JAVA Core

1. Java Syntax
    1. Execution of program and entry point
    2. Classes and Byte Code
    3. File Extensions (.java and .class)
    4. Methods and Fields in java
    5. lambda functions in java (Syntax)
    6. Compiling and Running Java File
    7. Building and Running Packages
    8. Class Path in java and its importance

1. Java Syntax

Any java file will be having one and only public class and by convention that class should represent the name of the java file, in which it is written.

Every Java Application will be started from the main function, any function which we want to write, should always be encapsulated in "class", now this function is known as "method" of that particular class.
The entry-point, from where the program will be executed should exactly be like :

```
public static void main(String[] args){ }
```

This represents the main method such as :

1. **public** : Will make the main method to access from outside the class.
2. **static** : While executing JVM can call this method without making the object of the class, and the memory is allocated until the program is finished.
3. **void** : it defines the return type of main method must be void as per the JVM conventions, without void it will not run it is a design convention.
4. **main** : while execution first JVM will look for the method named main and start execution.
5. **String[] args** : "args" is basically an array of String, it signifies the Command line arguments that are to be given while running the program, String array is used as the String can be converted to anything, be it an integer, double, boolean, or char itself.

Here is another class which is not public, that's why it can be created in this java file, the convention to create a class are :

1. **Pascal Case** : First letter is capital, space is denoted by capital letters. Ex: "Another Class" will be "AnotherClass"
2. **Syntax** : The identifier in this case are public, abstract, final.
   `"Identifier" "final/_" "class" "ClassName" { }`
3. **Java Do not permit to make class of Name class because class is a keyword.**

For any class Syntax it should be declared public in the file Syntax.java, this java file can have non-public classes as well

```
public class Syntax {
        public static void main(String[] args) {
        }
}

class AnotherClass {
}
```

```
abstract class AbstractClass {
}

final class FinalClass {
}
```

1.1 Execution of program and entry point

In Java, the execution of a program follows a specific sequence of steps :

1. **Compilation** :
    - Java is a compiled and interpreted language, which means that the source code is translated into an intermediate form called bytecode by the Java compiler. The compiler generates bytecode from the `.java` source file and create a `.class` file.
    - To compile a Java program, you use the `javac` command for a `Program.java` file.
    - `javac Program.java`
2. **Byte Code** :
    - Bytecode is a set of instructions that is platform-independent. It is not machine code but is designed to be executed by the Java Virtual Machine (JVM).
3. **Java Virtual Machine (JVM) :**
    - The JVM is an integral part of the Java Runtime Environment (JRE). It is responsible for executing the bytecode by using interpretation, produced during the compilation phase.
    - When you run a Java program, you use the `java` command followed by the name of the class containing the `main` method (the entry point of the program).
    - `java Program`
    - The JVM loads the bytecode, interprets it, and executes the program.
4. **Execution:**
    - The JVM executes the program by interpreting the bytecode or by using Just-In-Time (JIT) compilation to convert the bytecode into native machine code for the specific platform at runtime.
    - The `main` method is the starting point of execution. The JVM calls the `main` method to begin the program's execution.
5. **Runtime** :
    - During runtime, the program performs the specified tasks and interacts with the environment as defined in the Java code.

It's important to note that Java is designed to be platform-independent, and the JVM plays a crucial role in achieving this. Once the bytecode is generated, it can be executed on any system that has a compatible JVM installed.

JAVA Features/Buzzwords/Characteristics

1. **Simple:**
    - Java was designed to be easy to use and simple. It eliminates complex features like pointers, operator overloading, and explicit memory management that can be error-prone.
2. **Object-Oriented:**

- Java is an object-oriented programming language, promoting the use of classes and objects. It supports concepts like encapsulation, inheritance, and polymorphism.

