

various kinds of data. This can prove challenging because text and binary files require specific classes and methods to read, write, and manage the data effectively.


Remember

So far, you've learned about the importance of files in Java and how to write and read text and binary files using character and binary streams. As you work through this lab, remember a text file contains human-readable characters, typically encoded in formats like ASCII or UTF-8, and usually has a `.txt` extension. In contrast, a binary file stores non-human-readable data as byte sequences (like images, videos, or programs) and cannot be viewed with a text editor.

In this lab, you'll follow a series of steps to practice writing text and binary files using different approaches. These exercises will equip you to handle various file operations in future projects, like logging application events in a text file or storing encrypted data in a binary format.

Goal


Create and manipulate text and binary files to store various types of data, such as string representations of objects and numerical values.

 **Note:** When you encounter this icon, it's time to get into your IDE and start coding!

In your lab environment, open IntelliJ by double-clicking on the icon.

First up, you'll create a basic *Person* class with attributes for name, age, and weight. This class will serve as the foundation for the objects you'll later save to text and binary files. You'll also implement a *toString()* method to generate a string representation of the *Person* object, which is essential for saving the object's data to a text file.

From the *src* folder in the starter project, open and review the code for *Person* class.

 **It's time to get coding!**

- **TODO 1:** Add the instance variables *name* (a *String*), *age* (int), and *weight* (double).

```
public class Person {
```

```
//TODO 1: declare instance variables
```

- TODO 2: Define a constructor for *Person* class with three arguments.

```
Person(. . .) {  
  
    //TODO 2: complete the constructor  
  
}
```

- TODO 3: Provide a *toString()* method in the class to return a String representation of the object.

```
public String toString() {  
  
    //TODO 3: use the String.format() method to return string representation of the  
    object  
  
}  
  
}
```

Tip


Use the *String.format()* method to easily insert variable data into placeholders within a string. In Java, *%s*, *%d*, and *%f* are used as placeholders for String, integer, and float variables, respectively.

For example: “*String.format("Hello, my name is %s and my age is %d", "Nick", 25)*”; returns “*Hello, my name is Nick and my age is 25*”.

Pause and check

To check that your constructor and *toString()* method are correctly implemented, try creating a few *Person* objects and printing them to the console. This will help you verify that the object attributes are being properly initialized and formatted as a string.

Text files are used everywhere, from configuration settings to simple data logs – but why? Data needs to be easily readable by both humans and other programs, making text files an essential way to store information. In this task, you'll write the string representation of a *Person* object to a text file using the *FileWriter* class.

 It's time to get coding!

Open the *Main.java* file from the *src* folder and the static method *textFileWrite()* in it. This method uses the *FileWriter* class to write a text file in the current folder.

- TODO 4: Add a static method named *textFileWrite()* that returns no value. Inside the method, declare a *Person* object and use its *toString()* method to get the string representation.
- TODO 5: Create an object of the *FileWriter* class, pointing to the *file1.txt* file, and use the *write()* method to save the *Person* object's data to the file.
- TODO 6: Place the file writing code inside a *try* block, and catch any *IOException* that may occur.

And now what do you find when you open the project folder? If everything went smoothly, the *file.txt* file should be in the project folder with the data of the created object. Nice!

Now you'll enhance your file writing skills by using the *PrintWriter* class to write user-inputted *Person* object data to a text file. This method offers more flexibility in terms of how data is formatted and written to the file and is often used when you need to format text data before saving it.



It's time to get coding


- TODO 7: Define *printWrite()* method, a static method that doesn't return any data.
- TODO 8: Take inputs from the user with *Scanner* class for *name*, *age* and *weight* and create an instance of a *Person* object with the inputs.
- TODO 9: Declare an object of the *PrintWriter* class, pointing to the *file2.txt* file in the current folder. Then, use its *println()* method to write the String returned by the *toString()* method of the object.
- TODO 10: Place the file writing code inside a *try* block, and catch any *IOException* that may occur.

Pause and check

To be sure of your efforts, check that *file2.txt* is available in the project *src* folder by opening it up matching its content with the given inputs. If everything aligns, great work! If the contents don't match, check for errors like incorrect file paths, input handling, or data formatting, and adjust your code as needed.

Text files are one thing, but what about non-human-readable data like byte sequences? Sometimes, developers need to store data in a compact and efficient format – such as images, audio, or executable programs – and that's where binary files come in.

In this task, you'll use the *FileOutputStream* class to create a binary file and store a list of numbers using basic binary operations.


 It's time to get coding!

- TODO 11: In the *Main.java* file, declare a byte array to store a list of numbers.
- TODO 12: Open a *FileOutputStream* object, pointing towards *file3.txt* file.
- TODO 13: Use a *for* loop to iterate over the array and call the *write()* method of the *FileOutputStream* object to print each element.
- TODO 14: Place all the file activity inside the *try-catch* block.

Note


The *file3.txt* file (and the *file4.dat* in the next task) is actually a binary file, so you'll not be able to see its content with the help of any text editor.

Handling structured data in binary form is useful for saving objects in a more compact, non-readable format. In this final task, you'll use the *DataOutputStream* class to write the attributes of a *Person* object to a binary file.

 It's time to get coding!

- TODO 15: Declare (in the *Main.java* file) an object of the *DataOutputStream* class, using the *FileOutputStream* object as an argument in its constructor. (The *FileOutputStream* object should refer to *file4.dat*.)
- TODO 16: Use the *Scanner* class to read name, age, and weight. Then, instantiate a *Person* object from the inputs.
- TODO 17: Use the *writeUTF()*, *writeInt()*, and *writeDouble()* methods of the *DataOutputStream* object to write the *Person* object's name, age, and weight attributes, respectively. Methods like *writeUTF()*, *writeInt()*, and *writeDouble()* are used for writing binary files in Java. These are more efficient than the *write()* method in character stream writer classes, which are used to store only text data.
- TODO 18: Place the entire file inside a *try-catch* block.
- TODO 19: Call all the above static methods from the *main()* method in *Main.java*.

Finally, run through your program to verify that all file operations have been implemented correctly. This critical act of checking ensures that your program can successfully read and write data to both text and binary files, confirming that the *file I/O* functionality works as expected.

 It's time to get coding!

- TODO 20: Run the code in your IDE and check the output.

File written with FileWriter...

```
File written with FileOutputStream...
```

```
Enter name:
```

```
Nick
```

```
Enter age:
```

```
25
```

```
enter weight:
```

```
65
```

```
File written with PrintWriter...
```

```
File written with DataOutputStream...
```

```
Process finished with exit code 0
```

Congratulations! In this lab, you practiced writing data to both text and binary files using Java's *FileWriter*, *PrintWriter*, *FileOutputStream*, and *DataOutputStream* classes. These exercises have equipped you with essential skills needed to handle a variety of file operations, which are crucial across a wide range of real-world software development projects, from data storage to processing complex datasets.

As you continue through the course, the techniques you've learned here will enable you to manage *file I/O* tasks effectively, preparing you for more advanced projects where file handling is a key component. Good luck!