Java 11 Notes

➢ Why is Java 11 important ?

- Java 11 is the second LTS release after Java 8. Since Java 11, Oracle JDK would no longer be free for commercial use.
- You can use it in developing stages but to use it commercially, you need to buy a license. If you don't, you can get an invoice bill from Oracle any day!
- Java 10 was the last free Oracle JDK that could be downloaded.
- Oracle stops Java 8 support from January 2019. You'll need to pay for more support.
- You can continue using it, but won't get any patches/security updates.

➢ We can run the Java program without compilation.

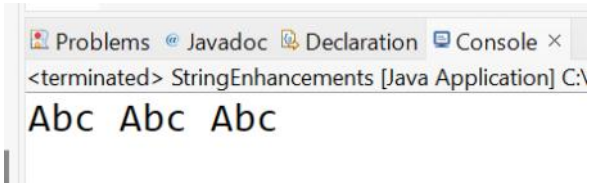
- From Java 11 we just run the program no need to compile separately. The Java command itself will compile internally.
- Java Test.java

➢ String Enhancements:

- **repeat(int)**: This methos can be called on string object, it returns the given string object for the mentioned int number.

```
public static void main(String[] args) {
    String s="Abc ";
    s=s.repeat(3);
    System.out.println(s);
}
```

Out put is:



- To this repeat() method if we provides negative value then it gives illegal argument exception
- This method accepts the integer number, Since it accepts the integer, after creating the new string the length should not cross the integer limit, if it crosses, it throws out of memory error.

```
public static void main(String[] args) {
    String s="Abc ";
    s=s.repeat(Integer.MAX_VALUE);
    System.out.println(s);
}
```

- **isBlank():** It returns true, if the given string is empty or having only the spaces.
- **strip():** this method is same as trim() which removes the leading and trailing spaces from the string.
- **stripLeading():** This method is used to delete the leading spaces only
- **stripTrailing()**: This method is used to delete the trailing spaces

```
String sample = " Karthik ";
System.out.println(sample.repeat(2)); // " Karthik  Karthik "
System.out.println(sample.isBlank()); // false
System.out.println("".isBlank()); // true
System.out.println("   ".isBlank()); // true
System.out.println(sample.trim());//"Karthik"
System.out.println(sample.strip()); // "Karthik"
System.out.println(sample.stripLeading()); // "Karthik "
System.out.println(sample.stripTrailing());//" Karthik"
```

What is the difference between strip() and trim();

- trim() removes only characters <= U+0020 (space); strip() removes all Unicode whitespace characters (but not all control characters, such as \0)
- However, the strip() method uses Character.isWhitespace() method to check if the character is a whitespace. This method uses Unicode code points whereas trim() method identifies any character having codepoint value less than or equal to 'U+0020' as a whitespace character.
- The strip() method is the recommended way to remove whitespaces because it uses the Unicode standard.
- Refer to link https://bugs.openjdk.org/browse/JDK-8200373
- String::trim has existed from early days of Java when Unicode had not fully evolved to the standard we widely use today.
- The definition of space used by String::trim is any code point less than or equal to the space code point (\u0020), commonly referred to as ASCII or ISO control characters.

```
String string = '\u2001'+"String    with    space"+ '\u2001';
  System.out.println("Before: \"" + string+"\"");
  //output: "Before: "?String    with    space?""
  System.out.println("After trim: \"" + string.trim()+"\"");
  //output: After trim: "?String    with    space?"
  System.out.println("After strip: \"" + string.strip()+"\"");
  //output: After strip: "String    with    space"
```

lines():

- Returns a stream of lines extracted from this string, separated by line terminators
- A line terminator is one of the following:
- a line feed character {@code "\n"} (U+000A),
- a carriage return character {@code "\r"} (U+000D),
- or a carriage return followed immediately by a line feed
- {@code "\r\n"} (U+000D U+000A).

Lines example with \n symbol

```
sample = "This\nis\na\n big text which has multiple\nlines.";

List<String> lines = new ArrayList<>();

sample.lines().forEach(line -> lines.add(line));
lines.forEach(line -> System.out.println(line));
//output is
//This
//is
//a
// big text which has multiple
//lines.
```

Lines example with \r symbol

```
sample = "This\ris\ra\r big text which has multiple\rlines.";

List<String> lines1 = new ArrayList<>();

sample.lines().forEach(line -> lines1.add(line));
lines.forEach(line -> System.out.println(line));
//output is
//This
//is
//a
// big text which has multiple
//lines.
```

➢ Collection Enhancements in Java 11:

- **toArray()**: In Java 11 new default method has been introduced in collection interface.

```
default <T> T[] toArray(IntFunction<T[]> generator) {
  return toArray(generator.apply(0));
}
```

