互联网编程

简答题+编程题

1.简要介绍 InetAddress 类，URI 类，URL 类和 URLConnection 类的作用，并回答它们分别在什么编程需求下可以选用。

① InetAddress 类

InetAddress 类用于表示 IP 地址，可以获取主机名和 IP 地址之间的映射关系。通过 InetAddress 类可以进行 DNS 查询，获取域名对应的 IP 地址等操作。在网络编程中，InetAddress 类常用于实现基于 IP 地址的网络通信。

适用编程需求：需要进行 DNS 查询，获取域名对应的 IP 地址；需要实现基于 IP 地址的网络通信。

②URI 类

URI 类用于表示一个统一资源标识符（Uniform Resource Identifier），可以用来标识互联网上的资源。URI 包含了对资源的描述，例如协议、主机名、端口号、路径等信息。在网络编程中，URI 类常用于解析和构造 URL。

适用编程需求：需要解析或构造 URL。

③URL 类

URL 类用于表示一个统一资源定位符（Uniform Resource Locator），可以用来定位互联网上的资源。URL 包含了对资源的描述，例如协议、主机名、端口号、路径等信息，还可以包含查询字符串和片段标识符。在网络编程中，URL 类常用于获取网络资源，例如下载文件、访问网页等。

适用编程需求：需要获取网络资源，例如下载文件、访问网页等。

④ URLConnection 类

URLConnection 类用于表示应用程序和 URL 之间的通信链接，它可以与远程服务器进行交互，发送请求和接收响应。通过 URLConnection 类可以配置请求头信息、请求方法、超时时间等，还可以获取响应头信息、响应码等。

适用编程需求：需要与远程服务器进行交互，发送请求和接收响应；需要配置请求头信息、请求方法、超时时间等；需要获取响应头信息、响应码等。

2. 创建一个简单的 HTTP 客户端程序：EasyHTTPClient 类，它访问 www.szu.edu.cn/xxgk/xxjj.htm，把得到的 HTTP 响应结果保存到本地文件系统的一个文件。代码的开头部分已给出：

```
Import java.net.*
Import java.io.*
public class EasyHTTPClient{
    String host = "www.szu.edu.cn/xxgk/xxjj.htm";
    int port = 80;
import java.net.*;
import java.io.*;
//以下是代码
public class EasyHTTPClient {
    public static void main(String[] args) {
        String host = "www.szu.edu.cn";
        String path = "/xxgk/xxjj.htm";
```

```
            int port = 80;
            String fileToSave = "response.txt";
            try {
                InetAddress address = InetAddress.getByName(host);
                Socket socket = new Socket(address, port);
                OutputStreamWriter out = new OutputStreamWriter(socket.getOutputStream());
                String request = "GET " + path + " HTTP/1.1\r\n";
                request += "Host: " + host + "\r\n";
                request += "Connection: close\r\n";
                request += "\r\n";
                out.write(request);
                out.flush();
                BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
                String line;
                boolean headerFinished = false;
                BufferedWriter fileWriter = new BufferedWriter(new FileWriter(fileToSave));
                while ((line = in.readLine()) != null) {
                    if (!headerFinished) {
                        if (line.isEmpty()) {
                            headerFinished = true;
                        }
                    } else {
                        fileWriter.write(line);
                        fileWriter.newLine();
                    }
                }
                in.close();
                fileWriter.close();
                socket.close();
                System.out.println("Response saved to " + fileToSave);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
}
```

3. 使用 httpURLConnection 类实现 web 页面的下载

```
import java.net.HttpURLConnection;
import java.net.URL;
import java.io.InputStream;
import java.io.FileOutputStream;

public class WebPageDownloader {
```

```
    public static void main(String[] args) {
        String urlStr = "https://www.example.com/index.html"; // 需要下载的网页地址
        try {
            URL url = new URL(urlStr);
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();

            // 设置请求方法和超时时间
            conn.setRequestMethod("GET");
            conn.setConnectTimeout(5000);

            // 获取响应状态码
            int responseCode = conn.getResponseCode();
            if (responseCode == 200) {
                // 获取响应输入流
                InputStream inputStream = conn.getInputStream();

                // 将响应内容写入文件
                FileOutputStream outputStream = new FileOutputStream("index.html");
                byte[] buffer = new byte[1024];
                int len = 0;
                while ((len = inputStream.read(buffer)) != -1) {
                    outputStream.write(buffer, 0, len);
                }

                // 关闭输入输出流
                outputStream.close();
                inputStream.close();
                System.out.println("Web 页面下载成功！");
            } else {
                System.out.println("Web 页面下载失败，响应状态码：" + responseCode);
            }
            conn.disconnect();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

4. 实现具有缓存功能的 HTTP 客户端

