

Module 4

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Lecture 4 – Binary I/O, JavaFX, Lambdas and Streams

Learning Objectives

After successfully completing the module, students will be able to do the following:

1. Distinguish between text I/O and binary I/O.
2. Read and write bytes using `FileInputStream` and `FileOutputStream`.
3. Read and write primitive types and Strings using `DataInputStream` and `DataOutputStream`.
4. Read and write objects using `ObjectInputStream` and `ObjectOutputStream`.
5. Implement the `Serializable` interface.
6. Develop Java GUI applications.
7. Use event-driven programming.
8. Use event handling with inner classes, anonymous inner classes, and lambdas.
9. Write lambda expressions that implement functional interfaces.
10. Create `Stream<T>` objects from data sources.
11. Perform intermediate and terminal operations on `Stream<T>`.

Module 4 Study Guide and Deliverables

Topics: Lecture 4 Binary I/O, JavaFX, Lambdas and Streams

Readings:

- Binary I/O, object serialization: Deitel & Deitel, Chapter 15
- Java SE 8 Lambdas and Streams: Deitel & Deitel, Chapter 17
- JavaFX: Deitel & Deitel, Chapter 22
- Module 4 online content

Assessments: Interim Assessment 4 due Sunday, February 6 at 6:00 AM ET

Assignments: Assignment 4 due Wednesday, February 9 at 6:00 AM ET

Live Classrooms:

- Wednesday, February 2, from 8:00 PM to 9:00 PM ET
- Thursday, February 3, from 9:00 PM to 10:00 PM ET

Module Welcome and Introduction

met_cs622_18_su1_ebraude_mod4 video cannot be displayed here

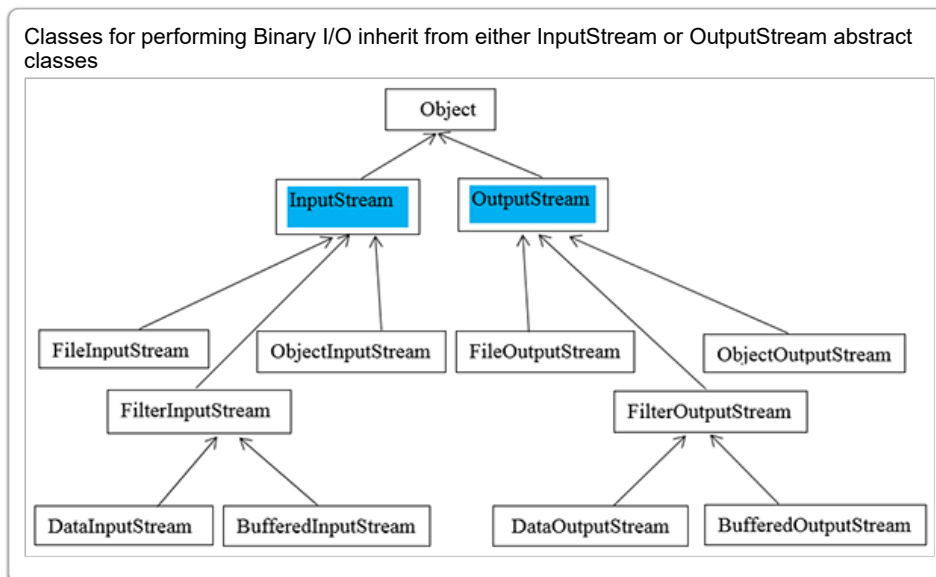
Binary I/O

Binary versus Text I/O

As mentioned in Module 2, files can in general be classified as either text or binary. A text file can be thought of as a sequence of characters, and a binary file as a sequence of bits (or bytes); this description, however, is not technically flawless, because characters, too, are represented using bits (bytes). The real difference is one of representation: for example, the decimal number 167 would be stored (represented) as three (ASCII or Unicode) characters 1, 6, and 7 in a text file, consuming one (or two or more – e.g., a Unicode character consists of two or more bytes) bytes per character. The same number 167 would be represented as hexadecimal A7 (because A is 10, and $167 = 16 * 10 + 7$), or its binary equivalent 10100111 ($A = 1010, 7 = 0111$) in a binary file, consuming, in this example, only one byte. A text file can be created (written) and read using a text editor, but a binary file must be read from or written to using a program (a text editor will show garbage if a binary file is used to read it).

Binary I/O Class Hierarchy

The classes needed for performing binary I/O all inherit from either `InputStream` or `OutputStream`; see the following diagram. Keep in mind that `InputStream` and `OutputStream` (drawn in blue) are abstract classes.



FileInputStream and FileOutputStream

The following example demonstrates creating a binary file named `first.dat`, writing 10 values - 0 through 9 - into it, and then reading those values back.

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

```

public class FileStream
{
    public static void main(String[] args) throws IOException
    {
        try (FileOutputStream outfile = new FileOutputStream("first.dat");)
        {
            for (int i = 0; i < 10; i++)
                outfile.write(i);
        }

        try (FileInputStream infile = new FileInputStream("first.dat");)
        {
            int smallvalue;
            while ((smallvalue = infile.read()) != -1)
                System.out.println(smallvalue);
        }
    }
}

```

The `outfile.write(i)` statement writes the byte for the int value `i`, that is `(byte)i`, to the output stream `outfile`.

The `read()` method on the `infile` stream reads the next byte of data from the input stream `infile`. The byte that is read is returned as an int value in 0 to 255. When the end of the file is reached, no byte is available for reading, and `infile.read()` returns -1.

Note that `try-with-resources` is used for both writing to and reading from the stream. This is okay because both `InputStream` and `OutputStream` are auto-closeable (they each have the `close()` method for closing the stream and releasing any resources associated with it).

A `FileNotFoundException` occurs if we try to create a `FileInputStream` with a nonexistent file.

Instead of declaring that `main()` throws `IOException`, one could explicitly catch and process `IOExceptions` in the body of `main()`, as shown below:

```

import java.io.*;

public class FileStreamException
{
    public static void main(String[] args)
    {
        try (FileOutputStream outfile = new FileOutputStream("first.dat");)
        {
            for (int i = 0; i < 10; i++)
                outfile.write(i);
        }

        catch (IOException ex)
        {
            ex.printStackTrace();
        }

        try (FileInputStream infile = new FileInputStream("first.dat");)
        {
            int smallvalue;
            while ((smallvalue = infile.read()) != -1)
                System.out.println(smallvalue);
        }

        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    }
}

