Advanced Java Topics

Topics

- Encryption
- Testing
- Regular Expressions
- Maps
- Lambda Expressions
- HttpRequest
- Nonblocking Server Using Socket Channels
- Asynchronous Server Using Asynchronous Socket Channels
- Deployment
- Web Sockets

Encryption and Testing

Maven

- We will use the Maven software project management tool for testing encryption
- We can install the Maven package manager using
  sudo apt-get install maven
- Note that the Ubuntu version of Maven is not necessarily up to date; maven.apache.org has the latest version

Exercise

- Please install Maven on your VM now
- Then enter the following command (on 1 line):

  mvn archetype:generate -DgroupId=ca.camosun.ICS226 -DartifactId=EncryptionExample -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false

Maven Project File Structure

- ./:
- pom.xml
- src/
- target/

- ./src/:
- main/
- test/

Maven Project File Structure - Main Branch

- ./src/main/:
- java/
- resources/

Maven Project File Structure - Test Branch

- ./src/test/:
- java/
- resources/

pom.xml

- Describes the project environment, including dependencies and output

pom.xml - General

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>ca.camosun.ICS226</groupId>

    <artifactId>EncryptionExample</artifactId>

    <version>1.0</version>

    <packaging>jar</packaging>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
```

- Your pom may be different

pom.xml - Dependencies

```
<dependencies>

    <dependency>
        <groupId>com.google.crypto.tink</groupId>
        <artifactId>tink</artifactId>
        <version>1.5.0</version>
    </dependency>

    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.7.0</version>
        <scope>test</scope>
    </dependency>

</dependencies>
```

pom.xml - Plugins

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.4</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer
">
                                    <mainClass>EncryptionExample</mainClass>
                                </transformer>
                            </transformers>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
```

pom.xml - Plugins

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.0.0-M5</version>
                <configuration>
                    <argLine>
                        --illegal-access=permit
                    </argLine>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

Encryption

```
import java.io.*;
import java.security.GeneralSecurityException;
import java.util.*;
import com.google.crypto.tink.Aead;
import com.google.crypto.tink.aead.AeadConfig;
import com.google.crypto.tink.aead.AesGcmKeyManager;
import com.google.crypto.tink.CleartextKeysetHandle;
import com.google.crypto.tink.JsonKeysetReader;
import com.google.crypto.tink.JsonKeysetWriter;
import com.google.crypto.tink.KeysetHandle;
```

Encryption

```
public class EncryptionExample {

    protected Aead aead; // Used to encrypt/decrypt
    protected KeysetHandle handle; // Used to manage a key

    public EncryptionExample() throws GeneralSecurityException {
        // Initialize Tink
        AeadConfig.register();

        // Generate a key securely
        this.handle = KeysetHandle.generateNew(AesGcmKeyManager.aes128GcmTemplate());

        // Generate an object that encrypts/decrypts using the key
        aead = this.handle.getPrimitive(Aead.class);
    }
```

Encryption

```
    public String encrypt(String plainText) throws GeneralSecurityException {
        return new String(Base64.getEncoder().encode(aead.encrypt(plainText.getBytes(),
null)));
    }

    public String decrypt(String cipherText) throws GeneralSecurityException {
        return new String(aead.decrypt(Base64.getDecoder().decode(cipherText.getBytes()),
null));
    }

    public void saveKey(String filename) throws IOException, GeneralSecurityException {
        CleartextKeysetHandle.write(this.handle, JsonKeysetWriter.withFile(new
File(filename)));
    }

    public void loadKey(String filename) throws IOException, GeneralSecurityException {
        this.handle = CleartextKeysetHandle.read(JsonKeysetReader.withFile(new
File(filename)));
        aead = this.handle.getPrimitive(Aead.class);
    }

}
```

Sample Test File

```
import java.io.FileNotFoundException;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class TestEncryptionExample {

    @Test
    public void testCrypto() throws Exception {
        EncryptionExample e = new EncryptionExample();
        Assertions.assertEquals("Hello World", e.decrypt(e.encrypt("Hello World")));
    }

}
```

Maven Commands

- mvn test # Run only the tests
- mvn package # Create a jar (includes testing phase)
- java -jar target/EncryptionExample-1.0.jar # Run a jar file
- mvn clean # Remove generated files

Exercise

- Copy the pom and the Java files from D2L into the correct directories
- Make sure that `mvn test` passes
- Ignore the illegal reflective access error; this is a known bug in this version of Maven
- Modify the files so that `java -jar target/EncryptionExample-1.0.jar e filename keyfile` generates a key and saves it in *keyFileName*, then reads in the file *fileName* and saves the encrypted version in *fileName.encrypted* and `java -jar target/EncryptionExample-1.0.jar d filename.encrypted keyfile` saves a decrypted version of fileName to *fileName.encrypted.decrypted*

Regular Expressions

Regular Expressions

- Java supports regular expression searches
  - The *Pattern* class is used to describe the regular expression
  - The *Matcher* class is used to apply the regular expression described by the *Pattern* class to a particular string

Pattern Class

- Format:

```
Pattern pattern = Pattern.compile(regex, options);
```

- Example Problem:
  Consider a data file with structure
  `EmployeeID,FirstName,LastName,Street,City`
  and look for all streets that contain the word *Drive*

