WHAT IS JAVA
==========
Java is an object-oriented, class-based, concurrent, secured and general-purpose computer-programming language. It is a widely used robust technology.

Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995.
James Gosling is known as the father of Java. Before Java, its name was Oak.
Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Platform: Any hardware or software environment in which a program runs, is known as a platform.
Since Java has a runtime environment (JRE) and API, it is called a platform.

Applications :

1.Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

JVM , JDK AND JRE IN JAVA
==========

JVM, JDK, and JRE are important components in the Java programming and runtime environment. Here's what each of them represents:

1. **JVM (Java Virtual Machine)**:

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist.

It is a specification that provides a runtime environment in which Java bytecode can be executed.

It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other.

However, Java is platform independent. There are three notions of the JVM: specification, implementation, and instance.

The JVM performs the following main tasks:
i. Loads code
ii. Verifies code
iii. Executes code
iv. Provides runtime environment

2. **JDK (Java Development Kit)**:

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets.

It physically exists. It contains JRE + development tools.

3. **JRE (Java Runtime Environment)**:

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment.

It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

JAVA OOPS:
==========

Object-Oriented Programming is a paradigm that provides many concepts, such as inheritance, data binding, polymorphism, etc.

Simula is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

Smalltalk is considered the first truly object-oriented programming language.

The popular object-oriented languages are Java, C#, PHP, Python, C++, etc.

1. **Classes and Objects**:

Class :

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
It is a logical entity. It can't be physical.A class in Java can contain:

Fields
Methods
Constructors

Object :

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc.

It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

State: represents the data (value) of an object.
Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.
Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

2. **Encapsulation**:

Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on that data into a single unit (a class).

Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which

is mixed of several medicines.

Encapsulation in java : We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The Java Bean class is the example of a fully encapsulated class.

It helps in data hiding, where the internal details of a class are hidden from the outside and can only be accessed through well-defined interfaces (methods).

Access modifiers like `private`, `protected`, and `public` control the visibility and accessibility of class members.

It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.

3. **Inheritance**:

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.

It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.

When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

The `extends` keyword is used in Java to create subclasses that inherit from a superclass.

The `implements` keyword is used in java to create subclass that inherit from a interface.

Uses : Runtime polymorphism can be achieved(Method Overriding) and Code Reusablity.

Types i. Single Inheritance 2. Mutlilevel Inheritance 3. Hierarchical Inheritance

Note : To reduce the complexity and simplify the language, multiple inheritance is not supported in java. It is possiable with interfaces.

4. **Polymorphism**:

Polymorphism in Java is a concept by which we can perform a single action in different ways

Polymorphism is achieved through method overriding and method overloading.

Method overloading(CompileTime Polymorphism) involves defining multiple methods with the same name in the same class, differing by the number or type of parameters.

Method overriding (RunTime Polymorphism) involves creating a method in a subclass with the same name as a method in the superclass, allowing the subclass to provide a specific implementation.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

5. **Abstraction**:
Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java:

i. Abstract class (0 to 100%)

ii.Interface (100%)

Abstract Class:

i. **Definition**:

- An abstract class is a class that can have both abstract (i.e., method without a body) and concrete (i.e., method with a body) methods.
- You can define fields (variables) and constructors in an abstract class.

ii. **Usage**:

- Abstract classes are used to provide a common base for multiple related classes.
- They can serve as a blueprint for subclasses, allowing code reuse.

iii. **Inheritance**:

- An abstract class can be extended (i.e., a subclass can inherit from it).
- A subclass must implement (override) all the abstract methods of its abstract superclass unless the subclass itself is declared as abstract.

iv. **Access Modifiers**:

- Abstract classes can have access modifiers for their fields and methods.

v. **Multiple Inheritance**:

- A Java class can extend only one abstract class, so Java supports single inheritance with abstract classes.

vi. **Constructor**:

- Abstract classes can have constructors, and these constructors can be called by the constructors of their subclasses.

Interface:

i. **Definition**:

- An interface is a completely abstract class, which means it can only contain abstract methods and constants (public static final fields).
- All methods declared in an interface are implicitly public and abstract, and all fields are implicitly public, static, and final.

ii. **Usage**:

- Interfaces are used to define contracts that classes must adhere to.
- They allow multiple unrelated classes to implement the same set of methods, promoting code interoperability.

iii. **Inheritance**:

- A class can implement multiple interfaces (i.e., a single class can adhere to multiple contracts), enabling multiple inheritance of types.

iv. **Access Modifiers**:

- All members (methods and constants) of an interface are implicitly public.

v. **Multiple Inheritance**:

- Java allows multiple inheritance through interfaces because a single class can implement multiple interfaces.

vi. **Constructor**:

- Interfaces do not have constructors. You cannot create an instance of an interface.

METHOD OVERLOADING AND METHOD OVERRIDING IN JAVA

==========

Method overloading and method overriding are two important concepts in object-oriented programming, particularly in Java. They both involve defining multiple methods with the same name, but they serve different purposes and have distinct characteristics. Let's explore each concept:

Method Overloading:

Method overloading, also known as compile-time polymorphism or static polymorphism, allows you to define multiple methods in a class with the same name but different parameters. The method signature, including the method name and the number or types of its parameters, must be different for overloaded methods. The compiler determines which method to call based on the number and types of arguments passed during a method invocation.

Key points about method overloading:

1. The return type of the method doesn't play a role in method overloading; it's not sufficient to differentiate overloaded methods.

2. Overloaded methods are invoked at compile time, based on the method's signature and the arguments passed.

3. Method overloading is a way to provide multiple ways to call a method with different argument lists for convenience and flexibility.

Here's an example of method overloading in Java:

```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public String add(String str1, String str2) {
        return str1 + str2;
    }
}
```

In the example above, the `add` method is overloaded with different parameter types.

Method Overriding:

Method overriding, also known as runtime polymorphism or dynamic polymorphism, allows a subclass to provide a specific implementation of a method that is already defined in its superclass.

The overridden method in the subclass has the same name, return type, and parameters as the method in the superclass.

Method overriding is used to implement the "is-a" relationship and allows you to define behavior specific to the subclass.

Key points about method overriding:

1. The return type, method name, and parameter list must be the same in the overriding method as in the overridden method.

2. The `@Override` annotation is commonly used to indicate that a method is intended to override a method in the superclass. While not strictly required, it helps catch errors at compile time.