This toArray() takes the functional interface and converts into Array.

```
public static void main(String[] args) {
    List<Employee> empList=new ArrayList<Employee>();
    empList.add(new Employee("Test", 123, 10000));
    empList.add(new Employee("Test2", 124,20000));
    empList.add(new Employee("Test3", 125,30000));
    empList.add(new Employee("Test4", 126,40000));

    System.out.println("**** With Lamda ***");
    Employee[] empArray= empList.toArray((e)->new Employee[e]);
    Stream.of(empArray).forEach((p)->System.out.println(p));
    System.out.println("**** With Method reference ****");
    Employee[] empArray1= empList.toArray(Employee[]::new);
    Stream.of(empArray1).forEach((p)->System.out.println(p));
}
```

➢ File API Changes:

- Reading and writing text files has been continuously simplified since Java 6. In Java 6, we had to open a FileInputStream, wrap it with an InputStreamReader and a BufferedReader, then load the text file line by line into a StringBuilder (alternatively, omit the BufferedReader and read the data in char[] blocks) and close the readers and the InputStream in the finally block.

- Luckily, we had libraries like Apache Commons IO that did this work for us with the FileUtils.readFileToString() and writeFileToString() methods.

```
//General way of reading the file
File file = new File("C:\\Users\\DELL\\Desktop\\Contact.txt");
BufferedReader br   = new BufferedReader(new FileReader(file));
String st=null;
while ((st = br.readLine()) != null)
    System.out.println(st);
```

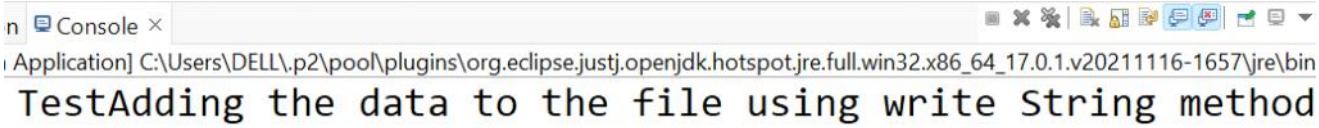
- The above is the general way of reading the data from the file.
- Below is the Java 11 implementation.
- **readString():** This method reads all content from a file into a String. Using UTF-8 charset it decodes from bytes to characters. This function makes sure that the file is closed when all content have been read or an I/O Error or other runtime exception, is thrown.

```
System.out.println("Java 11 Improvement of File API");
//Java 11 implementation
Path p= Path.of("C:\\\\Users\\\\DELL\\\\Desktop\\\\Contact.txt");
String s=Files.readString(p);
System.out.println(s);
```

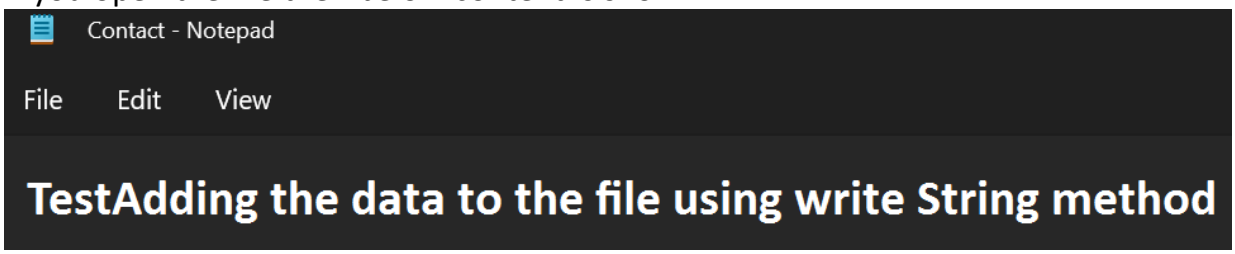
- **writeString():** This method is used to write the data to the file.
  Syntax: static Path writeString(Path path, CharSequence csq, OpenOption... options)
  - The first argument is path where the file is present.
  - The Second argument is Char Sequence which is nothing but the String that we want to write to the file
  - The third parameter var args options ->options specifies how the file is opened.

```
Files.writeString(p, "Adding the data to the file using "
        + "write String method",StandardOpenOption.APPEND);
String s1=Files.readString(p);
System.out.println(s1);
```

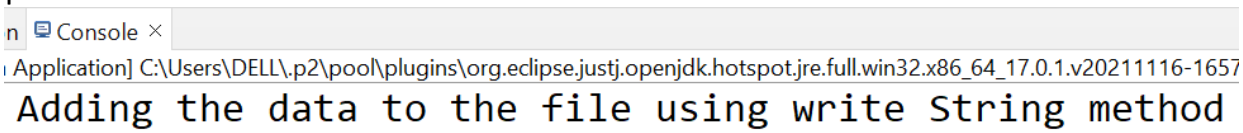
  - Here third parameter is **StandardOpenOption.Append** -> This will append the text to the data that is already present in the file. The append will happen at the end of the data in the file
    Example: in the contact.txt -> the file contains Test as word then output of the WriteString above method is shown below