- Answer:

```
Pattern pattern = Pattern.compile(",[^,]*Drive[^,]*,[^,]*$");
```

12345678,Kim,North,Oakridge Drive SW,Vancouver

Pattern Class

- Format:

```
Pattern pattern = Pattern.compile(regex, options);
```

- Example Problem:
  Consider a data file with structure
  `EmployeeID,FirstName,LastName,Street,City`
  and look for all streets that contain the word *Drive*

- Answer:

```
Pattern pattern = Pattern.compile(",[^,]*Drive[^,]*,[^,]*$");
```

Comma

12345678,Kim,North,Oakridge Drive SW,Vancouver

Pattern Class

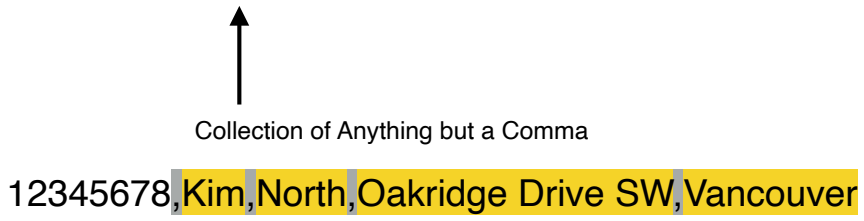
- Format:

```
Pattern pattern = Pattern.compile(regex, options);
```

- Example Problem:
  Consider a data file with structure
  `EmployeeID,FirstName,LastName,Street,City`
  and look for all streets that contain the word *Drive*

- Answer:

```
Pattern pattern = Pattern.compile(",[^,]*Drive[^,]*,[^,]*$");
```



Pattern Class

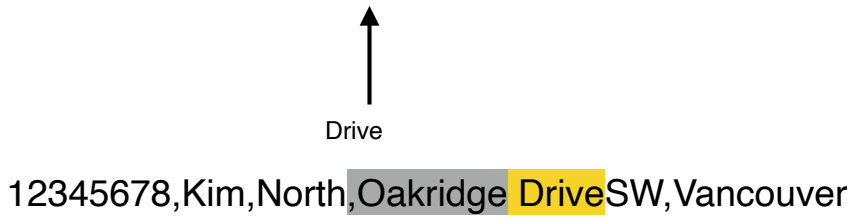
- Format:

```
Pattern pattern = Pattern.compile(regex, options);
```

- Example Problem:
  Consider a data file with structure
  `EmployeeID,FirstName,LastName,Street,City`
  and look for all streets that contain the word *Drive*

- Answer:

```
Pattern pattern = Pattern.compile(",[^,]*Drive[^,]*,[^,]*$");
```



Pattern Class

- Format:

```
Pattern pattern = Pattern.compile(regex, options);
```

- Example Problem:
  Consider a data file with structure
  `EmployeeID,FirstName,LastName,Street,City`
  and look for all streets that contain the word *Drive*

- Answer:

```
Pattern pattern = Pattern.compile(",[^,]*Drive[^,]*,[^,]*$");
```

Collection of Anything but a Comma

12345678,Kim,North,Oakridge Drive SW,Vancouver

Pattern Class

- Format:

```
Pattern pattern = Pattern.compile(regex, options);
```

- Example Problem:
  Consider a data file with structure
  `EmployeeID,FirstName,LastName,Street,City`
  and look for all streets that contain the word *Drive*

- Answer:

```
Pattern pattern = Pattern.compile(",[^,]*Drive[^,]*,[^,]*$");
```

12345678,Kim,North,Oakridge Drive SW,Vancouver

Pattern Class

- Format:

```
Pattern pattern = Pattern.compile(regex, options);
```

- Example Problem:
  Consider a data file with structure
  `EmployeeID,FirstName,LastName,Street,City`
  and look for all streets that contain the word *Drive*

- Answer:

```
Pattern pattern = Pattern.compile(",[^,]*Drive[^,]*,[^,]*$");
```

Pattern Class

- Format:

```
Pattern pattern = Pattern.compile(regex, options);
```

- Example Problem:
  Consider a data file with structure
  `EmployeeID,FirstName,LastName,Street,City`
  and look for all streets that contain the word *Drive*

- Answer:

```
Pattern pattern = Pattern.compile(",[^,]*Drive[^,]*,[^,]*$");
```

12345678,Kim,North,Oakridge Drive SW,Vancouver

Matcher Class

- Format:

```
Matcher matcher = pattern.matcher(searchstring);
while(matcher.find())
{
    System.out.println(matcher.group());
}
```

- Applies the previously-compiled pattern to *searchstring*
- While there are matches, prints out the matches one at a time
- e.g.,

```
,Oakridge Drive SW,Vancouver
,Iris Drive,Quebec City
,Breckenridge Drive,Oakville
,Baywind Drive,Winnipeg
,Chiniak Bay Drive,Brantford
```

Refining the Regular Expression

- Say we only want to print out the name of the street (i.e., without the city), if it contains the word *Drive*:

```
import java.util.regex.*
Pattern pattern = Pattern.compile(",([^,]*Drive[^,]*),[^,]*$");
Matcher matcher = pattern.matcher(searchstring);
while(matcher.find())
{
    System.out.println(matcher.group(1));
}
```

