Exception and I/O

1. Exceptions

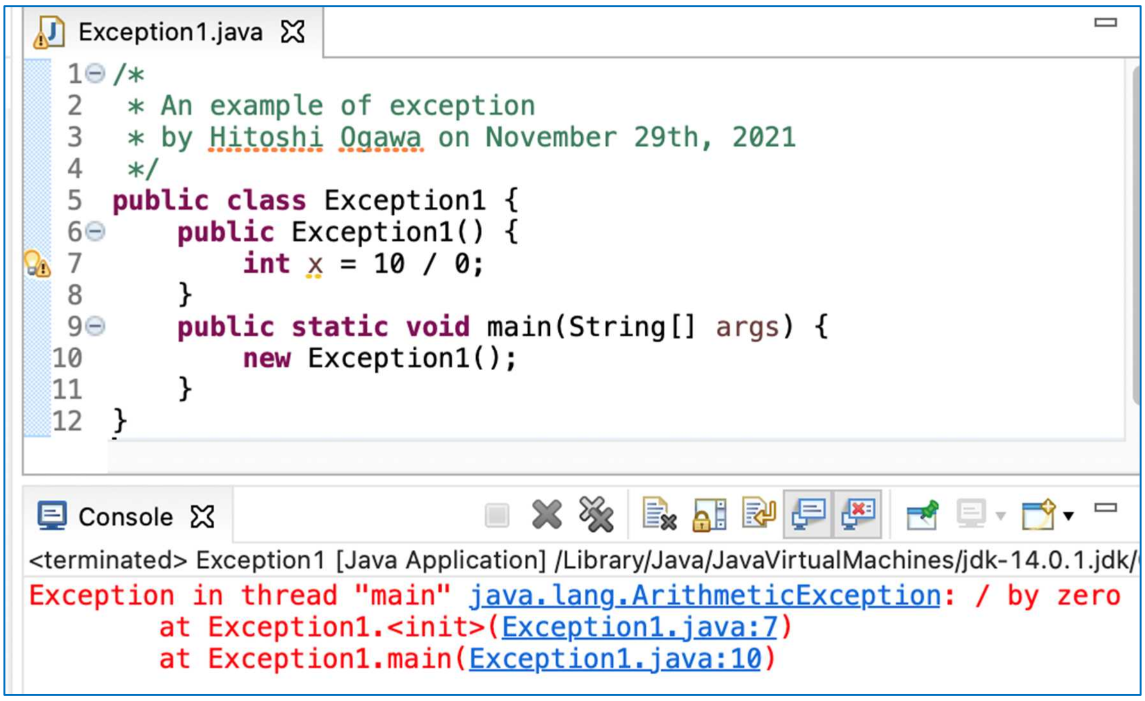
An unexpected event (error) in program operation is called an exception. An exception (or exceptional event) is a problem that arises during the execution of a program. When an exception occurs the normal flow of the program is disrupted and the program terminates abnormally, which is not recommended, therefore, the exception is to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- ✓ A user has entered an invalid data.
- ✓ A file that needs to be opened cannot be found.
- ✓ A network connection has been lost in the middle of communications or the JVM (Java Virtual Machine) has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Figure 1 shows a program that causes an error to divide by zero, and the result that the execution was stopped.



Exception1.java

```java
/*
 * An example of exception
 * by Hitoshi Ogawa on November 29th, 2021
 */
public class Exception1 {
    public Exception1() {
        int x = 10 / 0;
    }
    public static void main(String[] args) {
        new Exception1();
    }
}
```

Console

```
<terminated> Exception1 [Java Application] /Library/Java/JavaVirtualMachines/jdk-14.0.1.jdk/
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at Exception1.<init>(Exception1.java:7)
	at Exception1.main(Exception1.java:10)
```

Figure 1. A program that causes an error to divide by zero.

2. Try-Catch block

A try-catch block is used for exception handling. A try-catch block is placed around the code that might generate an exception. Code within a try-catch block is referred to as protected code, and the syntax for using try-catch looks like the following.

```
try {
    // Protected code
} catch (ExceptionType1 e) {
    // Catch block
} catch (ExceptionType2 e) {
    // Catch block
} finally {
    // The finally block always executed
}
```

The code which is prone to exception is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

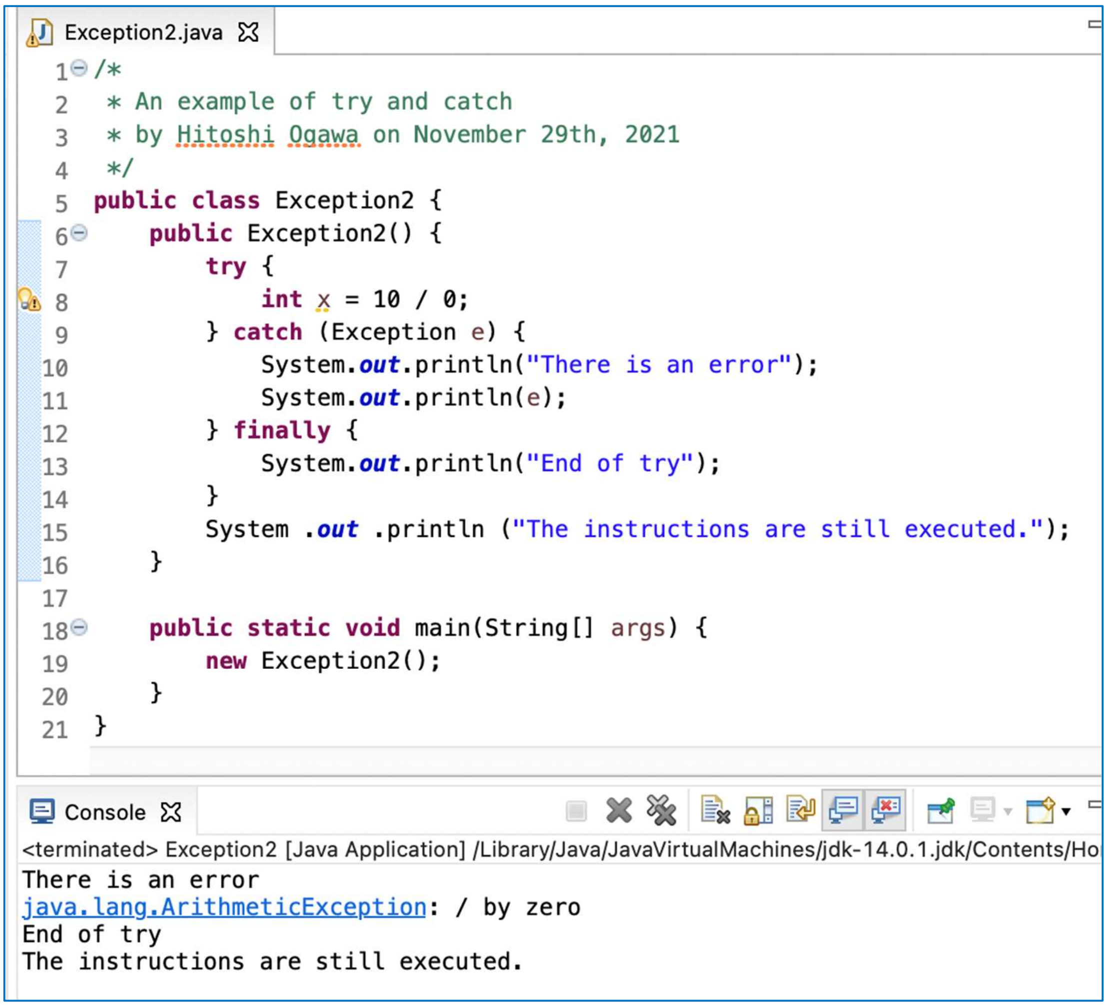
The syntax demonstrates two catch blocks, but we can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches *ExceptionType1*, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches. The main exception classes that can be used as *ExceptionType* are shown in Table 1.

