

## Module 2

---

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 2 – Handling Text and Exceptions

## Learning Objectives

---

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

1. Explain exceptions and exception handling.
2. Write a try-catch block to handle exceptions.
3. Throw exceptions in a method.
4. Declare exceptions in a method header.
5. Handle checked and unchecked exceptions.
6. Re-throw exceptions.
7. Create chained exceptions.
8. Use the finally clause.
9. Use try-with-resources.
10. Write data to a file using PrintWriter and Formatter classes.
11. Read from a file using Scanner classes.
12. Explain regular expressions.
13. Use the StringBuilder class.

### Module 2 Study Guide and Deliverables

January 23 – January 29

<b>Topics:</b>	Lecture 2: Handling Exceptions and Files
<b>Readings:</b>	<ul style="list-style-type: none"><li>• Module 2 online content</li><li>• Deitel &amp; Deitel, Chapter 11, Chapter 14, and Chapter 15</li><li>• Instructor's live class slides</li></ul>
<b>Assignments:</b>	Assignment 2 due <b>Tuesday, February 6 at 6:00 AM ET</b> (Access at "Assignments" on the left-hand course menu).
<b>Live Classrooms:</b>	Join the live classroom and the facilitator's live office hour session at "Live Classroom/Offices" > "Live Classroom" > Launch Meeting. <ul style="list-style-type: none"><li>• <b>Friday, January 26, 6:00 – 8:00 PM ET</b></li><li>• Live Office: Schedule with facilitators as long as there are questions</li></ul>

## Module Welcome and Introduction

---

met\_cs622\_18\_su1\_ebraude\_mod2 video cannot be displayed here

## Exception Handling

---

Exception handling features of Java allow a program to continue normal execution by handling exceptional (or erroneous or bad) situations that may arise at run time. Exception handling is used to tackle run-time—as opposed to compile-time—issues. We say that run-time problems are "thrown" as exceptions. An "exception" is a Java object that represents a condition that prevents execution from proceeding normally. If those problems are not dealt with by exception handling features, the program would crash (abort) without getting a chance to come to a natural halt.

### Try-catch Block

Exception handling is achieved in Java with a special syntax called "try and catch blocks". We illustrate the use of try-catch by considering the simple problem of dividing an integer by another. Look at the following two programs (the first without and the second with exception handling).

The first program (class DivByZeroNoExcep) aborts when the divisor is zero, but the second (class DivByZeroExcep) comes to a controlled stop under the same situation. In other words, in the first program, if the divisor is zero, it is not possible to have the program perform any other computation after the division step; the program must terminate immediately and unconditionally. In the second program, however, we can continue with further computation for as long as the logic would allow, eventually coming to what might be called a natural or normal halt; that is, the program comes to a complete stop at a time of the programmer's choosing, having executed any number of programmer-chosen steps even *after* the division has resulted in an error. In that sense, exception handling can be thought of as a form of "graceful degradation."

First program - without exception handling:

```
import java.util.Scanner;

public class DivByZeroNoExcep
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);

        int dividend = input.nextInt();
        int divisor = input.nextInt();

        int result = dividend / divisor;

        System.out.println("This line and the following are not reached when divisor is zero");
        System.out.printf("Result = %d\n", result);
    }
}
```

Output of the program – without exception handling

```

Administrator: Command Prompt
C:\Users\uday\UdayJava\excep>javac DivByZeroNoExcep.java
C:\Users\uday\UdayJava\excep>java DivByZeroNoExcep
8
5
This line and the following are not reached when divisor is zero
Result = 1
C:\Users\uday\UdayJava\excep>java DivByZeroNoExcep
5
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at DivByZeroNoExcep.main(DivByZeroNoExcep.java:12)
C:\Users\uday\UdayJava\excep>java DivByZeroNoExcep
67
23456
This line and the following are not reached when divisor is zero
Result = 0
C:\Users\uday\UdayJava\excep>

```

First program – with exception handling:

```

import java.util.Scanner;

public class DivByZeroExcep
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);

        int dividend = input.nextInt();
        int divisor = input.nextInt();

        try
        {
            int result = dividend / divisor;
            System.out.printf("Result = %d\n", result);
        }

        catch (ArithmeticException ex)
        {
            System.out.println("Attempt to divide by zero; re-run with non-zero divisor");
        }

        System.out.println("This line is reached even when divisor is zero");
    }
}