3. **Platform-Independent:**
   - One of Java's most significant features is its platform independence. Java programs are compiled into bytecode, which can be executed on any device with a Java Virtual Machine (JVM), making Java applications portable across different platforms.
4. **Distributed Computing:**
   - Java has built-in support for distributed computing with the Remote Method Invocation (RMI) and Java Remote Method Protocol (JRMP). This allows objects in a Java virtual machine to invoke methods on objects in another JVM.
5. **Multithreaded:**
   - Java provides built-in support for multithreading, allowing the concurrent execution of multiple threads within a program. This is particularly useful for developing responsive and efficient applications.
6. **Robust and Secure:**
   - Java is designed to be robust and provides features like automatic memory management (garbage collection) to minimize the risk of system crashes due to memory issues. Additionally, Java's security features, such as the sandbox model and bytecode verification, enhance the overall security of Java applications.
7. **Dynamic:**
   - Java supports dynamic loading of classes, which means classes are loaded on-demand during runtime. This allows for more flexibility and efficiency in managing and using resources.
8. **Architecturally Neutral:**
   - Java is designed to be architecturally neutral, meaning that the implementation of the language does not depend on the underlying hardware and operating system.
9. **High Performance:**
   - While Java is an interpreted language, its performance is optimized through Just-In-Time (JIT) compilation. JIT compilation translates bytecode into native machine code at runtime, improving execution speed.
10. **Rich Standard Library:**
    - Java comes with a comprehensive standard library (Java API) that provides a wide range of pre-built classes and methods, making it easier for developers to perform common tasks without having to write extensive code.

1.2 Classes and Byte Code

In Java every `.java` file is to be converted in `.class` file before execution, however if one java file is having more than one class, every class which is determined by the syntax will corresponds to a new `.class` file these files consists of the Byte Code which are the instructions that is only understood by the JVM (Java Virtual Machine), JVM will then execute the program using those byte code instructions.

Classes and its significance in java

In a java program a class is treated as a component of the program or the whole program, while writing an application in java a class is a crucial component in java that will help in writing much scalable and maintainable code.

There are some convention that is to be followed while writing classes in java :

1. Every java file should only be having one class, which is responsible for certain action only
2. The interface should only be used to make document for a certain class or for abstraction purpose.
3. States that are to be used in whole application should be detached and independent
4. Program to the interface while writing business logic
5. Embrace the naming conventions of Pascal Case for classes and all lower case in packages

Byte Code

The Byte Code for the `.java` file is created in the form of `.class` this class file will also have the execution of the application and how the program should look into the right class for execution of the program, while running an java application, byte code of all the classes should work in a flow, to get assured of this factor one need to take a look at how the JVM execute multiple class files.

While compiling multiple classes in java, JVM searches for the class which consist of main method and then use other classes whenever they are to be called in the application, while running Byte Code, it is necessary to build it in a right manner

The JVM will find the classes to be run using $CLASSPATH the class path is determine by the package, so if you need to run an application having 4 to 5 different packages you need to run it from the application package where these 4 to 5 packages are.

```
MyJavaApp
|-- src
|      `-- main
```

```
|             `-- java
|                     `-- com
|                             `-- learn
|                                     `-- test-app
|                                         |-- Main.java
|                                         `-- service
|                                             `-- ServiceClass.java
```

In the above directory tree the main function is in the test-app directory in Main.java file, so the class path till the main function from the execution directory will be `com.learn.test-app.Main` in here the `com.learn.test-app` is the main package where all the classes are to be written that is the reason JVM will make this as a class path and use it to run the main function and execute the program.

1.3 File Extensions (.java and .class)

The file extension for java source file and java compiled file is completely different, the file in which you need to write the java code is to be saved as `YourClassName.java` after compilation using the command `javac YourClassName.java` it will generate a separate file in the directory named as `YourClassName.class` which is not readable but can be opened up in a text editor, it consist of symbols and other characters that is created by using `javac`

However using `java` command you can also run the program directly from `.java` extension files, This is to be done by REPL (Read Eval Print Loop), this environment can also be accessed by using `jshell` command it will facilitate you with a (REPL console) in which you can write and run java code line by line