3. Overridden methods are invoked at runtime based on the actual object type, allowing polymorphic behavior.

Here's an example of method overriding in Java:

```java
class Animal {
    void makeSound() {
        System.out.println("The animal makes a sound.");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Polymorphism
        animal.makeSound(); // Calls the makeSound method of the Dog class
    }
}
```

In this example, the `makeSound` method in the `Dog` class overrides the method with the same name in the `Animal` class.

In summary, method overloading allows multiple methods with the same name but different parameters within the same class, while method overriding allows a subclass to provide a specific implementation for a method inherited from its superclass.
Method overriding is a key feature of polymorphism in object-oriented programming.

COMPILE-TIME AND RUNTIME POLYMORPHISM

Compile-time polymorphism (also known as static polymorphism) and runtime polymorphism (also known as dynamic polymorphism) are two types of polymorphism in object-oriented programming languages like Java.

They occur when different methods with the same name are invoked but are determined at different stages of the program's lifecycle.

Compile-Time Polymorphism (Static Polymorphism):

1. **Method Overloading**: Compile-time polymorphism is mostly achieved through method overloading. It occurs when there are multiple methods with the same name in a class, but they have different parameter lists (i.e., a different number or types of parameters).

2. **Determined at Compile Time**: The method to be called is determined by the compiler based on the number and types of arguments passed during the method call.

3. **No Need for Inheritance**: Compile-time polymorphism can occur within a single class without the need for inheritance.

4. **Examples**: Method overloading is a common example of compile-time polymorphism. The specific method to be invoked is resolved during compilation.

Here's an example of compile-time polymorphism using method overloading:

```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }
}
```

Runtime Polymorphism (Dynamic Polymorphism):

1. **Method Overriding**: Runtime polymorphism is primarily achieved through method overriding. It occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

2. **Determined at Runtime**: The method to be called is determined at runtime based on the actual object type, allowing for polymorphic behavior.

3. **Involves Inheritance**: Runtime polymorphism typically involves inheritance, where a subclass extends a superclass and overrides one or more of its methods.

4. **Examples**: Method overriding is a common example of runtime polymorphism. The specific method to be invoked depends on the actual runtime type of the object.

Here's an example of runtime polymorphism using method overriding:

```java
class Animal {
    void makeSound() {
        System.out.println("The animal makes a sound.");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Polymorphism
        animal.makeSound(); // Calls the makeSound method of the Dog class
    }
}
```

In this example, the method to be invoked is determined at runtime based on the actual object type (`Dog`), demonstrating runtime polymorphism.

In summary, compile-time polymorphism is resolved by the compiler based on the method's signature, whereas runtime polymorphism is determined at runtime based on the actual object type, allowing for more dynamic and flexible behavior.

UPCASTING AND DOWNCASTING IN JAVA

Object TypeCasting

A process of converting one data type to another is known as Typecasting and Upcasting and Downcasting is the type of object typecasting.

In Java, the object can also be typecasted like the datatypes. Parent and Child objects are two types of objects.

So, there are two types of typecasting possible for an object, i.e., Parent to Child and Child to Parent or can say Upcasting and Downcasting.

Upcasting:

Upcasting is a type of object typecasting in which a child object is typecasted to a parent class object. By using the Upcasting, we can easily access the variables and methods of the parent class to the child class.

Here, we don't access all the variables and the method. We access only some specified variables and methods of the child class. Upcasting is also known as Generalization and Widening.

Key points about upcasting:

1. Upcasting is an implicit or automatic type conversion that is done by the compiler. No casting operator is needed.

2. It promotes code reusability and allows you to use a more general reference to an object of a specific subclass.

3. Upcasting allows you to access only the members (fields and methods) of the superclass from the reference.

Here's an example of upcasting in Java:

```java
class Animal {
	void makeSound() {
		System.out.println("The animal makes a sound.");
	}
}

class Dog extends Animal {
	void makeSound() {
		System.out.println("The dog barks.");
	}

	void fetch() {
		System.out.println("The dog fetches a ball.");
	}
}

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

```
        Animal myAnimal = new Dog(); // Upcasting
        myAnimal.makeSound(); // Calls the makeSound method of the Dog class
    }
}
```

In this example, the `Dog` object is upcast to the `Animal` reference, and you can access the overridden `makeSound` method.

Downcasting:

Downcasting is the opposite process of upcasting. It involves casting a reference from a superclass type to a subclass type.

Downcasting allows you to access specific members of the subclass that are not present in the superclass.

However, downcasting requires an explicit cast operator and is not always safe. It can lead to a `ClassCastException` if the actual object type is not compatible with the downcast.

Key points about downcasting:

1. Downcasting requires an explicit type cast operator, and it's subject to runtime checks to ensure type safety.

2. You can access members specific to the subclass after downcasting.

3. If the object is not an instance of the target subclass, a `ClassCastException` may occur.

Here's an example of downcasting in Java:

```java
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        if (myAnimal instanceof Dog) {
            Dog myDog = (Dog) myAnimal; // Downcasting
            myDog.fetch(); // Calls the fetch method of the Dog class
        }
    }
}
```

In this example, downcasting is performed after checking whether the object is an instance of the `Dog` class. If the check is successful, you can safely downcast and access the `fetch` method.

In Java, we rarely use Upcasting. We use it when we need to develop a code that deals with only the parent class. Downcasting is used when we need to develop a code that accesses behaviors of the child class.

Type casting in JAVA
==========

In Java, type casting is a method or process that converts a data type into another data type in both ways manually and automatically.
The automatic conversion is done by the compiler and manual conversion performed by the programmer.

Widening Type Casting :

Converting a lower data type into a higher one is called widening type casting. It is also known as implicit conversion or casting down. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

Both data types must be compatible with each other.

The target type must be larger than the source type.

