Java Features

Shristi Technology Labs

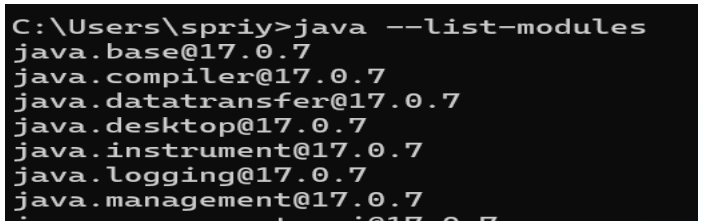
Java 9 Features

- JShell
- Factory Methods for Immutable List,Set,Map
- Private Method in interfaces
- Process API improvement
- Try With Resources improvement
- Enhanced @Deprecated Annotation
- DiamondOperator for anonymous inner classes
- Optional Class Improvements
- Reactive Streams
- ControlPanel - *removed in latest versions*

Java Modules

- Modules are a new level of abstraction above packages
- A module is a group of packages and resources together with a module descriptor file
- package of packages(separate project for each module)
- The naming rules for a module are similar to packages naming

  - reverse DNS names/project-style
- By default all packages are module private.
- Modules can be distrubuted as JAR or expolded compiled project
- can only have one module per JAR file.
- Modules are split into four groups - java,
  - java modules are implementation classes for core
  - **`java --list-modules`**



Java Modules

Packages

- Packages are used to determine what code is publicly accessible outside of the module.

Resources

- Resources like media or configuration files are available in the module

Module Descriptor

- **Name** – the name of the module
- **Dependencies** – a list of modules that this module depends on
- **Public Packages** – a list of packages that can be accessed outside the module
- **Services Offered** – service implementations that can be consumed by other modules
- **Services Consumed** – allows the current module to be a consumer of a service
- **Reflection Permissions** – explicitly allows other classes to use reflection to access the private members of a package

Module Directives

- Requires - both runtime and compile time
- Requires Static - required only during compile time
- Requires Transitive
- Exports - export the module for other modules to use
- Exports ... To - exports to specific package
- uses - consumes the service
- Provides ... With - provides the service implementation
- Open
- Opens
- Opens ... To

Example

```
module com.shristi.second {
    uses com.shristi.demo.Calculator;
    requires com.shristi.firstmodule;
    requires com.shristi.fourthmodule;
    // available only in compile time
    requires static com.shristi.thirdmodule;
}
```

```
module com.shristi.fourthmodule {
    provides com.shristi.demo.Calculator with com.shristi.demo.CalculatorImpl;
    // export package name to another package
    exports com.shristi.demo to com.shristi.second;
}
```

```
Greeter greet = new Greeter();
System.out.println(greet.greetUser("Sri"));
// consumes the service
Iterable<Calculator> calcIter = ServiceLoader.load(Calculator.class);
Calculator calculator = calcIter.iterator().next();
calculator.calculate(20,30);
```

JShell

- JShell is Java Shell and known as REPL (Read Evaluate Print Loop).

- It is used to execute and test any Java Constructs like class, interface, enum, object, statements etc. very easily.

```
C:\Users\spriy>jshell
|  Welcome to JShell -- Version 17.0.7
|  For an introduction type: /help intro

jshell> int a=90;
a ==> 90

jshell> System.out.print("Hello");
Hello
jshell>
```

Example

```
jshell> 3+1
$1 ==> 4

jshell> 2+10
$2 ==> 12

jshell> String call(String name){
   ...>     return "Great Day "+name;
   ...> }
|  created method call(String)

jshell> System.out.println(call("Sri"));
Great Day Sri
```

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
```

Example - UnmodifiableList, Set

```
// pre java 9
List<String> courses = new ArrayList<>();
courses.add("Java");
courses.add("Angular");
List<String> newcourses = Collections.unmodifiableList(courses);
newcourses.add("Node");

// from java 9 unModifiableList
List<String> ncourses = List.of("Perl", "Java", "Angular", "Node");
ncourses.add("Node");
ncourses.forEach(System.out::println);
List<String> emptyList = List.of();
System.out.println();