```

```

Administrator: Command Prompt
C:\Users\uday\UdayJava\excep>javac DivByZeroExcep.java
C:\Users\uday\UdayJava\excep>java DivByZeroExcep
6
0
Attempt to divide by zero; re-run with non-zero divisor
This line is reached even when divisor is zero
C:\Users\uday\UdayJava\excep>java DivByZeroExcep
45
21
Result = 2
This line is reached even when divisor is zero
C:\Users\uday\UdayJava\excep>java DivByZeroExcep
0
5
Result = 0
This line is reached even when divisor is zero
C:\Users\uday\UdayJava\excep>

```

As already mentioned, class DivByZeroNoExcep includes no programmer-written exception handling features. In DivByZeroNoExcep, the statement

```
int result = dividend / divisor;
```

automatically throws a problem (an ArithmeticException) (there is no explicit "throw" statement in the program, an idea that we re-visit later in this module) when the divisor is zero, and the program aborts.

In `DivByZeroExcep`, the potentially error-causing statement, the division statement, is placed in a "try" block, which is followed by a "catch" block. The idea is that an error (exception) thrown from a try block can be caught and processed in a catch block. Like the `DivByZeroNoExcep` class, the `DivByZeroExcep` class issues no explicit "throw" statement; the exception is automatically thrown.

Class `DivByZeroExcep` has only one catch block, but in general multiple catch blocks can be present for a given try block (in that case, when an exception is thrown, the catch blocks are scanned in sequence – from top to bottom – and the first catch block with an argument match is executed; the others are ignored).

When a statement in a try block causes a run-time problem (throws an exception), the remaining statements in the try block are skipped and control goes to one of the catch blocks (the matching catch block).

Note that an exception (error) is "thrown" from a method, and either the method itself or its caller is responsible for "catching" and processing the exception. Both exception throwing and exception catching can take place in the same method, as in class `DivByZeroExcep`. The next example, a slight modification of `DivByZeroExcep`, shows two methods, one throwing the exception, the other catching it.

```
import java.util.Scanner;

public class DivByZeroExcepTwoMethods
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);

        int dividend = input.nextInt();
        int divisor = input.nextInt();

        try
        {
            int result = producequotient(dividend, divisor);
            System.out.printf("Result = %d\n", result);
        }

        catch (ArithmeticException ex)
        {
            System.out.println("Attempt to divide by zero; re-run with non-zero divisor");
        }

        System.out.println("This line is reached even when divisor is zero");
    }

    public static int producequotient(int dividend, int divisor)
    {
        return dividend / divisor;
    }
}
```

In the above program, method `producequotient()` throws (again, implicitly) the exception (an `ArithmeticException`), and `main()` catches it.

We can make `producequotient()` declare the exception by adding the clause "throws `ArithmeticException`" after the method header, as shown below. Because the exception `ArithmeticException` is an "unchecked" exception (to be discussed shortly), this "throws `ArithmeticException`" clause is optional in this example.

```
public static int producequotient(int dividend, int divisor) throws ArithmeticException
{
    return dividend / divisor;
}
```

An explicit throwing of the exception is shown in the following class which is a modification to `DivByZeroExcepTwoMethods`. Observe how the `producequotient()` method's inside differs from the version discussed earlier. The "throws `ArithmeticException`" declaration, again, is optional after the `producequotient()` header because `ArithmeticException` is an unchecked exception.

Note the important difference between the following two (we will have occasion to provide more illustrations of this in the examples that will follow):

- "throw new `Exception()`" in the method body
- "throws `Exception`" in the same method's declaration

```
import java.util.Scanner;

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

```

{
    Scanner input = new Scanner(System.in);

    int dividend = input.nextInt();
    int divisor = input.nextInt();

    try
    {
        int result = producequotient(dividend, divisor);
        System.out.printf("Result = %d\n", result);
    }

    catch (ArithmeticException ex)
    {
        System.out.println("Attempt to divide by zero; re-run with non-zero divisor");
    }

    System.out.println("This line is reached even when divisor is zero");
}

public static int producequotient(int dividend, int divisor) throws ArithmeticException
{
    if (divisor == 0)
        throw new ArithmeticException();

    return dividend / divisor;
}
}

```

As the above code shows, the primary advantage of exception processing is the separation of the detection of a run-time error (usually in a called method) from the handling (processing) of the error (usually in the calling method).

The try and catch blocks are programming language blocks, in the sense that a block defines its own scope and local variables. The following example illustrates this issue (please read the comments in the code):