The finally block follows a try block or a catch block. The code of finally block is always executed regardless of the occurrence of the exception. We do not have to write a finally block.

A program “Exception2” using try-catch is shown in Figure 2. Division of 0 on line 8 raises an exception (ArithmeticException) and the exception is thrown to the catch block. Since Exception matches this exception, lined 10 and 11 are executed. Then, a finally block is executed. Next, the instruction is executed directly under the try-catch block. The execution result also shown in Figure 2 indicates that processing is continued without stopping the program.

Table 1. The main classes for exception

Class	Description
Exception	Superclass of the classes shown below
IOException	It is thrown when an input-output operation failed or interrupted
FileNotFoundException	This Exception is raised when a file is not accessible or does not open
NullPointerException	This exception is raised when referring to the members of a null object. Null represents nothing
ArithmeticException	It is thrown when an exceptional condition has occurred in an arithmetic operation.
IndexOutOfBoundException	It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.


Exception2.java

```java
/*
 * An example of try and catch
 * by Hitoshi Ogawa on November 29th, 2021
 */
public class Exception2 {
    public Exception2() {
        try {
            int x = 10 / 0;
        } catch (Exception e) {
            System.out.println("There is an error");
            System.out.println(e);
        } finally {
            System.out.println("End of try");
        }
        System .out .println ("The instructions are still executed.");
    }

    public static void main(String[] args) {
        new Exception2();
    }
}
```

Console

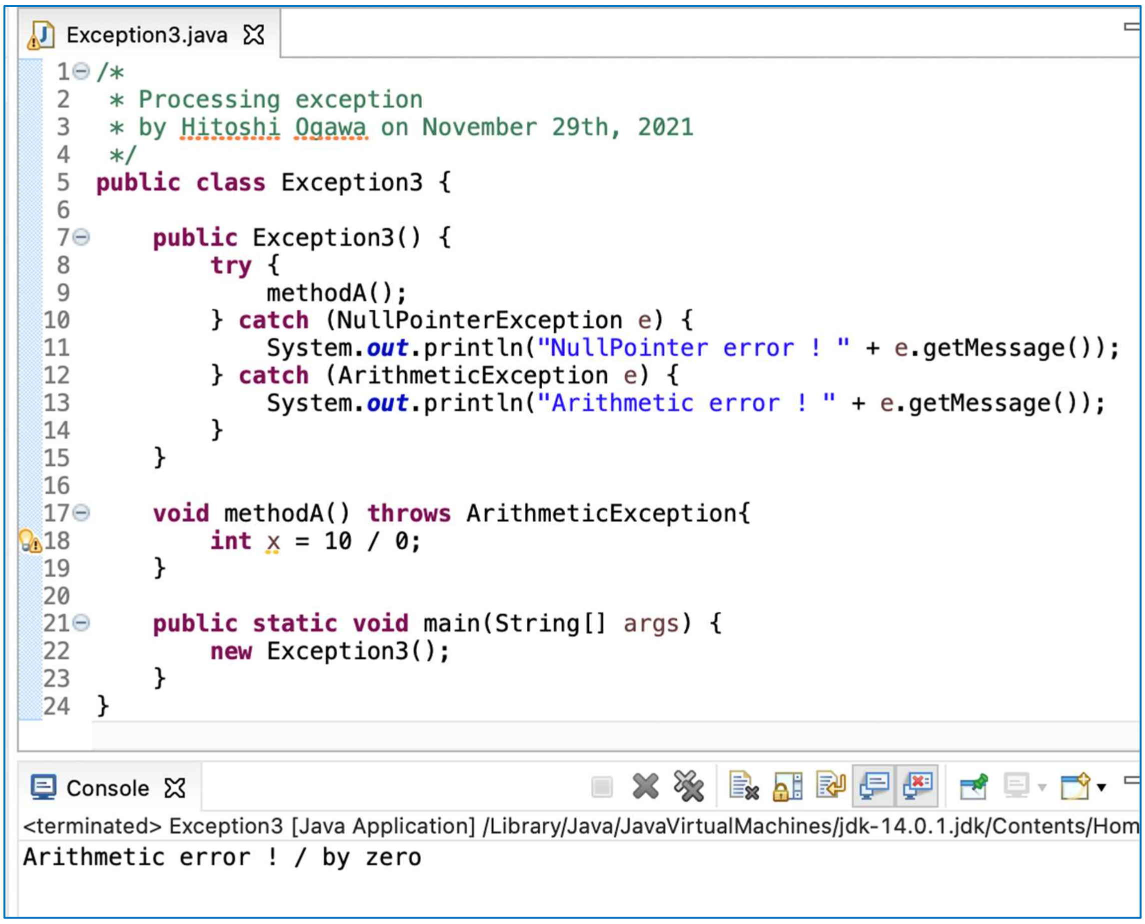
```
<terminated> Exception2 [Java Application] /Library/Java/JavaVirtualMachines/jdk-14.0.1.jdk/Contents/Ho
There is an error
java.lang.ArithmeticException: / by zero
End of try
The instructions are still executed.
```

Figure 2. The program using try-catch and the result.

If a method is called on a try block, we can program it to catch the exception raised by that method. The method to be called must be written in the following format.

```
method_head throws ExceptionType {
    ...
}
```

A program “Exception3” using **throws** is shown in Figure 3. An exception occurs with methodA (line 18). In this case, the “ArithmeticException” is thrown where methodA is called (line 9) and is not received in the catch block on line 10, but also received in the catch block on line 12. The statement “e.getMessage()” is an instruction to retrieve exception information. The execution result of Exception3 is also shown in Figure 3.



```
1  /*
2   * Processing exception
3   * by Hitoshi Ogawa on November 29th, 2021
4   */
5  public class Exception3 {
6
7      public Exception3() {
8          try {
9              methodA();
10         } catch (NullPointerException e) {
11             System.out.println("NullPointer error ! " + e.getMessage());
12         } catch (ArithmeticException e) {
13             System.out.println("Arithmetic error ! " + e.getMessage());
14         }
15     }
16
17     void methodA() throws ArithmeticException{
18         int x = 10 / 0;
19     }
20
21     public static void main(String[] args) {
22         new Exception3();
23     }
24 }
```

```
<terminated> Exception3 [Java Application] /Library/Java/JavaVirtualMachines/jdk-14.0.1.jdk/Contents/Hom
Arithmetic error ! / by zero
```

Figure 3. The program using throw and the result.

3. Standard Input Stream

A stream is a concept for data input/output. Data to be handled includes standard input/output data, data in the file, and communication data. Standard input/output generally means input/output by command prompt. Depending on the direction of data flow, there are two types of input stream and output stream.