// immutable Set
Set<String> imSet = Set.copyOf(ncourses);
imSet.add("Node");
imSet.forEach(System.out::println);
```

Example - Immutable Map

```
Map.of(1, "Java", 2, "Angular")
        .forEach((courseId, courseName) ->
                    System.out.println(courseName + " " + courseId));

// using Map.Entry
Map<Integer,String> map = Map.ofEntries(
                    Map.entry(1, "Leadership"),
                    Map.entry(2, "Five Point"),
                    Map.entry(3, "Life"));
map.forEach((bookId, title) ->
    System.out.println(bookId + " " + title));
```

Private methods in an interface

```
@FunctionalInterface
public interface Checker {

    boolean checkName(String username, String city);
    // private methods
    private void call() {
        System.out.println("Hello");
    }
    default void show() {
        System.out.println("Great Day");
        call();
    }

}
```

Process API improvement

- Process API is responsible to control and manage OS processes
- Has new classes and methods to manage the OS processes
- ProcessHandle class helps to get
  - the native process Id,
  - start time,
  - accumulated CPU time,arguments,command,
  - user and parent process
  - check pocess liveliness and to destroy processes

Example for ProcessHandle

```
ProcessBuilder builder = new ProcessBuilder("notepad.exe");
// opens the process notepad
Process process = builder.start();
ProcessHandle currentProcess = ProcessHandle.current();     // Current processhandle
System.out.println("Process Id: "+currentProcess.pid());    // Process id
System.out.println("Direct children: "+ currentProcess.children()); // Direct children of the process
System.out.println("Class name: "+currentProcess.getClass());       // Class name
System.out.println("All processes: "+ProcessHandle.allProcesses()); // All current processes
System.out.println("Process info: "+currentProcess.info());         // Process info
System.out.println("Is process alive: "+currentProcess.isAlive());
System.out.println("Process's parent "+currentProcess.parent());  // Parent of the process
```

Try With Resources Improvement

```
// Using Java 7
FileReader fileReader = new FileReader("demo.txt");
try (FileReader reader11 = fileReader;) {
    // logic goes here
}
// Using Java 11
FileReader reader = new FileReader("demo.txt");
try (reader) {
    logic goes here
}
```

```
RandomAccessFile rFile = new RandomAccessFile("demo.txt", "rw");
FileWriter writer = new FileWriter("demo1.txt");
try with resources 9
try (rFile;writer) {
    FileChannel channel = rFile.getChannel();
    ByteBuffer buffer = ByteBuffer.allocate(100);
```

Enhanced @Deprecated annotation

- @Deprecated annotation is from java 5
- It means it should not be uased as it may lead to errors.

```
// forRemoval - true this method may be removed in future
@Deprecated(since = "8",forRemoval = true)
public String call() {
    return "Hello";
}
```

Diamond operator in inner Class

- use <> operator with anonymous class after java 9

```
// before java 9
Mapper<String> mapper = new Mapper<String>("John") {
    @Override
    public void mapData() {
        System.out.println(data.toUpperCase());
    }
};
mapper.mapData();

// post java 9
Mapper<String> mapperOne = new Mapper<>("John") {
    @Override
    public void mapData() {
        System.out.println(data.toUpperCase());
    }
};
mapperOne.mapData();
```

Reactive Streams

- **takeWhile()** - takes all the values till the predicate returns false
- **dropWhile()** - throws all the values at the start till the predicate returns true.
- **iterate()**
- **ofNullable()**

Example of takeWhile and dropWhile

```
System.out.println("Prints till the blank line- exclusive");
//Java, Angular
Stream.of("Java", "Angular", " ", "Node", "Spring", "React")
    .takeWhile(s -> !s.isBlank()).forEach(System.out::println);

System.out.println("Starts after the blank line- inclusive");
//" ", Node,Spring,React
Stream.of("Java", "Angular", " ", "Node", "Spring", "React")
.dropWhile(s -> !s.isBlank()).forEach(System.out::println);