```
// 导入相关的类
import java.io.*;
import java.net.*;
import java.util.*;
```

```
// 定义一个缓存类，用于存储 URL 和对应的响应内容
class Cache {
    // 使用一个 HashMap 来保存 URL 和响应内容的映射关系
    private HashMap<String, String> map;
    // 构造方法，初始化 HashMap
    public Cache() {
        map = new HashMap<>();
    }
    // 判断缓存中是否存在某个 URL 的响应内容
    public boolean contains(String url) {
        return map.containsKey(url);
    }
    // 从缓存中获取某个 URL 的响应内容，如果不存在则返回 null
    public String get(String url) {
        return map.get(url);
    }
    // 将某个 URL 和对应的响应内容存入缓存中
    public void put(String url, String content) {
        map.put(url, content);
    }
}
// 定义一个 HTTP 客户端类，用于发送请求和接收响应
class HttpClient {
    // 使用一个 Cache 对象来保存缓存内容
    private Cache cache;
    // 构造方法，初始化 Cache 对象
    public HttpClient() {
        cache = new Cache();
    }
    // 发送 GET 请求的方法，参数为目标 URL，返回值为响应内容
    public String get(String url) throws IOException {
        // 如果缓存中存在该 URL 的响应内容，则直接返回
        if (cache.contains(url)) {
            return cache.get(url);
        }
        // 否则，创建一个 URL 对象和一个 URLConnection 对象
        URL u = new URL(url);
        URLConnection conn = u.openConnection();
        // 设置请求头，指定只接受文本类型的响应
        conn.setRequestProperty("Accept", "text/*");
        // 打开输入流，读取响应内容
        BufferedReader in = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
        StringBuilder sb = new StringBuilder();
```

```
            String line;
            while ((line = in.readLine()) != null) {
                sb.append(line).append("\n");
            }
            in.close();
            // 将响应内容转换为字符串，并存入缓存中
            String content = sb.toString();
            cache.put(url, content);
            // 返回响应内容
            return content;
        }
}
// 定义一个测试类，用于测试 HTTP 客户端的功能
public class Test {
    public static void main(String[] args) throws IOException {
        // 创建一个 HTTP 客户端对象
        HttpClient client = new HttpClient();
        // 定义一个测试用的 URL 数组
        String[] urls = {"https://www.baidu.com", "https://www.google.com",
"https://www.bing.com"};
        // 循环两次，分别测试第一次请求和第二次请求的效果
        for (int i = 0; i < 2; i++) {
            System.out.println("Round " + (i + 1) + ":");
            for (String url : urls) {
                System.out.println("Requesting " + url);
                long start = System.currentTimeMillis(); // 记录请求开始时间
                String content = client.get(url); // 发送请求并获取响应内容
                long end = System.currentTimeMillis(); // 记录请求结束时间
                System.out.println("Response length: " + content.length()); // 打印响应内容
的长度
                System.out.println("Response time: " + (end - start) + " ms"); // 打印请求耗
时
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

5. 简述什么是中间人攻击，如何有效解决？

中间人攻击（Man-in-the-Middle Attack）是一种常见的网络安全威胁，攻击者在通信过程中潜在地插入自己作为"中间人"来窃取、篡改或伪造通信内容。这种攻击通常发生在通信的两个实体之间，攻击者能够截获、修改或篡改双方之间传输的数据。

中间人攻击的工作原理是攻击者在目标实体之间建立起虚假的通信连接。攻击者可能会使用一些技术手段来实现这一点，如 ARP 欺骗、DNS 欺骗、Wi-Fi 劫持等。一旦攻击者成功地插入自己作为中间人，他们可以监视通信内容、修改数据、注入恶意代码或伪装成合法实体进行欺骗。