The main classes necessary for handling input streams are introduced below.

- ✓ InputStreamReader(InputStream in)
  converts byte stream data to character stream. For standard input, use "System.in" for InputStream.
- ✓ FileReader(File file)
  reads a character stream from a character file.
- ✓ BufferedReader(Reader in)
  reads efficiently text from the character input stream by buffering letters, arrays and lines, since FIleReader reads only one character at a time. Table 2 shows the methods often used in this class. When using these methods, use exception handling to catch IOException (Exception).

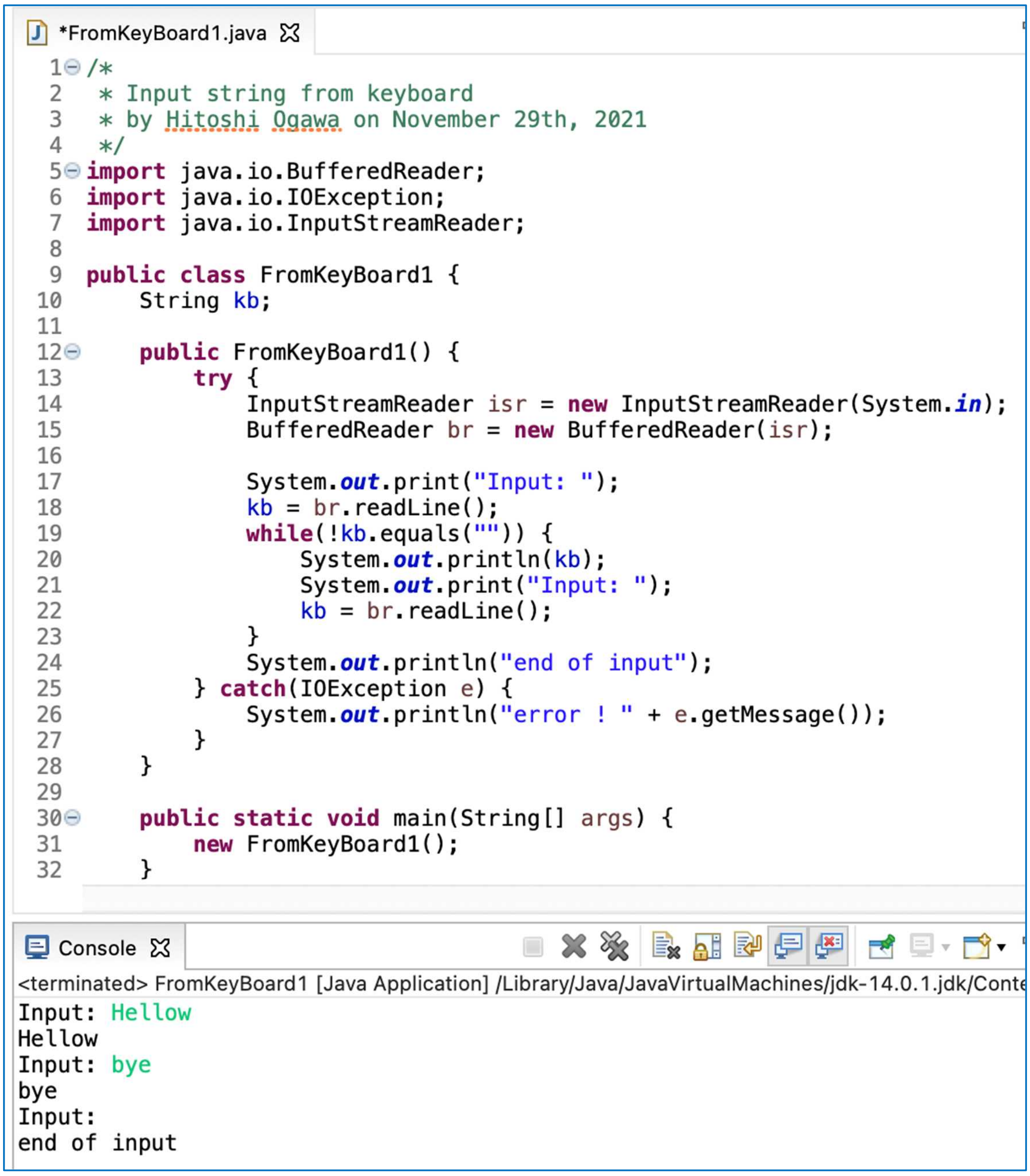
"FromKeyBoard1" class shown in Figure 4 is an example program that input data from standard input/output. Lines 5 to 7 are declarations for using Java.io packages. Exception handling of input stream processing is set (lines 13 to 27). IOException is caught in the catch block. A stream "isr" from standard input is set (line 14). The stream "isr" is registered in BufferedReader "br" (line 15). In line 17, "input: " is printed to indicate the position of the input to the user. In order not to break a line, "System.out.print" is used. The input string is assigned to the variable "str" (line 18). The condition "!str.equals("")" in line 19 checks whether the contents of "str" are empty.

While statement is to be repeated unless the contents of "str" are empty. The string entered is standard output in line 20. The user inputs the string again at lines 21 and 22. This process continues until a space is entered.

An example of execution of FromKeyBoard1 is also shown in Figure 4.

Table 2. The main methods of BufferedReader class

Method	Description
void **close()** throws IOException	Closes the stream and release all system resources associated with it
int **read()** throws IOException	Reads single character
String **readLine()** throws IOException	Reads a line of text



```java
/*
 * Input string from keyboard
 * by Hitoshi Ogawa on November 29th, 2021
 */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class FromKeyBoard1 {
    String kb;

    public FromKeyBoard1() {
        try {
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);

            System.out.print("Input: ");
            kb = br.readLine();
            while(!kb.equals("")) {
                System.out.println(kb);
                System.out.print("Input: ");
                kb = br.readLine();
            }
            System.out.println("end of input");
        } catch(IOException e) {
            System.out.println("error ! " + e.getMessage());
        }
    }

    public static void main(String[] args) {
        new FromKeyBoard1();
    }
```

```
<terminated> FromKeyBoard1 [Java Application] /Library/Java/JavaVirtualMachines/jdk-14.0.1.jdk/Conte
Input: Hellow
Hellow
Input: bye
bye
Input:
end of input
```

Figure 4. An example program that input data from standard input/output and an example of execution.

Figure 5 shows a program “FromKeyBoard2.java” that accepts only positive integers and not real numbers or other characters. This program will exit when a negative integer is entered. In general, we can determine whether characters can be converted to numbers by examining the code of the input character. However, we need to analyze every character. Especially in the case of real numbers it is very complicated. When changing input characters to numbers, distinction can be made relatively easily by using exception handling which cannot be changed.

The variable “br” of BufferedReader type is prepared to accept data from standard input (lines 15 and 16). Since the variable “br” is also used in getData method, it is declared in advance in the field (line 10). In the getData method, characters are read from the user in the same way as FromKeyBoard1 (lines 33 and 34). The entered characters are converted to integers (line 35) and returns (line 36). If it cannot be converted to an integer, it is thrown to the catch block. In the catch block, the value 0 is returned after printing the error message (line 39).

If the value from getData method is greater than 0, its value is printed (lines 20 and 21). Otherwise, the message that it is not a positive number is printed (line 23). When the value from getData method is 0 or more, input processing is continued. When the value is less than 0, the processing is terminated.