- While there are matches, prints out the expression inside the parentheses, one at a time

- e.g.,

```
Oakridge Drive SW
Iris Drive
Breckenridge Drive
Baywind Drive
Chiniak Bay Drive
```

Refining the Regular Expression Further

- Say we only want to print out the name of the street, if it contains the word *Drive* or *drive* or any other combination of upper and lower case:

```
Pattern pattern = Pattern.compile(",([^,]*Drive[^,]*),[^,]*$", Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher(searchstring);
while(matcher.find())
{
    System.out.println(matcher.group(1));
}
```

Full Regular Expression Example

- Write a class that reads in the name of a file with structure
  `EmployeeID,FirstName,LastName,Street,City`
  and then prints out all street names with the word *Drive* in it; the search should be case insensitive

Full Regular Expression Example

```
import java.io.*;
import java.util.regex.*;

public class Regex {

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Expected a single file name as argument");
            System.exit(-1);
        }

        try (
            BufferedReader fileIn = new BufferedReader(new InputStreamReader(
                new FileInputStream(args[0])));
        ) {
            Pattern pattern = Pattern.compile(",([^,]*Drive[^,]*),[^,]*$",
                Pattern.CASE_INSENSITIVE);
            String line = "";
            while ((line = fileIn.readLine()) != null) {
                Matcher matcher = pattern.matcher(line);
                while(matcher.find())
                {
                    System.out.println(matcher.group(1));
                }
            }
```

Full Regular Expression Example

```
        } catch (Exception e) {
            System.err.println(e.getMessage());
            System.exit(-2);
        }
    }
}
```

On Your Own Exercise

- Write a Java program that takes the name of a database file of structure
  `EmployeeID,FirstName,LastName,Street,City`
  and a name, and then prints out all employees with that given name

Maps

Traditional Approach

- Say we want to write a method that accepts a list of decimal numbers and converts them to a list of hexadecimal numbers

```
public static List<String> convertToHex1(List<String> list) {
    ArrayList<String> convertedList = new ArrayList<String>();
    for (String s : list) {
        try {
            convertedList.add(Integer.toHexString(Integer.valueOf(s)));
        }
        catch (Exception e) {
            // Ignore invalid numbers
        }
    }
    return convertedList;
}
```

Mapping Approach

- Applying the same operation to all elements of an array is so common that many languages now offer a mapping approach

```
public static List<String> convertToHex2(List<String> list) {
    return list
        .stream()
        .map(Integer::parseInt).map(Integer::toHexString)
        .collect(Collectors.toList());
}
```

- *stream()* enables the *map()* function to operate on the list

- *map(mapper)* creates a new stream by applying *mapper* to each element of the underlying stream; because a stream is returned, multiple *map* calls can be made in a row

- *collect(Collectors.toList())* takes the underlying stream and turns it into a *List*

Mapping Approach

- Applying the same operation to all elements of an array is so common that many languages now offer a mapping approach

```
public static List<String> convertToHex2(List<String> list) {
    return list
        .stream()
        .map(Integer::parseInt).map(Integer::toHexString)
        .collect(Collectors.toList());
}
```

- *stream()* enables the *map()* function to operate on the list

- *map(mapper)* creates a new stream by applying *mapper* to each element of the underlying stream; because a stream is returned, multiple *map* calls can be made in a row

- *collect(Collectors.toList())* takes the underlying stream and turns it into a *List*

Mapping Approach

- Applying the same operation to all elements of an array is so common that many languages now offer a mapping approach

```
public static List<String> convertToHex2(List<String> list) {
    return list
        .stream()
        .map(Integer::parseInt).map(Integer::toHexString)
        .collect(Collectors.toList());
}
```

- *stream()* enables the *map()* function to operate on the list
- *map(mapper)* creates a new stream by applying *mapper* to each element of the underlying stream; because a stream is returned, multiple *map* calls can be made in a row
- *collect(Collectors.toList())* takes the underlying stream and turns it into a *List*

Lambda Expressions

Lambda Expression Motivation 1

- Integer has a built-in *toHexString* method; but what if we want to write a custom method and apply that to each element of an array?

- e.g., return a list of booleans indicating whether the element is even or odd

```
public static List<Boolean> evenOrOdd(List<Integer> list) {
    ArrayList<Boolean> convertedList = new ArrayList<Boolean>();
    for (Integer num : list) {
        convertedList.add(new Boolean(Integer.valueOf(num) % 2 == 0));
    }
    return convertedList;
}
```

Lambda Expression Motivation 2

- Now what if we want to use Maps instead?
- We could write our own conversion method as part of the class containing the *evenOrOdd2* method

```
public static Boolean isEven(Integer i) {
    return i % 2 == 0;
}

public static List<Boolean> evenOrOdd2(List<Integer> list) {
    return list.stream().map(Maps::isEven).collect(Collectors.toList());
}
```

- Note that *isEven* is *static*

Lambda Expression

- Again, this is so common that there is another approach: Lambda expressions

```
public static List<Boolean> evenOrOdd3(List<Integer> list) {
    return list.stream().map(num -> num % 2 == 0).collect(Collectors.toList());
}
```

- Format is *(argument list) -> body of method*

- No method name is required (or allowed)