为了有效解决中间人攻击，可以采取以下几种措施：

①加密通信：使用安全的通信协议和加密算法，例如 HTTPS、SSH 等，能够保护通信内容的机密性和完整性，使攻击者无法获取或篡改数据。

②证书和公钥基础设施（PKI）：使用数字证书和 PKI 来验证通信实体的身份，确保双方的身份是合法的，防止攻击者冒充通信方。

③安全的网络连接：确保在连接网络时使用受信任的网络，特别是公共 Wi-Fi 网络容易受到中间人攻击，因此应尽量避免使用不安全的公共 Wi-Fi。

④安全软件和防火墙：使用防火墙和安全软件来监测和阻止中间人攻击，这些工具可以检测到异常的通信行为并采取相应的措施。

⑤安全意识教育：教育用户提高安全意识，包括警惕钓鱼邮件、恶意链接等网络攻击手段，以免被诱导进入中间人攻击的陷阱。

综上所述，中间人攻击是一种严重的网络安全威胁，但通过使用加密通信、证书和 PKI、安全的网络连接、安全软件和防火墙以及安全意识教育等措施，可以有效地减轻中间人攻击的风险。

6. 基于 UDP 传输协议使用 java 编程实现一个 UDP 客户端与服务器端，进行通信。

Clinet.java

```
import java.io.IOException;
import java.net.*;

public class Client {
    public static void main(String[] args) {
        String serverHostname = "localhost";
        int serverPort = 12345;
        String message = "Hello, server!";

        try {
            InetAddress serverAddress = InetAddress.getByName(serverHostname);
            DatagramSocket socket = new DatagramSocket();

            byte[] sendData = message.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, serverAddress, serverPort);

            socket.send(sendPacket);

            byte[] receiveData = new byte[1024];
            DatagramPacket receivePacket = new DatagramPacket(receiveData,
```

```
receiveData.length);

                socket.receive(receivePacket);

                String receivedMessage = new String(receivePacket.getData(), 0,
receivePacket.getLength());
                System.out.println("Server response: " + receivedMessage);

                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

Server.java

```
import java.io.IOException;
import java.net.*;

public class Server {
    public static void main(String[] args) {
        int serverPort = 12345;

        try {
            DatagramSocket socket = new DatagramSocket(serverPort);

            byte[] receiveData = new byte[1024];
            DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length);

            socket.receive(receivePacket);

            String receivedMessage = new String(receivePacket.getData(), 0,
receivePacket.getLength());
            System.out.println("Client message: " + receivedMessage);

            InetAddress clientAddress = receivePacket.getAddress();
            int clientPort = receivePacket.getPort();
            String responseMessage = "Hello, client!";

            byte[] sendData = responseMessage.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendData,
sendData.length, clientAddress, clientPort);
```

```
            socket.send(sendPacket);

            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

7. 使用 Java 实现一个 PC 的 telnet 客户端，能够根据输入服务器地址（例如 bbs.pku.edu.cn，但不限于，任意 BBS 均可）连接服务器，实现以下功能： 1. 登录 bbs.pku.edu.cn（但不限于），显示服务器发来的文本数据； 2. 将用户输入的用户名 guest 发给服务器； 3. 显示更多服务器发来的文本数据； 4. 将键盘方向键“上下左右”的命令发给服务器端，操纵 BBS。

```
import java.io.*;
import java.net.*;

public class TelnetClient {
    public static void main(String[] args) {
        String serverHostname = "bbs.pku.edu.cn";
        int serverPort = 23;

        try {
            Socket socket = new Socket(serverHostname, serverPort);
            InputStream inputStream = socket.getInputStream();
            OutputStream outputStream = socket.getOutputStream();

            BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));
            PrintWriter writer = new PrintWriter(outputStream, true);

            // 读取并显示服务器发来的文本数据
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }

            // 将用户名发送给服务器
            writer.println("guest");

            // 读取并显示更多服务器发来的文本数据
            while ((line = reader.readLine()) != null) {
```