```

public class Test1
{
    public static void main(String[] args)
    {
        try
        {
            int i = 10;
            // other statements here
        }
        catch (Exception ex)
        {
            int j = 20;
            // other statements here
        }

        i = 20;    // wrong: i cannot be sensed; try block local variable i
        j = 100;  // wrong: try and catch each is a "block" - usual scope rules of blocks apply
    }
}

```

## Checked and Unchecked Exceptions

Exceptions are objects of classes. The exception hierarchy has the Throwable class at the top, and the class Exception is a sub-class of Throwable. Frequently encountered sub-classes of Exception include RuntimeException, IOException, and ClassNotFoundException. All exceptions except RuntimeException and Error (and their subclasses) are checked exceptions. Unchecked exceptions comprise RuntimeException and Error and their subclasses. Check out [further details about checked and unchecked exceptions](#).

The golden rule in exception handling is that **checked exceptions must be caught or declared** (the compiler objects if this policy is violated). No corresponding stipulation applies to unchecked exceptions. This critically important aspect is illustrated in the following examples.

In class Test2 below, the exception type Exception must be treated as a checked exception, because it is a super-class that has both checked and unchecked exceptions as its sub-classes.

The catch block of m1() does catch the exception thrown by its try block, but re-throws it. This means that the caller of m1(), namely main(), must either catch (and process) this exception or declare that main() throws Exception.

```
public class Test2
{
    public static void main(String[] args)
    {
        try
        {
            m1();
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }

    public static void m1() throws Exception //This "throws Exception" is a must here
    {
        try
        {
            throw new Exception();
        }
        catch (Exception ex)
        {
            throw ex;
        }
    }
}
```

In the following code (class Test2A), method m1() does not have to declare the exception, because it catches and processes it (does not re-throw it).

```
public class Test2A
{
    public static void main(String[] args)
    {
        try
        {
            m1();
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }

    public static void m1() //m1() does NOT have to declare Exception because it catches it
    {
        try
        {
            throw new Exception();
        }
        catch (Exception ex)
        {
            System.out.println("exception caught and processed in m1()");
        }
    }
}
```

The following examples (class Test3) shows a case where the main() method must declare the exception (because it does not catch-and-process it inside its body).

```
public class Test3
{
    public static void main(String[] args) throws Exception // excep. declaration is a must
    {
        int i = 1;
        if (i > 0)
            throw new Exception();
    }
}
```

If a method invokes another that throws but does not catch an exception, the first method must either catch-and-process the exception or declare the exception. See how `m1()` invokes `m2()` (and also how `main()` invokes `m1()`) in class `Test4` below:

```
public class Test4
{
    public static void main(String[] args)
    {
        try
        {
            m1();
        }
        catch (Exception ex)
        {
            System.out.println("Exception caught and processed in main()");
            ex.printStackTrace();
        }
    }

    // Use any one of the following two versions of m1()

    /*****
    public static void m1() throws Exception //This "throws Exception" is a must here
    {
        m2();
    }
    *****/

    /*****/
    public static void m1()
    {
        try
        {
            m2();
        }
        catch(Exception ex)
        {
            System.out.println("Exception caught and processed in m1()");
            ex.printStackTrace();
        }
    }
    /*****/

    public static void m2() throws Exception //This "throws Exception" is a must here
    {
        int i = 1;
        if (i > 0)
            throw new Exception();
    }
}
```

## Order of the Catch Blocks

Note that:

- In a catch block, a formal argument of a superclass exception matches an actual argument (a thrown exception) of a subclass exception. This makes the next item important.
- The order in which different exceptions appear in multiple catch blocks is important. The rule is to place subclasses before a superclass (otherwise, if the superclass is placed before subclasses, it will catch all the subclasses). The compiler checks for this.

Consider class `Test5` (below). First, note that `m1()` catches `IOException`, but not `Exception`. But `Exception` is a super-class of `IOException`; therefore `m1()` must declare `Exception` (recall the either-catch-or-declare rule for checked exceptions). The following program compiles and runs fine.

```
import java.io.*;

public class Test5
{
```

```

public static void main(String[] args)
{
    try
    {
        m1();
    }
    catch (Exception ex)
    {
        System.out.println("Exception caught and processed in main()");
        ex.printStackTrace();
    }
}

public static void m1() throws Exception
    // Must either catch or declare Exception (that is thrown by m2()).
    // Catching IOException alone is not enough.
{
    try
    {
        m2();
        m3();
    }

    catch(IOException ioex)
    {
        System.out.println("IOException caught and processed in m1()");
        ioex.printStackTrace();
    }
}

public static void m2() throws Exception // "throws Exception" is a must here
{
    int i = 1;
    if (i > 0)
        throw new Exception();
}

public static void m3() throws IOException // "throws IOException" is a must here
{
    int i = 1;
    if (i > 0)
        throw new IOException();
}
}

```

Next, consider altering m1() in the above code to provide a second catch block for Exception. Will the following version do?