//1,2 ,3
Stream.of(1,2,3,4,5,6).takeWhile(i->i<4).forEach(System.out::println);
System.out.println();
//4,5,6
Stream.of(1,2,3,4,5,6).dropWhile(i->i<4).forEach(System.out::println);
```

Example of iterate

- iterate()

Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)

```
syntax
Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)

for (int index=seed; hasNext.test(index); index = next.applyAsInt(index)) {}

// initial value- seed,condt using predicate,increment using UnaryOperator
IntStream.iterate(10,x->x<100,x->x+10).forEach(System.out::println);
```

Optional class improvements

New methods of Optional class

- stream()
- ifPresentOrElse()
- or()

```
Optional<String> opt = Optional.empty();
opt.ifPresentOrElse(System.out::println,
            ()->System.out.println("No data"));
```

```
Optional<String> opt = Optional.empty();
// returns an Optional using Supplier
Optional<String> opt2 = opt.or(()->Optional.of("No data"));
System.out.println(opt2.get()); // no data

Optional<String> opt1 = Optional.of("Hello");
Optional<String> opt3 = opt1.or(()->Optional.of("No data"));
System.out.println(opt3.get()); // Hello
```

Java 10 Features

- Local-Variable Type Inference
- Consolidate the JDK Forest into a Single Repository
- Parallel Full GC for G1
- Heap Allocation on Alternative Memory Devices

Local-Variable Type Inference

- is the new feature in Java 10
- Adds type inference to declarations of local variables with initializers.
- It can be used in for each / for loop as index
- variable with ‘var’ should be initialized
- It cannot be initialized to null
- It cannot be used with non local variables

```
Illegal Declaration
var num; // should be initialized
var myList = null; // should be initialized
public var name = "Sri"; // cannot use on public variables
var con = (num)->{} ;// lambda needs explicit target type
var myarr = {1,2,3,4}; // array needs explicit target type
```

Example - var

```
// Example
// inferred value for List
var numbers = List.of(1, 2, 3, 4, 5);

// in for each loop
for (var number : numbers) {
    System.out.println(number);
}
// in for loop for index
for (var i = 0; i < numbers.size(); i++) {
    System.out.println(numbers.get(i));
}
```

```
// Ok. But not a good idea
var result = greet();// dont know the return type
// no var in streams
var res = Arrays.asList("Ram","Raj")
        .stream()
        .findFirst().map(String::length)
        .orElse(0);
// this becomes a ArrayList<Object> not book
var bookList = new ArrayList<>();
var books = new ArrayList<Book>(); //ok
```

Consolidated JDK Forest as Single Repository

- In JDK 9, there are eight module based directories termed as repos.
  - root, corba, hotspot, jaxp, jaxws, jdk, langtools, nashorn

- From Java 10 onwards, JDK forests are organized into single repository to streamline development.
- Now code in organized as -

$ROOT/src/java.base

$ROOT/src/java.compiler

Stream Methods

- **toUnModifiableList()**
- **copyOf()**
- **OrElseThrow() -** to throw NoSuchElementException

```
// Immutable List
List<String> newcourses =
        Arrays.asList("Java","Angular")
        .stream()
        .collect(Collectors.toUnmodifiableList());

