive information such as login credentials or personal data.
5. Security: The CookiesHandler class can also be used to ensure the security of cookies by setting options such as the cookie's expiration time, its domain, and its security level (i.e., whether it can only be transmitted over a secure connection).

Cookie Manager and its importance

A cookie manager is a software tool that is used to manage HTTP cookies in a web browser. Here are some of the key importance of a cookie manager:

1. Privacy: A cookie manager can help protect the privacy of web users by allowing them to control which cookies are stored on their computer or mobile device. This can prevent the collection of personal information and tracking of user activity by third-party websites.
2. Security: A cookie manager can help improve the security of web browsing by allowing users to delete cookies that may contain sensitive information such as login credentials or personal data. This can prevent unauthorized access to user accounts and protect against identity theft.
3. Efficiency: A cookie manager can improve the efficiency of web browsing by allowing users to save and manage cookies that store website preferences and login credentials. This can streamline the login process and make the user's browsing experience more efficient.
4. Customization: A cookie manager can allow users to customize their web browsing experience by allowing them to control which cookies are accepted or rejected. This can help users tailor their browsing experience to their specific needs and preferences.
5. Compliance: A cookie manager can help website owners comply with privacy laws and regulations by providing users with clear and transparent information about the use of cookies on their websites. This can help build trust with users and ensure compliance with data protection laws.

Difference between Default cookie manager and Custom cookie manager



Feature	Default Cookies Manager	Custom Cookies Manager
User Interface	Simple and basic	Customizable and flexible
Cookie Management	Basic management of cookies	Advanced management of cookies, including the ability to create, edit, and delete cookies
Security	Basic security features, such as the ability to block third-party cookies	Advanced security features, such as the ability to block specific cookies or domains
Privacy	Basic privacy features, such as the ability to clear cookies	Advanced privacy features, such as the ability to block cookies from specific websites
Compatibility	Built-in to most web browsers	May require installation or integration with a web browser
Cost	Free and included with web browsers	May require purchase or subscription
User Support	Limited support options	Customizable support options, including user forums, documentation, and technical support

Co

Cookie Store

A cookie store is a data structure used by web browsers to store HTTP cookies. When a user visits a website, the website can set cookies on the user's computer, which are then stored in the cookie store. The cookie store is typically organized by domain, with each domain having its own set of cookies. Cookies in the cookie store can contain information such as user preferences, login credentials, and session data. When a user revisits a website, the website can read the cookies from the cookie store to personalize the user's experience and remember their previous activity. The cookie store is an important component of web browsing and allows websites to provide personalized and efficient user experiences. The key purposes of HTTP cookie stores:

1. State Management: HTTP cookie stores allow websites to store information about user sessions, such as login credentials and session IDs. This allows users to remain authenticated and maintain their session state across multiple page views or visits.
2. Personalization: HTTP cookie stores allow websites to personalize the user experience by storing user preferences and activity data. This can include items such as language preferences, product recommendations, and search history.
3. Tracking: HTTP cookie stores can be used to track user behaviour and deliver targeted advertising. This can include items such as personalized product recommendations or targeted display ads based on a user's browsing history.
4. Shopping Carts: HTTP cookie stores can be used to store information about a user's shopping cart, allowing users to add items to their cart and return to complete their purchase at a later time.
5. Security: HTTP cookie stores can be used to enhance website security by storing information such as authentication tokens and session IDs. This can help prevent unauthorized access to user accounts and protect against identity theft.

Program to display HTTP Header Format

```
import java.net.*;
import java.io.*;
public class HttpHeaderExample {
  public static void main(String[] args) throws Exception {
    URL url = new URL("https://www.example.com");
    HttpURLConnection connection = (HttpURLConnection) url.openConnection();
    // Set request method
    connection.setRequestMethod("GET");
    // Set request headers
    connection.setRequestProperty("User-Agent", "Mozilla/5.0");
    connection.setRequestProperty("Accept-Language", "en-US,en;q=0.5");
    // Print response headers
    System.out.println("Response Code: " + connection.getResponseCode());
    System.out.println("Response Message: " + connection.getResponseMessage());
    System.out.println("Content-Type: " + connection.getContentType());
    System.out.println("Content-Length: " + connection.getContentLength());
    // Print request headers
    System.out.println("Request Method: " + connection.getRequestMethod());
    System.out.println("Request URL: " + connection.getURL());
    System.out.println("User-Agent: " + connection.getRequestProperty("User-Agent"));
    System.out.println("Accept-Language: " + connection.getRequestProperty("Accept-Language"));
    // Close connection
    connection.disconnect();
  }
}
```

Unit 4

1. What is the Keep-Alive protocol and how does it work in the context of HTTP? How can developers use this protocol to improve the performance and efficiency of their web applications, and what are some potential drawbacks to be aware of when working with Keep-Alive?
2. What are the different HTTP methods available for communicating with web servers, and what are some of the key features and use cases for each method? How can developers determine which method to use for a given application, and what are some best practices for implementing these methods in Java-based web applications?
3. What is the Request Body in HTTP, and how does it function within the larger context of HTTP communication? How can developers use the Request Body to send and receive data from web servers, and what are some common use cases for this feature?
4. What are cookies in the context of web development, and how do they function within the larger framework of HTTP? How can developers use the CookieManager and CookiesStore classes to manage cookies in Java-based web applications, and what are some best practices for ensuring that cookies are used effectively and securely?