

# Creating, Reading, Writing and Deleting a File

by Sophia



## WHAT'S COVERED

In this lesson, you will learn more about how to read and write to files. Specifically, this lesson covers:

### Table of Contents

- [1. Reading and Writing to Files](#)
  - [1a. Writing to a File](#)
  - [1b. Appending to a File](#)
  - [1c. Writing Integers and Floats](#)
  - [1d. Reading Integers](#)
  - [1e. Deleting a File](#)

## 1. Reading and Writing to Files

In the previous tutorial, you worked with a `File` object that represents an abstraction of a file in that it may point to a file that already exists or to a file that doesn't exist yet (but will be created).

### 1a. Writing to a File

Since working with files is generally more interesting when they have content, let's first look a bit more into writing text to a file.

In the previous tutorial, you used this code to write to a new file using the `Files.write()` method (in the `java.nio` library package):

```
import java.io.*;
import java.nio.file.*;
import java.util.ArrayList;

public class WriteTextToFile {
```

```

public static void main(String[] args) {
    // Create ArrayList of Strings & add lines of text
    ArrayList<String> lines = new ArrayList<>();
    lines.add("Line 1");
    lines.add("Line 2");
    lines.add("Line 3");
    // File object pointing to output.txt file (which may not exist yet)
    File outputFile = new File("output.txt");

    try {
        // StandardOpenOption.CREATE creates a new file. It will create the file
        // if it doesn't exist or overwrite it if it does.
        Files.write(outputFile.toPath(), lines, StandardOpenOption.CREATE);
    }
    catch(IOException ex) {
        System.out.println("Error writing to file: " + ex.getMessage());
    }
}

```

This code creates a File object named outputFile that points to a file called output.txt:

#### ➤ EXAMPLE

```
File outputFile = new File("output.txt");
```

This next line then writes the text in the ArrayList called lines to the output.txt file:

#### ➤ EXAMPLE

```
Files.write(outputFile.toPath(), lines, StandardOpenOption.CREATE);
```

Note how the File object, outputFile, is converted to a Path object using the File class's toPath() method. The StandardOpenOption.CREATE option tells the JVM to create the file on disk, if the file doesn't already exist.

If there is a collection of lines of text to write to the file, the Files.writeString() method can be used like this:

```

class WriteStringToFile {
    public static void main(String[] args) {
        File output = new File("output.txt");
        try {
            Files.writeString(output.toPath(), "Hello, world", StandardOpenOption.CREATE);
        }
        catch(IOException ex) {
            System.out.println("Error: " + ex.getMessage());
        }
    }
}

```

```
}  
}  
}
```

This program creates a file named output.txt and writes the string "Hello, world" to it. If the file already exists, the contents are overwritten when using the `StandardOpenOption.CREATE` option.

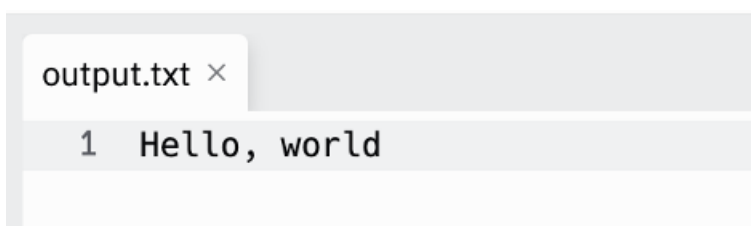


#### BIG IDEA

If the same file is written to more than once using multiple calls to `Files.writeString()`, there will only be one line of text in the output file because `StandardOpenOption.CREATE` mode will either create a new file or overwrite an existing file.

```
import java.io.*;  
import java.nio.*;  
import java.nio.file.*;  
  
public class WriteStringToFile {  
    public static void main(String[] args) {  
        File output = new File("output.txt");  
        try {  
            // Try writing 2 lines to the file in StandardOpenOption.CREATE mode  
            Files.writeString(output.toPath(), "Hello, world", StandardOpenOption.CREATE);  
            Files.writeString(output.toPath(), "Hello, world", StandardOpenOption.CREATE);  
        }  
        catch(IOException ex) {  
            System.out.println("Error: " + ex.getMessage());  
        }  
    }  
}
```

Each time we run this example, our output.txt file will just have the output file always looking like the following (remember we need to move to the output.txt file to see this output):



#### TRY IT

**Directions:** Enter this code in a file named `WriteStringToFile.java`. If your Replit directory already contains an `output.txt` file, delete it, and then run the program (`java WriteStringToFile.java`). Look for the new file called `output.txt` and click on it to see the output. Try running it a few times to see if the output is the same.



#### REFLECT

The output file always contains one line because each time you write to the file, you are using the

StandardOpenOption.CREATE option. This means that each time the file is opened, it creates a new file and overwrites all of the old content.

