JAVA 7

* **Try with resources statement**
* We can add the resources in the try statement, which will close the resource automatically after the execution of the try block. **finally()** is not required here.

***try (BufferedReader reader = new BufferedReader(new FileReader(inputFilePath));***

***BufferedWriter writer = new BufferedWriter(new FileWriter(outputFilePath))) {***

***}***

* The resources which we define in try with resources block are effectively final because JVM needs to close that resource. This is because if we create a resource with the same variable defined in try with resources it will lead to confusion to the JVM.
* ***java.lang.Autocloseable*** interface is used for this purpose.
* ***java.lang.Autocloseable*** interface has a method void close which throws generic exceptions. Prior Java 7 there was a interface ***java.io.Closeable*** interface which throws IOException, which will be used to close the resource. Example***: br.close(), conn.close().***
* We can create a custom resource class by implementing ***AutoCloseable*** interface.
* A try-with-resources statement can itself have catch and finally clauses for other requirements inside the application.
* Till Java 1.6, try block should be followed by either catch or finally block but from Java 7 we can have only try with resource block with out catch & finally blocks
* **Suppressed Exceptions**
* Suppressed exceptions are the exceptions thrown in the code but were ignored somehow. One of the classic example is in the scenario’s ‘try-catch-finally’ block execution, where we received an exception in try block and again there is one more exception thrown inside the catch block or finally block, due to which the super exception from try block will be ignored.
* To support suppressed exceptions better handling, a new constructor and two new methods were added to the ***Throwable*** class (***parent of Exception and Error classes***) in JDK 7.

***Throwable.addSupressed(aThrowable)***

***Throwable.getSupressed(); // Returns Throwable[]***

* **Catching multiple exceptions using a single catch block**
* Before java7 we should have multiple catch statements to catch specific Exceptions.
* To reduce the code duplication in the scenarios where we have a common action/business logic need to be performed for different run time exceptions we can use a single catch block with multiple exceptions separated by a **|** operator.

***catch(NullPointerException | ArrayIndexOutOfBoundException ex){***

***}***

* When you catch multiple exceptions, the exception variable is implicitly final. So, you cannot assign a new value to ex in the body of the catch clause.
* **Rethrowing exceptions with more inclusive type checking**
* **Easier Exception handling for reflective methods**
* Clubed exceptions such as *IllegelAccessException*, *InvocationTargetException*, *NoSuchMethodException*, *ClassNotFoundException* into one exception ***ReflectiveOperationException*** which is the super class of the mentioned exceptions.
* **Objects class & Null checks**
* Java.util.Objects is introduced in Java7 which has static utility methods for null checks, computing hash code, comparing two objects, returning a string for an object.
* There are two methods to null-check the objects. Both of them will throw NullPointerException if the object is null. We can send custom message also along with the NullPointerException
* requireNonNull(T Obj)
* requireNonNull(T Obj, String message)
* **Close method inside URLClassLoader**
* URLClassLoader is used to load classes or resources from a search path of URLs referring to both jar files or directories.
* In Java7 adds a close method to close the URLClassLoader by implementing AutoCloseable. We can add the URLClassLoader in the try-with-resources statement.
* **@SafeVarargs Annotation**
* It is used suppress the unsafe operation warnings at the compile time.
* **@SafeVarargs** annotation is used to indicate that methods will not cause heap pollution. These methods are considered to be safe.
* @SafeVarargs can only be applied on
* Static methods
* Final methods
* Constructors
* From Java9 it can also be used in private instance methods.
* We cannot use this annotation on public methods because we cannot use this on any overridden methods.
* **Enhancements related to files and directories**
* Below are the newly created interfaces and classes created in Java 7
* java.nio.file.Path
* java.nio.file.Paths
* java.nio.file.Files
* ***java.io*** package is blocking in nature, whereas ***java.nio*** package is non blocking in nature, and needs less threads.
* ***java.io*** package can be used to handle large file operation, ***java.nio*** package is suitable for smaller files.
* **WatchService**
* ***java.nio.file.WatchService*** is used to keep watch of any changes in a file or a directory.
* Before Java 7 this could be done by using the pooling of threads
* **Binary Literals**
* A binary literal is a number that is represented in 0s and 1s (binary digits). It is a representation of integral types (byte, short, int, and long) in a binary number system especially when we are dealing with the bit-wise operators.
* We just need to add a prefix of either 0b/0B in Infront of the binary digit representation.

***int num = 0B111;***

***System.out.println(“The number value is = "+ num); // This will print 7 on the console***

* **Using String in switch statements**
* From Java 7, Java allows to use String objects in the expression of switch statement.
* String value is case sensitive and null objects are not allowed. In case if you use null value, java will throw an NullPointerException.
* It must be only String object and normal Object is not allowed
* **Diamond Operator or Type Interference Operator**
* From Java 7, Java provides a improved compiler which is smart enough to infer the type of generic instance. It simplifies the use of generics when creating an object.
* We can’t use this diamond operator feature inside the anonymous inner class. But this limitation is resolved in Java 9
* **Using underscore in numeric values**
* We can use the underscore(\_) inside the numeric values to improve the readability of the code. However, the compiler will remove them internally while processing the numeric values
* Below are some restrictions while using the underscore inside your numeric values
* It is not allowed at the beginning or end of a number
* It is not allowed adjacent to a decimal point
* It is not allowed prior to an L or F suffix that we use to indicate long/float numbers
* It is not allowed where a string of digits is expected.
* **JDBC Improvements**
* JDBC 4.1, which is part of Java 7, introduces the following features
* The ability to use a try-with-resources statement to automatically close resources of type Connection, ResultSet, and Statement
* **RowSet 1.1**: The introduction of the ***RowSetFactory*** interface and the ***RowSetProvider*** class, which enable you to create all types of row sets supported by your JDBC driver
* **Fork and Join Framework**
* Fork & Join framework supports parallel processing by splitting (forking) big tasks into a multiple small subtasks that can be processed independently by using all the available CPU cores and eventually joining the results from all the subtasks to get the final results.
* ForkJoin framework follows ‘Work Stealing Algorithm’ where free threads try to “steal” work from deques of busy threads
* Java 8's parallel streams underline uses the ForkJoin framework