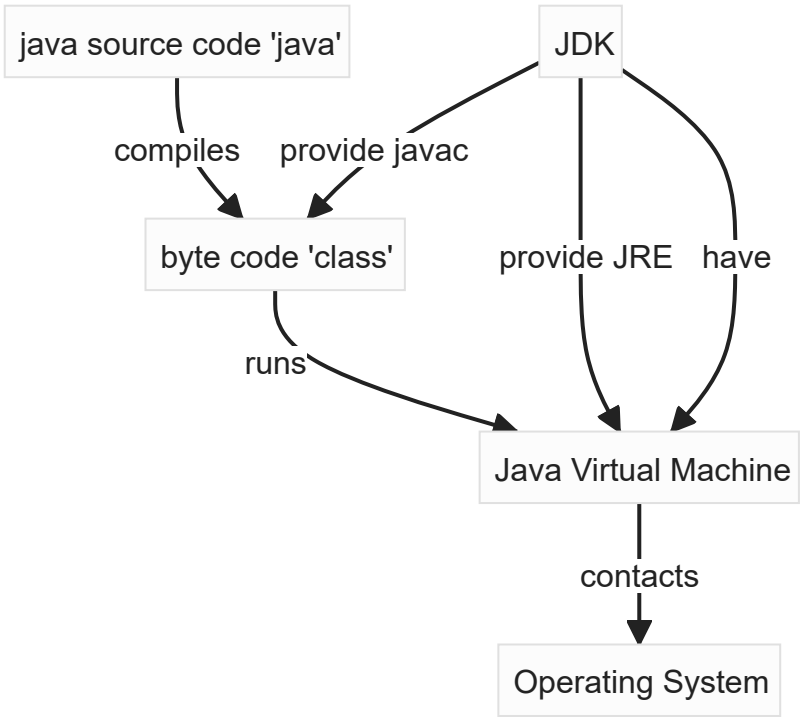
```
❯ jshell
|  Welcome to JShell -- Version 17.0.7
|  For an introduction type: /help intro

jshell> System.out.println(1>>>2);
0

jshell> /exit
|  Goodbye
```

The above example shows how to access jshell from a command line where jdk is installed, JDK is also known as Java Development Kit it is a combination of JVM and JRE (Java Virtual Machine and Java Runtime Environment), The Byte Code can be run on the JDK irrespective of the Platform.

JDK is a directory which consist of binaries and executables for specific machines that provide initiation and rules to run the Byte Code as a Program or Application.



1. **Java Virtual Machine (JVM):**
   - The JVM is an abstract machine that provides the runtime environment in which Java bytecode can be executed.
   - It abstracts the underlying hardware and operating system details, providing a consistent platform for Java programs to run.
2. **Java Runtime Environment (JRE):**
   - The JRE is a package of software components that provides the necessary runtime environment for Java applications to run.
   - It includes the JVM, libraries, and other components required for running Java applications but does not include development tools.
3. **Java Development Kit (JDK):**
   - The JDK is a full-featured software development kit that includes everything in the JRE, plus additional tools and utilities needed for Java development.
   - It includes the compiler (`javac`), debugger, and other development tools. Developers use the JDK to write, compile, and debug Java applications.

1.4 Methods and Fields in java

In java everything should be encapsulated in a class except a code block that is why, whenever we are writing functions in the class these function are known as methods, the syntax for writing a function is :

```
"Access modifier" "final,_" "static,_" "return type" "method name"("parameters")
{
        // function body
}
```

This method syntax indicates the following :

1. In this case **"Access modifier"** can be **"public, private, protected"**
2. **"final"** indicates that if the method can be overridden or not.
3. **"static"** indicates that this method can be used without making an object of class also it can be used in the "static context" meaning it is now the member of class directly associated with it.
4. **"return type"** indicates that what this function should return.
5. **"method name"** indicates the function name conventionally given in camel case.
6. **"parameters"** indicates the values on which the function depends upon.

In java we can define method wherever needed in the class file it is not compulsory to define it before using it, that is if we want to use x method in main method we can define it after main method or before main method it will act as same.

The below class is named method and have all the different types of method that can be there in java.

```
class Method {

        // cannot be overridden while inheriting this class
        public final String doSomething(String time) {
                return "done something at this" + time;
        }

        // is not accessible to outer classes
        private void hasMethod() {
                System.out.println("this class has methods");
        }

        // accessible in same package
        // but in different package by only those who inherit this class using
extend
        protected void hasRealMethod() {
```

```
                System.out.println("this class really have methods");
        }

        // a class member method
        public static String directAccess() {
                return "can be access using the classname only no need to make
object";
        }
}
```

The Method class is inherited to show how override is done while using protected keyword in different package.

```
class InheritMethod extends Method {
        // this annotation determines that the overriding process can only apply
on methods
        @Override
        protected void hasRealMethod() {
                System.out.println("now implementation is followed that is used
in this class");
        }

}
```

The entry point and usage of the above methods :

```
public class TopicMethods {

        public final static String thisMethod() {
                return "this method cannot be overridden also is static";
        }

        public static void main(String[] args) {
                // static methods
                System.out.println(thisMethod());

                // static method of class
                System.out.println(Method.directAccess());

                // public members of class
                Method m = new Method();
                System.out.println(m.doSomething("2023"));

        }
}
```

While designing any application methods determines the behavior for the real world entity, and field determines the state of the real world entity. This can be shown by a class diagram

Class Diagram Representation (UML)
A class diagram is a type of UML (Unified Modeling Language) diagram that represents the structure of a system by showing the classes of the system, their attributes, methods, and the relationships among them.