An example of execution of FromKeyBoard2 is shown in Console window in Figure 5.

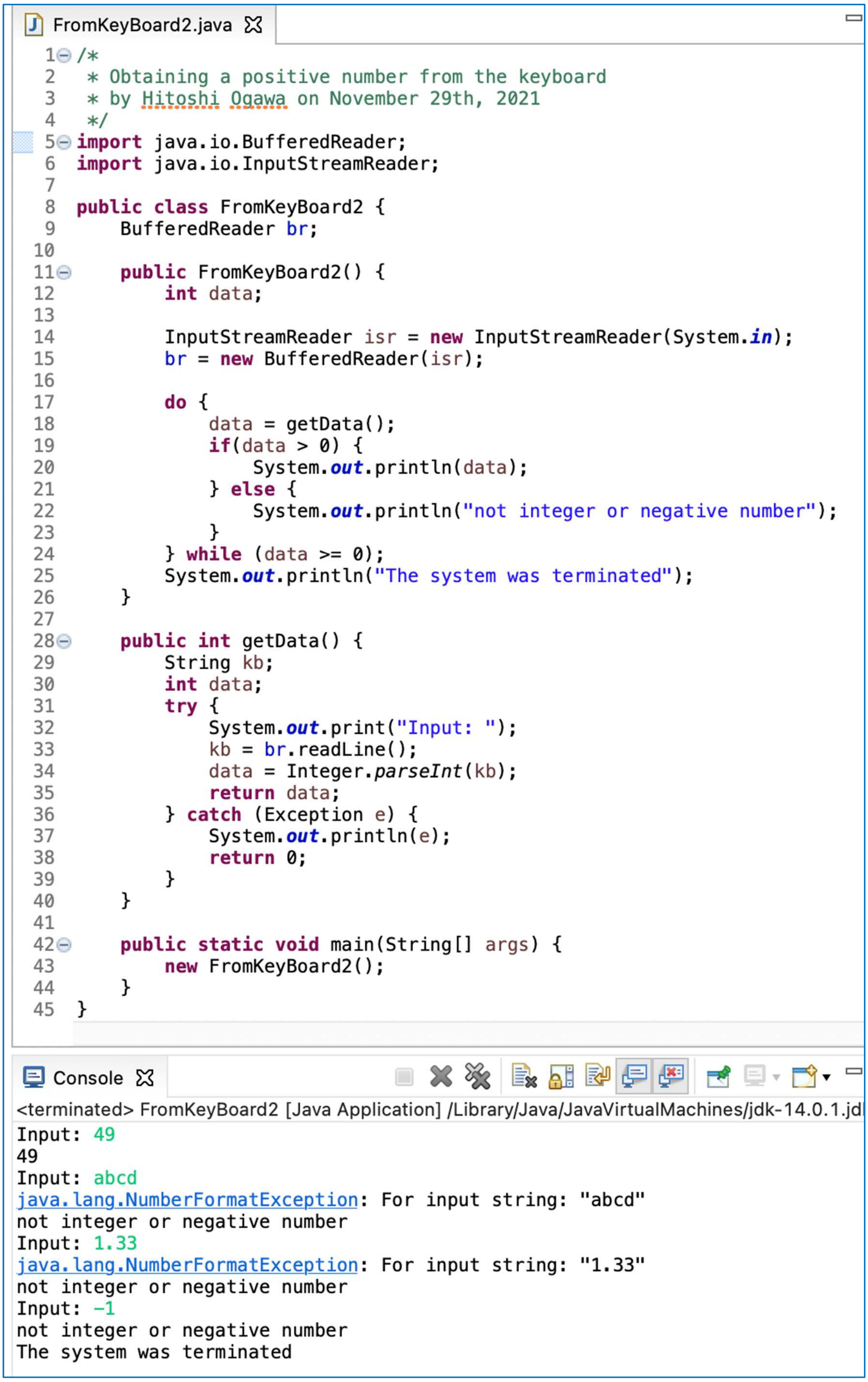
This program can be used to acquire positive numbers.

4. File Output Stream

File Output Stream is used to create a file and write data into it. The main classes necessary for handling file onput streams are introduced below.

- ✓ FileWriter(File file)
  writes a character stream to the file.
- ✓ BufferedWriter(Writer out)
  writes efficiently string to output stream by buffering characters, arrays and lines.

Table 3 shows the methods often used in this class. When using these methods, use exception handling to catch IOException (Exception).



```java
/*
 * Obtaining a positive number from the keyboard
 * by Hitoshi Ogawa on November 29th, 2021
 */
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class FromKeyBoard2 {
    BufferedReader br;

    public FromKeyBoard2() {
        int data;

        InputStreamReader isr = new InputStreamReader(System.in);
        br = new BufferedReader(isr);

        do {
            data = getData();
            if(data > 0) {
                System.out.println(data);
            } else {
                System.out.println("not integer or negative number");
            }
        } while (data >= 0);
        System.out.println("The system was terminated");
    }

    public int getData() {
        String kb;
        int data;
        try {
            System.out.print("Input: ");
            kb = br.readLine();
            data = Integer.parseInt(kb);
            return data;
        } catch (Exception e) {
            System.out.println(e);
            return 0;
        }
    }

    public static void main(String[] args) {
        new FromKeyBoard2();
    }
}
```

```
<terminated> FromKeyBoard2 [Java Application] /Library/Java/JavaVirtualMachines/jdk-14.0.1.jd
Input: 49
49
Input: abcd
java.lang.NumberFormatException: For input string: "abcd"
not integer or negative number
Input: 1.33
java.lang.NumberFormatException: For input string: "1.33"
not integer or negative number
Input: -1
not integer or negative number
The system was terminated
```

Figure 5. A program that accepts only integers

Table 3. The main methods of BufferedWriter class

Method	Description
void close() throws IOException	Flushs the stream first (forcibly output the contents of the buffer) and exit Write a character string
void newLine() throws IOException	Writes a line feed character
void write(int *c*) throws IOException	Write a single character *c*
void write(String *str*) throws IOException	Reads a line of text *str*