```

DataInputStream and DataOutputStream

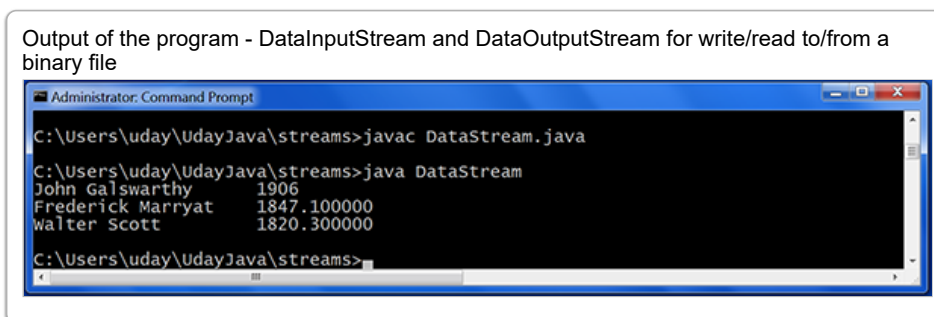
DataInputStream reads bytes from the stream and delivers them as corresponding primitive types or Strings. Similarly, DataOutputStream converts primitive-type values or Strings into bytes and writes the bytes to the stream. The DataInputStream constructor takes an InputStream object as its argument; similarly, the DataOutputStream constructor takes an OutputStream object as its argument. The following program illustrates DataInputStream and DataOutputStream for write/read to/from a binary file named second.dat:

```
import java.io.*;

public class DataStream
{
    public static void main(String[] args) throws IOException
    {
        try (DataOutputStream outfile = new DataOutputStream(new FileOutputStream("second.dat")))
        {
            outfile.writeUTF("John Galsworthy");
            outfile.writeInt(1906);
            outfile.writeUTF("Frederick Marryat");
            outfile.writeDouble(1847.1);
            outfile.writeUTF("Walter Scott");
            outfile.writeDouble(1820.3);
        }

        try (DataInputStream infile = new DataInputStream(new FileInputStream("second.dat")))
        {
            System.out.printf("%-20s %d\n", infile.readUTF(), infile.readInt());
            System.out.printf("%-20s %f\n", infile.readUTF(), infile.readDouble());
            System.out.printf("%-20s %f\n", infile.readUTF(), infile.readDouble());
        }
    }
}
```

Note that data must be read in the same order and format in which they were written into the binary file. The output of the above program is captured in the screenshot below:



If an attempt is made to read data past the end of the file, an EOFException occurs. This exception can be used to detect the end-of-file condition, as shown in the following program that does I/O on a binary file named third.dat:

```
import java.io.*;

public class DataStreamEOF
{
    public static void main(String[] args)
    {
        try
        {
            try (DataOutputStream outfile = new DataOutputStream(new
                FileOutputStream("third.dat"));
```

```

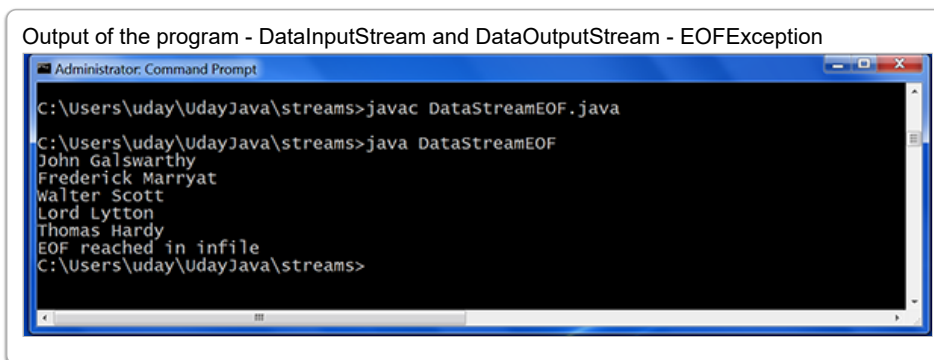
    {
        outfile.writeUTF("John Galsworthy");
        outfile.writeUTF("Frederick Marryat");
        outfile.writeUTF("Walter Scott");
        outfile.writeUTF("Lord Lytton");
        outfile.writeUTF("Thomas Hardy");
    }

    try (DataInputStream infile = new DataInputStream(new FileInputStream("third.dat")));
    {
        while(true)
        {
            System.out.printf("%-20s\n", infile.readUTF());
        }
    }
}
catch (EOFException ex)
{
    System.out.printf("EOF reached in infile");
}

catch (Exception ex)
{
    System.out.printf("Exception caught in main()");
    ex.printStackTrace();
}
}
}

```

The program's output is shown below.



ObjectInputStream and ObjectOutputStream; The Serializable Interface

ObjectInputStream and ObjectOutputStream can be used to read/write whole objects (not only primitive types and Strings) from/to binary files. Now, not every object can be written to an output stream; only those whose classes are "Serializable" can. A Serializable object is an instance of the java.io.Serializable interface; this means the object's class must implement the Serializable interface.

The Serializable interface has no methods; thus it is a marker interface.

The following program creates a Serializable class Employee and writes (and also reads) Employee objects to/from a binary file named objectfile.dat.

```

import java.io.*;

public class Serializable1
{
    public static void main(String[] args)
    {
        try

```

```

    {
        try (ObjectOutputStream outfile = new ObjectOutputStream(new
            FileOutputStream("objectfile.dat"));)
        {
            outfile.writeObject(new Employee(1, "Keats", 400.50));
            outfile.writeObject(new Employee(2, "Shelly", 200.75));
            outfile.writeObject(new Employee(3, "Byron", 500.00));
        }

        try (ObjectInputStream infile = new ObjectInputStream(new
            FileInputStream("objectfile.dat"));)
        {
            while (true)
            {
                System.out.printf("%s%n", (Employee) (infile.readObject()));
            }
        }
    }

    catch (EOFException ex)
    {
        System.out.println("EOF reached in objectfile.dat");
    }

    catch (FileNotFoundException ex)
    {
        System.out.println("FileNotFoundException");
        ex.printStackTrace();
    }

    catch (IOException ex)
    {
        System.out.println("IOException");
        ex.printStackTrace();
    }

    catch (ClassNotFoundException ex)
    {
        System.out.println("ClassNotFoundException");
        ex.printStackTrace();
    }
}

class Employee implements Serializable
{
    private int id;
    private String name;
    private double salary;

    public Employee(int empid, String nm, double sal)
    {
        id = empid;
        name = nm;
        salary = sal;
    }

    public String toString()
    {

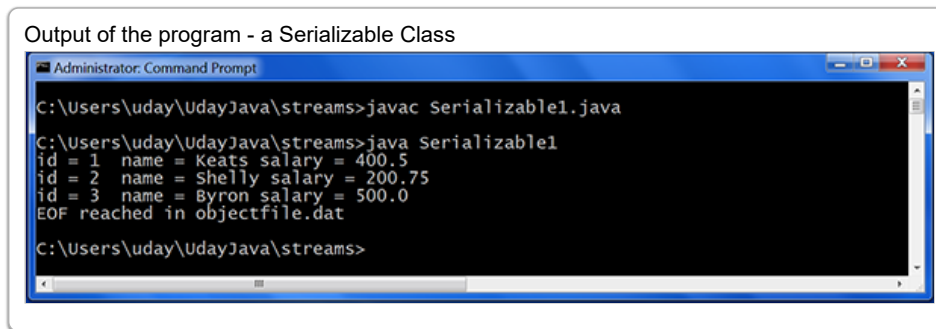
```

```

        return "id = " + id + "   name = " + name + " salary = " + salary;
    }
}

```

The output is shown below:



GUI: JavaFX

JavaFX is a versatile framework for developing graphical user interface (GUI) programs in Java. JavaFX has replaced Swing and AWT.