```
public class WideningTypeCastingExample
{
	public static void main(String[] args)
	{
		int x = 7;
		//automatically converts the integer type into long type
		long y = x;
		//automatically converts the long type into float type
		float z = y;
		System.out.println("Before conversion, int value "+x);
		System.out.println("After conversion, long value "+y);
		System.out.println("After conversion, float value "+z);
	}
}
```

Narrowing Type Casting :

Converting a higher data type into a lower one is called narrowing type casting. It is also known as explicit conversion or casting up.

It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

```
public class NarrowingTypeCastingExample
{
	public static void main(String args[])
	{
		double d = 166.66;
		//converting double data type into long data type
		long l = (long)d;
		//converting long data type into int data type
		int i = (int)l;
		System.out.println("Before conversion: "+d);
		//fractional part lost
		System.out.println("After conversion into long type: "+l);
		//fractional part lost
		System.out.println("After conversion into int type: "+i);
	}
}
```

Wrapper classes in Java
==========

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java : Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc.

1. **Wrapper Classes**: In Java, there are eight wrapper classes, each corresponding to a primitive data type:

- `Byte`: Represents a byte value.
- `Short`: Represents a short value.
- `Integer`: Represents an int value.
- `Long`: Represents a long value.
- `Float`: Represents a float value.
- `Double`: Represents a double value.
- `Character`: Represents a char value.
- `Boolean`: Represents a boolean value.

2. **Autoboxing and Unboxing**:

Autoboxing : The automatic conversion of primitive data type into its corresponding wrapper class(Object) is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1{
    public static void main(String args[]){
        //Converting int into Integer
        int a=20;
        Integer i=Integer.valueOf(a);//converting int into Integer explicitly
        Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a)
internally

        System.out.println(a+" "+i+" "+j);
    }
}
```

Unboxing : The automatic conversion of wrapper type(Object) into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
public class WrapperExample2{
    public static void main(String args[]){
        //Converting Integer to int
        Integer a=new Integer(3);
        int i=a.intValue();//converting Integer to int explicitly
        int j=a;//unboxing, now compiler will write a.intValue() internally

        System.out.println(a+" "+i+" "+j);
    }
}
```

3. **Parsing Methods**: Wrapper classes provide methods for parsing strings and converting them to primitive data types. These methods are especially useful when you need to convert user input or data from files into the appropriate data types.

- `Integer.parseInt(String)`: Parses a string and returns an `int`.
- `Double.parseDouble(String)`: Parses a string and returns a `double`.
- `Boolean.parseBoolean(String)`: Parses a string and returns a `boolean`.

Example of parsing methods:

```java
String numberString = "123";
int number = Integer.parseInt(numberString); // Converts the string "123" to an int
```

Using wrapper classes, autoboxing, unboxing, and parsing methods, you can work with both primitive data types and objects more flexibly and conveniently in Java.

Java Arrays

Array:

1. Normally, an array is a collection of similar type of elements which has contiguous memory location.
2. Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location.
3. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
4. Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Java String

In Java, string is basically an object that represents sequence of char values. The java.lang.String class is used to create a string object. An array of characters works same as Java string. For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
```

is same as:

```
String s="javatpoint";
```

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements Serializable, Comparable and CharSequence interfaces.

How to create a string object?
There are two ways to create String object:

i. By string literal
ii. By new keyword

1) String Literal :

Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```

Java String

In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as the "string constant pool".

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

String s=new String("Welcome");//creates two objects and one reference variable.

EXCEPTION HANDLING
==========

Exception : Exception is an abnormal condition.In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception-Handling : The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

Exception are two types : 1. Checked Exception 2. Unchecked Exception.

1. Checked Exceptions (Compile-Time Exceptions):

The classes that directly inherit the Throwable class except RuntimeException are known as checked exceptions.

Checked exceptions are exceptions that are checked at compile time by the Java compiler.

This means that if a method can throw a checked exception, the programmer is required to handle it using `try-catch` blocks or declare it using the `throws` clause in the method signature.

Examples of checked exceptions include:

- `IOException`: Occurs when there are problems with input and output operations.
- `FileNotFoundException`: Occurs when an attempt is made to access a file that does not exist.
- `SQLException`: Occurs when there are database-related errors.

Checked exceptions are typically related to external factors and resources.

Example of handling a checked exception:

```java
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try {
            // Code that may throw a checked exception
        } catch (IOException e) {
            // Handle the exception here
        }
    }
}
```

2. Unchecked Exceptions (Runtime Exceptions):

The classes that inherit the RuntimeException are known as unchecked exceptions.

Unchecked exceptions, also known as runtime exceptions, are exceptions that are not checked at compile time. They typically represent programming errors, and it's not required to handle them explicitly using `try-catch` blocks.

Examples of unchecked exceptions include:
- `NullPointerException`: Occurs when trying to access an object or variable that is `null`.
- `ArithmeticException`: Occurs when there is an arithmetic error, such as division by zero.
- `ArrayIndexOutOfBoundsException`: Occurs when trying to access an array element with an index that is out of bounds.

Unchecked exceptions are usually the result of mistakes in the code, so the focus is on fixing the underlying issue rather than handling the exception.

Example of an unchecked exception:

```java
public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        int result = numbers[4]; // Throws ArrayIndexOutOfBoundsException
    }
}
```

"Throw" and "throws" are related concepts in Java used for exception handling, but they serve different purposes and are used in different contexts:

1. **`throw`**:

- **Definition**: `throw` is a keyword in Java used to manually throw an exception. When you use `throw`, you are explicitly raising an exception in your code, indicating that something exceptional has occurred.

- **Purpose**: It is used to throw exceptions, indicating that a specific error condition has been encountered. You can throw built-in exceptions or custom exceptions.

- **Example**:

```java
public void divide(int numerator, int denominator) {
    if (denominator == 0) {
        throw new ArithmeticException("Division by zero");
    }
    int result = numerator / denominator;
}
```

2. **`throws`**:

- **Definition**: `throws` is a modifier used in a method signature to declare that the method may throw one or more exceptions. It is used to specify which exceptions the method might throw, allowing the calling code to handle them.

- **Purpose**: It is used to declare that a method may throw exceptions, providing information to the caller about potential exceptions that need to be handled or propagated.

- **Example**:

```java
public int divide(int numerator, int denominator) throws ArithmeticException {
    if (denominator == 0) {
        throw new ArithmeticException("Division by zero");
    }
```

```
        return numerator / denominator;
    }
```

In the "throw" example, we manually throw an exception when a specific error condition is encountered. In the "throws" example, we declare in the method signature that the method may throw an exception, allowing the calling code to be aware of the potential exception and handle it.

It's important to note that "throw" is used within the method where the exception occurs, while "throws" is used in the method signature to declare exceptions that may be thrown by that method. These concepts work together to facilitate proper exception handling in Java.

