8.4 I/O over the Internet

Up to now, we have performed I/O from standard keyboard input and various types of local files. **We can also collect input from websites on the Internet.** While the `java.io` library deals mainly with *file* I/O, the Java API provides `java.net` package to interact with Internet resources. Specifically, the `java.net.URL` class works with URL objects. URL stands for Uniform Resource Locator. **Each URL is a unique address directing you to an Internet resource, such as a file, image, or website.** For example, www.google.com is the URL for Google's homepage.

To digest the data at an URL, Java needs to *stream* the data at the address to the computer program. Conveniently, the `java.net.URL` class exposes a constructor that takes in a String parameter—the URL address—and creates a `URL` object linked to that address. Then, users can invoke the `URL` object's `openStream()` method to establish a connection between the client application and the Internet resource. This connection gives the computer access to an `InputStream` that it can use to read data from the remote data source. Figure 8-10 depicts the interactions between `java.net.URL`, `java.io`, the client application, and the remote Internet resource.

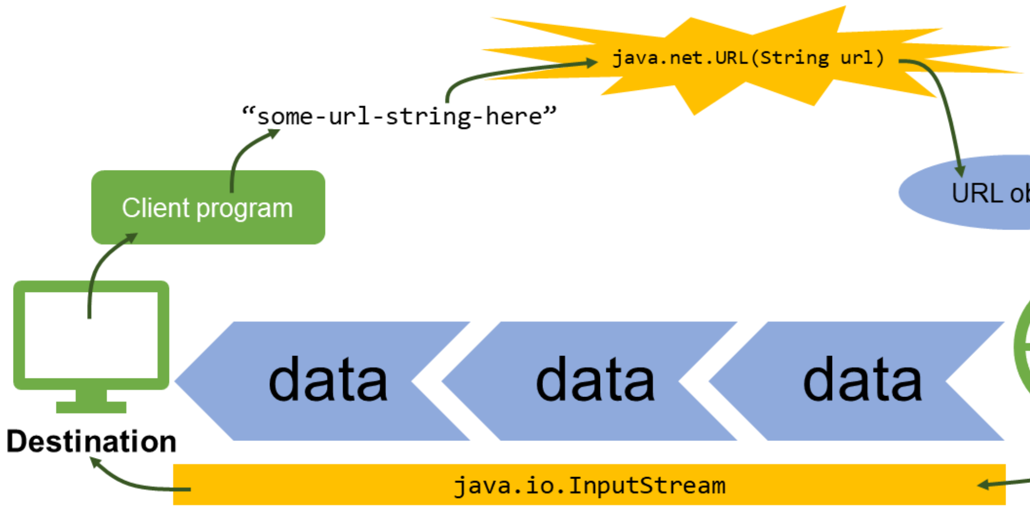

Figure 8-10 Interactions between URL *objects and streams between client program and Internet resource*

8.4.1 I/O from a URL: Example

We'll walk through an example, **JavaIOEx6_URL**, to illustrate how to read input from a website, scrape the website for links, and then write the links to a local file.

JavaIOEx6_URL

Goal: Read and write data from remote Internet resource

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.regex.*;
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;

public class JavaIOEx6_URL {

    public static void main(String[] args) {
        String address =
            "https://en.wikipedia.org/wiki/Java_(programming_language)";

        ArrayList<String> links = readURLForLinks(address);
        writeLinksToFile("files/urls.txt", links);
```

A1

```
        int num = (int) links.stream()
                .filter(item -> item.startsWith("https:"))
                .count();
        System.out.println(num);
```

A2

```
        ArrayList<String> listOfHttps = new ArrayList<>();
        links.stream()
             .filter(item -> item.startsWith("https:"))
             .forEach(item -> listOfHttps.add(item));
           //.forEach((item) -> {System.out.println(item);});

        listOfHttps.stream()
                .forEach((item) -> {System.out.println(item);});
        System.out.println(listOfHttps.size());
    }
```

A3

```
        BufferedImage img = null;
        try {
            URL url = new URL(
                "http://clipart-library.com/data_images/379246.png");
            img = ImageIO.read(in);
            ImageIO.write(img, "png", new File("files/snail.png"));
        } catch(IOException e){
            e.printStackTrace();
        }
```

A4

```java
        try {
            File file = new File("files/snail.png");
            if(!file.exists())
                throw new FileNotFoundException("File not found");
            System.out.println("Does the file exist? " + file.exists());
            System.out.println("The file name is: " + file.getName());
            System.out.println("The path is: " + file.getPath());
            System.out.println("The parent is: " + file.getParent());
            System.out.println("The file has " + file.length() + " bytes");
            System.out.println("Can the file be read? " + file.canRead());
            System.out.println("Can the file be written? " +
                file.canWrite());
            System.out.println("File can be executed? " +
                file.canExecute());
            System.out.println("Absolute path is " +
                file.getAbsolutePath());
            System.out.println("Last modified on " +
                new java.util.Date(file.lastModified()));
        } catch(FileNotFoundException e) {
            e.printStackTrace();
        }
```

A5