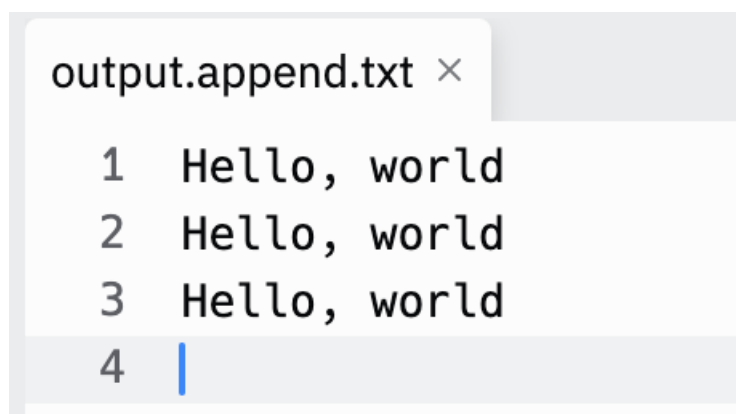
## 1b. Appending to a File

When using a combination of StandardOpenOption.CREATE and StandardOpenOption.APPEND, a new file will be created if the file doesn't exist. It will append a line to the file, if the file already exists.

```
import java.io.*;
import java.nio.*;
import java.nio.file.*;

public class AppendStringToFile {
    public static void main(String[] args) {
        File output = new File("output.append.txt");
        try {
            // Use both StandardOpenOption.CREATE and StandardOpenOption.APPEND so that
            // file is created if it doesn't exist. If the file already exists, text is
            // appended.
            Files.writeString(output.toPath(), "Hello, world\n", StandardOpenOption.CREATE,
                             StandardOpenOption.APPEND);
        }
        catch(IOException ex) {
            System.out.println("Error: " + ex.getMessage());
        }
    }
}
```

Each time the file is opened and written to, a new line is added. If we run it three times, the results will look like this:



```
output.append.txt x
1 Hello, world
2 Hello, world
3 Hello, world
4 |
```



**Directions:** Try entering the code above in a file in Replit named AppendStringToFile.java. Run the program again. Check the output.append.txt file to see the output.



Notice you see a new appended line each time you run the `AppendStringToFile.java` file.

## 1c. Writing Integers and Floats

Writing to a file works the same as we have been doing it. The only exception is that integers or floats must be converted to a string before the string can be written to the file. Since this conversion needs to be done for numbers one at a time, the program needs to use `Files.writeString()`.

Let's try an example using integers. In this example, it will be years that need to be converted to strings. First, we will define an `ArrayList` of `Integers`, then we will write this `List` using `StandardOpenOption.CREATE` and `StandardOpenOption.APPEND` within a `for` loop:

```
import java.io.*;
import java.io.IOException;
import java.nio.file.*;
import java.util.ArrayList;

public class WriteNumbersToFile {

    public static void main(String[] args) {
        ArrayList<Integer> years = new ArrayList<>();
        years.add(1975);
        years.add(1979);
        years.add(1983);
        File numbersOutput = new File("years.txt");
        // Iterate over ArrayList of years
        for(int year : years) {
            try {
                // Use Integer.toString() to convert years to strings.
                // Need to convert years one at a time, so they have to be
                // written one at a time using Files.writeString(). Add \n after each
                Files.writeString(numbersOutput.toPath(), Integer.toString(year) + "\n",
                    StandardOpenOption.CREATE, StandardOpenOption.APPEND);
            }
            catch(IOException ex) {
                System.out.println("Error: " + ex.getMessage());
            }
        }
    }
}
```



**Directions:** Try adding the code above to create a new file called `years.txt`. Run the program, then see if the years as strings show up in the `years.txt` file as below:

➞ EXAMPLE

years.txt ✕	
1	1975
2	1979
3	1983
4	



REFLECT

It is important to remember that when reading and writing values to and from text files, all of the data is read and written as strings. This means that numeric data will need to be converted to and from Java Strings, since the rest of the code needs these to be of the correct data type.

## 1d. Reading Integers

An example reading our just written data works the same as before, except that the year string must be converted to an integer:

```
import java.io.*;
import java.nio.file.Files;
import java.util.ArrayList;
import java.util.List;

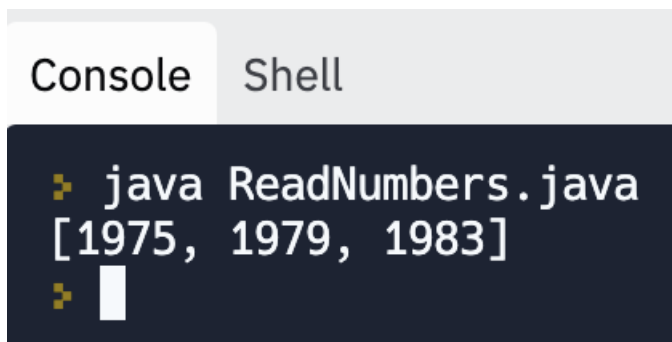
public class ReadNumbers {

    public static void main(String[] args) {
        File yearsFile = new File("years.txt");
        ArrayList<Integer> years = new ArrayList<>();
        try {
            List<String> yearsAsStrings = Files.readAllLines(yearsFile.toPath());
            for(String yearString : yearsAsStrings) {
                years.add(Integer.parseInt(yearString));
            }
        }
        catch(IOException ex) {
            System.out.println("Error reading file: " + ex.getMessage());
        }
        catch(NumberFormatException ex) {
            System.out.println("Number format error: " + ex.getMessage());
        }
        System.out.println(years.toString());
    }
}
```

[TRY IT](#)

Try typing the code above into a file named `ReadIntegers.java` and run the program.