A JavaFX program is written with a class that extends the abstract class `javafx.application.Application` (class [Application](#)). A simple application (class `Javafx1`) for displaying a single button is shown below.

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class Javafx1 extends Application
{
    public void start(Stage mainStage)
    {
        StackPane pane = new StackPane();
        Button firstButton = new Button("First Button");
        pane.getChildren().add(firstButton);
        Scene scene = new Scene(pane, 200, 100);
        mainStage.setTitle("One button in a pane");
        mainStage.setScene(scene);
        mainStage.show();
    }
}

```

The `main()` method is not needed when the application is run from the command line (it may be needed when an IDE is used). When a JavaFX application has no `main()` method, the Java Virtual Machine (JVM) automatically invokes the `launch()` method to run the application. Class `Javafx1` overrides the `start()` method defined in `javafx.application.Application`.

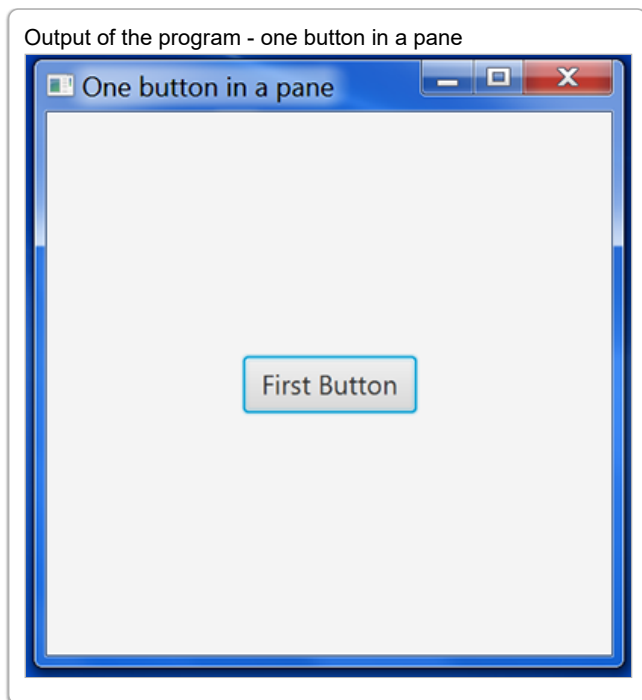
The statement

```
Button firstButton = new Button("First Button");
```

creates a button called "First Button". Similarly, the program creates a pane and a scene; the pane is put in the scene; the scene is put on the stage. The statement

```
mainStage.setTitle("One button in a pane");
```

gives a title to the stage. The stage object `mainStage` is automatically created by the JVM when the application is launched. The program displays the following:



The next program illustrates the use of font and color:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.text.*;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class JavafxColorFont extends Application
{
    public void start(Stage mainStage)
    {
        StackPane pane = new StackPane();

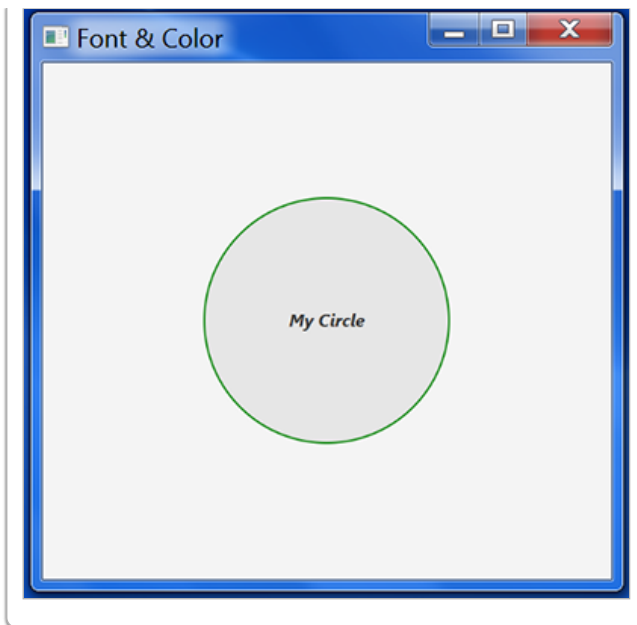
        Circle circ = new Circle();
        circ.setRadius(70);
        circ.setStroke(Color.GREEN);
        circ.setFill(new Color(0.7, 0.7, 0.7, 0.2));
        pane.getChildren().add(circ);

        Label la = new Label("My Circle");
        la.setFont(Font.font("Courier", FontWeight.BOLD, FontPosture.ITALIC, 10));
        pane.getChildren().add(la);

        Scene scene = new Scene(pane);
        mainStage.setTitle("Font & Color");
        mainStage.setScene(scene);
        mainStage.show();
    }
}
```

Note how we set the Circle's color and the font of the label "My Circle":

Output of the program - circle's color and font

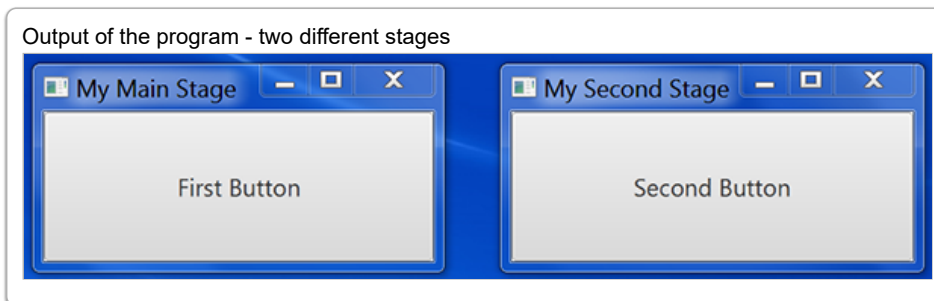


The use of two different stages is demonstrated by the following program.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class JavafxTwoStages extends Application
{
    public void start(Stage mainStage)
    {
        Button firstButton = new Button("First Button");
        Scene scene = new Scene(firstButton, 200, 200);
        mainStage.setTitle("My Main Stage");
        mainStage.setScene(scene);
        mainStage.show();