```java
    public static ArrayList<String> readURLForLinks(String address) {
        BufferedReader in = null;
        ArrayList<String> links = new ArrayList<String>();
        try {
            URL url = new URL(address);
            in = new BufferedReader(
                    new InputStreamReader(url.openStream()));
            String line;
            String regex = "<a href\\s*=\\s*\"(.*?)\".*?>";
            Pattern pattern = Pattern.compile(regex);
```

B1-1

```java
            while ((line = in.readLine()) !=  null) {
                Matcher matcher = pattern.matcher(line);
                while (matcher.find()) {
                    // extract just the link with a group
                    String link = matcher.group(1);
                    links.add(link); // add link to the ArrayList
                }
            }
```

B1-2

```java
        } catch (MalformedURLException e) {
            System.out.println("Invalid URL");
        }
        catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                in.close();
            } catch (IOException e) {
                System.out.println("Problem closing reader");
            }
        }
        return links;
    }
```

B1-3

```java
    public static void writeLinksToFile(
        String filename, ArrayList<String> links) {

        BufferedWriter out = null;
        try {
            out = new BufferedWriter(new FileWriter(filename));
            for (String link: links) {
                out.write(link);
                out.write('\n');
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (out != null) {
                try {
                    out.flush();
                    out.close();
                } catch (IOException ex) {
                    System.out.println("Problem closing writer");
                }
            }
        }
    }
}
```

B2

First, we need to retrieve information from a given website. The `readURLForLinks()` method does this. We begin at **Cell B1-1**:

```
public static ArrayList<String> readURLForLinks(String address) {
    BufferedReader in = null;
    ArrayList<String> links = new ArrayList<String>();
    try {
        URL url = new URL(address);
        in = new BufferedReader(
                new InputStreamReader(url.openStream()));
        String line;
        String regex = "<a href\\s*=\\s*\"(.*?)\".*?>";
        Pattern pattern = Pattern.compile(regex);
```

Given a URL address, we create a `URL` object named `url`. We open a stream from the Internet resource at that address and pass its contents to an `InputStreamReader`, which is wrapped within a `BufferedReader`. We do this so that while the `InputStreamReader` converts the data into raw bytes, the `BufferedReader` can read the data one line of characters at a time.

To look for links, we use a regular expression that leverages the patterns of HTML text. In HTML, all links will be tagged by `<a href = “. . .”>`. The address that follows `“href”` in quote marks is the link. Web developers may have whitespace around the equals sign, and there may be additional information after the quote marks. By this knowledge, we build a regular expression and compile it into a `Pattern` object.

In **Cell B1-2**, we will begin scraping the webpage for links:

```
while ((line = in.readLine()) !=  null) {
    Matcher matcher = pattern.matcher(line);
    while (matcher.find()) {
        String link = matcher.group(1); // extract just the link with a group
        links.add(link); // add link to the ArrayList
    }
}
```

One line at a time, the program will read the HTML text of the website. At each line, it will make a `Matcher` object and try to match the regular expression above to the contents of the line. If there are any matches, we will extract the link. To extract the link portion of the regular expression, we use the `group()` method and pass in the number one (we only have one group). After extracting the link, we add it to an `ArrayList` named `links`.

To handle any possible `java.net Exceptions` or `java.io Exceptions`, we provide the catch blocks in **Cell B1-3**. We also remember to close the `BufferedReader` resource.

```
            } catch (MalformedURLException e) {
                System.out.println("Invalid URL");
            }
            catch (IOException e) {
                e.printStackTrace();
            } finally {
                try {
                    in.close();
                } catch (IOException e) {
                    System.out.println("Problem closing reader");
                }
            }
            return links;
        }
```

At last, we return the `ArrayList` of links.

To write the links to a local file, we provide the `writeLinksToFile()` method, outlined in **Cell B2**:

```
    public static void writeLinksToFile (
        String filename, ArrayList<String> links) {

        BufferedWriter out = null;
        try {
            out = new BufferedWriter(new FileWriter(filename));
            for (String link: links) {
                out.write(link);
                out.write('\n');
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (out != null) {
                try {
                    out.flush();
                    out.close();
                } catch (IOException ex) {
                    System.out.println("Problem closing writer");
                }
            }
        }
    }
```

This method creates a `BufferedWriter` object which wraps around a `FileWriter` object that will access the file at the provided file name. The method then writes all links in the parameter `ArrayList` to the file.

In the client program—`main()`—we will scrape the Wikipedia page for the Java programming language. **Cell A1** does this:

```
public static void main(String[] args) {
    String address = "https://en.wikipedia.org/wiki/" +
        "Java_(programming_language)";

    ArrayList<String> links = readURLForLinks(address);
    writeLinksToFile("files/urls.txt", links);
```

Running the code above generates the file `urls.txt`. Below, we list the first few lines of the file:

```
/wiki/Java_(software_platform)
/wiki/Java_Platform,_Standard_Edition
/wiki/Java_(disambiguation)
/wiki/JavaScript
/wiki/File:Java_programming_language_logo.svg
/wiki/Object-oriented
/wiki/Class-based_programming
```

Because Wikipedia pages are constantly changing, you may see different output when you run the program.

8.4.2 Applying Functional Operations to URL I/O

As you scan the generated `urls.txt` file, you will probably come across links that begin with `/wiki/` and `#`, in addition to the web address prefix that you're used to seeing: `http://` or `https://`. The links beginning with `/wiki/` and `#` refer to shortcuts within the Wikipedia universe and sections within the Java (programming language) article, respectively. If we want to see the "true" links that begin with `http://` or `https://`, we'll need to filter the list of links that we receive.

We could do this by changing the readURLForLinks() method, but we may want a general purpose link web scraper. We could also write code that iterates over each element of the ArrayList and checks for the "http"-style prefixes. Rather than following either of these two options, we choose instead to apply the principles of **functional operations, a powerful and relatively new paradigm that allows us to "stream" elements within data structures—the same way that we stream data from a source—and simultaneously operate on them**.

In **Cell A2**, we count the number of all links that start with the "`https`" protocol:

```
int num = (int) links.stream()
        .filter(item -> item.startsWith("https:"))
        .count();
System.out.println(num);
```

Notice how we didn't need to iterate using any for loops or for-each loops. Instead, we made three chained method calls that execute one after the other in a **pipeline**. First, by using the `stream()` method, we created a **stream** containing the String elements of the **source**: the `links` list. Then, by using the `filter()` method, we filtered the items of the stream so that only those that start with “`https:`” pass through the pipeline to the rest of the operations. Finally, we **aggregate** the items that had passed through the pipeline; in our case, we find the number of items with the `count()` method.

Within the `filter()` invocation, we need to pass in a **functional interface** of type `Predicate`, some function that we would like the program to execute for every item in the incoming stream that passes into `filter()`. We can satisfy the functional interface parameter requirement by writing a **lambda expression**, an **anonymous method** that has no name and quickly transform a set of parameters (the variables to the left of the arrow `->`) into a matching set of results (the expression to the right of the arrow `->`). In out case, we transform each incoming stream item into a boolean, whether it begins with “`https:`”. The `filter()` operation will then look at the stream of boolean results and produce an output stream of items containing *only those whose result was true*.

The (most likely smaller) output stream of `filter()` becomes the input stream to `count()`, the final aggregation operation. At the end of the pipeline, the aggregation function transforms the stream of elements back into a result.

The pipeline we have just created is depicted in Figure 8-11.

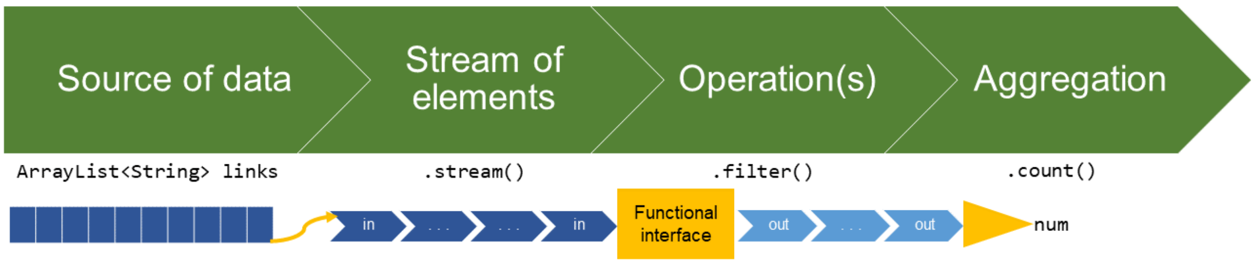

Figure 8-11 Pipeline using functional operations

After running the program, we see that there are 132 links on the Java (programming language) Wikipedia page that begin with “`https:`”.

The next block of code in **Cell A3** shows another pipeline:

```
ArrayList<String> listOfHttps = new ArrayList<>();
links.stream()
     .filter(item -> item.startsWith("https:"))
     .forEach(item -> listOfHttps.add(item));
   //.forEach((item) -> {System.out.println(item);});

listOfHttps.stream()
        .forEach((item) -> {System.out.println(item);});
System.out.println(listOfHttps.size());
```

In this pipeline, we again turn the `links` list into a stream and filter the stream so only those items that begin with “`https:`” remain. Using the filtered stream, we then add each remaining element into an `ArrayList` called `listOfHttps`. Here, we have the option to print out each filtered stream element, or to stream the `listOfHttps` resource and print out the elements there. We arbitrarily chose the latter option.

The first few “`https:`” links printed when we ran the program were:

```
https://en.wikibooks.org/wiki/Java_Programming
https://en.wiktionary.org/wiki/Java
https://commons.wikimedia.org/wiki/Category:Java_(programming_language)
https://en.wikibooks.org/wiki/Subject:Java_programming_language/Java
```

Using pipelines, streams, and functional interfaces circumvents the need for loops (for, for-each, and while) and control flow statements (if-else). This is the fundamental difference between **functional programming** and **imperative programming**. In functional programming, all operations are conducted within functions; we present expressions in a **declarative** manner instead of using control flow logic. The pipelines we demonstrated above are an excellent example of functional programming in Java. Imperative programming, on the other hand, executes statements iteratively (e.g. within a loop) depending on condition checks (e.g. the result of control flow statements). Functional programming emphasizes *what* the computer should do (e.g. make a stream, filter the stream, count the remaining stream) instead of *how* to do it (e.g. for each element in the array, count it if it begins with a certain prefix). **One of primary benefits of using streams and functional programming is that it is easy to convert code for parallel processing.**

Now it's your turn! Try **Knowledge Check 1 for** `URL`**:** Create a pipeline in the client program that prints out each of the links in the links list beginning with “`https://`”, but only the portion of the link that comes *after* the “`https://`” prefix.

8.4.3 Image I/O through URLs

In this last section, we will illustrate briefly how to download an image from the Internet and inspect some key properties of `File` objects.

First, we will be using the `ImageIO` class and a `BufferedImage` to quickly load the image, so we need to import those respective libraries.

```
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
```

Then, in **Cell A4**, we download a clipart of a cartoon snail:

```
BufferedImage img = null;
try {
    URL url = new URL("http://clipart-library.com/data_images/379246.png");
    img = ImageIO.read(in);
    ImageIO.write(img, "png", new File("files/snail.png"));
} catch(IOException e){
    e.printStackTrace();
}
```

We first create the `URL` object and then feed it into the `ImageIO` class's `read()` method, which will open an `InputStream` and begin processing the image data. After the entire image has been read, we write it to a `.png` image file by creating a new `File` object.

In **Cell A5**, we open the file we just downloaded and inspect some basic properties:

```
try {
    File file = new File("files/snail.png");
    if(!file.exists())
        throw new FileNotFoundException("File not found");
    System.out.println("Does the file exist? " + file.exists());
    System.out.println("The file name is: " + file.getName());
    System.out.println("The path is: " + file.getPath());
    System.out.println("The parent is: " + file.getParent());
    System.out.println("The file has " + file.length() + " bytes");
    System.out.println("Can the file be read? " + file.canRead());
    System.out.println("Can the file be written? " + file.canWrite());
    System.out.println("File can be executed? " + file.canExecute());
    System.out.println("Absolute path is " + file.getAbsolutePath());
    System.out.println("Last modified on " +
        new java.util.Date(file.lastModified()));
} catch(FileNotFoundException e) {
    e.printStackTrace();
}
```

We can create a `File` object for the computer to process and inspect properties like the relative and absolute file path, its size (i.e. number of bytes), whether it is readable and writable, and the date it was last modified.

The output from the program is:

```
Does the file exist? true
The file name is: snail.png
The path is: files\snail.png
The parent is: files
The file has 52040 bytes
Can the file be read? true
Can the file be written? true
File can be executed? true
Absolute path is
    C:\Users\username\eclipse-workspace\JavaIOEx6_URL\files\snail.png
Last modified on Fri Jul 03 16:48:16 CDT 2020
```

We've covered a lot of ground, and this concludes our exploration of Java I/O. We've seen I/O from standard keyboard input, from files as character streams and byte streams. We've tokenized character streams by using a `Scanner`, and handled bytes as both primitive types and objects. To make programs more efficient, we wrapped buffered streams around the underlying streams. We've also conducted I/O over the Internet by streaming input from a webpage. Using the principles of functional programming, we streamed the data that our program read in. Finally, we downloaded an image from the Web and inspected its `File`-related properties. From the sheer variety of applications of I/O, we hope that you are convinced that I/O is an important topic in computer science and have become familiar with the many different ways you could solve I/O problems.

You may have been curious about some of the gaps around interfaces and abstract classes in this chapter. The next chapter will take a break from I/O and explore those higher-level abstraction topics in more detail.