Exercise

- Using lambda expressions and maps, write a Java program that reads in a list of integers and returns *true* for every number that is a power of 2, *false* otherwise

- e.g.,
  java Maps 1 2 3 4 5 6 7 8 9 10
  should return
  [true, true, false, true, false, false, false, true, false, false]

- Note that *num & (num - 1) == 0* is true if and only if *num* is a power of 2

Solution

```
import java.util.*;
import java.util.stream.*;

public class Maps {

  public static List<Boolean> power2(List<Integer> list) {
    return list.stream().map(num -> ((num & (num - 1)) == 0)).collect(Collectors.toList());
  }

  public static void main(String[] args) {
    ArrayList<Integer> currentMatches = new ArrayList<Integer>();
    for (String arg : args) {
      currentMatches.add(Integer.valueOf(arg));
    }
    System.out.println(power2(currentMatches));
  }
}
```

HttpRequest

Download a Page - Synchronous Approach

- Traditionally, downloading a web page was a synchronous (blocking) operation

```
try (
    Socket socket = new Socket(args[0], 80);
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));
) {
    String message = "";
    out.println("GET / HTTP/1.1\nHost:" + args[0] + "\nAccept-Language: en-
us\n\n");
    while(true) {
        message = in.readLine();
        if (message == null) {
            break;
        }
        System.out.println(message);
    }
} catch (Exception e) {
    System.err.println(e);
    System.exit(-1);
}
```

Synchronous Approach

- This approach is fine if network response is quick and the app has nothing else to do in the meantime
- In other cases, it makes more sense to let the GET request complete in the background and to go on to other tasks in the meantime
- e.g.,
  - crawling a website
  - search engines tend to following all links on a web site in order to build a searchable index
  - blocking GET requests do not scale, given the size of the Internet

Asynchronous Approach

- Starting with JDK 11, Java officially supports asynchronous web requests
- Consists of 2 parts:
  - HttpRequest - sets up the HTTP request parameters, such as the URL and any headers
  - HttpClient - makes the actual HTTP request

HttpRequest

```
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(URI.create("https://www.google.ca"))
    .build();
```

- Follows a builder pattern:
  - *newBuilder()* creates the builder; allows subsequent calls to define what web server should be contacted and what headers should be set
  - *uri(uri)* is used to indicate what web server to contact
  - *setHeader(String name, String value)* is used to set a particular header (e.g., preferred language)
  - *build()* is used to return the completed construction (in this case, an HttpRequest)

HttpRequest

```
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(URI.create("https://www.google.ca"))
    .build();
```

- Follows a builder pattern:
  - *newBuilder()* creates the builder; allows subsequent calls to define what web server should be contacted and what headers should be set
  - *uri(uri)* is used to indicate what web server to contact
  - *setHeader(String name, String value)* is used to set a particular header (e.g., preferred language)
  - *build()* is used to return the completed construction (in this case, an HttpRequest)

HttpRequest

```
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(URI.create("https://www.google.ca"))
    .build();
```

- Follows a builder pattern:
  - *newBuilder()* creates the builder; allows subsequent calls to define what web server should be contacted and what headers should be set
  - *uri(uri)* is used to indicate what web server to contact
  - *setHeader(String name, String value)* is used to set a particular header (e.g., preferred language); optional
  - *build()* is used to return the completed construction (in this case, an HttpRequest)

HttpRequest

```
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(URI.create("https://www.google.ca"))
    .build();
```

- Follows a builder pattern:
  - *newBuilder()* creates the builder; allows subsequent calls to define what web server should be contacted and what headers should be set
  - *uri(uri)* is used to indicate what web server to contact
  - *setHeader(String name, String value)* is used to set a particular header (e.g., preferred language)
  - *build()* is used to return the completed construction (in this case, an HttpRequest)

HttpClient

- e.g.,

```
HttpClient client = HttpClient.newHttpClient();
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

- *sendAsync(request, handler)* returns immediately with a *CompletableFuture* instance; it requires an *HttpRequest* and a response handler (e.g., *HttpResponse.BodyHandlers.ofString()*, which returns a handler that will store the response as a String)
- *thenApply(method)* applies the given method to the returned response (e.g., *HttpResponse::body* returns the body in a form that can be manipulated further
- *thenAccept(method)* applies the given method to the returned response (e.g., *System.out::println*) but no further processing is possible (*println* returns void)

HttpClient

- e.g.,

```
HttpClient client = HttpClient.newHttpClient();
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

- *sendAsync(request, handler)* returns immediately with a *CompletableFuture* instance; it requires an *HttpRequest* and a response handler (e.g., *HttpResponse.BodyHandlers.ofString()*, which returns a handler that will store the response as a String)

- *thenApply(method)* applies the given method to the returned response (e.g., *HttpResponse::body* returns the body in a form that can be manipulated further

- *thenAccept(method)* applies the given method to the returned response (e.g., *System.out::println*) but no further processing is possible (*println* returns void)

HttpClient

- e.g.,

```
HttpClient client = HttpClient.newHttpClient();
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

- *sendAsync(request, handler)* returns immediately with a *CompletableFuture* instance; it requires an *HttpRequest* and a response handler (e.g., *HttpResponse.BodyHandlers.ofString()*, which returns a handler that will store the response as a String)

- *thenApply(method)* applies the given method to the returned response (e.g., *HttpResponse::body* returns the body in a form that can be manipulated further

- *thenAccept(method)* applies the given method to the returned response (e.g., *System.out::println*) but no further processing is possible (*println* returns void)

HttpClient

- Incidentally, this approach can also be made synchronous
- e.g.,