// Immutable List
Set.copyOf(newcourses);
```

Time-Based Release Versioning

- Now it is time-based release of Java - a new release every six months.
- March 2018 release is JDK 10, September 2018 release is JDK 11 - - These are called feature releases and contains at least one or two significant features
- Java 11 is an LTS release

Feature Release

- contains language specific features, JVM features, New/Improved APIs, Removal/Deprecation of APIs.

Update Release

- includes bug fixes, security issue fix, regression fixes etc.
- Each update release happens per quarter in Jan, April, July and Oct months

Long Term Stable Release

- Long term support release will be announced after every three years

Version Format

version format: **$FEATURE.$INTERIM.$UPDATE.$PATCH**

$FEATURE − It denotes the major feature release and will get incremented by 1 after every Feature Release. like 10,11

$INTERIM − It denotes any non-feature, non-update release which contains bug fixes and enhancements.

$UPDATE − It denotes the Update release done after a feature release. eg,, an update release of Java in Apr 2018 is JDK 10.0.1 and for July 2018 is JDK 10.0.2 and so on.

$PATCH − It denotes any emergency release incremented only in case an critical issue is to be promoted on emergent basis.

Parallel Full GC for G1

- Before java 10 - GC (Garbage Collector) implementation components were scattered within code base
- In Java 9 - **G1 (Garbage First) garbage collector** was used.
- G1 avoids full garbage collection but in case of concurrent threads look for collection and memory is not revived fast.
- With Java 10, now G1 will use a fall back Full Garbage Collection
- Moved from single threaded to parallel thread
- With JEP 307, a parallel thread will start mark-sweep-compact algorithm.
- The number of threads can be controlled using following option.

```
$java -XX:ParallelGCThreads=4
```

Heap Allocation on alternative memory

- User can specify an alternative memory device, to allocation the java heap space.
- User need to pass a path to the file system using a new option **-XX:AllocateHeapAt**.

  -XX:AllocateHeapAt=~/etc/heap
- It takes file path and do a memory mapping

Java 11 Features

- Running Java File with single command
- New utility methods in String class
- Local-Variable Syntax for Lambda Parameters
- Nested Based Access Control

Running Java File with single command

- No need to compile the java source file with **javac** tool first.
- Directly run the file with **java** command and it implicitly compiles

```
D:\2021\apps\eclipseexamples\javanew23>java Trial.java
Hello
```

New utility methods in String class

- **isBlank()** - checks if the string is blank
- **lines()** - returns a stream of strings, is a collection of all substrings split by lines.
- **strip()** - removes all kinds of whitespaces leading and trailing
- **stripTrailing()** - removes all kinds of trailing whitespaces
- **stripLeading()** - removes all kinds of leading whitespaces
- **repeat(int n)** - repeat the string n number of times

Example

```
//isBlank and strip
System.out.println("".isBlank()); // true
System.out.println(" ".isBlank()); // true
// removes trailing and leading
System.out.println("Great!!!"+(" Sripriya ").strip()+"Bye");

String message = " Have a great day\n  Reading is fun\n Enjoy Sports ";

// converts a String separated by new lines into a Stream<String>
message.lines().forEach(str -> System.out.println(str.strip()));
// strip Leading
System.out.println("Have fun!"+(" Music Lovers ").stripLeading() + "Bye");
// strip Trailing
System.out.println("Music Lovers".concat(" Have fun ").stripTrailing() + "Bye");
// repeat
System.out.println("Good Life ".repeat(3));
```

Local-Variable Syntax for Lambda Parameters

- This is the only language feature release in Java 11
- If an annotation like @Nullable is applied, then type definition is required
- Specify for all parameters or none
- Even in case of one paramater paranthesis is needed

```
BiConsumer<String,String> con =
        (var s1,var s2)->System.out.println(s1.concat(s2));
con.accept("Welcome", "home");

// error as both paramaters should have type declared
BiConsumer<String,String> con1 =
        (s1, var s2)->System.out.println(s1.concat(s2));

