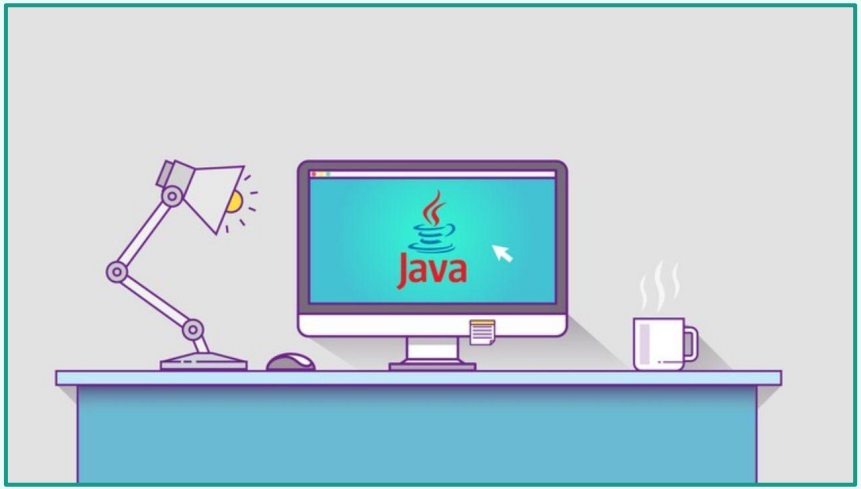
Java SE & Spring

Module 1: Java SE

13.Java IO API



What is an Java IO?

Java IO is a collection of classes and interfaces which you may use to perform almost every possible IO operation through your java application. Reading from a file or writing to a file, creating files, transferring files through a network is a few examples of IO API. For these operations, Java has a standard API which is named Java IO API. But there is another API with name Java NIO API (NIO stands for New IO) too. There are several differences between these two API. Such as blocking and non-blocking issue. While IO API is blocking, NIO is non-blocking. Blocking means, while carried out an IO operation, current thread waits for IO operation is completed. After IO operation finished, then normal program sequence continues. But in NIO API, it is not reuired that current thread waits for IO operation’s finish. While IO operation continues, current thread concurrently work for other operations.

Basic Input & Output

Basic Output:

As we have seen until now from beginning of the course, we can use System.out class for printing our program-generated throughputs. For a quick recap, we can mention println() and other common printing methods under System class.

- **print()**: prints string inside methods double quoted parameter, or the other types of parameters that converted internally to string. (e.g int, double etc.)
- **println()**: has same logic with print() method, additionally it appends new line char to the end of the parameter.
- **printf()**: provides formatted printing as we talked about it in chapter 10.

Basic Input & Output

Basic Input:

Java provides different ways to get input from the user. However, in this tutorial, you will learn to get input from user using the object of Scanner class.

Example:

```
1  Scanner scanner = new Scanner(System.in));
2  int number = scanner.nextInt();
3  System.out.println(number);
```

File in Java

File handling is an important part of any application. Java has several methods for creating, reading, updating, and deleting files. The File class from the java.io package, allows us to work with files.

To use the File class, we need to create an object of the class, and specify the filename or directory name

Handling a file:

```
1  import java.io.File;
2  File myObj = new File("filename.txt"); // Specify the filename
```

File in Java

The **File** class has many useful methods for creating and getting information about files.

- `boolean` **canRead()**: Tests whether the file is readable or not
- `boolean` **canWrite()**: Tests whether the file is writable or not
- `boolean` **createNewFile()**: Creates an empty file
- `boolean` **delete()**: Deletes a file
- `boolean` **exists()**: Tests whether the file exists
- String **getName()**: Returns the name of the file
- String **getAbsolutePath()**: Returns the absolute pathname of the file
- `long` **length()**: Returns the size of the file in bytes
- String[] **list()**: Returns an array of the files in the directory
- `boolean` **mkdir()**: Creates a directory

Creating Files

To create a file in Java, you can use the createNewFile() method. This method returns a boolean value: true if the file was successfully created, and false if the file already exists. Note that the method is enclosed in a try...catch block. This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason).

Example:

```
try {
     if(fileForReading.createNewFile()){
         System.out.println("File created!");
     }
} catch (IOException e) {
        e.printStackTrace();
}
```

Note: To create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "\" character (for Windows). On Mac and Linux you can just write the path, like: /Users/name/filename.txt

Reading Files

There are many available classes in the Java API that can be used to read and write files in Java: FileReader, BufferedReader, Files, Scanner, FileInputStream, FileWriter, BufferedWriter, FileOutputStream, etc. Which one to use depends on the Java version you're working with and whether you need to read bytes or characters, and the size of the file/lines etc. We will discuss 3 types of reading in this section, Scanner, BufferedReader and Files. First two classes are member of Java IO API, the last one is a member of Java NIO API.

Example:

```
1  File fileForReading = new File("D:\\users\\E_BATUR\\Desktop\\captmidn.txt");
2  try (Scanner fileScanner = new Scanner(fileForReading)) {
3      System.out.println(fileScanner.nextLine());
4      while (fileScanner.hasNextLine()) {
5            System.out.println(fileScanner.nextLine());
6      }
7
8  } catch (IOException e) {
9      e.printStackTrace();
10 }
```

Writing to Files

In the following example, we use the **FileWriter** class together with its write() method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the close() method

Example:

```
try {
    FileWriter myWriter = new FileWriter("filename.txt");
    myWriter.write("Files in Java might be tricky, but it is fun enough!");
    myWriter.close();
    System.out.println("Successfully wrote to the file.");
} catch (IOException e) {
    System.out.println("An error occurred.");
    e.printStackTrace();
}
```

Deleting Files

- To delete a file in Java, use the delete() method:

  Example:

```
1  File myObj = new File("filename.txt");
2  if (myObj.delete()) {
3     System.out.println("Deleted the file: " + myObj.getName());
4  } else {
5     System.out.println("Failed to delete the file.");
6  }
```

- We can also delete a folder, but the folder must be empty.

```
1  File myObj = new File("C:\\Users\\MyName\\Test");
2  if (myObj.delete()) {
3     System.out.println("Deleted the folder: " + myObj.getName());
4  } else {
5     System.out.println("Failed to delete the folder.");
6  }
```

Practice Time: Let’s Code Together!

Questions ?

Next:Java Annotations