You should get the following result (based on the years written to the file previously):

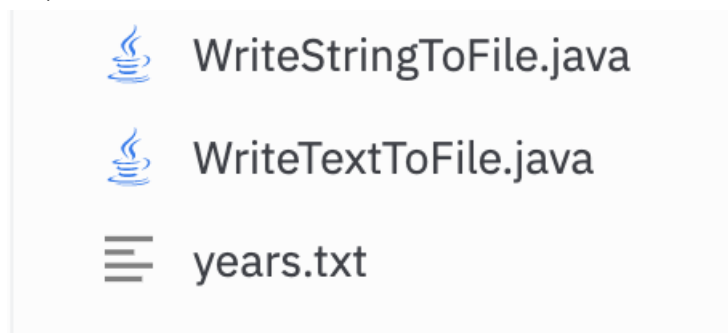


The screenshot shows a Replit interface with two tabs: 'Console' and 'Shell'. The 'Console' tab is active, displaying the output of a Java program. The first line shows a yellow prompt character followed by the command `java ReadNumbers.java`. The second line shows the output `[1975, 1979, 1983]`. A third line shows a yellow prompt character followed by a white cursor.

## 1e. Deleting a File

The `java.nio` package's `Files.deleteIfExists()` handles deleting a file (assuming that the file exists). This method returns `true` or `false`, indicating if the file was deleted successfully or not. Notice that before using the following code example, our File structure looks like this (from Replit's left side File panel):

➔ EXAMPLE



To delete the `years.txt` file that we have created, we could use the following code.

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;

public class DeleteFile {

    public static void main(String[] args) {
        File yearsFile = new File("years.txt");
        boolean fileDeleted = false;
        try {
            fileDeleted = Files.deleteIfExists(yearsFile.toPath());
        }
        catch(IOException ex) {
            System.out.println("Error deleting file: " + ex.getMessage());
        }
    }
}
```

```

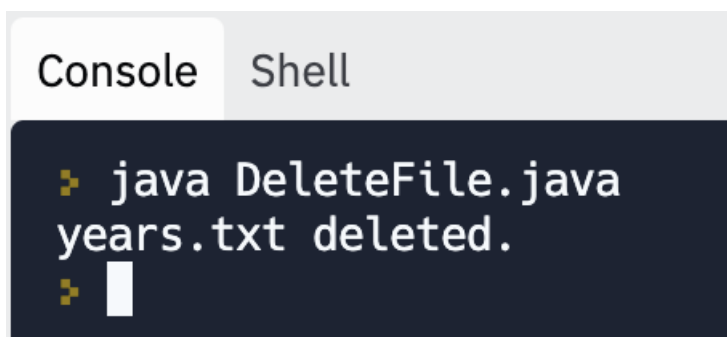
}
if(fileDeleted) {
    System.out.println(yearsFile.getName() + " deleted.");
}
else {
    System.out.println(yearsFile.getName() + " not deleted.");
}
}
}
}

```

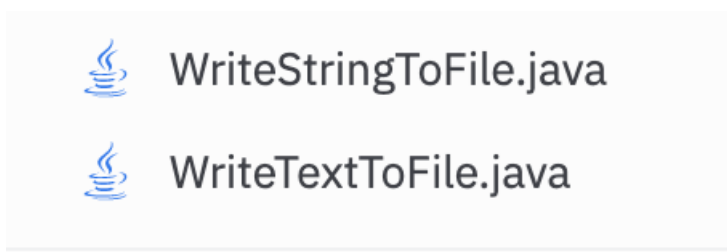


**Directions:** Try typing the code above into a file named `ReadIntegers.java` and run the program. Now type in the code above (in a file named `DeleteFile.java`) and see if you can remove the `years.txt` file.

The output from running the program should look like this:



Notice that after running the code, the file is removed from the Files directory.



## SUMMARY

In this lesson, you had a chance to **read and write to files**. You learned how to create a file, **write to a file**, and **append a file**. We also looked at an example of **writing numeric data or integers and floats** to a file. You also learned how to **read integers stored as strings in a text file**. Finally, you learned how to **delete a file**.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source [cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf](https://cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf)

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source [py4e.com/html3/](https://py4e.com/html3/)