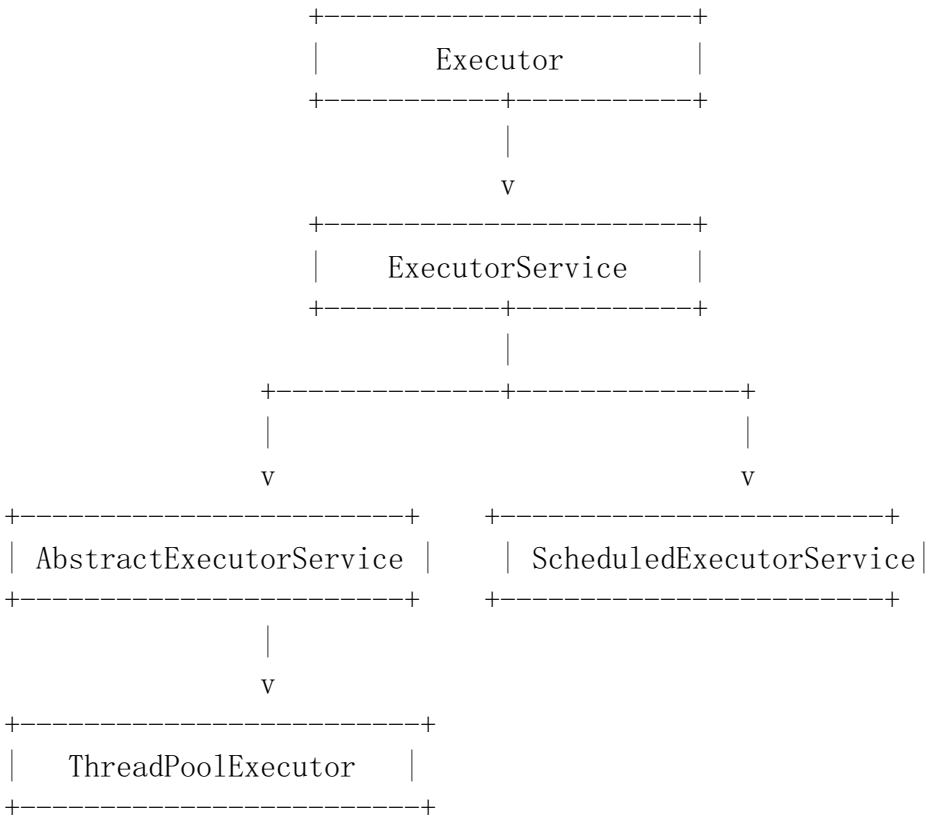
```
                System.out.println(line);
            }

            // 键盘输入，发送命令给服务器
            BufferedReader keyboardReader = new BufferedReader(new
InputStreamReader(System.in));
            while (true) {
                String command = keyboardReader.readLine();
                writer.println(command);

                // 读取并显示服务器发来的文本数据
                while ((line = reader.readLine()) != null) {
                    System.out.println(line);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

8. Java 实现线程池有哪些类？试画出相应实现程序的类结构图
在 Java 中，实现线程池的主要类有以下几个：

Executor: 是一个顶层接口，定义了执行任务的方法。
ExecutorService: 继承自 Executor 接口，提供了更丰富的任务执行控制方法，比如提交任务、关闭线程池等。
AbstractExecutorService: 是 ExecutorService 接口的抽象实现，提供了默认的实现方法，便于自定义线程池的实现。
ThreadPoolExecutor: 是 AbstractExecutorService 的具体实现类，提供了线程池的核心功能，包括线程池的创建、管理和任务调度等。
ScheduledExecutorService: 继承自 ExecutorService 接口，支持任务的定时执行和周期性执行。



在类结构图中，Executor 是顶层接口，ExecutorService 是具体的线程池接口，AbstractExecutorService 是 ExecutorService 的抽象实现。ScheduledExecutorService 继承自 ExecutorService，支持任务的定时执行和周期性执行。最后，ThreadPoolExecutor 是 AbstractExecutorService 的具体实现类，提供了线程池的核心功能。

9. Java 实现线程池使用了工厂模式，什么是工厂模式？请试使用这种设计模式

工厂模式（Factory Pattern）是一种创建型设计模式，它提供了一种创建对象的接口，但将具体对象的创建逻辑封装在工厂类中，使得客户端无需知道具体对象的创建过程，只需要通过工厂类来创建对象。

工厂模式通常包含以下几个关键角色：

1. 抽象产品（Abstract Product）：定义了产品的共同接口，客户端通过这个接口访问具体产品。
2. 具体产品（Concrete Product）：实现了抽象产品接口，是工厂模式所创建的对象。
3. 抽象工厂（Abstract Factory）：声明了创建产品的工厂方法，可以是接口或抽象类。
4. 具体工厂（Concrete Factory）：实现了抽象工厂接口，负责创建具体产品的对象。

下面是一个使用工厂模式实现线程池的示例：