```
HttpClient client = HttpClient.newHttpClient();
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println)
    .join();
```

- *join()* waits until the operation has completed

Download a Page - Asynchronous Approach

```
import java.net.*;
import java.net.http.*;

public class Request {

    public static void main(String[]args) {
        if (args.length != 1) {
            System.err.println("1 URL required");
            System.exit(-1);
        }

        HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create("https://" + args[0]))
            .build();

        HttpClient client = HttpClient.newHttpClient();
        client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
            .thenApply(HttpResponse::body)
            .thenAccept(System.out::println)
            .join(); // Need join here since we otherwise drop out of main()
    }
}
```

Take a deep breath...

...really deep...

Asynchronous Approach Using Maps

- Going back to the web crawling example, given a list of URLs, we want to follow multiple links simultaneously and finish when all links have been followed

```
List<HttpRequest> requests = urlList
    .stream()
    .map(url -> HttpRequest.newBuilder(url).build())
    .collect(Collectors.toList());
```

- Recall that *stream -> map -> collect* is used to apply an operation to each element in a list

- Recall that *newBuilder(url).build()* is used to create an HttpRequest that can be used by HttpClient

Asynchronous Approach Using Maps

- Going back to the web crawling example, given a list of URLs, we want to follow multiple links simultaneously and finish when all links have been followed

```
List<HttpRequest> requests = urlList
    .stream()
    .map(url -> HttpRequest.newBuilder(url).build())
    .collect(Collectors.toList());
```

- Recall that *stream -> map -> collect* is used to apply an operation to each element in a list
- Recall that *newBuilder(url).build()* is used to create an HttpRequest that can be used by HttpClient

Asynchronous Approach Using Maps

- Now that we have a list of *HttpRequests*, we set up *HttpClients* to access the URLs concurrently using the pattern *sendAsync -> thenApply -> thenAccept*

```
HttpClient client = HttpClient.newHttpClient();
CompletableFuture<?>[] asyncs = requests
    .stream()
    .map(request -> client
        .sendAsync(request, HttpResponse.BodyHandlers.ofString())
        .thenApply(HttpResponse::body)
        .thenAccept(System.out::println))
    .toArray(CompletableFuture<?>[]::new);

CompletableFuture.allOf(asyncs).join();
```

- *stream -> map -> toArray* is used to ensure that each *HttpRequest* is passed to an *HttpClient* to create a list of asynchronous network requests

- *map* itself takes the *HttpRequest* and converts it to an asynchronous call which will convert the response to a *String*, then shift the focus to the response body, then print it out

Asynchronous Approach Using Maps

- Now that we have a list of *HttpRequests*, we set up *HttpClients* to access the URLs concurrently using the pattern *sendAsync -> thenApply -> thenAccept*

```
HttpClient client = HttpClient.newHttpClient();
CompletableFuture<?>[] asyncs = requests
    .stream()
    .map(request -> client
        .sendAsync(request, HttpResponse.BodyHandlers.ofString())
        .thenApply(HttpResponse::body)
        .thenAccept(System.out::println))
    .toArray(CompletableFuture<?>[]::new);

CompletableFuture.allOf(asyncs).join();
```

- *stream -> map -> toArray* is used to ensure that each *HttpRequest* is passed to an *HttpClient* to create a list of asynchronous network requests

- *map* itself takes the *HttpRequest* and converts it to an asynchronous call which will convert the response to a *String*, then shift the focus to the response body, then print it out

Asynchronous Approach Using Maps

- Now that we have a list of *HttpRequests*, we set up *HttpClients* to access the URLs concurrently using the pattern *sendAsync -> thenApply -> thenAccept*

```
HttpClient client = HttpClient.newHttpClient();
CompletableFuture<?>[] asyncs = requests
    .stream()
    .map(request -> client
        .sendAsync(request, HttpResponse.BodyHandlers.ofString())
        .thenApply(HttpResponse::body)
        .thenAccept(System.out::println))
    .toArray(CompletableFuture<?>[]::new);

CompletableFuture.allOf(asyncs).join();
```

- *stream -> map -> toArray* is used to ensure that each *HttpRequest* is passed to an *HttpClient* to create a list of asynchronous network requests
- *map* itself takes the *HttpRequest* and converts it to an asynchronous call which will convert the response to a *String*, then shift the focus to the response body, then print it out

Asynchronous Approach Using Maps

- Now that we have a list of *HttpRequests*, we set up *HttpClients* to access the URLs concurrently using the pattern *sendAsync -> thenApply -> thenAccept*

```
HttpClient client = HttpClient.newHttpClient();
CompletableFuture<?>[] asyncs = requests
    .stream()
    .map(request -> client
        .sendAsync(request, HttpResponse.BodyHandlers.ofString())
        .thenApply(HttpResponse::body)
        .thenAccept(System.out::println))
    .toArray(CompletableFuture<?>[]::new);