```

public static void m1()
{
    try
    {
        m2();
        m3();
    }

    catch(Exception ex)
    {
        System.out.println("Exception caught and processed in m1()");
        ex.printStackTrace();
    }

    catch(IOException ioex)
    {
        System.out.println("IOException caught and processed in m1()");
        ioex.printStackTrace();
    }
}

```

The answer is no. Because an IOException thrown by m3() (from the invocation from within m1()) will always be caught by the first catch in m1(), which was not the purpose behind providing a separate catch on IOException. (Happily, the compiler catches this as a syntax error.) The solution, therefore, is to switch the order of the two catch blocks,



as follows:

```
public static void m1()
{
    try
    {
        m2();
        m3();
    }

    catch(IOException ioex)
    {
        System.out.println("IOException caught and processed in m1()");
        ioex.printStackTrace();
    }

    catch(Exception ex)
    {
        System.out.println("Exception caught and processed in m1()");
        ex.printStackTrace();
    }
}
```

Consider another question: In the above situation, is it OK to dispense with the subclass catch, keeping only the superclass catch block, on the ground that both types of exceptions—IOException and Exception—would be caught by catch (Exception)? The answer is (probably) no, because apparently the user wanted two *different* types of processing corresponding to these two types of exception.

In this context, let us note an important restriction (we will see an example of the effect of this restriction in the discussion of exception handling for the Thread.sleep() method in multithreading in Module 6):

If a method does not declare exceptions in the superclass, no subclass can override it to declare exceptions.

Let us reiterate that the *either-catch-or-declare* rule does not apply to unchecked exceptions (see the class Test7 below), even though it may often be good programming practice to do so. Be careful, however, that sprinkling exception checking too liberally is not a good idea, either. (Why not?)

```
// For unchecked exceptions, catch-or-declare is optional

public class Test7
{
    public static void main(String[] args)
    {
        try
        {
            m1();
        }
        catch(IllegalArgumentException ex)
        {
            System.out.println("Exception caught and processed in main()");
            ex.printStackTrace();
        }
        System.out.println("The part after exception executed in main()");
    }

    public static void m1() // catch or declare optional, since unchecked excep
    {
        m2(555);
    }

    public static void m2(int i) throws IllegalArgumentException // declaration optional
    {
        if (i > 0)
            throw new IllegalArgumentException();

        System.out.println("Unreachable if excep thrown in m2()");
    }
}
```

## Re-throw an Exception

The following example shows checked exceptions and re-throw:

```
public class Test8
{
    public static void main(String[] args)
    {
        try
        {
            m1();
        }
        catch(Exception ex)
        {
            System.out.println("Exception caught and processed in main()");
            ex.printStackTrace();
        }
        System.out.println("The part after exception executed in main()");
    }

    public static void m1() throws Exception // mandatory declaration here
    {
        try
        {
            m2(555);
        }
        catch (Exception ex)
        {
            System.out.println("Exception from m2() caught in m1() ... and rethrown");
            throw ex;
        }
        System.out.println("The part after exception executed in m1()");
    }

    public static void m2(int i) throws Exception // mandatory declaration here
    {
        if (i > 0)
            throw new Exception();

        System.out.println("Unreachable if excep thrown in m2()");
    }
}
```

## The Finally Clause

A "finally" block can be added after the catch block(s). The code in the finally block executes regardless of whether or not an exception occurs in the try block or of whether or not an exception is caught. Here's an example:

```
//finally

public class Test9
{
    public static void main(String[] args)
    {
        try
        {
            m1();
        }
        catch (Exception ex)
        {
            System.out.println("Exception caught and processed in main()");
            ex.printStackTrace();
        }
        System.out.println("The part after exception executed in main()");
    }

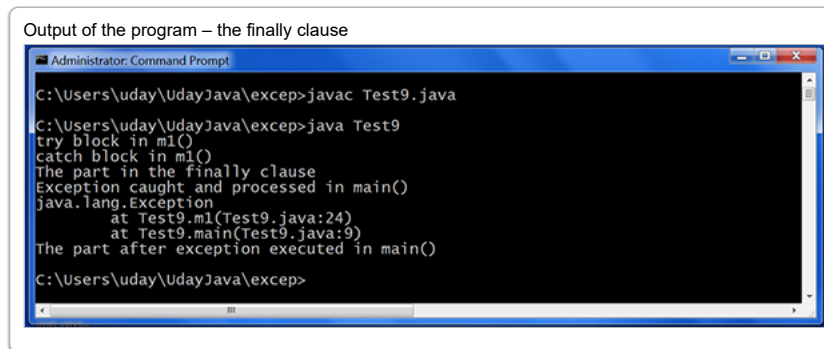
    public static void m1() throws Exception //This "throws Exception" is a must here
    {
        try
        {
```