INNER CLASS IN JAVA
==========

In Java, an inner class is a class defined within another class. Inner classes provide a way to logically group classes within another class, and they have several use cases and benefits, such as encapsulation and code organization. There are several types of inner classes in Java:

1. **Non-static Inner Class (Member Inner Class):**
   - A non-static inner class is a class that is defined inside another class without the `static` keyword.
   - It has access to the instance variables and methods of the outer class, including private members.
   - To create an instance of a non-static inner class, you typically need to create an instance of the outer class first.

   ```java
   public class OuterClass {
       private int outerField;

       public class InnerClass {
           public void display() {
               System.out.println("Outer field: " + outerField);
           }
       }
   }
   ```

2. **Static Inner Class (Nested Class):**
   - A static inner class is a class defined inside another class with the `static` keyword.
   - It does not have access to the instance variables and methods of the outer class directly, but it can access static members.
   - You can create an instance of a static inner class without creating an instance of the outer class.

   ```java
   public class OuterClass {
       private static int outerStaticField;

       public static class InnerClass {
           public void display() {
               System.out.println("Outer static field: " + outerStaticField);
           }
       }
   }
   ```

3. **Local Inner Class:**
   - A local inner class is defined inside a method or block (e.g., a code block within a method).
   - It has local scope and can access final or effectively final local variables from the enclosing method.

```java
public class OuterClass {
    public void display() {
        final int localVar = 42;

        class LocalInnerClass {
            public void printLocalVar() {
                System.out.println("Local variable: " + localVar);
            }
        }

        LocalInnerClass inner = new LocalInnerClass();
        inner.printLocalVar();
    }
}
```

4. **Anonymous Inner Class:**
   - An anonymous inner class is a class without a name defined inside a method's argument or within an expression.
   - It is often used to implement an interface or extend a class and override its methods.

```java
public class OuterClass {
    public void displayMessage() {
        new Thread(new Runnable() {
            public void run() {
                System.out.println("Anonymous inner class thread is running.");
            }
        }).start();
    }
}
```

Inner classes can be useful for implementing encapsulation, achieving better code organization, and enhancing code readability.

The choice of which type of inner class to use depends on the specific requirements of your program and the level of access needed to the outer class's members.

Functional Interface
==========

A functional interface in Java is an interface that has exactly one abstract method. Functional interfaces are also known as Single Abstract Method (SAM) interfaces. These interfaces are a fundamental concept in Java's support for lambda expressions and the Java Stream API. Functional interfaces are used to represent single behaviors, and they can be implemented as lambda expressions or method references, making code more concise and expressive.

Here are the key characteristics and rules for functional interfaces in Java:

1. **Single Abstract Method (SAM)**:
   - A functional interface should have exactly one abstract (unimplemented) method. This method is referred to as the "functional method" or "SAM method."
   - A functional interface can have additional default or static methods, but there must be only one abstract method.

2. **@FunctionalInterface Annotation**:
   - While it's not strictly required, you can use the `@FunctionalInterface` annotation to explicitly declare an interface

as a functional interface. This annotation helps provide clarity and allows the compiler to generate an error if there are multiple abstract methods.

3. **Lambda Expressions**:
   - You can use lambda expressions to create instances of functional interfaces. The lambda expression provides the implementation of the single abstract method defined in the interface.
   - Lambda expressions are a concise way to represent behavior or functionality.

Here's an example of a simple functional interface in Java:

```java
@FunctionalInterface
interface MyFunctionalInterface {
    void doSomething();
}

public class Main {
    public static void main(String[] args) {
        // Using a lambda expression to implement the functional interface
        MyFunctionalInterface functionalObj = () -> {
            System.out.println("Doing something...");
        };
        functionalObj.doSomething();
    }
}
```

In this example, `MyFunctionalInterface` is a functional interface with a single abstract method, `doSomething()`. We use a lambda expression to provide the implementation for the `doSomething()` method.

Functional interfaces are widely used in Java, especially in the context of the Stream API, where they enable concise and expressive code for operations on collections. Java provides several built-in functional interfaces in the `java.util.function` package, such as `Predicate`, `Function`, `Consumer`, and `Supplier`, to cover common use cases for lambda expressions and the Stream API.
These functional interfaces simplify the development of functional-style programming in Java.

Lambda Expressions
==========

Lambda expressions in Java provide a concise way to represent anonymous functions (functions that don't have a name) and are a part of the Java programming language since Java 8. Lambda expressions are primarily used for writing code more briefly and with less boilerplate code, making your code more readable and expressive.

Here's a breakdown of lambda expressions in Java:

Syntax:
Lambda expressions have the following syntax:

```
(parameters) -> expression
```

- `parameters`: This represents the input parameters to the lambda expression.
- `->`: It is the lambda operator, which separates the parameters from the expression.
- `expression`: This is the body of the lambda expression, where you specify what the lambda should do. This can be a single expression or a block of code enclosed in curly braces `{}`.

Usage:
Lambda expressions are often used in places where functional interfaces are expected. A functional interface is an

interface with a single abstract method. Java provides many built-in functional interfaces in the `java.util.function` package, such as `Predicate`, `Consumer`, and `Function`, which can be used with lambda expressions.

Here are some common examples of using lambda expressions:

1. **Functional Interfaces**:

   ```java
   // Using a lambda expression to define a Predicate
   Predicate<Integer> isEven = (num) -> num % 2 == 0;

   // Using a lambda expression with a Consumer
   List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
   names.forEach(name -> System.out.println("Hello, " + name));
   ```

2. **Threads**:

   ```java
   // Using a lambda expression to define a Runnable for a new thread
   Thread thread = new Thread(() -> {
       for (int i = 0; i < 10; i++) {
           System.out.println("Thread: " + i);
       }
   });
   thread.start();
   ```

3. **Sorting**:

   ```java
   List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
   names.sort((name1, name2) -> name1.compareTo(name2));
   ```

Lambda expressions are a powerful feature that simplifies code, especially when working with collections, event handling, and functional programming constructs. They allow you to define behavior directly where it's needed, reducing the need for writing separate classes or anonymous inner classes.

Java Streams
==========

Java Streams are a powerful and versatile feature introduced in Java 8 for processing sequences of elements (e.g., collections) in a functional and declarative manner. They provide a concise and expressive way to perform operations on data, making your code more readable and efficient.