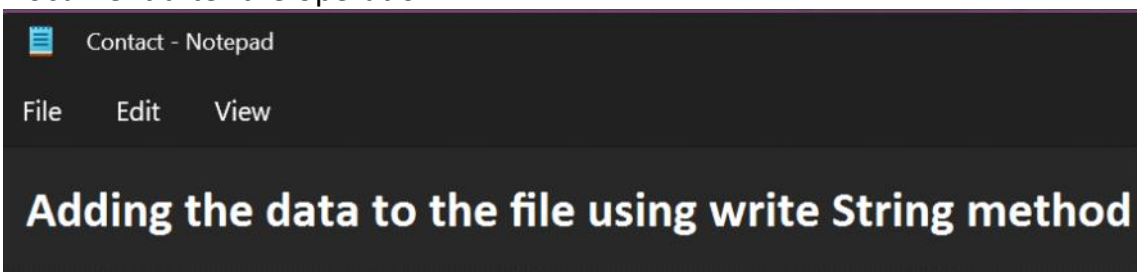
If you open the file then below content is shown



StandardOpenOption.CREATE ->This method will override the content that is present in the mentioned file.



Document after the operation:



The below is the other way of creating the file using createTempFile method

```
Path path = Files.writeString(Files.createTempFile("test", ".txt"), "Java 11 features");
```

➢ Local Variable Declaration for Lamda:

- Java 11 adds the support for Local-Variable syntax for lambda expressions. Lambdas can infer the type, but using the var keyword allows us to use annotations like @NotNull or @Nullable with the parameters.
- (@NotNull var str) -> "$" + str
- In order to mention the local variable we can use **var.**
- In Java 10, Local-variable type-inference was introduced.
- It allows us to use var as data type of local variable instead of actual data type. (Example: var x=10)
- Local variable are those variables, which are defined inside a particular block of code like method, constructor and init blocks
- We cannot use this var for the method parameter, constructor parameter
- This var should be initialized where it is declared
- Java compiler infers the data type based on the value assigned to the variable.

  Example:

```
void m1(){
        String x="ABC"
        Int y=10;
} //Here in the above m1 method compiler knew that x is String type and y is int type
```

Same method we can like below from java 10.

void m1(){
var x="ABC"
Var y=10; // here by looking at the 10 compiler understand sthat it takes int value but late you can re initialize this y with another int value like below
y=25 but not you cannot change the type (y="A")
}//The difference here is that instead of specifying the data type, by assigned value compiler will understand infer the data type.

- Here when we use local variable like var-> Initialization is mandatory.
- From Java 11 the new feature is we can declare this local variable var In lamda expression
- **Rules for Lamda local variables.**
  - (var s1, s2) -> s1 + s2 //no skipping allowed
  - (var s1, String y) -> s1 + y //no mixing allowed
  - var s1 -> s1 //not allowed. Need parentheses if you use var in lambda.

1. If there are multiple parameters then use **var** with all the parameters. No skipping allowed.
2. If one parameter use **var** then other parameters must use **var** rather than other types like **int**, **float**, etc.
3. Must use parenthesis () while using the **var** with the parameters.

➢ Predicate Interface Changes:

- **The not() Method**
  A static not() method has been added to the Predicate interface in Java 11. As the name suggests, this method is used to negate a Predicate. The not() method can also be used with method references.
- Predicate.not() is a static method which is introduced in Java 11 to negate the supplied Predicate
- Syntax of not method of predicate interface

```
@SuppressWarnings("unchecked")
static <T> Predicate<T> not(Predicate<? super T> target) {
    Objects.requireNonNull(target);
    return (Predicate<T>)target.negate();
}
```

This not() static method is taking the predicate as input and returning the negate [predicate as output. Internally it is calling the negate() method

Before Java 11 : Negating The Predicate
This is the implementation before Java 11

```
List<String> listOfStrings = Arrays.
        asList(" ", "\t", "\n", "One", "Two", "Three");
//To get the blank String List, below is the Lamda
List<String> blankStrings = listOfStrings.stream().
        filter(String::isBlank).collect(Collectors.toList());
//To get the non blank String List, below is the Lamda
List<String> nonBlankStrings = listOfStrings.stream().
        filter(str -> ! str.isBlank()).collect(Collectors.toList());
System.out.println(blankStrings);//Output is -> [ ,     , ]
System.out.println(nonBlankStrings);//Output is -> [One, Two, Three]
```