```

        System.out.println("try block in m1()");
        throw new Exception();
    }
    catch (Exception ex)
    {
        System.out.println("catch block in m1()");
        throw ex;
    }
    finally
    {
        System.out.println("The part in the finally clause");
    }
}
}

```

The output is shown below:



If the finally clause is present, catch may be omitted, as shown in the following:

```

public class Test9A
{
    public static void main(String[] args)
    {
        try
        {
            m1();
        }
        catch (Exception ex)
        {
            System.out.println("Exception caught and processed in main()");
            ex.printStackTrace();
        }
        System.out.println("The part after exception executed in main()");
    }

    public static void m1() throws Exception // declaration is a must here
    {
        try
        {
            System.out.println("try block in m1()");
            throw new Exception();
        }

        finally
        {
            System.out.println("The part in the finally clause");
        }
    }
}

```

## Chained Exceptions

Chaining of exceptions arises when an exception is thrown along with another exception. For instance, the catch block in m1() in the following example creates (and throws) a chained exception by passing two arguments to the constructor of Exception: a string, and a previously thrown exception from m2().

```

public class Test10
{
    public static void main(String[] args)
    {
        try
        {
            m1();
        }
        catch(Exception ex)
        {
            System.out.println("Exception caught and processed in main()");
            ex.printStackTrace();
        }
        System.out.println("The part after exception executed in main()");
    }

    public static void m1() throws Exception
    {
        try
        {
            m2(555);
        }
        catch (Exception ex)
        {
            throw new Exception("Excep from m1", ex); // chained exception
        }
    }

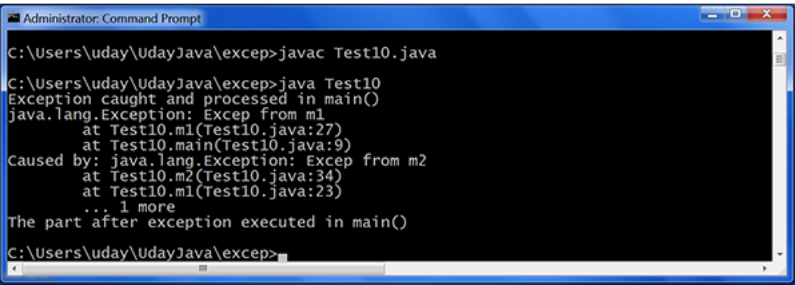
    public static void m2(int i) throws Exception
    {
        if (i > 0)
            throw new Exception("Excep from m2");

        System.out.println("Unreachable if excep thrown in m2()");
    }
}

```

The output of the above program is shown below:

Output of the program – chained exceptions



```

Administrator: Command Prompt
C:\Users\uday\UdayJava\excep>javac Test10.java
C:\Users\uday\UdayJava\excep>java Test10
Exception caught and processed in main()
java.lang.Exception: Excep from m1
    at Test10.m1(Test10.java:27)
    at Test10.main(Test10.java:9)
Caused by: java.lang.Exception: Excep from m2
    at Test10.m2(Test10.java:34)
    at Test10.m1(Test10.java:23)
    ... 1 more
The part after exception executed in main()
C:\Users\uday\UdayJava\excep>

```

## Text Input/Output

Data stored in a program's memory variables are temporary in nature and are no longer available once the program ends. Data stored on secondary storage, such as disks, are of a more permanent nature. Typically, data are stored in secondary storage in the form of files. In Java, files are basically streams of bytes. Two types of files are considered in programming: text files (character-based representation, with one character consuming two bytes), and binary or raw files (byte-based representation, with an integer taking four bytes, etc.). This module focuses on sequential-access text files; binary files are discussed in Module 4.

## Formatter

A sequential-access text file can be created with the `Formatter` object in Java. The following program creates a file named `poets.txt` in the current directory on disk.