// data type is must if annotation is used with variable
// can use String or var
BiConsumer<String,String> con2 =
    (@Nullable var s1, var s2)->System.out.println(s1.concat(s2));
```

Nested Based Access Control

Problem:

- The nested types have unrestricted access to each other, including to private fields,

Solution

- One nest member (typically the top-level class) is designated as the nest host.
- It contains an attribute (NestMembers) to identify the other statically known nest members.
- Each of the other nest members has an attribute (NestHost) to identify its nest host.

Performance Tuning

- Use a profiler to find the real bottleneck
- Avoid BigDecimal or BigInteger
- Use StringBuilder to concatenate Strings programmatically
- Use + to concatenate Strings in in one statement(in case of a query)
- Use primitives whereever possible
- Cache expensive resources, like database connections and creating Integer objects

Performance Tuning

- Use the Latest Stable Java Version
- Size the Java Heap memory correctly - using Xmx, Xms
  - Donot have the heapsize too small
- Set the initial heap size ***java -Xms256m -Xmx2048m***
  - Xms is the initial setting of the heap memory size
- Use StoredProcedures instead of queries as SP are precompiled
- Choose the right garbage collector (SerialGC, Parallel GC)
- Tune the JVM Garbage Collector to reduce the time required for a full GC
- Avoid Database connection delays. Use Pools

JNI - Java Native Interface

- JNI is an interface that allows Java to interact with code written in another language.
- Supports code reusability and performance.
- Can reuse existing/legacy code with Java (mostly C/C++).
- The native code used to be up to 20 times faster than Java, when running in interpreted mode.
- JNI can be used to invoke Java code from within natively-written applications written in C/C++.
- Java command-line utility is an example, that launches Java code in a Java Virtual Machine.

JNI Components

- **Java Code** – The class with native method.
- **Native Code** – the actual logic of native methods, coded in C or C++.
- **JNI header file** – this header file for C/C++ (include/jni.h into the JDK directory) includes all definitions of JNI elements that we may use into our native programs.
- **C/C++ Compiler** – Choose between GCC, Clang, Visual Studio, or to generate a native shared library for our platform.

JNI Data Mapping to variables

boolean	jboolean
byte	jbyte
char	jchar
double	jdouble
float	jfloat
int	jint
long	jlong
short	jshort
void	void

Java 16

- Records

Records

- Record Class is the common base class of all record classes

- It is a transparent carrier for a fixed set of values
- The variables are called the record components - they are final
- Records can have static fields, static block
- Records have *fields, all-args constructor, getters, toString, and equals/hashCode* methods

- Records are immutable. No setter methods
- Record classes are final. Records can implement other interfaces
- Records are used as data transfer objects (DTOs)

Example

```
// username and city are record components
public record User(String username, String city) {

    // can have static fields, block
    static String message;
    static {
        message = "Great day";
    }

    // can have static methods
    public static void show() {
        System.out.println("static method");
    }

    // can have concrete methods
    public void call() {
        System.out.println("in record");
    }
}
```

```
    User userRecord = new User("Sri", "Bangalore");
    // to get the record components
    System.out.println(userRecord.username());
    System.out.println(userRecord);
    System.out.println(User.message);
    User.show();
    userRecord.call();
}
```

Java 17

- Sealed Classes

Sealed Classes

- A sealed class allows to choose its sub-classes.
- Use the keyword ***sealed*** to create a sealed class.
- A sealed class must be followed by ***permits*** keyword along with the list of classes that can extend it.
- The permits clause specifies the classes that are permitted to extend the sealed class/implement the sealed interface
- The extending classes should be sealed/non-sealed/final

eg.,

```
public sealed class Vehicle permits Car, Bike{}
```

```
public non-sealed class Car extends Vehicle {}
```

Non-Sealed Classes

- If a class is non-sealed the parent-child hierarchy ends here.
- But the non-sealed class can be extended by other classes
- Use the keyword ***non-sealed*** to create a non-sealed class.
- This class cannot be extended

Rules

- A sealed class must define the classes that may extend it using ***permits***
- The permitted child class must extend the sealed class and must either be final, sealed or non-sealed.
- If the parent sealed class is in a module, then the child classes must also be in the same module
- If the parent sealed class is in an unnamed module, then the child classes must in the same package,
- The non-sealed class can be extended

Sealed Interface

- An interface may allow to choose its child interfaces or classes that can extend it using permits.

```
public sealed interface Calculator permits  CalculatorImpl,Scientific{}
```

```
public non-sealed class CalculatorImpl implements Calculator {}
```

```
public non-sealed interface Scientific extends Calculator {}
```

Sealed Records

- A record can not extend a normal class,but can implement a ***sealed*** interface.
- Further, a record is implicitly final.

```
sealed interface Device permits Mobile{}
public record Mobile() implements Device {}
```

Thank you