```
// 抽象产品：任务接口
public interface Task {
    void execute();
}

// 具体产品：具体任务类
public class ConcreteTask implements Task {
    @Override
    public void execute() {
        // 执行具体任务的逻辑
    }
}

// 抽象工厂：线程池工厂接口
public interface ThreadPoolFactory {
    Task createTask();
}

// 具体工厂：线程池工厂实现类
public class ThreadPoolFactoryImpl implements ThreadPoolFactory {
    @Override
    public Task createTask() {
        return new ConcreteTask();
    }
}

// 客户端使用示例
public class Client {
    public static void main(String[] args) {
        ThreadPoolFactory factory = new ThreadPoolFactoryImpl();
        Task task = factory.createTask();
        task.execute();
    }
}
```

在上述示例中，抽象产品是`Task`接口，定义了任务的共同接口。具体产品是`ConcreteTask`类，实现了`Task`接口，代表具体的任务类。抽象工厂是`ThreadPoolFactory`接口，声明了创建任务的工厂方法。具体工厂是`ThreadPoolFactoryImpl`类，实现了`ThreadPoolFactory`接口，负责创建具体的任务对象。

客户端通过工厂类来创建任务对象，而无需关心具体的任务类和创建过程。这种设计方式可以使客户端代码与具体产品类解耦，提高代码的灵活性和可维护性。

10. 如何快速判断垃圾邮件发送者？

```
import java.net.*;
public class SpamCheck{
    public static final String BLACKHOLE = "sbl.spamhaus.org";
    public static void main(String[] args) throws
UnknownHostException{
        for (String arg: args){
            if (isSpammer(arg)){
                System.out.println(arg + " is a known spammer.");
            }
            else{
                System.out.println(arg + " appears legitimate.");
            }
        }
    }