        Stage secondstage = new Stage();
        secondstage.setTitle("My Second Stage");
        secondstage.setScene(new Scene(new Button("Second Button"), 100, 100));
        secondstage.show();
    }
}
```



You can check out [a rich source of information on JavaFX](#).

Event-driven Programming

In event-driven programming, the program interacts with the user in such a way that the user's action—e.g., a keystroke, mouse/button click or mouse movement—controls the program's execution. An event is an action initiated by the user, such as a button click. The program can choose to respond to an event in a certain specific way (this includes the possibility that no action is taken by the program in response to an event). The program is said to be event-driven in the sense that the program logic is executed depending on the sequence of events. Java GUI programs are frequently event-driven.

We illustrate even-driven programming with a simple example of a hypothetical online voting scenario. When the program starts, a window appears and shows two buttons—"yes" and "no"—that the user might click to indicate either a "yes" vote or a "no" vote. As each vote is "cast," a message appears stating whether it was a yes vote or a no vote. At the program's termination, which is triggered by the closing of the window, the total counts of yes and no votes are displayed. The following program (class `JavafxEvents`) demonstrates events and event handling.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.text.*;
import javafx.scene.paint.Color;
import javafx.scene.layout.HBox;
import javafx.scene.shape.Circle;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;

public class JavafxEvents extends Application
{
    private static int yesvotecount = 0;
    private static int novotecount = 0;

    public void start(Stage mainStage)
    {
        HBox hbx = new HBox(50);
        hbx.setAlignment(Pos.CENTER);
        Button buttyes = new Button("Yes");
        Button buttno = new Button("NO");
        YesHandler yhandler = new YesHandler();
        buttyes.setOnAction(yhandler);
        NoHandler nhandler = new NoHandler();
        buttno.setOnAction(nhandler);
        hbx.getChildren().addAll(buttyes, buttno);

        Scene scene = new Scene(hbx);
        mainStage.setTitle("Voting Choices");
        mainStage.setScene(scene);
        mainStage.show();
    }

    public void stop()
    {
        System.out.printf("Voting completed; results: yes = %3d, no = %3d\n",
            yesvotecount, novotecount);
    }

    public static void onmoreyes()
    {
        yesvotecount++;
    }

    public static void onmoreno()
    {
        novotecount++;
    }
}
```

```

    }
}

class YesHandler implements EventHandler<ActionEvent>
{
    public void handle(ActionEvent event)
    {
        JavafxEvents.onemoreyes();
        System.out.println("Another yes vote received");
    }
}

class NoHandler implements EventHandler<ActionEvent>
{
    public void handle(ActionEvent event)
    {
        JavafxEvents.onemoreno();
        System.out.println("Another no vote received");
    }
}

```

The above program creates two source objects (buttons in this case), `buttyes` and `buttno`, with the following statements:

```

Button buttyes = new Button("Yes");
Button buttno = new Button("NO");

```

Two handler objects, `yhandler` and `nhandler` are created as follows:

```

YesHandler yhandler = new YesHandler();
NoHandler nhandler = new NoHandler();

```

The handlers are "registered" with their corresponding source objects as follows:

```

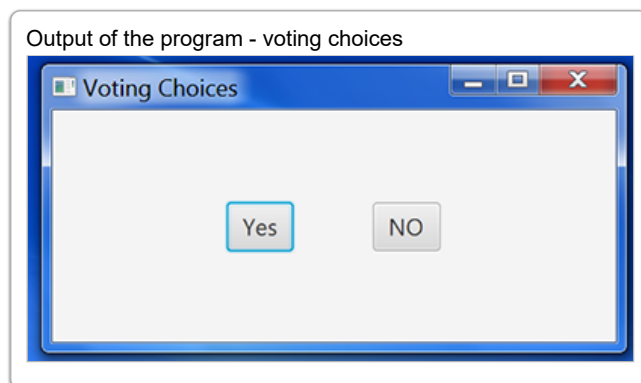
buttyes.setOnAction(yhandler);
buttno.setOnAction(nhandler);

```

The `EventHandler<ActionEvent>` interface contains the `handle(EventAction)` method. A handler class (such as `YesHandler` or `NoHandler` in the present example) should override the `handle()` method to specify what needs to be done to process the event.

When a button is clicked, the button object fires an `ActionEvent` and invokes its (the button's) handler's `handle(ActionEvent)` method, passing the `ActionEvent` as the argument to the `handle()` method. For example, a click on the "Yes" button causes the `yhandler` object's `handle()` method to be invoked, thereby incrementing the yes count (with `JavafxEvents.onemoreyes()`) and printing a message to that effect.

When the program starts, the Yes and No buttons appear on the "Voting Choices" window:



The program finishes when the (last) window is closed, at which point the `stop()` method is invoked. For a comprehensive description of the life-cycle of an event-driven program such as this one, see [the documentation of the abstract class Application](#) and [the helpful tutorial on event handling](#).

Here is a screenshot of a single session (one execution) of the program:

Output of the program - single session

```

Administrator: Command Prompt
C:\Users\uday\UdayJava\gui>javac JavafxEvents.java
C:\Users\uday\UdayJava\gui>java JavafxEvents
Another yes vote received
Another no vote received
Another no vote received
Another yes vote received
Another no vote received
Another no vote received
Another no vote received
Another yes vote received
Another yes vote received
Voting completed; results: yes = 4, no = 5
C:\Users\uday\UdayJava\gui>

```

Now, a look at the design of the above program (class `JavafxEvents`) will be useful for understanding the motivation for the following program's (class `JavafxEventsInnerClass`) design. In class `JavafxEvents`, it was mandatory for the two variables `yesvotecount` and `novotecount` to be private static. Why static? Because otherwise `yesvotecount` and `novotecount` would be `JavafxEvent`'s instance fields, and classes `YesHandler` and `NoHandler` would need `JavafxEvents` instances (objects) to access those objects' `yesvotecount` and `novotecount`. Why private? Simply because this is good programming practice; otherwise any other class would be able to access them directly, without having to go through `JavafxEvent`'s accessor methods (the accessor methods are public).

Inner Classes

The design can be simplified if we write `YesHandler` and `NoHandler` as "inner classes," that is, (non-static) classes nested (housed) within `JavafxEvents`. That way, `YesHandler` and `NoHandler` classes would have direct access to the private fields of `JavafxEvents`, eliminating the need for public accessor methods for `yesvotecount` and `novotecount`. This is shown in the following program, class `JavafxEventsInnerClass`:

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.text.*;
import javafx.scene.paint.Color;
import javafx.scene.layout.HBox;
import javafx.scene.shape.Circle;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;

public class JavafxEventsInnerClass extends Application
{
    private int yesvotecount = 0;
    private int novotecount = 0;

    public void start(Stage mainStage)
    {
        HBox hbx = new HBox(50);
        hbx.setAlignment(Pos.CENTER);
        Button buttyes = new Button("Yes");
        Button buttno = new Button("NO");
        YesHandler yhandler = new YesHandler();
        buttyes.setOnAction(yhandler);
        NoHandler nhandler = new NoHandler();
        buttno.setOnAction(nhandler);
        hbx.getChildren().addAll(buttyes, buttno);

        Scene scene = new Scene(hbx);
    }
}

```

```

        mainStage.setTitle("Voting Choices");
        mainStage.setScene(scene);
        mainStage.show();
    }

    public void stop()
    {
        System.out.printf("Voting completed; results: yes = %3d, no = %3d\n", yesvotecount, novotecount);
    }

    // inner class
    class YesHandler implements EventHandler<ActionEvent>
    {
        public void handle(ActionEvent event)
        {
            yesvotecount++;
            System.out.println("Another yes vote received");
        }
    } // end class YesHandler