```
// Formatter: Create and write to a textfile

import java.util.Formatter;
import java.io.FileNotFoundException;

public class File1
{
    public static void main(String[] args)
    {
        Formatter outfile = null;

        try
        {
            outfile = new Formatter("poets.txt"); // open file
        }
        catch (FileNotFoundException ex)
        {
            System.err.println("Cannot open file ... quitting");
        }

        outfile.format("Wilfred Owen           %3d\n", 25);
        outfile.format("W.B. Yeats           %3d\n", 50);
        outfile.format("Rabindranath Tagore      %3d\n", 2000);

        outfile.close();
    }
}
```

The Formatter creates a named file if the file doesn't already exist. If the file did exist, its contents are discarded. In the present example, a Formatter object named outfile is created, and outfile's format() method is used to write data on the file in the specified format. The close() statement is used to close the file.

The output of the program is the file poets.txt with the following contents:

```
Wilfred Owen           25
W.B. Yeats             50
Rabindranath Tagore    2000
```

## Scanner

The Scanner class is used to read data from a text file. The following class (class File2) provides an example of file reading using Scanner. The program assumes that a textfile named ghosts.txt already resides in the same directory on disk, with the following content:

```
12  Ebenezer           25  Dickens
45  Baskervilles       50  Doyle
99  Canterville        20  Wilde
```

This example uses the File class. The File class contains the methods for retrieving the properties of a file/directory. Note that the File class does not contain methods for reading or writing file contents. Constructing a File instance does not create a file on disk. A File instance can be created for any filename no matter whether the named file exists or not. The constructor File(String) creates a File instance (object) for the specified string, where the string may be a file or a directory. (Other overloads of the constructor exist; see java.io.File.) Note that the Scanner constructor uses a File object as its argument.

```
// Scanner: read from a textfile

import java.util.Scanner;
import java.io.File;
import java.io.IOException;

public class File2
{
    public static void main(String[] args) throws IOException
    {
        Scanner infile = new Scanner(new File("ghosts.txt"));

        while (infile.hasNext())
        {
            System.out.printf("%-3d %-20s %-3d %-20s\n",
                              infile.nextInt(),

```

```

        infile.next(),
        infile.nextInt(),
        infile.next());
    }

    infile.close();
}

```

The above program reads data from the file `ghosts.txt` until the end-of-file marker of `ghosts.txt` is reached (at that point `hasNext()` returns false). The token-reading methods `nextInt()` and `next()` read the next integer and string respectively (the default delimiter is one or more white space characters).

## PrintWriter

In addition to the `Formatter` class, the `PrintWriter` class can be used to create and write to a textfile. The following example creates a file named `scientists.txt`. The `PrintWriter` object supports both the `format()` and `printf()` methods.

```

import java.io.PrintWriter;
import java.io.IOException;

public class File3
{
    public static void main(String[] args) throws IOException
    {
        PrintWriter outfilepw = new PrintWriter("scientists.txt");

        outfilepw.format("I. Newton    %3d\n", 25);
        outfilepw.format("S.N. Bose    %3d\n", 25);
        outfilepw.format("J.C. Bose    %3d\n", 25);

        outfilepw.printf("C.V. Raman    %3d\n", 50);
        outfilepw.printf("S. Ramanujan %3d\n", 20);

        outfilepw.close();
    }
}

```

When we run the above program, a file named `"scientists.txt"` with the following content is created:

```

I. Newton    25
S.N. Bose    25
J.C. Bose    25
C.V. Raman   50
S. Ramanujan 20

```

## Try-with-resources

Try with autocloseable resources is a useful feature of the try block that often comes in handy in file processing. A resource (e.g., a file) is created as part of entering the try statement (see the special syntax in the example code below). The resource is then used in the try block and, most importantly, auto-closed. Note that:

- The resource must be a subclass of `AutoCloseable`, that is, an object of a class with a `close()` method.
- The resource (the `PrintWriter` object in this example) is (implicitly) closed when the try block terminates or when there is an exception.

If the try-with-resources statement is in a loop (as when the same sequential textfile needs to be read more than once in the same program), the resource is opened (and closed) in each iteration through the loop.