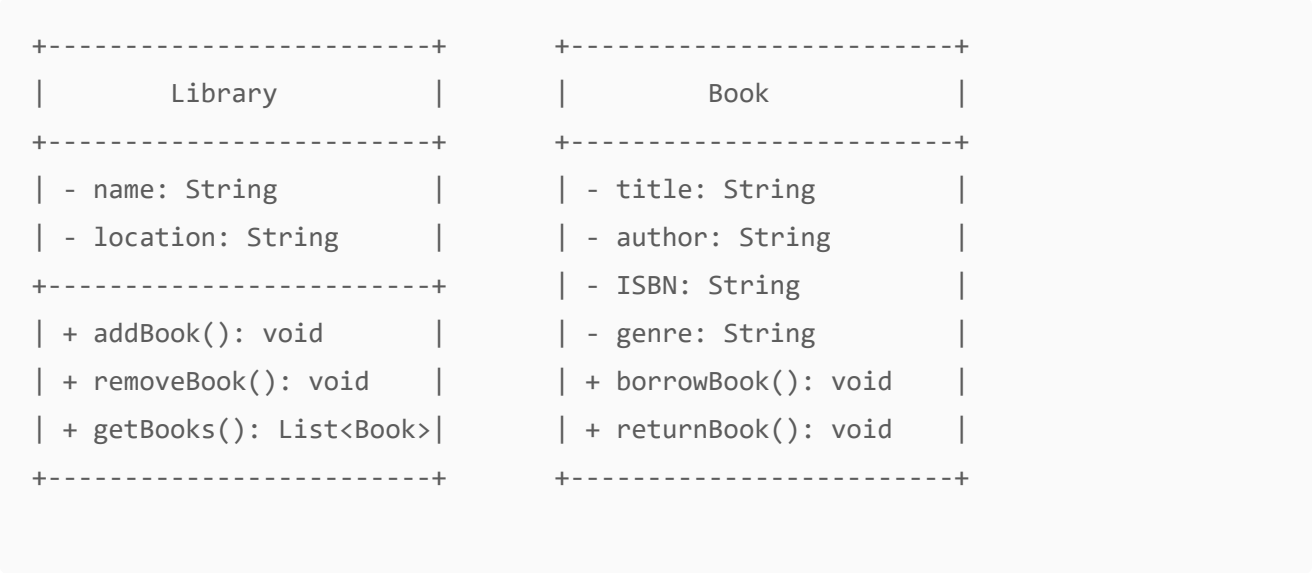
It has mainly 6 different component

1. **Classes :**
    - Represents a blueprint or template for creating objects. It typically includes the class name, attributes, and methods.

```
+-------------------------+
|      Car                |
+-------------------------+
| - make: String          |
| - model: String         |
| - year: int             |
+-------------------------+
| + start(): void         |
| + accelerate(): void    |
| + brake(): void         |
+-------------------------+
```

2. **Attributes:**
    - Represent the characteristics or properties of a class. They are typically listed under the class name and include their data types.
    - Example: `make: String`, `model: String`, `year: int` in the above class diagram.
3. **Methods:**
    - Represent the behaviors or operations that the class can perform. They are also listed under the class name and include their return types.
    - Example: `start(): void`, `accelerate(): void`, `brake(): void` in the above class diagram.
4. **Visibility Notation:**
    - Indicates the visibility or access level of attributes and methods. Common notations include `+` for public, `-` for private, and `#` for protected.
    - Example: `- make: String` (private attribute), `+ start(): void` (public method) in the above class diagram.
5. **Relationships:**

- Show how classes are related to each other. Common types of relationships include associations, aggregations, and compositions.
- Example: For a library and books relationship



In Above UML's, fields are basically the variables that are outside the methods which determine a certain value for the object of the class, this helps every object for having a unique nature and characteristics.