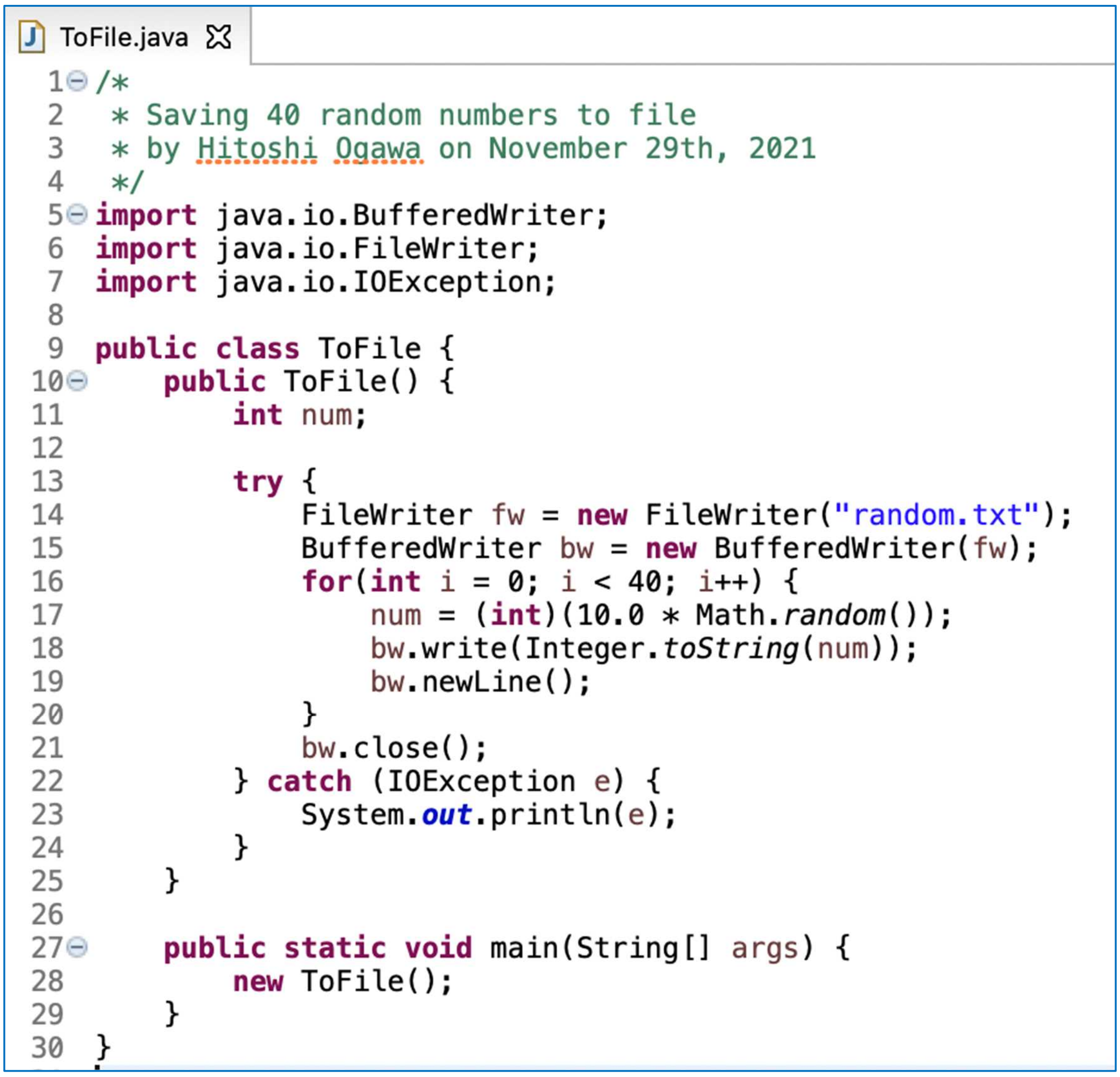
Figure 6 shows the “ToFile” class as an example of writing characters to a file. This program randomly generates 40 integers from 0 to 9 and writes it to the file “random.txt”.

A stream “fw” from file “random.txt” is set (line 14). The stream “fw” is registered in BufferedWriter “bw” (line 15).

Math.random () on line 17 calls the random method of Math class. The Math class is a special class that can be directly used without creating an instance. Randomly generate decimal numbers between 0.0 and less than 1.0. It is multiplied by 10.0, and a decimal number of 0.0 or more and less than 10.0 is obtained. Real numbers are cast, converted to integers, and assigned to the variable “num” (line 17). Refer Chapter 5 of “AWT Event Handling” (The sixth day).

On line 18, the integer num is converted to a String by” Integer.toString (num)” and written to the file. (The data written to the file must be a character string) Line feed characters are written on line 19. After repeating for statement 40 times (lines 16 to 20), “dandom.txt” is closed (line 21).

If the file “random.txt” is not displayed in Package Explorer, reflesh the Package ((3) of Appendix) or refer to (1), (2) and (3) of the Appendix.



```java
/*
 * Saving 40 random numbers to file
 * by Hitoshi Ogawa on November 29th, 2021
 */
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class ToFile {
    public ToFile() {
        int num;

        try {
            FileWriter fw = new FileWriter("random.txt");
            BufferedWriter bw = new BufferedWriter(fw);
            for(int i = 0; i < 40; i++) {
                num = (int)(10.0 * Math.random());
                bw.write(Integer.toString(num));
                bw.newLine();
            }
            bw.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }

    public static void main(String[] args) {
        new ToFile();
    }
}
```

Figure 6. An example program of writing characters to a file “random.txt”.

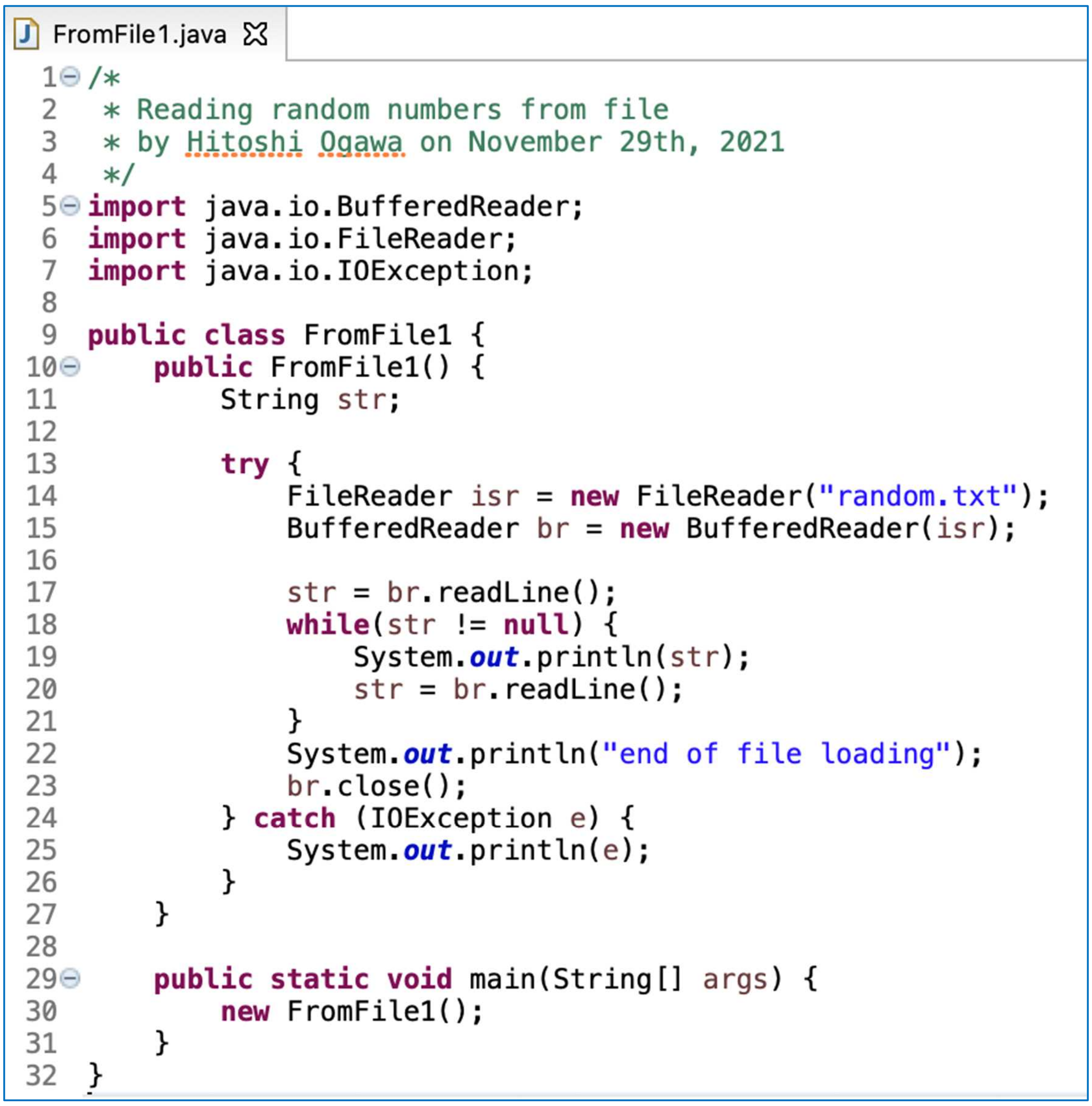
5. File Input Stream

Figure 7 shows a program “FromFile1.java” that reads the contents of the file “random.txt” created by “ToFile”.