Using the same example with the Predicate.not() method

```
List<String> java11NonBlankList=listOfStrings.stream().filter(Predicate.not
        ((s)->s.isBlank())).collect(Collectors.toList());
System.out.println(java11NonBlankList);//output => [One, Two, Three]
```

➢ Http Client:

- In Java 11, HTTP Client has introduced newly, through which we can send the request and get the response.
- This HttpClient is present in java.net.http package.
- Making HTTP requests is a core feature of modern programming, and is often one of the first things you want to do when learning a new programming language. For Java programmers there are many ways to do it - core libraries in the JDK and third-party libraries
- Using HttpURLConnection or ApacheHttpClient HTTP request and response can be obtained.
- An enhanced HttpClient API was introduced in Java 9 as an experimental feature. With Java 11, now HttpClient is a standard. It is recommended to use instead of other HTTP Client APIs like Apache Http Client API. It is quite feature rich and now Java based applications can make HTTP requests without using any external dependency.

Steps

- Following are the steps to use an HttpClient.
  - Create HttpClient instance using HttpClient.newBuilder() instance
  - Create HttpRequest instance using HttpRequest.newBuilder() instance
  - Make a request using httpClient.send() and get a response object

```
public void get(String uri) throws Exception {
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(uri))
            .build();

    HttpResponse<String> response =
            client.send(request, BodyHandlers.ofString());
```

```
public void get(String uri) throws Exception {
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
          .uri(URI.create(uri))
          .build();

    HttpResponse<String> response =
          client.send(request, BodyHandlers.ofString());

    System.out.println(response.body());
}
```

The above example uses the `ofString` `BodyHandler` to convert the response body bytes into a `String`. A `BodyHandler` must be supplied for each `HttpRequest` sent. The `BodyHandler` determines how to handle the response body, if any.

Asynchronous Get

The asynchronous API returns immediately with a `CompletableFuture` that completes with the `HttpResponse` when it becomes available. `CompletableFuture` was added in Java 8 and supports composable asynchronous programming.

Response body as a String

```
public CompletableFuture<String> get(String uri) {
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
          .uri(URI.create(uri))
          .build();

    return client.sendAsync(request, BodyHandlers.ofString())
          .thenApply(HttpResponse::body);
}
```

The `CompletableFuture.thenApply(Function)` method can be used to map the `HttpResponse` to its body type, status code, etc.

Post

A request body can be supplied by an `HttpRequest.BodyPublisher`.

```
public void post(String uri, String data) throws Exception {
    HttpClient client = HttpClient.newBuilder().build();
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(uri))
            .POST(BodyPublishers.ofString(data))
            .build();

    HttpResponse<?> response = client.send(request, BodyHandlers.discarding()
    System.out.println(response.statusCode());
}
```

The above example uses the `ofString BodyPublisher` to convert the given `String` into request body bytes.

The `BodyPublisher` is a reactive-stream publisher that publishes streams of request body on-demand. `HttpRequest.Builder` has a number of methods that allow setting a `BodyPublisher`; `Builder::POST`, `Builder::PUT`, and `Builder::method`. The `HttpRequest.BodyPublishers` class has a number of convenience static factory methods that create a `BodyPublisher` for common types of data; `ofString`, `ofByteArray`, `ofFile`.

➢ Nested Based Access

- Java 11 introduced a concept of nested class where we can declare a class within a class. This nesting of classes allows to logically group the classes to be used in one place, making them more readable and maintainable. Nested class can be of four types –
  - Static nested classes
  - Non-static nested classes
  - Local classes
  - Anonymous classes
- Java 11 also provide the concept of nestmate to allow communication and verification of nested classes.
- Java 11 introduced **nest-based access control** that allows classes to access each other's private members **without the need for bridge methods** created by the compiler. These methods are called **accessibility-broadening bridge methods** and the compiler inserts these into the code during the program execution.
- Before Java 11, if we have private members in our code then the compiler creates accessibility-broadening bridge methods that increase the size of the deployed applications and may lead to confusion. That's why Java improved nest-based access control.
- Java 11 allows classes and interfaces to be nested within each other. These nested type can be private fields, methods, and constructors.

Java added some new methods to Java Reflection API Class. We can use these methods to get information on nest-based access control.

```
public Class<?> getNestHost()
public boolean isNestmateOf(Class<?> c)
public Class<?>[] getNestMembers()
```

The `getNestHost()` method is used to get the name of nest host while the `isNestmateOf()` method is used to check whether a class is a nestmate.

The `getNestMembers()` method returns an array of nest members including classes and interfaces.