CompletableFuture.allOf(asyncs).join();
```

- *stream -> map -> toArray* is used to ensure that each *HttpRequest* is passed to an *HttpClient* to create a list of asynchronous network requests

- *map* itself takes the *HttpRequest* and converts it to an asynchronous call which will convert the response to a *String*, then shift the focus to the response body, then print it out

Asynchronous Approach Using Maps

```
import java.io.*;
import java.net.*;
import java.net.http.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

public class Crawl {

    public static void main(String[]args) {
            ArrayList<URI> urlList = new ArrayList<URI>();
            for(String url : args) {
                urlList.add(URI.create("http://" + url));
            }

            List<HttpRequest> requests = urlList
                .stream()
                .map(url -> HttpRequest.newBuilder(url).build())
                .collect(Collectors.toList());
```

Asynchronous Approach Using Maps

```
        HttpClient client = HttpClient.newHttpClient();
        CompletableFuture<?>[] asyncs = requests
            .stream()
            .map(request -> client
                .sendAsync(request, HttpResponse.BodyHandlers.ofString())
                .thenApply(HttpResponse::body)
                .thenAccept(System.out::println))
            .toArray(CompletableFuture<?>[]::new);

        CompletableFuture.allOf(asyncs).join();
    }
}
```

Exercise

- Write a Java program that takes a list of names from the command line, and then repeatedly prints out these names, each from its own async

- Hint 1: You must write the infinite loop inside a CompletableFuture, or the async will not yield CompletableFuture.supplyAsync(() -> { // loop goes in here }

- Hint 2: Use TimeUnit.SECONDS.sleep(1) after every println

```
instructor@ics226:~/Maps$ javac Print.java ; java Print abc def ghi
[abc, def, ghi]
abc
def
ghi
abc
def
ghi
abc
def
ghi
abc
def
ghi
...
```

Solution

```
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

public class Print {
  public static void main(String[]args) {
      ArrayList<String> nameList = new ArrayList<String>();
      for(String name : args) {
        nameList.add(name);
      }
      System.out.println(nameList);
      CompletableFuture<?>[] asyncs = nameList
        .stream()
        .map((name) -> CompletableFuture.supplyAsync(() -> {
               while (true) {
                System.out.println(name);
                try {
                 TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e) {
                 System.err.println(e);
                }
               }}))
        .toArray(CompletableFuture<?>[]::new);
      CompletableFuture.allOf(asyncs).join();
  }}
```

Nonblocking Server Using Socket Channels

Nonblocking Servers

- As mentioned before, the *accept* and *readLine* operations are blocking, i.e., they do not return until a connection has been accepted or a newline has been read
- To avoid locking up the server, we used threads to serve individual network connections
- In Java, it is possible to network in a non-blocking way by using channels
- Channels connect network streams to buffers, and can be configured to avoid blocking

Sample Run (Nonblocking Server)

```
# java Client localhost 12345 Hi

main Hi

main Hi

main Hi

...
```

```
# java Client localhost 12345 Bye

main Bye

main Bye

main Bye

...
```

Setting Up a Nonblocking Channel

- Open a nonblocking channel

```
ServerSocketChannel channel = ServerSocketChannel.open();
channel.configureBlocking(false);
```

- Associate the channel with a server socket

```
ServerSocket serverSocket = channel.socket();
serverSocket.bind(new InetSocketAddress(this.HOST, this.port));
```

- Register to be notified when a connection is accepted

```
Selector selector = Selector.open();
channel.register(selector, SelectionKey.OP_ACCEPT);
```

Receiving Events

- Await an event (we can eliminate this wait by using *selectNow*, but then we end up with a busy loop); but unlike waiting for a *accept* or *readLine*, this block is fairly innocuous (because it means that no one is trying to contact the server)

```
selector.select();
```

- Determine the event (new connection or incoming data) and act on it

```
Set keys = selector.selectedKeys();
Iterator i = keys.iterator();
while (i.hasNext()) {
    SelectionKey key = (SelectionKey) i.next();
    i.remove();
    if (key.isAcceptable()) {
        acceptConnection(key);
    }
    else if (key.isReadable()) {
        readConnection(key);
    }}
```

Not yet set up; must do this when the connection has been accepted

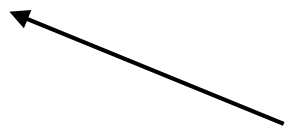
Accepting Connections

- Get the *SocketChannel* from the *ServerSocketChannel* and accept the connection

```
void acceptConnection(SelectionKey key) {
    try {
        ServerSocketChannel channel = (ServerSocketChannel)
            key.channel();
        SocketChannel socketChannel = (SocketChannel)
            channel.accept();
```

- Register for non-blocking read operations

```
        socketChannel.configureBlocking(false);
        socketChannel.register(key.selector(),
            SelectionKey.OP_READ);
    } catch (Exception e) {
        System.err.println(e);
    }
}
```



Now key.isReadable() will work

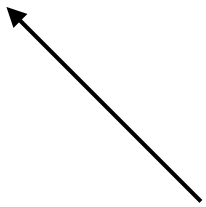
Reading/Writing

- Access the buffer associated with the network connection

```
void readConnection(SelectionKey key) {
    try {
        SocketChannel channel = (SocketChannel) key.channel();
        ByteBuffer readBuffer = ByteBuffer.allocate(BUF_SIZE);
        int bytesCount = channel.read(readBuffer);
        if (bytesCount > 0) {
```

- Echo back the message + Thread ID