In order to read the data from the file, “FileReader” and “BufferedReader” introduced on page 5 are used. A stream “isr” from file “random.txt” is set (line 14). The stream “isr” is registered in BufferedReader “br” (line 15).

The structure of the program is the same as “FromKeyBoard1”. However, the input request to the use is unnecessary.

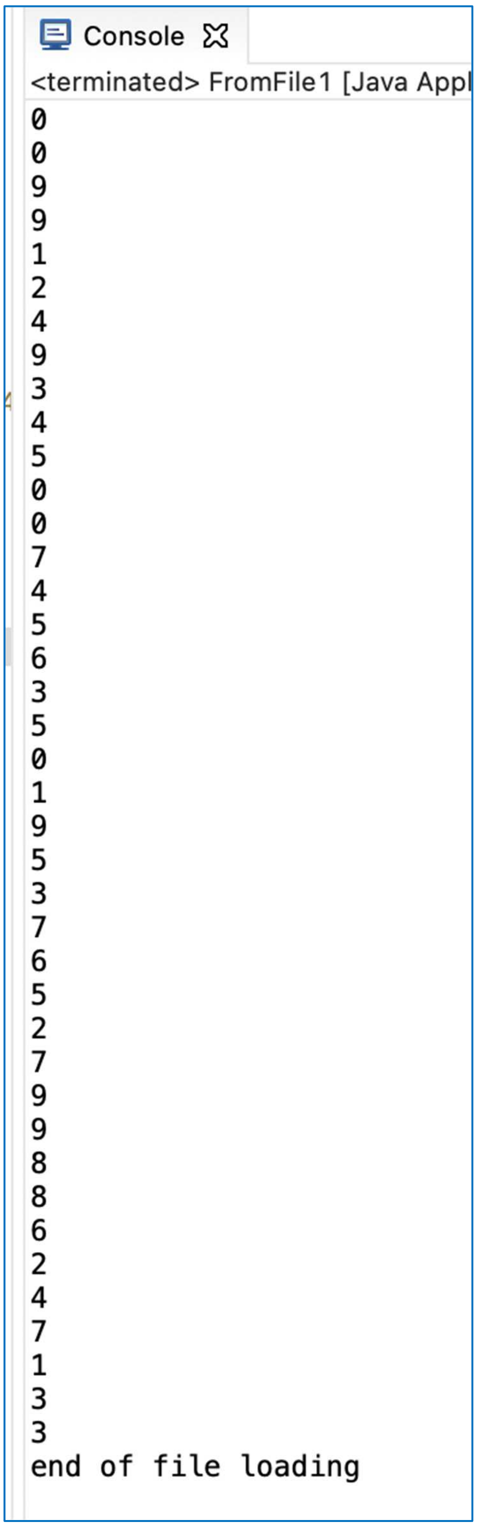
If the read data is “null”, it means that all the data of the file has been read. Figure 8 shows the execution result of “FromFile1.java”.



FromFile1.java

```java
/*
 * Reading random numbers from file
 * by Hitoshi Ogawa on November 29th, 2021
 */
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class FromFile1 {
    public FromFile1() {
        String str;

        try {
            FileReader isr = new FileReader("random.txt");
            BufferedReader br = new BufferedReader(isr);

            str = br.readLine();
            while(str != null) {
                System.out.println(str);
                str = br.readLine();
            }
            System.out.println("end of file loading");
            br.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }

    public static void main(String[] args) {
        new FromFile1();
    }
}
```

Figure 7. A program that reads the contents of the file “random.txt”.



```
<terminated> FromFile1 [Java Appl
0
0
9
9
1
2
4
9
3
4
5
0
0
7
4
5
6
3
5
0
1
9
5
3
7
6
5
2
7
9
9
8
8
6
2
4
7
1
3
3
end of file loading
```

Figure 8. The execution result of “FromFile1.java”.

Figure 9 shows the result of graphing the frequence of the numbers in the file “random.txt”. Red letters indicate the numbers created, and blue letters indicate the number created.

Figure 10 shows the program “FromFile2” that reads the data from file “random.txt” and creates a graph. To record the number for each integer, an array variable “record” is provided in the field. For “countData” method called from the constructor, the number read is converted to number and used as the index of the array “record” (lines 44 to 48).

At the beginning of the paint method, the font is defined and registered (lines 26 and 27). For numbers 0 to 9, the target, the counted number, and a bar graph of that number are drawn. The target number is displayed in red, the counted number and the bar graph are displayed in blue. Notice that each position is specified by the value of the variable i.

The value 50 of "50 * record [i]" on line 33 is set assuming that the maximum number counted is 7.