```

import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class File4
{
    public static void main(String[] args)
    {
        try (PrintWriter outfilepw = new PrintWriter("scientists.txt");)
        {
            outfilepw.format("I. Newton    %3d\n", 25);
            outfilepw.format("S.N. Bose    %3d\n", 25);
        }
    }
}

```

```

        outfilepw.format("J.C. Bose      %3d\n", 25);

        outfilepw.printf("C.V. Raman    %3d\n", 50);
        outfilepw.printf("S. Ramanujan  %3d\n", 20);
    }
    catch (FileNotFoundException ex)
    {
        ex.printStackTrace();
    }
}
}

```

## Regular Expression

---

It is assumed that the student has a basic familiarity with the String class and the methods it supports. Check out [the best source to refresh one's memory on Java strings](#).

Instances of the String class are immutable in that the content of a String object cannot be altered once the string is created.

A regular expression is a string that describes a pattern or template for representing a set of strings. For example,

- The regular expression "." (a single period) represents (matches) any single character. E.g., the string "Book" matches all of the following regular expressions:
  - "B..."
  - "Bo.k"
  - "B.ok"
  - "Boo."
  - ".ook"
  - "..."
  - "B..k"
- The regular expression "[abc]" means one of the three characters, that is, a or b or c. For example, the strings "bxb", "bxc", "bxa" all match the regular expression "bx[abc]".
- The regular expression "(nd|m)" represents the two characters nd (in that order) or the single character m. For example, both "And" and "Am" match the regular expression "A(nd|m)".

We discuss the use of regular expressions in the following applications involving strings:

- string matching,
- creating a new string by replacing characters of an existing string,
- splitting a given string into a number of new strings.

Several representative uses of regular expressions and their matching patterns in string processing are illustrated in the following program:

```

public class File5
{
    public static void main(String[] args)
    {
        System.out.println("USA".matches("US")); // f
        System.out.println("USA".matches("USA")); // t

        System.out.println("United States".matches("Unite.*")); // t
        System.out.println("United Airlines".matches("Unite.*")); // t
        System.out.println("United States".matches("Unite*")); // f

        System.out.println("617-353-3000".matches("\\d{3}-\\d{3}-\\d{4}")); // t

        String s = "express".replaceAll("e", "E"); // ExprEss
        System.out.println(s); // ExprEss

        s = "express".replaceFirst("e", "E");
        System.out.println(s); // Express

        s = "express".replaceAll("ex", "Im");
        System.out.println(s); // Impress

        s = "express".replaceAll("[spye]", "END");
        System.out.println(s); // ENDxENDrENDENDEND
    }
}

```

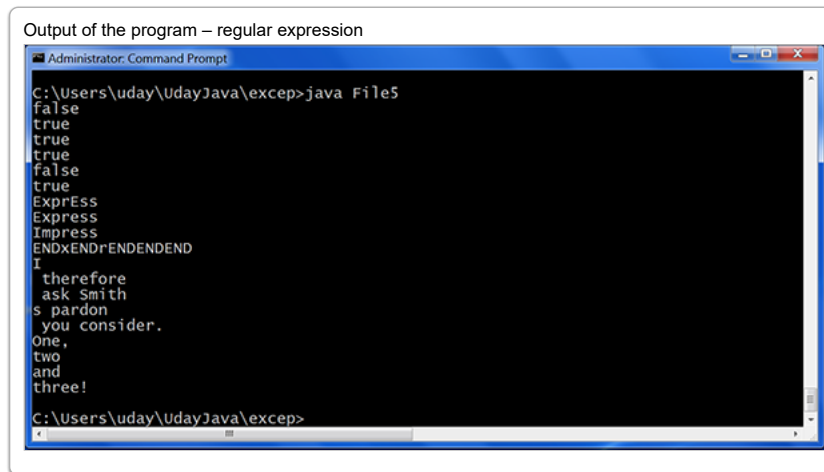
```

String[] arr = "I, therefore, ask Smith's pardon; you consider.".split("[;,']");
for (String word: arr)
    System.out.println(word);

arr = "One, two and three!".split("\\s+");
for (String word: arr)
    System.out.println(word);
}
}

```

The output is shown below:



Check out [more information on regular expressions](#).

## StringBuilder

The classes `StringBuilder` and `StringBuffer` can, in general, be used wherever the `String` class is used. The most important difference between `String` and `StringBuilder` is that objects of type `StringBuilder` can be modified (content added, deleted, appended, reversed, etc.), whereas `String` objects are immutable.

The `StringBuilder` class allows its instance object's contents to be modified as follows:

- Append to the string
- Delete characters from the string
- Delete the character at a specified position
- Insert into the string at a specified position
- Replace the characters at specified positions
- Set a new character at a specified position in the string
- Reverse the string

The following program illustrates some of the most frequently used features of the `StringBuilder` class.