1.5 lambda functions in java (Syntax)

Read Syntax only, advanced java concept.

lambda expressions were introduced in Java 8 as a way to provide concise syntax for writing anonymous functions also called implementations of functional interfaces. The functional interface tells java that this class is having methods that can be referenced to the lambda function. as like in the example :

```
// defining the lambda function, what and how many parameters to take
@FunctionalInterface
interface LambdaExample {
        String methodExmpl(int a, int b);
}

public class TopicLambdaFunction {
        public static void main(String[] args) {
                // referencing the lambda function to the object
                LambdaExample exmp = (a,b) -> {
                        int result = a+b;
                        return "this is the result: " + result;
                };
                // using object to call upon desired method
                System.out.println(exmp.methodExmpl(4, 5));
        }
}
```

The lambda function syntax is as follows :

```
(a,b) -> {
        //function body
        return "something";
}
```

1.6 Compiling and Running Java Files

1. **Write Your Java Code:**
   - Create a new text file with a `.java` extension. For example, you can use a text editor or an Integrated Development Environment (IDE) like Eclipse, IntelliJ, or Visual Studio Code. Let's call the file `MyProgram.java`.

```
public class MyProgram {
        public static void main(String[] args) {
         System.out.println("Hello, World!");
        }
}
```

2. **Save the File:**
   - Save the file with the `.java` extension (e.g., `MyProgram.java`).
3. **Open a Terminal or Command Prompt:**
   - Open a terminal or command prompt window.
4. **Navigate to the Directory:**
   - Use the `cd` command to navigate to the directory where your Java file is located.
   `cd path/to/your/directory`
5. **Compile the Java File:**
   - Use the `javac` command to compile the Java file.
   - If there are no errors in your code, this will generate a file named `MyProgram.class`.
   `javac MyProgram.java`
6. **Run the Java Program:**
   - Use the `java` command to run the compiled program.
   `java MyProgram`

however in IDE's the execution can be changed because ide lets you to decide which JDK you want to use, instead of using `javac` and `java` command in ide the exact path of the JDK executable is used along with the exact path of the class files.

```
C:\Users\kushagra\.jdks\corretto-17.0.7\bin\java.exe "-
javaagent:D:\programs\IntelliJ IDEA
2023.2\lib\idea_rt.jar=64978:D:\programs\IntelliJ IDEA 2023.2\bin" -
Dfile.encoding=UTF-8 -classpath D:\learn-workspace\learn\design-
patterns\out\production\design-patterns Main
```

IDE runs the code like the above code.

1.7 Building and Running Packages

1. **Create Project Structure:**
   - for the following project structure, `MyApp.java` is in the package `com.example` which is having main method

```
Project
|-- src
|   `-- com
|       `-- example
|           `-- MyApp.java (main method)
|           |-- service
|               `-- ServiceForApp.java
```

2. **Navigate to Project Directory:**
   - Open a terminal or command prompt and navigate to your project directory (`MyProject` in this example).
   `cd path/to/Project`
3. **Compile Java Code:**
   - Compile your Java code using `javac`. Specify the source file (`MyApp.java`) and use the `-d` option to specify the destination directory for the compiled `.class` files.
   - This command compiles `MyApp.java` and places the compiled class file in the `out` directory.
   `javac .\com\example\*.java -d .\out\`
4. **Run Java Program:**
   - Navigate to the directory containing the compiled classes (`out` in this example) and use the `java` command to run your program.
   `java com.example.MyApp`

1.8 Class Path in java and its importance

the classpath is a parameter that tells the Java Virtual Machine (JVM) where to find user-defined classes and packages. It specifies the locations where Java should look for classes to load and execute.

classpath is crucial when working with Java, especially for compiling and running programs that involve multiple classes or external libraries.

1. **Classpath Elements:**
   - The classpath can include directories, JAR files (Java Archive files), and ZIP files. These elements contain compiled Java classes that your program needs to execute.
2. **Setting the Classpath:**
   - You can set the classpath using the `-cp` or `-classpath` option when running Java commands, if any application needs a certain class file to run
     `java -cp /path/to/classes MyApp`
3. **Classpath Order:**
   - The order of elements in the classpath matters. When searching for a class, Java looks in the directories and JAR files in the order they appear in the classpath. The first occurrence of the class found is used.
4. **Importance for Compiling:**
   - When you compile a Java program, the compiler needs to find classes and dependencies. The classpath tells the compiler where to look for these classes.
5. **Importance for Running:**
   - When you run a Java program, the JVM needs to locate and load classes during runtime. The classpath is crucial for the JVM to find the necessary classes and resources.