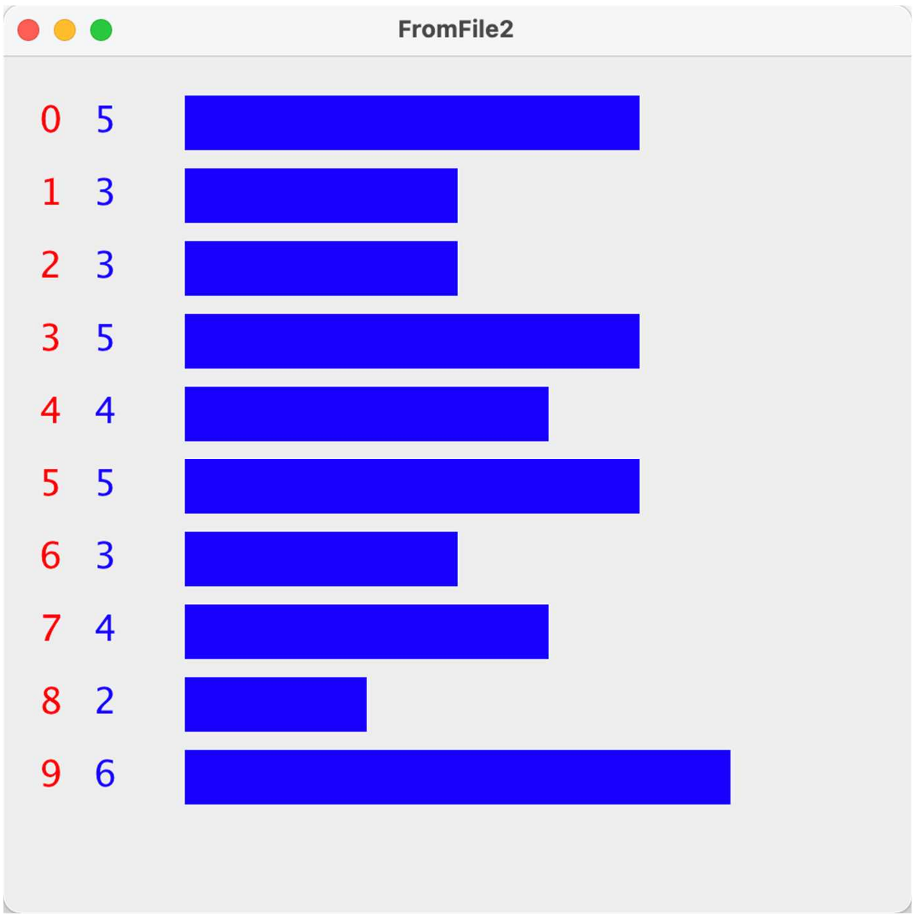
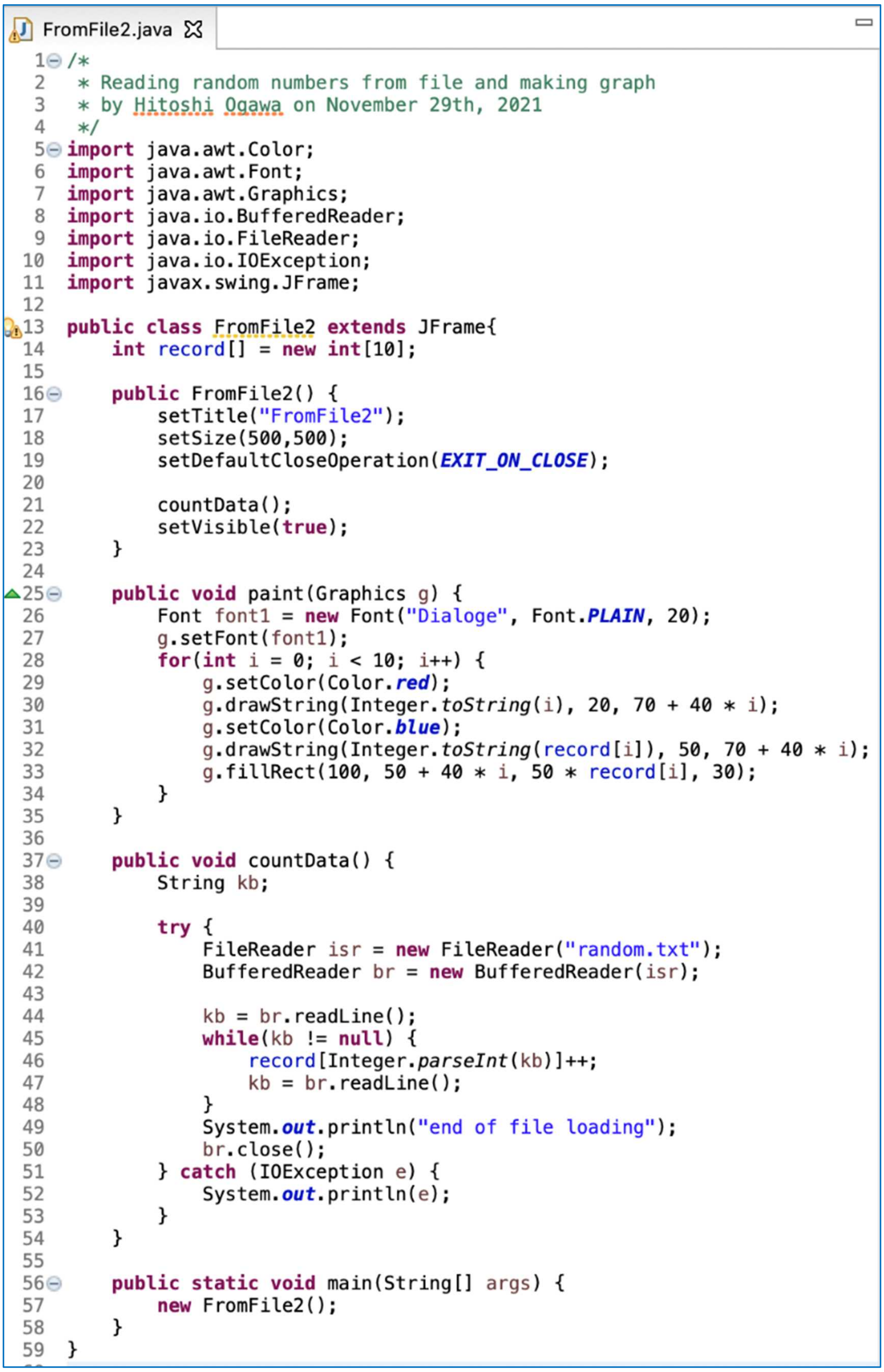


Figure 9. The execution result of “FromFile2.java”.



FromFile2.java

```java
/*
 * Reading random numbers from file and making graph
 * by Hitoshi Ogawa on November 29th, 2021
 */
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import javax.swing.JFrame;

public class FromFile2 extends JFrame{
    int record[] = new int[10];

    public FromFile2() {
        setTitle("FromFile2");
        setSize(500,500);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        countData();
        setVisible(true);
    }

    public void paint(Graphics g) {
        Font font1 = new Font("Dialoge", Font.PLAIN, 20);
        g.setFont(font1);
        for(int i = 0; i < 10; i++) {
            g.setColor(Color.red);
            g.drawString(Integer.toString(i), 20, 70 + 40 * i);
            g.setColor(Color.blue);
            g.drawString(Integer.toString(record[i]), 50, 70 + 40 * i);
            g.fillRect(100, 50 + 40 * i, 50 * record[i], 30);
        }
    }

    public void countData() {
        String kb;

        try {
            FileReader isr = new FileReader("random.txt");
            BufferedReader br = new BufferedReader(isr);

            kb = br.readLine();
            while(kb != null) {
                record[Integer.parseInt(kb)]++;
                kb = br.readLine();
            }
            System.out.println("end of file loading");
            br.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }

    public static void main(String[] args) {
        new FromFile2();
    }
}
```

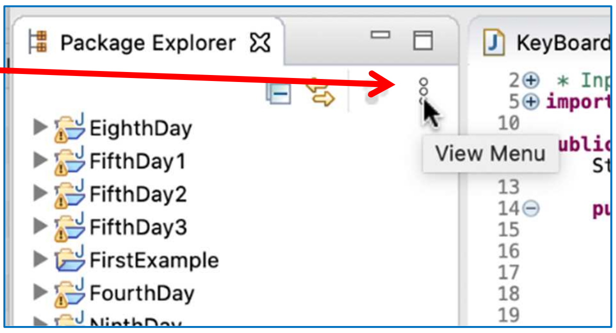
Figure 10. A program that creates a graph of file “random.txt” data.