private static boolean isSpammer(String arg) {
        try {
            InetAddress address = InetAddress.getByName(arg);
            byte[] quad = address.getAddress();
            String query = BLACKHOLE;
            for (byte octet : quad){
                int unsignedByte = octet < 0 ? octet + 256 : octet;
                query = unsignedByte + "." + query;
            }
            InetAddress.getByName(query);
            return true;
        }
        catch (UnknownHostException e) {
            return false;
        }
    }
}
```

11. 试从程序输入输出处理数据的角度思考并解释一下，从 PC 浏览器地址栏输入一个 url，到可以在浏览器中看到这个网页内容，什么程序从哪里获得了什么数据？处理了什么数据？

从 PC 浏览器地址栏输入一个 URL 到最终在浏览器中看到网页内容，涉及多个程序的协作和数据处理过程。下面是一个简化的描述：

1. 用户在 PC 浏览器的地址栏中输入 URL，例如 https://www.example.com。
2. 浏览器解析 URL，提取出主机名（www.example.com）和协议（https）。
3. 浏览器通过域名系统（DNS）将主机名解析为对应的 IP 地址，以便建立网络连接。
4. 浏览器通过网络协议（如 HTTP 或 HTTPS）创建一个网络连接，连接到目标服务器的 IP 地址。
5. 浏览器向服务器发送 HTTP 请求，请求获取网页内容。请求包括请求方法（GET、POST 等）、路径（URL 中的路径部分）、请求头（包含浏览器相关信息和用户的一些请求参数）等。
6. 目标服务器接收到浏览器发送的 HTTP 请求，根据请求内容生成相应的响应。
7. 服务器将响应以 HTTP 报文的形式发送回浏览器，包括响应状态码、响应头和响应体。
8. 浏览器接收到服务器的响应，开始处理响应数据。
9. 浏览器首先解析响应头，获取服务器返回的元数据信息，如响应状态码、内容类型等。
10. 浏览器根据响应头中的内容类型确定如何处理响应体的数据。如果是 HTML 文档，浏览器会解析 HTML，构建 DOM 树，并开始渲染网页内容。
11. 浏览器根据 HTML 文档中的 CSS 样式和 JavaScript 脚本等，进行页面布局和交互处理。
12. 最终，浏览器将解析渲染后的网页内容显示给用户。

从程序的角度来看，涉及的主要程序和数据处理过程包括：

- 浏览器程序：负责解析 URL、发送 HTTP 请求、接收和解析服务器响应、渲染和显示网页内容。
- 域名系统（DNS）：将主机名解析为 IP 地址。
- 网络协议栈：负责建立网络连接，并将 HTTP 请求和响应通过网络传输。
- 服务器程序：接收浏览器发送的 HTTP 请求，根据请求生成响应，并将响应发送回浏览器。

数据处理过程包括：

- URL 解析：从输入的 URL 中提取出主机名和协议。
- DNS 解析：将主机名解析为对应的 IP 地址。
- HTTP 请求生成和发送：根据 URL 和用户的请求生成 HTTP 请求，并发送给目标服务器。
- HTTP 响应接收和解析：接收服务器发送的 HTTP 响应，并解析出响应状态码、响应头和响应体。
- 响应数据

处理：根据响应头中的内容类型，确定如何处理响应体的数据，例如解析 HTML、渲染页面等。

总之，从 PC 浏览器地址栏输入 URL 到浏览器中显示网页内容，涉及多个程序的协作和数据处理过程，包括 URL 解析、DNS 解析、HTTP 请求和响应的发送和接收，以及浏览器对响应数据的解析和页面渲染等。

12. 下载一个 Web 页面，在控制台输出该页面的原始 HTML。

思路：从命令行读取一个 URL（http://www.oreilly.com），从这个 URL 打开一个 InputStream，将此 InputStream 串链到默认编码方式的 InputStreamReader，使用其 read()方法从文件读取连续的字符，将字符在控制台显示输出。

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import java.nio.charset.StandardCharsets;

public class WebPageDownloader {
    public static void main(String[] args) {
        String url = "http://www.oreilly.com";

        try {
            URL webpageUrl = new URL(url);
            InputStream inputStream = webpageUrl.openStream();
            InputStreamReader reader = new
InputStreamReader(inputStream, StandardCharsets.UTF_8);
            BufferedReader bufferedReader = new
BufferedReader(reader);

            String line;
            while ((line = bufferedReader.readLine()) != null) {
                System.out.println(line);
            }

            bufferedReader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述代码中，我们使用 URL 类创建了一个 webpageUrl 对象，然后通过 openStream()方法获取与该 URL 连接的输入流。接下来，我们使用 InputStreamReader 将输入流与默认字符编码（UTF-8）的 InputStreamReader

相连接。然后，我们使用 BufferedReader 逐行读取 HTML 内容，并将每行输出到控制台。

13. 编写一个名为 ClipherSuites 的类，实现以下功能：
    1. 生成安全 socket （协议为 https），端口为：443
    2. 按行打印出当前 socket 所支持所有密码组
    3. 并输出密码组的个数

```
import java.io.*;
import javax.net.ssl.*;

public class ClipherSuites {

  public static void main(String[] args) {

    if (args.length == 0) {
      System.out.println("Usage: java ClipherSuites host");
      return;
    }

    int port = 443; // default https port
    String host = args[0];

    SSLSocketFactory factory
        = (SSLSocketFactory) SSLSocketFactory.getDefault();
    SSLSocket socket = null;
 try {
   socket = (SSLSocket) factory.createSocket(host, port);

   String[] supported = socket.getSupportedCipherSuites();
   for(String s : supported)
     System.out.println(s);
   System.out.println("There are " + supported.length + "clipher suites.");

 } catch (IOException ex) {
   System.err.println(ex);
 } finally {
    try {
      if (socket != null) socket.close();
    } catch (IOException e) {}
 }
```