    // inner class
    class NoHandler implements EventHandler<ActionEvent>
    {
        public void handle(ActionEvent event)
        {
            novotecount++;
            System.out.println("Another no vote received");
        }
    } // end class NoHandler
} // end class JavafxEventsInnerClass

```

Here's a simple example of an outer class and an inner class:

```

// Two separate classes, none inner
public class Class1
{
    // ...
}

public class Class2
{
    // ...
}

// Class2 is an inner class
public class Class1
{
    // ...
    class Class2
    {
        // ...
    }
}

// Inner class can access enclosing class's fields and items
public class Class1
{
    // ...
    private int k;

    public void method1()
    {
        // ...
    }
}

```

```

    }

    class Class2
    {
        public void method2()
        {
            k++;
            method1();
            // ...
        }
    } // end Class2
} // end Class1

```

Note that the static methods `onemoreyes()` and `onemoreno()` (of class `JavafxEvents`) are no longer required in the revised design, nor are `yesvotecount` and `novotecount` required to be declared static.

Anonymous Inner Classes

Class `JavafxEventsInnerClass` can be further simplified by using "anonymous inner classes" for the two event handling classes. An anonymous inner class is a name-less class that offers the unique advantage of defining the class and creating an instance of the class in a single step. An anonymous inner class looks like this:

```

new Interface or Superclass()
{
    // Method(s) of interface or superclass overridden or implemented here
    // Other (optional) methods
}

```

For our event-handling example, the inner class `YesHandler` (in class `JavafxEventsInnerClass`) can be replaced with the following anonymous inner class:

```

new EventHandler<ActionEvent>()
{
    public void handle(ActionEvent event)
    {
        yesvotecount++;
        System.out.println("Another yes vote received");
    }
}

```

And thus the statement

```
buttyes.setOnAction(yhandler);
```

in class `JavafxEventsInnerClass` can now be replaced with the following:

```

buttyes.setOnAction(new EventHandler<ActionEvent>()
{
    public void handle(ActionEvent event)
    {
        yesvotecount++;
        System.out.println("Another yes vote received");
    }
});

```

We are now in a position to write class `JavafxEventsAnonymousInnerClass`:

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.text.*;
import javafx.scene.paint.Color;

```

```

import javafx.scene.layout.HBox;
import javafx.scene.shape.Circle;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;

public class JavafxEventsAnonymousInnerClass extends Application
{
    private int yesvotecount = 0;
    private int novotecount = 0;

    public void start(Stage mainStage)
    {
        HBox hbx = new HBox(50);
        hbx.setAlignment(Pos.CENTER);
        Button buttyes = new Button("Yes");
        Button buttno = new Button("NO");

        buttyes.setOnAction(new EventHandler<ActionEvent>()
        {
            public void handle(ActionEvent event)
            {
                yesvotecount++;
                System.out.println("Another yes vote received");
            }
        });

        buttno.setOnAction(new EventHandler<ActionEvent>()
        {
            public void handle(ActionEvent event)
            {
                novotecount++;
                System.out.println("Another no vote received");
            }
        });

        hbx.getChildren().addAll(buttyes, buttno);

        Scene scene = new Scene(hbx);
        mainStage.setTitle("Voting Choices");
        mainStage.setScene(scene);
        mainStage.show();
    }

    public void stop()
    {
        System.out.printf("Voting completed; results: yes = %3d, no = %3d%n",
            yesvotecount, novotecount);
    }

} // end class JavafxEventsAnonymousInnerClass

```

Java 8 Lambda Expressions

The above program can be further simplified with the help of "lambda expressions."

The following piece of code

```
buttyes.setOnAction(new EventHandler<ActionEvent>()
{
    public void handle(ActionEvent event)
    {
        yesvotecount++;
        System.out.println("Another yes vote received");
    }
});
```

can be simplified to:

```
buttyes.setOnAction(    event ->
{
    yesvotecount++;
    System.out.println("Another yes vote received");
}

);
```

where the part

```
event-> {
    yesvotecount++;
    System.out.println("Another yes vote received");
}
```

is a lambda expression. The compiler treats the lambda as if it is an anonymous inner class object. In the present example, the compiler infers that the object is an instance of `EventHandler<ActionEvent>`, and that `event` is a parameter of type `ActionEvent`.

The statement(s) in the lambda must all be for the same method (it is an error to try to put more than one method in the same lambda). This means that the interface from which the lambda is created (e.g., the `EventHandler` interface in our example) must have exactly one abstract method. Such an interface is called a "functional interface".

Lambda expressions take the form:

```
(parameter list) -> {statements}
```

That is, a lambda looks like

```
(type1 parameter1, type2 parameter2, ...) -> {one or more statements;}
```

The types of the parameter(s) may be explicitly stated or implicitly understood by the compiler. If there is only one statement, the braces may be omitted, as in

```
(int i, int j) -> {return i + j;}
```

or

```
(int i, int j) -> return i + j
```

or

```
(i, j) -> i + j
```

In the last example above, the type of the expression `i+j` is implicitly inferred. The parentheses surrounding the parameters may be omitted if there is only one parameter with an implicitly inferred type, as in

```
i -> System.out.println(i)
```

An empty parameter list can be represented thus:

```
() -> System.out.println("Lambdas are fun!")
```


Other special short-hand syntax notations for lambdas exist.

The code for class `JavafxEventsLambda` is shown below, demonstrating different syntactical styles of writing lambdas. Note that the outputs of `JavafxEventsLambda`, `JavafxEventsAnonymousInnerClass`, `JavafxEventsInnerClass`, and `JavafxEvents` are all the same.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.text.*;
import javafx.scene.paint.Color;
import javafx.scene.layout.HBox;
import javafx.scene.shape.Circle;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;

public class JavafxEventsLambda extends Application
{
    private int yesvotecount = 0;
    private int novotecount = 0;

    public void start(Stage mainStage)
    {
        HBox hbx = new HBox(50);
        hbx.setAlignment(Pos.CENTER);
        Button buttyes = new Button("Yes");
        Button buttno = new Button("NO");

        buttyes.setOnAction(
            (ActionEvent event) ->
            {
                yesvotecount++;
                System.out.println("Another yes vote received");
            }
        );

        buttno.setOnAction(
            (event) ->
            {
                novotecount++;
                System.out.println("Another no vote received");
            }
        );

        /*****      An alternative syntax
        buttno.setOnAction(
            event ->
            {
                novotecount++;
                System.out.println("Another no vote received");
            }
        );