```
            String msg = Thread.currentThread().getName()
                + " " + new String(readBuffer.array());
            channel.write(ByteBuffer.wrap(msg.getBytes()));
        }
    } catch (Exception e) {
        System.err.println(e);
    }
}
```



Converts byte array into a ByteBuffer

A Nonblocking Server

```
import java.nio.ByteBuffer;
import java.io.*;
import java.net.*;
import java.nio.channels.*;
import java.util.*;

public class Server2 {
    protected final int BUF_SIZE = 1024;
    protected final String HOST = "";
    protected int port;

    public Server2(int port) {
        this.port = port;
    }

    void acceptConnection(SelectionKey key) {
        try {
            ServerSocketChannel channel = (ServerSocketChannel) key.channel();
            SocketChannel socketChannel = (SocketChannel) channel.accept();
            socketChannel.configureBlocking(false);
            socketChannel.register(key.selector(), SelectionKey.OP_READ);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
```

A Nonblocking Server

```
void readConnection(SelectionKey key) {
    try {
        SocketChannel channel = (SocketChannel) key.channel();
        ByteBuffer readBuffer = ByteBuffer.allocate(BUF_SIZE);
        int bytesCount = channel.read(readBuffer); // does not reset buf index
        if (bytesCount > 0) {
                String msg = Thread.currentThread().getName() + " " +
                new String(readBuffer.array());
            channel.write(ByteBuffer.wrap(msg.getBytes()));
        }
    } catch (Exception e) {
        System.err.println(e);
    }
}

public void serve() {
    try {
        ServerSocketChannel channel = ServerSocketChannel.open();
        channel.configureBlocking(false);
        ServerSocket serverSocket = channel.socket();
        serverSocket.bind(new InetSocketAddress(this.HOST, this.port));
        Selector selector = Selector.open();
        channel.register(selector, SelectionKey.OP_ACCEPT);
```

A Nonblocking Server

```
        while(true) {
            selector.select();
            Set keys = selector.selectedKeys();
            Iterator i = keys.iterator();
            while (i.hasNext()) {
                SelectionKey key = (SelectionKey) i.next();
                i.remove();
                if (key.isAcceptable()) {
                    acceptConnection(key);
                }
                else if (key.isReadable()) {
                    readConnection(key);
                }
            }
        }
    } catch (Exception e) {
        System.err.println(e);
        System.exit(-3);
    }
}

public static void main(String[] args) {
    if (args.length != 1) {
        System.err.println("Need <port>");
```

A Nonblocking Server

```
            System.exit(-99);
        }

        Server2 s = new Server2(Integer.valueOf(args[0]));
        s.serve();
    }
}
```

Asynchronous Server Using Asynchronous Socket Channels

An Asynchronous Server

- Combines ideas of the threaded server and the non-blocking server
- Individual requests are handled on separate threads

Sample Run (Asynchronous Server)

```
# java Client localhost 12345 Hi

Thread-3 Hi

Thread-1 Hi

Thread-2 Hi

Thread-2 Hi

Thread-1 Hi

Thread-3 Hi

...
```

```
# java Client localhost 12345 Bye

Thread-3 Bye

Thread-2 Bye

Thread-1 Bye

Thread-2 Bye

Thread-3 Bye

Thread-2 Bye

...
```

Accepting Connections

- Accepting connections is done on a separate thread

```
        try (
            AsynchronousServerSocketChannel serverSocket =
AsynchronousServerSocketChannel.open();
        ) {
            serverSocket.bind(new InetSocketAddress(this.HOST,
this.port));
            CompletionHandler<AsynchronousSocketChannel,Void>
handler = new CompletionHandler<AsynchronousSocketChannel,Void>() {
                public void completed(AsynchronousSocketChannel
channel, Void attachment) {
                    serverSocket.accept(null, this);
                    delegate(channel);
                }
                public void failed(Throwable e, Void att) {
                        System.err.println(e);
                }
            };
            serverSocket.accept(null, handler);
            Thread.currentThread().join();
        } catch (Exception e) {
```

Launches a Thread that waits for a connection, then executes the handler

Blocks until there are no threads left (forever in this case)

Reading/Writing

- Reading the buffer is also an asynchronous operation

```
    void delegate(AsynchronousSocketChannel channel) {
        try {
            ByteBuffer readBuffer = ByteBuffer.allocate(BUF_SIZE);
            CompletionHandler<Integer,Void> handler =
                new CompletionHandler<Integer,Void>() {
                public void completed(Integer result, Void
attachment) {
                    if (result <= 0) {
                        try {
                            channel.close();
                        } catch (Exception e) {
                            System.err.println(e);
                        }
                        return;
                    }
```

Number of bytes read
(-1 = end of stream)

Reading/Writing

- Continued from previous slide

Without this, the readBuffer will overflow because old message won't be cleared

```
                    String msg = Thread.currentThread().getName() +
" " + new String(readBuffer.array());
                     channel.write(ByteBuffer.wrap(msg.getBytes()));
                     readBuffer.clear();
                     channel.read(readBuffer, null, this);
                }
                public void failed(Throwable e, Void att) {
                          System.err.println(e);
                }