Appendix

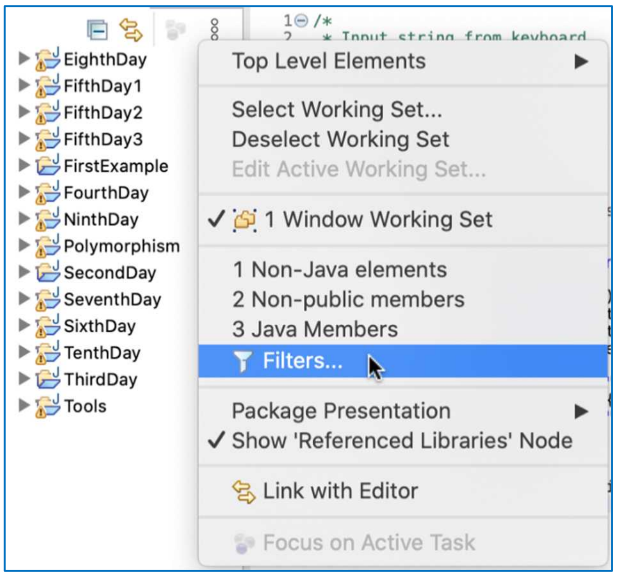
If you want to display the file created by the program Package Explorer, follow the steps below.

(1) Open Filters

- Press the "View Menu" mark

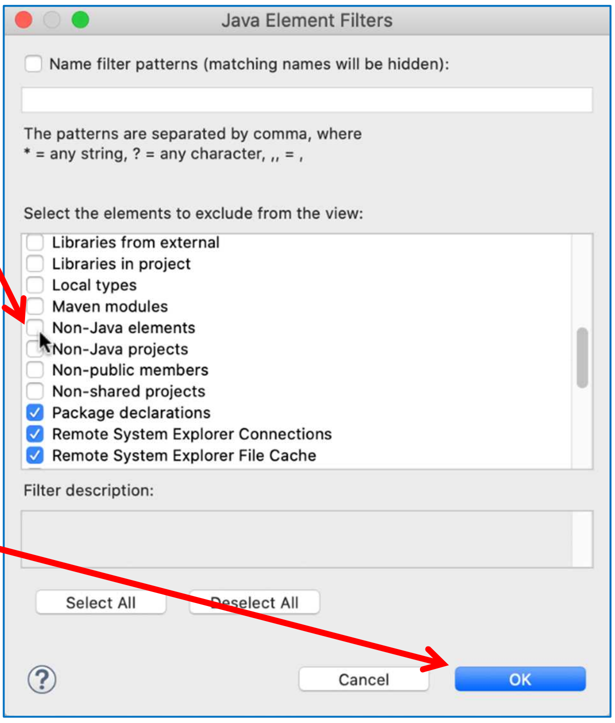


- Select Filters



(2) Set FilterRemove the mark

- Remove the mark of "Non-Java elements"



Java Element Filters

- [ ] Name filter patterns (matching names will be hidden):

The patterns are separated by comma, where
* = any string, ? = any character, ,, = ,

Select the elements to exclude from the view:

- [ ] Libraries from external
- [ ] Libraries in project
- [ ] Local types
- [ ] Maven modules
- [ ] Non-Java elements
- [ ] Non-Java projects
- [ ] Non-public members
- [ ] Non-shared projects
- [x] Package declarations
- [x] Remote System Explorer Connections
- [x] Remote System Explorer File Cache

Filter description:

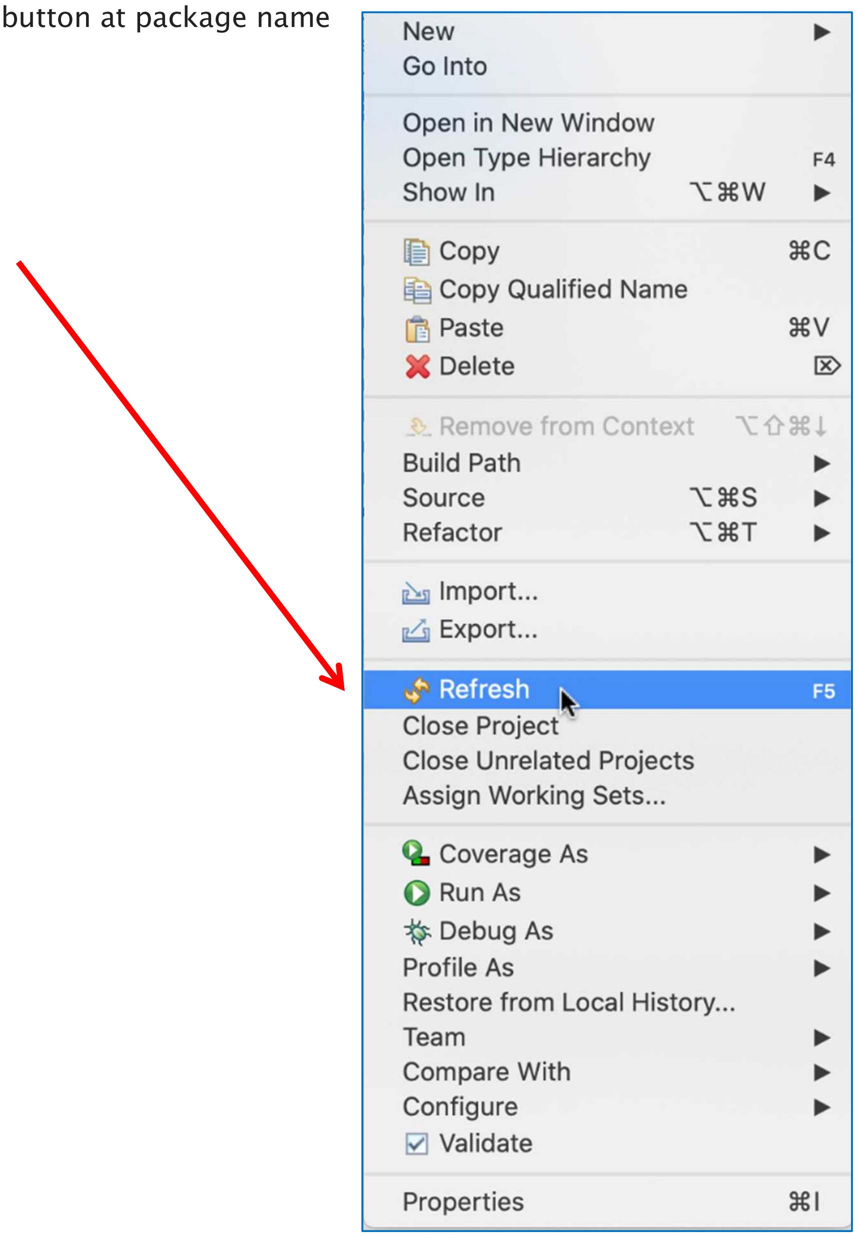
- Click "OK" button

If the file name is not displayed,

(3) Refresh Package

- Press by right button at package name

- Select Refresh


New ▶
Go Into
Open in New Window
Open Type Hierarchy F4
Show In ⌥⌘W ▶
Copy ⌘C
Copy Qualified Name
Paste ⌘V
Delete ⌦
Remove from Context ⌥⇧⌘↓
Build Path ▶
Source ⌥⌘S ▶
Refactor ⌥⌘T ▶
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Coverage As ▶
Run As ▶
Debug As ▶
Profile As ▶
Restore from Local History...
Team ▶
Compare With ▶
Configure ▶
Validate
Properties ⌘I