        *****/

        hbx.getChildren().addAll(buttyes, buttno);
    }
}
```

```

        Scene scene = new Scene(hbx);
        mainStage.setTitle("Voting Choices");
        mainStage.setScene(scene);
        mainStage.show();
    }

    public void stop()
    {
        System.out.printf("Voting completed; results: yes = %3d, no = %3d\n",
            yesvotecount, novotecount);
    }
} // end class JavafxEventsLambda

```

Java 8 Streams and Lambdas

Java 8 Streams are not to be confused with the traditional use of streams in Java input/output (e.g., `InputStream`, `OutputStream`, `DataInputStream`, `ObjectOutputStream`, etc.). Java 8 defines an interface `Stream<T>` (package `java.util.stream`) from which objects of type `Stream<T>` can be created. For example, an object of type `Stream<Student>` can be instantiated. This object can be thought of as providing a "stream" of `Student` objects (like the flow of a stream, metaphorically). A number of operations can be applied on the stream of `Students`, producing, typically, either a new stream or a single object or value.

Typically, a source (such as an array or a collection object) provides the elements (e.g., the individual `Student` objects) for a `Stream`; a number of intermediate operations are performed on the `Stream` (that is, on the elements of the `Stream`); a terminal operation produces a single value or a single object or performs some other operation (e.g., `print`) on each element of the `Stream`. Often the different methods applied on a `Stream` can be chained, because one (intermediate) operation works on a `Stream`, yielding another `Stream`. [Read more about Streams.](#)

The following program illustrates several different ways of creating `Stream` objects:

- Use the static factory method **of()** of `Stream`
- Use the **stream()** static method of the `Arrays` class
- Use the **stream()** method of a `Collection` object

```

import java.util.stream.Stream;
import java.util.*;

// Illustring different ways of creating a Stream
public class StreamStringCreation
{
    public static void main(String[] args)
    {
        String[] str = {"abc", "def", "ijk"};

        // Stream.of(T[])
        Stream<String> st1 = Stream.of(str);
        System.out.print("Stream.of(T[]): ");
        st1.forEach(s -> System.out.printf("%s ", s));
        System.out.println();

        // Stream.of(T[])
        Stream<String> st2 = Stream.of("one", "two", "three");
        System.out.print("Stream.of(T[]): ");
        st2.forEach(s -> System.out.printf("%s ", s));
        System.out.println();

        // static method stream(T[]) of Arrays class
        Stream<String> st3 = Arrays.stream(str);
    }
}

```

```

        System.out.print("Arrays.stream(T[]): ");
        st3.forEach(s -> System.out.printf("%s ", s));
        System.out.println();

        // CollectionObject.stream()
        List<String> aliststr = new ArrayList<String>();
        aliststr.add("abc");
        aliststr.add("def");
        aliststr.add("ijk");
        Stream<String> st4 = aliststr.stream();
        System.out.print("ArrayListObject.stream(): ");
        st4.forEach(s -> System.out.printf("%s ", s));
        System.out.println();
    }
}

```

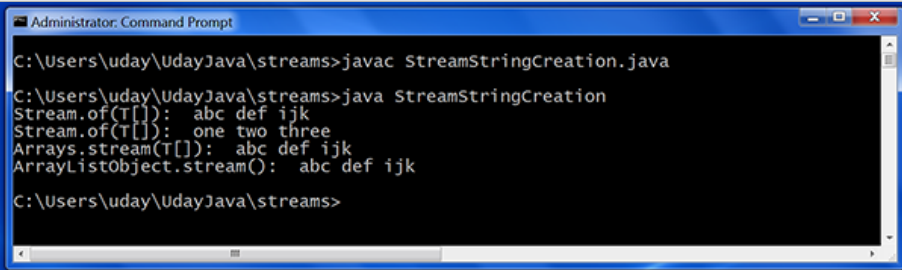
The above program creates four `Stream<String>` objects named `st1`, `st2`, `st3` and `st4`. The `forEach()` method takes each element of the stream and prints it. Note that

```
s -> System.out.printf("%s ", s)
```

is a Java 8 lambda (and a popular idiom) for the traditional functional interface.

The output is shown below.

Output of the program - creating Stream objects



```

Administrator: Command Prompt
C:\Users\uday\UdayJava\streams>javac StreamStringCreation.java
C:\Users\uday\UdayJava\streams>java StreamStringCreation
Stream.of(T[]): abc def ijk
Stream.of(T[]): one two three
Arrays.stream(T[]): abc def ijk
ArrayListObject.stream(): abc def ijk
C:\Users\uday\UdayJava\streams>

```

Functional Programming

The fundamental difference between a collection object and a Stream is that once a Stream has been processed, the result is not stored – this means that in order to obtain the same result, the operation has to be applied again. A simple analogy should help:

Suppose we want to find the sum of two numbers `x` and `y` because we will need the sum for two different purposes: one to print half of the sum, the other to print the square of the sum. In the Stream style (which is similar to the functional style), the following statements in a hypothetical programming language will perform the task:

```

Print add(x, y) / 2;
Print add(x, y) * add(x, y);

```

This is because, in the functional style, there is no provision for storing the result of adding `x` and `y`; we have to invoke the `add` method whenever the sum is needed, regardless of whether or not the sum was computed earlier. (Students familiar with a functional programming language, such as LISP, should be able to appreciate this point well.)

Terminal operations, such as `forEach()`, are literally the "terminal" or the very last operation on a Stream, as illustrated in the following code:

```

// Incorrect program - does not run
import java.util.stream.Stream;

public class Stream1
{
    public static void main(String[] args)

```

```

{
    String[] s = {"abc", "def", "ijk"};
    Stream<String> st1 = Stream.of(s);
    st1.forEach(System.out::println);

    Stream<String> st2 = Stream.of("one", "two", "three");
    st2.forEach(System.out::println);

    st2.forEach(System.out::println); // WRONG: stream st2 has already been operated on
}
}

```

The above program does not run, because the last `forEach()` is invalid; after the elements of a `Stream` have been processed by a `forEach()` method, no further operation can be performed on the `Stream`.

Stream<Integer>

As mentioned earlier, an intermediate operation on a `Stream` produces another `Stream`; study the following example, where several intermediate operations are illustrated (descriptions of the methods are given after the program's output is shown).

```

import java.util.stream.Stream;
import java.util.stream.Collectors;
import java.util.*;

// Illustring Stream<Integer>
public class StreamInteger
{
    public static void main(String[] args)
    {
        Integer[] myint = {0, 1, 2, 3, 4, 5};

        // Stream.of(T[])
        Stream<Integer> st1 = Stream.of(myint);
        System.out.print("Stream.of(T[]): ");
        st1.forEach(s -> System.out.printf("%d ", s));
        System.out.println();

        // Stream.of(T[])
        Stream<Integer> st2 = Stream.of(10, 11, 12, 13, 14, 15, 16, 17, 18, 19);
        System.out.print("Stream.of(T[]): ");
        st2.forEach(s -> System.out.printf("%d ", s));
        System.out.println();

        // static method stream(T[]) of Arrays class
        Stream<Integer> st3 = Arrays.stream(myint);
        System.out.print("Arrays.stream(T[]): ");
        st3.forEach(s -> System.out.printf("%d ", s));
        System.out.println();

        // CollectionObject.stream()
        List<Integer> a = new ArrayList<Integer>();
        for (int i = 5; i >= 0; i--)
            a.add(i);
        Stream<Integer> st4 = a.stream();
        System.out.print("ArrayListObject.stream(): ");
        st4.forEach(s -> System.out.printf("%d ", s));
        System.out.println();