            };
            channel.read(readBuffer, null, handler);
```

Launches a Thread that is called when there is data to be read

An Asynchronous Server

```
import java.io.*;
import java.net.*;
import java.nio.channels.*;
import java.nio.ByteBuffer;
import java.util.concurrent.*;

public class Server3 {

    protected final int BUF_SIZE = 1024;
    protected final String HOST = "";
    protected int port;

    public Server3(int port) {
        this.port = port;
    }

    void delegate(AsynchronousSocketChannel channel) {
        try {
            ByteBuffer readBuffer = ByteBuffer.allocate(BUF_SIZE);
            CompletionHandler<Integer,Void> handler =
                new CompletionHandler<Integer,Void>() {
                public void completed(Integer result, Void attachment) {
                    if (result <= 0) {
                        try {
                            channel.close();
```

An Asynchronous Server

```
                    } catch (Exception e) {
                            System.err.println(e);
                    }
                    return;
                }
                readBuffer.flip();
                String msg = Thread.currentThread().getName() + " " + new
String(readBuffer.array());
                channel.write(ByteBuffer.wrap(msg.getBytes()));
                channel.read(readBuffer, null, this);
            }
            public void failed(Throwable e, Void att) {
                    System.err.println(e);
            }

        };
        channel.read(readBuffer, null, handler);
    } catch (Exception e) {
        System.err.println(e);
        System.exit(-1);
    }
}
```

An Asynchronous Server

```
    public void serve() {
        try (
            AsynchronousServerSocketChannel serverSocket =
AsynchronousServerSocketChannel.open();
        ) {
            serverSocket.bind(new InetSocketAddress(this.HOST, this.port));
            CompletionHandler<AsynchronousSocketChannel,Void> handler = new
CompletionHandler<AsynchronousSocketChannel,Void>() {
                public void completed(AsynchronousSocketChannel channel, Void
attachment) {
                    serverSocket.accept(null, this);
                    delegate(channel);
                }
                public void failed(Throwable e, Void att) {
                            System.err.println(e);
                }
            };
            serverSocket.accept(null, handler);
            Thread.currentThread().join();
        } catch (Exception e) {
            System.err.println(e);
            System.exit(-3);
        }
    }
```

An Asynchronous Server

```
   public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Need <port>");
            System.exit(-99);
        }
        Server3 s = new Server3(Integer.valueOf(args[0]));
        s.serve();
    }
}
```

Deployment Using Tomcat

Tomcat

- We used Django to host a Python web application
- Similarly, Tomcat can be used to host a Java web application

Sample JSP Page

```
<html>

    <body>

        <form action = "Greeting" method = "post">
            First Name: <input type = "text" name = "firstName">
            <br/>
            Last Name: <input type = "text" name = "lastName" />
            <br/>
            <input type = "submit" value = "Submit" />
        </form>

    </body>

</html>
```

- *Greeting* is the name of the Java class that handles this form

Sample Java Class

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Greeting extends HttpServlet
{
        protected void doPost(HttpServletRequest req,
HttpServletResponse res) throws IOException, ServletException {
            String firstName = req.getParameter("firstName");
            String lastName = req.getParameter("lastName");

            PrintWriter writer = res.getWriter();
            res.setContentType("text/html");
            writer.println("Hello " + firstName + " " + lastName);
            writer.close();
        }
}
```

- *doPost* handles *POST* requests (*doGet* handles *GET* requests)
- *req* is the incoming request, *res* the outgoing response

Sample web.xml File

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd" >
<web-app>
    <display-name>Greeting</display-name>

    <servlet>
        <servlet-name>Greeting</servlet-name>
        <servlet-class>Greeting</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Greeting</servlet-name>
        <url-pattern>/Greeting</url-pattern>
    </servlet-mapping>

</web-app>
```

- *servlet* declares the name of the servlet and the name of the class that serves the request
- *servlet-mapping* maps a URL to the servlet (similar to *urls.py*)

Java Web App Launch

- Build the project locally using
  mvn package

- Run the project locally using
  java -jar target/dependency/webapp-runner.jar target/.war*

Heroku Deployment

- *Procfile* should contain
  web: java $JAVA_OPTS -jar target/dependency/webapp-runner.jar --port $PORT target/.war*
- Commit everything to git
- Now run
  - *heroku login*
  - *heroku create # Do this only once per project*
  - *git push heroku master*
  - *heroku open*

Heroku Deployment

- For details, consult https://devcenter.heroku.com/articles/java-webapp-runner

Docker

Docker Deployment

- Amazon's EC2 and DigitalOcean support deployment of docker containers
- Can create servers in docker containers then deploy them via the cloud

Websockets

Web Sockets

- Support socket programming on web pages (e.g., to add chat services and home monitoring to a web page)
- See https://github.com/mdn/samples-server/tree/master/s/websocket-chat for a simple client and server (in NodeJS)

Key Skills

- Test an app using Maven
- Encrypt data using Java
- Conduct regular expression searches in Java
- Use Java maps
- Use Java lambda expressions
- Make asynchronous Java HttpRequests

Key Skills

- Write a non-blocking, single-threaded server that can support multiple simultaneous client connections
- Write an asynchronous server that can support multiple simultaneous client connections
- Deploy a Java app to Heroku