Here are some key concepts and operations related to Java Streams:

1. **Stream Creation**:
   - You can create a Stream from various data sources, including collections, arrays, I/O channels, and more. For example:

     ```java
     List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
     Stream<Integer> stream = numbers.stream();
     ```

2. **Intermediate and Terminal Operations**:

- Streams consist of two types of operations: intermediate and terminal.
- Intermediate operations transform a Stream into another Stream. Examples include `filter`, `map`, and `sorted`.
- Terminal operations produce a result or a side effect. Examples include `forEach`, `collect`, and `reduce`.

3. **Filtering**:
   - You can use the `filter` method to selectively include elements in a Stream based on a given condition.

4. **Mapping**:
   - The `map` method allows you to transform elements in a Stream to another type using a provided function.

5. **Sorting**:
   - The `sorted` method orders the elements in a Stream based on a natural or custom order.

6. **Reduction**:
   - The `reduce` operation combines elements in a Stream into a single result using a specified binary operator.

7. **Collecting**:
   - The `collect` method is used to accumulate the elements of a Stream into a collection, such as a List or Set.

8. **ForEach**:
   - The `forEach` operation performs an action on each element in the Stream.

9. **Grouping and Partitioning**:
   - Streams support operations like `groupingBy` and `partitioningBy` for grouping elements into maps or partitions.

10. **Infinite Streams**:
    - Streams can be infinite. You can create infinite Streams using operations like `generate` and `iterate`.

11. **Parallel Processing**:
    - Streams support parallel processing, allowing for efficient use of multi-core processors. You can use the `parallel` and `parallelStream` methods to enable parallel execution.

12. **Lazy Evaluation**:
    - Streams are lazily evaluated, which means intermediate operations are only executed when a terminal operation is called. This allows for efficient processing of large data sets.

Here's an example of a simple Java Stream pipeline that filters and transforms a list of integers:

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sumOfSquares = numbers.stream()
        .filter(n -> n % 2 == 0) // Filter even numbers
        .map(n -> n * n)         // Square each number
        .reduce(0, Integer::sum);  // Sum the squared numbers
```

This example demonstrates how you can use Stream operations to perform data processing in a concise and readable way. Streams are a fundamental part of modern Java programming and are widely used in a variety of applications.

JDBC IN JAVA

JDBC, which stands for Java Database Connectivity, is a Java-based API (Application Programming Interface) that allows Java applications to interact with relational databases. It provides a standard interface for connecting to databases, executing SQL queries, and retrieving and manipulating data. JDBC is a crucial technology for database-driven Java applications and is part of the Java Standard Library.

Here are some key points about JDBC:

1. **Database Connectivity**: JDBC enables Java applications to establish connections to relational databases like MySQL, Oracle, PostgreSQL, SQL Server, and others. It provides a way for Java code to communicate with databases through drivers.

2. **Driver Types**:
   - JDBC drivers are used to establish a connection between the Java application and the database. There are four types of JDBC drivers:
     - Type-1 (JDBC-ODBC bridge)
     - Type-2 (Native-API driver)
     - Type-3 (Network Protocol driver)
     - Type-4 (Thin driver, also known as a native-protocol driver)
   - The most common and recommended driver type is Type-4 (Thin driver) because it doesn't require a native database client.

3. **Database Operations**: JDBC allows you to perform various database operations, including executing SQL queries, retrieving and updating data, and managing transactions.

4. **Basic Steps**:
   - To use JDBC, you typically follow these basic steps:
     1. Load the JDBC driver (if not using the Type-4 driver, you need to load the appropriate driver class).
     2. Establish a database connection using a connection URL, username, and password.
     3. Create a statement or prepared statement for executing SQL queries.
     4. Execute SQL queries to retrieve, insert, update, or delete data.
     5. Process the results, if any, from the database.
     6. Close the database connection when done.

5. **Exception Handling**: JDBC methods can throw various exceptions, including `SQLException`. Proper exception handling is important to deal with potential errors during database operations.

6. **Batch Processing**: JDBC supports batch processing, which allows you to execute multiple SQL statements in a single batch, reducing the overhead of repeated database communication.

7. **Transaction Management**: You can use JDBC to manage database transactions, including committing or rolling back changes to ensure data consistency.

8. **Connection Pooling**: In real-world applications, connection pooling is often used to efficiently manage database connections and improve performance.

Here's a basic example of how to use JDBC to connect to a database and execute a simple query in Java:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class JDBCDemo {
    public static void main(String[] args) {
        String jdbcURL = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "yourUsername";
        String password = "yourPassword";

        try {
            // Load the JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Establish a connection to the database
```

```
                Connection connection = DriverManager.getConnection(jdbcURL, username, password);

                // Create a statement
                Statement statement = connection.createStatement();

                // Execute a query
                String sqlQuery = "SELECT * FROM employees";
                ResultSet resultSet = statement.executeQuery(sqlQuery);

                // Process the results
                while (resultSet.next()) {
                    System.out.println("Name: " + resultSet.getString("name"));
                }

                // Close resources
                resultSet.close();
                statement.close();
                connection.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

JDBC provides a powerful and standardized way for Java applications to interact with databases, making it a fundamental technology for database-driven Java applications and enterprise systems.

MULTITHREADING

Thread : A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.
Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

Multithreading : Multithreading in java is process of executing multiple threads simultaneously.
Multiprocessing and multithreading, both are used to achieve multitasking.
However, we use multithreading than multiprocessing because threads use a shared memory area.
They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
Java Multithreading is mostly used in games, animation, etc.

1) Process-based Multitasking (Multiprocessing) :
Each process has an address in memory. In other words, each process allocates a separate memory area.
A process is heavyweight.
Cost of communication between the process is high.
Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading) :
Threads share the same address space.
A thread is lightweight.

Cost of communication between the thread is low.

Java Thread class : Java provides Thread class to achieve thread programming.
Thread class provides constructors and methods to create and perform operations on a thread.
Thread class extends Object class and implements Runnable interface.

Life cycle of a Thread (Thread States)
In Java, a thread always exists in any one of the following states. These states are:
i. New
ii. Active
iii. Blocked / Waiting
iv. Timed Waiting
v. Terminated

Collections in Java:
==========
The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
Java Collection means a single unit of objects.
Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Collection : is used to represent a single unit with a group of individual objects.

Collections : is used to operate on collection with several utility methods.

Iterable Interface
==========
The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
It contains only one abstract method. i.e.,

Iterator interface
==========
Iterator interface provides the facility of iterating the elements in a forward direction only.
1 public boolean hasNext() It returns true if the iterator has more elements otherwise it returns false.
2 public Object next() It returns the element and moves the cursor pointer to the next element.
3 public void remove() It removes the last elements returned by the iterator. It is less used.

There are various ways to traverse the collection elements:

-> By Iterator interface.
-> By for-each loop.

-> By ListIterator interface.
-> By for loop.
-> By forEach() method.
-> By forEachRemaining() method.

1. List Interface
==========
List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

List <data-type> list1= new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
List <data-type> list4 = new Stack();

i. ArrayList:
==========
1) ArrayList internally uses a dynamic array to store the elements.
2) Manipulation with ArrayList is slow because it internally uses an array. I
3) An ArrayList class can act as a list only because it implements List only.
4) ArrayList is better for storing and accessing data.
5) The memory location for the elements of an ArrayList is contiguous.
6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.
7) To be precise, an ArrayList is a resizable array.

--> ArrayList and LinkedList both implement the List interface and maintain insertion order. Both are non-synchronized classes.

ii. LinkedList
==========
1) LinkedList internally uses a doubly linked list to store the elements.
2) If any element is removed from the array, all the other elements are shifted in memory. Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) LinkedList is better for manipulating data.
5) The location for the elements of a linked list is not contagious.
6) There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.
7) LinkedList implements the doubly linked list of the list interface.

iii.Vector
==========

1) Vector is like the dynamic array which can grow or shrink its size.
2) Unlike array, we can store n-number of elements in it as there is no size limit.
3) It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the List interface, so we can use all the methods of List interface here.
4) It is recommended to use the Vector class in the thread-safe implementation only.
5) If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.
6) The Iterators returned by the Vector class are fail-fast.
7) In case of concurrent modification, it fails and throws the ConcurrentModificationException.

It is similar to the ArrayList, but with two differences-
a. Vector is synchronized.

b. Java Vector contains many legacy methods that are not the part of a collections' framework.

iv. Stack

==========

1) The stack is a linear data structure that is used to store the collection of objects.

2) It is based on Last-In-First-Out (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects.

3) One of them is the Stack class that provides different operations such as push, pop, search, etc.

4) In this section, we will discuss the Java Stack class, its methods, and implement the stack data structure in a Java program.

5) The stack data structure has the two most important operations that are push and pop.

6) The push operation inserts an element into the stack and pop operation removes an element from the top of the stack.

2. Queue Interface

==========

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed.

There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated.

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

i. PriorityQueue

==========

1) PriorityQueue is also class that is defined in the collection framework that gives us a way for processing the objects on the basis of priority.

2) It is already described that the insertion and deletion of objects follows FIFO pattern in the Java queue.

3) However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.

3. Deque Interface

==========

The interface called Deque is present in java.util package.

It is the subtype of the interface queue. The Deque supports the addition as well as the removal of elements from both ends of the data structure.

Therefore, a deque can be used as a stack or a queue.

We know that the stack supports the Last In First Out (LIFO) operation, and the operation First In First Out is supported by a queue.

As a deque supports both, either of the mentioned operations can be performed on it. Deque is an acronym for "double ended queue".

i. ArrayDeque class

==========

1) We know that it is not possible to create an object of an interface in Java.

2) Therefore, for instantiation, we need a class that implements the Deque interface, and that class is ArrayDeque.

3) It grows and shrinks as per usage. It also inherits the AbstractCollection class.

4) The important points about ArrayDeque class are:

a) Unlike Queue, we can add or remove elements from both sides.

b) Null elements are not allowed in the ArrayDeque.

c) ArrayDeque is not thread safe, in the absence of external synchronization.

d) ArrayDeque has no capacity restrictions.

e) ArrayDeque is faster than LinkedList and Stack.

4. Set Interface

Set Interface in Java is present in java.util package.
It extends the Collection interface.
It represents the unordered set of elements which doesn't allow us to store the duplicate items.
We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

i. HashSet Class

1) Java HashSet class is used to create a collection that uses a hash table for storage.
2) It inherits the AbstractSet class and implements Set interface.
3) The important points about Java HashSet class are:
    a) HashSet stores the elements by using a mechanism called hashing.
    b) HashSet contains unique elements only.
    c) HashSet allows null value.
    d) HashSet class is non synchronized.
    e) HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
    f) HashSet is the best approach for search operations.
    g) The initial default capacity of HashSet is 16, and the load factor is 0.75.

ii. LinkedHashSet Class

1) Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface.
2) It inherits the HashSet class and implements the Set interface.
3) The important points about the Java LinkedHashSet class are:
    a) Java LinkedHashSet class contains unique elements only like HashSet.
    b) Java LinkedHashSet class provides all optional set operations and permits null elements.
    c) Java LinkedHashSet class is non-synchronized.
    d) Java LinkedHashSet class maintains insertion order.

iii. TreeSet Class

1) Java TreeSet class implements the Set interface that uses a tree for storage.
2) It inherits AbstractSet class and implements the NavigableSet interface.
3) The objects of the TreeSet class are stored in ascending order.
4) The important points about the Java TreeSet class are:
    a) Java TreeSet class contains unique elements only like HashSet.
    b) Java TreeSet class access and retrieval times are quiet fast.
    c) Java TreeSet class doesn't allow null element.
    d) Java TreeSet class is non synchronized.
    e) Java TreeSet class maintains ascending order.
    f) Java TreeSet class contains unique elements only like HashSet.
    g) Java TreeSet class access and retrieval times are quite fast.
    h) Java TreeSet class doesn't allow null elements.
    i) Java TreeSet class is non-synchronized.
    j) Java TreeSet class maintains ascending order.
    k) The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.

5 Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry.

A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap.

A Map can't be traversed, so you need to convert it into Set using keySet() or entrySet() method.

Class	Description
HashMap	is the implementation of Map, but it doesn't maintain any order.
LinkedHashMap	is the implementation of Map. It inherits HashMap class. It maintains insertion order.
TreeMap	is the implementation of Map and SortedMap. It maintains ascending order.

i. HashMap

1) Java HashMap class hierarchy

2) Java HashMap class implements the Map interface which allows us to store key and value pair, where keys should be unique.

3) If you try to insert the duplicate key, it will replace the element of the corresponding key.

4) It is easy to perform operations using the key index like updation, deletion, etc.

5) HashMap class is found in the java.util package.

6) HashMap in Java is like the legacy Hashtable class, but it is not synchronized.

7) It allows us to store the null elements as well, but there should be only one null key.

8) Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value.

9) It inherits the AbstractMap class and implements the Map interface.

a) Java HashMap contains values based on the key.

b) Java HashMap contains only unique keys.

c) Java HashMap may have one null key and multiple null values.

d) Java HashMap is non synchronized.

e) Java HashMap maintains no order.

f) The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

ii. LinkedHashMap class

1) Java LinkedHashMap class hierarchy

2) Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

3) Points to remember:

a) Java LinkedHashMap contains values based on the key.

b) Java LinkedHashMap contains unique elements.

c) Java LinkedHashMap may have one null key and multiple null values.

d) Java LinkedHashMap is non synchronized.

e) Java LinkedHashMap maintains insertion order.

f) The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>

6) TreeMap class
==========

Java TreeMap class hierarchy: TreeMap class implements SortedMap interface and SortedMap interface extends Map interface.

Java TreeMap class is a red-black tree based implementation.

It provides an efficient means of storing key-value pairs in sorted order.

The important points about Java TreeMap class are:

Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.

Java TreeMap contains only unique elements.

Java TreeMap cannot have a null key but can have multiple null values.

Java TreeMap is non synchronized.

Java TreeMap maintains ascending order.

Singleton Class:
==========

Explanation:
A Singleton class is a design pattern that restricts the instantiation of a class to a single instance and provides a global point of access to that instance. It ensures that there is only one instance of the class in the application, and it provides a mechanism to access that instance.

Uses:
- *Resource Management:* When you want to control access to a shared resource, such as a database connection or a configuration manager.
- *Logging:* In scenarios where a single logging instance needs to manage logs across the application.
- *Caching:* To maintain a single cache manager instance throughout the application.

Example Code:
Certainly! Here's an example of a singleton class for resource management in MySQL and logging:

java
```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.logging.Logger;