        System.out.println("Number of elements: " + a.stream().count());
    }
}

```

```

System.out.println("Sum of elements: " +
    a.stream()
    .reduce(0, (i, j) -> i + j));
// a.stream() must be invoked again

System.out.println("Sum of cubes: " +
    a.stream()
    .reduce(0, (i, j) -> i + j * j * j));

System.out.println("Product of elements (excluding zero): " +
    a.stream()
    .filter(i -> i > 0)
    .reduce(1, (i, j) -> i * j));

System.out.println("Product of squares of elements (excluding zero): " +
    a.stream()
    .filter(i -> i > 0)
    .reduce(1, (i, j) -> i * j * j));

System.out.print("Sorted elements: ");
a.stream()
    .sorted()
    .forEach(i -> System.out.printf("%d ", i));
System.out.println();

System.out.print("Sorted elements (distinct) after int div by 2: ");
a.stream()
    .map(i -> i / 2)
    .distinct()
    .sorted()
    .forEach(i -> System.out.printf("%d ", i));
System.out.println();

System.out.print("Sorted squared elements after int div by 2 (those > 3): ");
a.stream()
    .map(i -> i * i / 2)
    .filter(i -> i > 3)
    .sorted()
    .forEach(i -> System.out.printf("%d ", i));
System.out.println();

System.out.printf("Sorted elements in a list: %s\n",
    a.stream()
    .sorted()
    .collect(Collectors.toList())
);

System.out.printf("Selected cubed elements in a sorted list: %s\n",
    a.stream()
    .map(i -> i * i * i)
    .filter(i -> i % 3 > 0)
    .sorted()
    .collect(Collectors.toList())
);

List<Integer> arrlst = a.stream()
    .map(i -> i * i * i)
    .filter(i -> i % 3 > 0)
    .sorted()
    .collect(Collectors.toList());

```

```

        System.out.printf("Numbers in list returned by stream(): ");
        for (Integer i: arrlst)
            System.out.printf("%d ", i);
        System.out.println();
    }
}

```

The output of the above program is shown below:

Output of the program - several intermediate stream operations

```

C:\Users\uday\UdayJava\streams>javac StreamInteger.java
C:\Users\uday\UdayJava\streams>java StreamInteger
Stream.of(T[]): 0 1 2 3 4 5
Stream.of(T[]): 10 11 12 13 14 15 16 17 18 19
Arrays.stream(T[]): 0 1 2 3 4 5
ArrayListObject.stream(): 5 4 3 2 1 0
Number of elements: 6
Sum of elements: 15
Sum of cubes: 225
Product of elements (excluding zero): 120
Product of squares of elements (excluding zero): 14400
Sorted elements: 0 1 2 3 4 5
Sorted elements (distinct) after int div by 2: 0 1 2
Sorted squared elements after int div by 2 (those > 3): 4 8 12
Sorted elements in a list: [0, 1, 2, 3, 4, 5]
Selected cubed elements in a sorted list: [1, 8, 64, 125]
Numbers in list returned by stream(): 1 8 64 125
C:\Users\uday\UdayJava\streams>

```

The above program illustrates the use of the following features:

- Data source:
 - An Integer array, named myint
 - An ArrayList<Integer>, named a
- Intermediate operations:
 - *sorted*: produces a stream with the original elements in sorted order
 - *distinct*: produces a stream with only distinct (unique or non-repeating) elements
 - *filter*: produces a stream with only those elements that satisfy a given condition (the condition can be a compound condition involving, for instance, && and ||)
 - *map*: produces a stream in which each original element is mapped (changed) to a new element (this includes the possibility that the new element will be of a different type)
- Terminal operations:
 - *reduce*: reduces the elements to a single value, using an association accumulation function, such as a lambda function for adding two numbers
 - *forEach*: performs a specified action on every single element
 - *collect*: creates a new Collection object (such as a List) using the results of the immediately preceding operation

The operation

```
a.reduce(0, (i, j) -> i + j)
```

produces a final value that is the sum of all the elements in stream a. This is done by obtaining pair-wise sums, starting with the addition of the initial (given) zero and the first element, proceeding to the second element, and then to the third, etc. The initial zero effectively means that the first and the second elements are added and the result is added to the third element, and so on.

Similarly,

```
a.reduce(1, (i, j) -> i * j)
```

produces the product of all the elements in stream a (note the initial 1). Again,

```
a.reduce(0, (i, j) -> i + j * j * j)
```

reduces the elements to the sum of their cubes.

Stream method `collect()` is a terminal operation; it produces a mutable (changeable) collection object, such as a `List` or `Set`. For example, the statement

```
a.collect(Collectors.toList())
```

uses the static method `toList()` of the `Collectors` class (`java.util.stream`) to produce a mutable `List<Integer>` object from a `Stream<Integer>` object (a in the above statement).

Note that `Collectors` is a class (full of static methods), while `Collector` is an interface; both can be found in package `java.util.stream`.

Stream<Student>

Let us reiterate that the `Stream` interface is a generic interface on any non-primitive type. Thus it does not make sense to talk about `Stream<int>` or `Stream<double>`; in fact, Java provides types `IntStream`, `DoubleStream`, and `LongStream` (class `Arrays` has overloaded versions of our familiar `stream()` method to create those types from, for example, arrays of `int`'s, `long`'s or `double`'s).

We see an example of a `Stream` of a user-defined type, `Student`, in the following code.

```
import java.util.stream.Stream;
import java.util.stream.Collectors;
import java.util.function.Predicate;
import java.util.*;

public class StreamStudent
{
    public static void main(String[] args)
    {
        ArrayList<Student> students = new ArrayList<Student>();

        students.add(new Student(1, "Philip Pullman", 4.9));
        students.add(new Student(2, "J.R.R. Tolkien", 4.7));
        students.add(new Student(3, "Terry Pratchette", 4.6));
        students.add(new Student(4, "Neil Gaiman", 4.5));

        Stream<Student> stustream = students.stream();

        System.out.println("Here come the students:");
        stustream.forEach(System.out::println);

        System.out.println("And here again:");
        students.stream()
            .forEach(System.out::println);

        System.out.println("Students with GPA greater than 4.6 or a long name:");
        students.stream()
            .filter(stu -> stu.getgpa() > 4.6 || (stu.getname()).length() >= 15)
            .forEach(System.out::println);

        Predicate<Student> highGPAOrLongName =
            stu -> (stu.getgpa() > 4.6 || (stu.getname()).length() >= 10);

        System.out.println("Second time: Students with GPA greater than 4.6 or a long name:");
        students.stream()
            .filter(highGPAOrLongName)
            .forEach(System.out::println);
    }
}
```

```

        System.out.printf("The first student with GPA greater than 4.6 or a long name:%n%s%n",
            students.stream()
                .filter(highGPAOrLongName)
                .findFirst()
                .get()
            );

        System.out.println("Students sorted on GPA:");
        students.stream()
            .sorted(Comparator.comparing(Student::getgpa))
            .forEach(System.out::println);