```

public class File6
{
    public static void main(String[] args)
    {
        StringBuilder sb = new StringBuilder("Hello");
        System.out.println(sb);

        sb.append(" My ");
        sb.append("Friend!");
        System.out.println(sb);

        sb.insert(9, "New ");
        System.out.println(sb);

        sb.delete(6, 9);
        System.out.println(sb);
    }
}

```



```

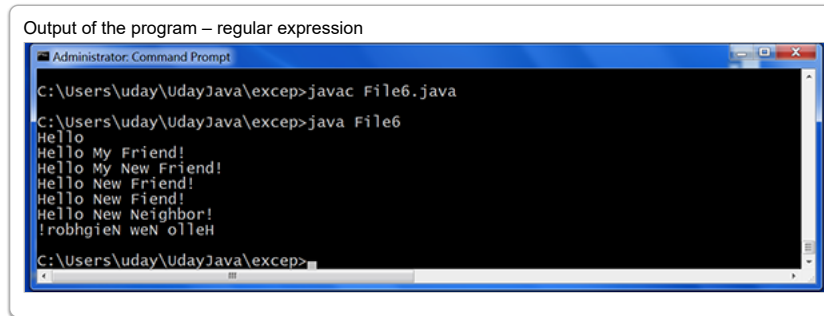
        sb.deleteCharAt(11);
        System.out.println(sb);

        sb.replace(10, 15, "Neighbor");
        System.out.println(sb);

        sb.reverse();
        System.out.println(sb);
    }
}

```

The output of the above code appears below:



The StringBuffer class is similar to the StringBuilder class, with the important difference that the former offers *synchronized* methods for thread-safe manipulation of strings (thread-safety is discussed in detail in Module 6). Use StringBuilder when there is no concurrency (no concurrency means no chance of data inconsistency and therefore no need for synchronization).

Check out the original documentation for a description of all of the methods of [StringBuilder](#) and [StringBuffer](#).

## Module 2 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.

Note: Test Yourself Questions 2.1 to 2.4 refer to the class File1 in the section on Text input/output.

### Test Yourself 2.1

Instead of having two separate statements to open the file (initializing outfile to null at the beginning and then invoking the constructor in the try block), why didn't we write only one statement in the try block, as follows?

```
Formatter outfile = new Formatter("poets.txt");
```

Suggested answer: That would make outfile local to the try block and thus inaccessible from the rest of main().

### Test Yourself 2.2

Must we put the "new Formatter()" statement in a try block (can't we avoid the try-catch altogether)?

Suggested answer: Yes (but read on), because the Formatter constructor throws a checked exception, namely FileNotFoundException (which is an IOException). (Recall that IOExceptions are checked exceptions.) The FileNotFoundException must be either caught or declared; this means that an alternative approach would be to have main() declare this exception (that would allow us to dispense with the try-catch idiom).

### Test Yourself 2.3

Following the answer to Q2 above, if we add "throws FileNotFoundException" after "main(String[] args)", will the program run?

Suggested answer: Yes.

### Test Yourself 2.4

Following the answer to Q2, if we add "throws IOException" after "main(String[] args)", will the program run, or do we have to do something more?

Suggested answer: Just adding "throws IOException" will not suffice (a compilation error will occur); we will have to import java.io.IOException as well. (This is because FileNotFoundException is a sub-class, not super-class, of IOException.)

### Test Yourself 2.5

Study the exception handling code in each of the following programs, and then say whether each program will (i) compile OK, (ii) run OK:

#### Program #1:

```
import java.io.*;

public class Test6
{
    public static void main(String[] args)
    {
        m1();
    }

    public static void m1()
    {
        try
        {
            int i = 1;
            if (i > 0)
                throw new IOException();
        }
        catch(IOException ioex)
        {
            System.out.println("IOException caught and processed in m1()");
            ioex.printStackTrace();
        }
        System.out.println("The part after exception throwing now executed");
    }
}
```

#### Program #2:

```
import java.io.*;

public class Test6A
{
    public static void main(String[] args)
    {
        try
        {
            m1();
        }
        catch(IOException ioex)
        {
            System.out.println("IOException caught and processed in main()");
            ioex.printStackTrace();
        }
        System.out.println("The part after exception throwing now executed");
    }

    public static void m1() throws IOException
    {
        int i = 1;
        if (i > 0)
            throw new IOException();
    }
}
```

#### Program #3:

```
import java.io.*;

public class Test6B
{
    public static void main(String[] args) throws IOException
    {
        m1();
        System.out.println("The part after exception throwing now executed");
    }

    public static void m1() throws IOException
    {
        int i = 1;
        if (i > 0)
            throw new IOException();
    }
}
```

Suggested answer: Yes to all the six questions.

## References

---

- Oracle. *The Java™ Tutorials: Exceptions*. Retrieved from <https://docs.oracle.com/javase/tutorial/essential/exceptions/>.