public class ResourceManager {
    private static ResourceManager instance;
    private Connection databaseConnection;
    private Logger logger;

    private ResourceManager() {
        // Private constructor to prevent instantiation from outside the class.
        initializeDatabaseConnection();
        initializeLogger();
    }

    public static synchronized ResourceManager getInstance() {
        if (instance == null) {
            instance = new ResourceManager();
        }
        return instance;
    }

    private void initializeDatabaseConnection() {
        // Initialize your MySQL database connection here.
```

```
        String url = "jdbc:mysql://localhost:3306/your_database";
        String user = "your_username";
        String password = "your_password";

        try {
            databaseConnection = DriverManager.getConnection(url, user, password);
        } catch (SQLException e) {
            e.printStackTrace();
            // Log the exception using the logger.
            logger.severe("Failed to initialize database connection: " + e.getMessage());
        }
    }

    private void initializeLogger() {
        // Initialize your logger here.
        logger = Logger.getLogger(ResourceManager.class.getName());
    }

    public Connection getDatabaseConnection() {
        return databaseConnection;
    }

    public void performDatabaseOperation() {
        // Example method for performing a database operation.
        // Use this method in other classes to interact with the database.
        // ...
    }

    public void logMessage(String message) {
        // Example method for logging messages.
        // Use this method in other classes to log messages.
        logger.info(message);
    }
}
```

In this example, the ResourceManager class is a singleton responsible for managing the MySQL database connection and logging. Other classes can obtain an instance of this class using the getInstance() method. They can then use methods like getDatabaseConnection() to get the database connection or logMessage() to log messages.

```
public class DatabaseOperationClass {
    public static void main(String[] args) {
        // Get an instance of ResourceManager.
        ResourceManager resourceManager = ResourceManager.getInstance();

        // Use the database connection obtained from ResourceManager.
        Connection databaseConnection = resourceManager.getDatabaseConnection();

        // Perform a database operation using the obtained connection.
        try (Statement statement = databaseConnection.createStatement()) {
            ResultSet resultSet = statement.executeQuery("SELECT * FROM your_table");

            // Process the resultSet or perform other database operations...

        } catch (SQLException e) {
            e.printStackTrace();
            // Log the exception using ResourceManager's logger.
            resourceManager.logMessage("Error in DatabaseOperationClass: " + e.getMessage());
        }
    }
```

```
}
```

In this example:

1. The ResourceManager instance is obtained using ResourceManager.getInstance().
2. The database connection is obtained through resourceManager.getDatabaseConnection().
3. A database operation is performed using the obtained connection. Any exceptions are logged using resourceManager.logMessage().

Similarly, other classes can use the ResourceManager singleton to log messages:

```
java
public class LoggingClass {
    public static void main(String[] args) {
        // Get an instance of ResourceManager.
        ResourceManager resourceManager = ResourceManager.getInstance();

        // Log a message using ResourceManager's logger.
        resourceManager.logMessage("This is a log message from LoggingClass.");
    }
}
```

This way, the ResourceManager encapsulates the details of resource management and logging, providing a centralized and consistent way for other classes to access these functionalities.

Please note that this is a simplified example, and you may need to adapt it based on your specific requirements and the details of your MySQL setup. Additionally, consider using connection pooling for more efficient resource management in a production environment

Object Cloning:
==========
Explanation:
Object cloning is the process of creating an exact copy of an object. In Java, the Cloneable interface is used, and the clone() method is overridden to achieve object cloning. Cloning can be shallow or deep, depending on whether it creates copies of referenced objects.

Uses:
- *Prototype Pattern:* Creating new objects by copying an existing object (prototype) rather than creating them from scratch.
- *Immutable Classes:* Cloning is often used in classes where you want to create copies without risking modification of the original instance.
- *Copying Collections:* Creating independent copies of collections to avoid shared references.

Example Code:
```
java
// Person.java
public class Person implements Cloneable {
    private String name;
    private int age;

    // Constructors, getters, setters...