        System.out.printf("Average GPA: %f%n",
            students.stream()
                .mapToDouble(Student::getgpa)
                .average()
                .getAsDouble()
            );

        System.out.printf("Lowest GPA: %f%n",
            students.stream()
                .mapToDouble(Student::getgpa)
                .min()
                .getAsDouble()
            );

        System.out.printf("Highest GPA: %f%n",
            students.stream()
                .mapToDouble(Student::getgpa)
                .max()
                .getAsDouble()
            );

        System.out.printf("Sum of the GPAs: %f%n",
            students.stream()
                .mapToDouble(Student::getgpa)
                .sum()
            );
    }
}

class Student
{
    private int id;
    private String name;
    private double gpa;

    public Student(int serialno, String nm, double grade)
    {
        id = serialno;
        name = nm;
        gpa = grade;
    }

    public double getgpa()
    {
        return gpa;
    }
}

```



```

public String getname()
{
    return name;
}

public String toString()
{
    return "Id = " + id + " Name = " + name + " GPA = " + gpa;
}
}

```

The output of this program is shown below.

Output of the program - a Stream of a user-defined type, Student

```

Administrator: Command Prompt
C:\Users\uday\UdayJava\streams>java StreamStudent
Here come the students:
Id = 1 Name = Philip Pullman GPA = 4.9
Id = 2 Name = J.R.R. Tolkien GPA = 4.7
Id = 3 Name = Terry Pratchette GPA = 4.6
Id = 4 Name = Neil Gaiman GPA = 4.5
And here again:
Id = 1 Name = Philip Pullman GPA = 4.9
Id = 2 Name = J.R.R. Tolkien GPA = 4.7
Id = 3 Name = Terry Pratchette GPA = 4.6
Id = 4 Name = Neil Gaiman GPA = 4.5
Students with GPA greater than 4.6 or a long name:
Id = 1 Name = Philip Pullman GPA = 4.9
Id = 2 Name = J.R.R. Tolkien GPA = 4.7
Id = 3 Name = Terry Pratchette GPA = 4.6
Second time: Students with GPA greater than 4.6 or a long name:
Id = 1 Name = Philip Pullman GPA = 4.9
Id = 2 Name = J.R.R. Tolkien GPA = 4.7
Id = 3 Name = Terry Pratchette GPA = 4.6
Id = 4 Name = Neil Gaiman GPA = 4.5
The first student with GPA greater than 4.6 or a long name:
Id = 1 Name = Philip Pullman GPA = 4.9
Students sorted on GPA:
Id = 4 Name = Neil Gaiman GPA = 4.5
Id = 3 Name = Terry Pratchette GPA = 4.6
Id = 2 Name = J.R.R. Tolkien GPA = 4.7
Id = 1 Name = Philip Pullman GPA = 4.9
Average GPA: 4.675000
Lowest GPA: 4.500000
Highest GPA: 4.900000
Sum of the GPAs: 18.700000
C:\Users\uday\UdayJava\streams>

```

The above program demonstrates the creation (with a lambda) and subsequent use of an object, named `highGPAOrLongName`, that is a functional interface `Predicate<Student>`:

```

Predicate<Student> highGPAOrLongName =
    stu -> (stu.getgpa() > 4.6 || (stu.getname()).length() >= 10);

```

An immediate advantage of defining a predicate like this is that the corresponding lambda can be used multiple times. Our program showed one use of the predicate as the (compound) condition in a filter:

```

students.stream().filter(highGPAOrLongName)

```

Learn [more detailed information on predicates](#).

The `findFirst()` method finds and returns the first element that matches the filtering criterion. This method is an example of short-circuit evaluation. Technically, `findFirst()` returns an `Optional<Student>`, which when acted on by the `get()` method, returns a `Student` object. (See [Class Optional<T>](#).)

The program demonstrated a typical use of the `comparing()` method of the `Comparator` interface (read more about [Interface Comparator<T>](#)):

```

students.stream().sorted(Comparator.comparing(Student::getgpa))

```

In the above statement, the `comparing()` method returns a `Comparator` object that calls the `getgpa()` method on each of two `Student` objects and returns zero if the two GPAs are identical, a negative value if the first GPA is less than the second, and a positive value if the first is more than the second.

The `Stream` method `mapToDouble()` maps each element (`Student` object) to a double value obtained by a call to the specified method, `getgpa()` in this example; the result is a `DoubleStream`.

The present example illustrates the following terminal operations that we did not encounter in the previously discussed class `StreamInteger`:

- `max()`
- `min()`
- `average()`
- `sum()`
- `findFirst()`

An excellent source of further information on Java 8 Streams is the original documentation [Aggregate Operations](#).

Module 4 Practice Questions

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer first, and then click "Show Answer" to compare yours to the suggested answer.

Test Yourself Questions 4.1 and 4.2 refer to the program involving class `FileStream` in the module content.

Test Yourself 4.1

Can we drop the `main()` header declaration "throws `IOException`"?

Suggested answer: No. Dropping it would cause the compiler to point out `IOException` and `FileNotFoundException` as neither declared nor caught. If we must drop this declaration, the exception(s) must be caught and processed explicitly inside `main()`.

Test Yourself 4.2

Will the program produce correct results if we write and read the int values 200 through 500?

Suggested answer: No. The values 200 through 255 will be fine, but 256 through 500 will cause wrong results. Note that it is a byte (= 8 bits) that is written or read ($2^8 = 256$).

Test Yourself 4.3

The question refers to the program involving class `DataStream` in the module content.

Is it okay to change `FileOutputStream` to `OutputStream` in the following statement? Why or why not?

```
DataOutputStream outfile = new DataOutputStream(new  
FileOutputStream("second.dat"));
```

Suggested answer: No. Because `OutputStream` is an abstract class and cannot be instantiated.

Test Yourself 4.4

The question refers to the (incorrect) program involving class `Stream1` in the section on Java 8 Streams and lambdas.

We know that the program in `Stream1` does not work because the second `forEach()` on `st2` is invalid. Would the program work if the offending `forEach()` statement is replaced with the following?

```
st2.sorted();
```

Suggested answer: No. Because `forEach()` is a terminal operation.

Test Yourself 4.5

Assuming that `IntStream mystream` contains the integers 1 through 10, what would be the value produced by

```
mystream.reduce(0, (i, j) -> i * j) ?
```

Suggested answer: Zero.

Test Yourself 4.6

In the first even-handling example in the notes (class JavafxEvents for yes-no voting), after voting has started, if the user presses control-c to get out of the program, will the total vote counts be printed?

Suggested answer: No. Because in that case the stop() method (of the Application interface, overridden in JavafxEvents) would not be called.

References

- Oracle. *JavaFX 8: Packages*. Retrieved from <https://docs.oracle.com/javase/8/javafx/api/toc.htm>.
- Oracle. *The Java™ Tutorials: Aggregate Operations*. Retrieved from <https://docs.oracle.com/javase/tutorial/collections/streams/>.

Boston University Metropolitan College