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
```

```
    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

Usage:

```
java
// CloningUsage.java
public class CloningUsage {
    public static void main(String[] args) throws CloneNotSupportedException {
        Person person1 = new Person("John", 30);
        Person person2 = (Person) person1.clone();

        System.out.println("Original: " + person1);
        System.out.println("Cloned: " + person2);
    }
}
```

In the Person class example, the clone method creates a shallow copy. If your class contains references to other objects, and you need to create a deep copy, you may need to override the clone method accordingly.

Remember to handle CloneNotSupportedException or use the Cloneable interface carefully, considering alternatives like copy constructors or serialization based on your specific requirements.

Note: The Cloneable interface is a marker interface, and it's recommended to override the clone method when implementing it.

Remember that cloning has some complexities, especially with deep cloning and handling mutable objects within the cloned object. It's often considered in conjunction with the clone method, and sometimes custom serialization and deserialization approaches are preferred.

IMPORTANTS
==========
Using List<Student> students = new ArrayList<>(); instead of ArrayList<Student> students = new ArrayList<>(); provides more flexibility and follows the principle of programming to interfaces. Here are the reasons and benefits:

1. *Flexibility and Abstraction:*
   - Using List<Student> allows you to switch the implementation later without changing the rest of your code. You can easily switch to a different List implementation like LinkedList or any other class that implements the List interface.

   ```
   java
   List<Student> students = new LinkedList<>();
   ```

2. *Programming to Interfaces:*
   - It's a good practice to program to interfaces or superclasses rather than concrete classes. This promotes flexibility and helps in adhering to the Dependency Inversion Principle of SOLID.

```
java
// Preferred way - Programming to the interface
List<Student> students = new ArrayList<>();

// Avoid - Concrete implementation
ArrayList<Student> students = new ArrayList<>();
```

3. *Easier Testing:*
   - When using interfaces, it's easier to mock objects for testing, leading to more effective unit testing.

Examples:

Programming to Interface:

```
java
// Programming to Interface
List<Student> students = new ArrayList<>();

students.add(new Student("John"));
students.add(new Student("Alice"));

for (Student student : students) {
    System.out.println(student.getName());
}
```

Programming to Concrete Class:

```
java
// Avoid - Programming to Concrete Class
ArrayList<Student> students = new ArrayList<>();

students.add(new Student("John"));
students.add(new Student("Alice"));

for (Student student : students) {
    System.out.println(student.getName());
}
```

In the first example, if you decide later to change the ArrayList to a different implementation, you only need to change the right side of the assignment. In the second example, if you used ArrayList throughout your code and later decided to switch to LinkedList, you would need to modify every line where you instantiated or used the ArrayList, which can be error-prone and time-consuming.

Programming to interfaces allows for more maintainable, modular, and testable code. It is a key principle in object-oriented design, promoting loose coupling between components and making your code more adaptable to changes.

Certainly! Let's dive into explanations of Spring Boot Security and OAuth, along with their uses, advantages, disadvantages, and some examples.

Spring Boot Security:
==========
Explanation:
Spring Boot Security is a powerful authentication and access control framework for Java applications.
It's built on top of the Spring Security framework and simplifies the configuration and integration of security features into Spring Boot applications.

Uses:
- *User Authentication and Authorization:* Secure your application by authenticating users and controlling access to different resources.
- *Form-based Login:* Easily set up login pages with minimal configuration.
- *Method-Level Security:* Restrict access to specific methods or services based on user roles.

Advantages:
- *Ease of Integration:* Seamless integration with Spring Boot applications.
- *Customization:* Highly customizable to suit various authentication and authorization requirements.
- *Community Support:* Extensive community support and documentation.

Disadvantages:
- *Learning Curve:* It might have a learning curve for beginners, especially when dealing with complex configurations.
- *Overhead:* In some cases, the comprehensive features might introduce more complexity than needed for simple applications.

Example Code:
Here's a basic example of a Spring Boot Security configuration class:

java
```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/public/**").permitAll()
                .anyRequest().authenticated()
                .and()
            .formLogin();
    }
}
```

OAuth 2.0:
==========
Explanation:
OAuth 2.0 is an open standard for authorization that enables secure access to resources on behalf of a user without exposing their credentials.
It's commonly used for third-party authentication and authorization.

Uses:
- *Third-Party Authentication:* Allows users to log in using credentials from another platform (e.g., Google, Facebook).
- *Secure API Access:* Enables secure access to APIs on behalf of a user without exposing their credentials.
- *Single Sign-On (SSO):* Users can authenticate once and access multiple applications.

Advantages:
- *Secure Authorization:* Enhances security by not exposing user credentials to third-party applications.
- *Single Sign-On:* Simplifies user experience by allowing a single login for multiple services.
- *Standardization:* A widely adopted standard with broad industry support.

Disadvantages:
- *Complexity:* Implementing OAuth 2.0 can be complex, especially for beginners.
- *Token Management:* Requires careful handling of access tokens to prevent security issues.

Example Code:
For a Spring Boot OAuth 2.0 example, consider using Spring Security's OAuth 2.0 client. Here's a simplified example:

properties

```
# application.properties
spring.security.oauth2.client.registration.google.client-id=<your-client-id>
spring.security.oauth2.client.registration.google.client-secret=<your-client-secret>
```

```
java
@SpringBootApplication
@EnableOAuth2Client
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

This example configures Google as an OAuth 2.0 provider for authentication.

These are simplified examples, and real-world implementations may involve additional configurations and considerations. Always refer to official documentation for comprehensive details and best practices.

Certainly! I can guide you through a brief tutorial on Spring Boot Security and OAuth 2.0. Keep in mind that this is a high-level overview, and you may want to refer to official documentation or more detailed tutorials for a comprehensive understanding.

Spring Boot Security Tutorial:

1. *Dependencies:*
   - Add the following dependencies to your pom.xml or build.gradle:
     ```
     xml
     <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-security</artifactId>
     </dependency>
     ```

2. *Security Configuration:*
   - Create a class extending WebSecurityConfigurerAdapter to customize security configurations.
   ```
   java
   @EnableWebSecurity
   public class SecurityConfig extends WebSecurityConfigurerAdapter {
       @Override
       protected void configure(HttpSecurity http) throws Exception {
           http.authorizeRequests().antMatchers("/public/**").permitAll()
               .anyRequest().authenticated().and().formLogin();
       }
   }
   ```

3. *User Authentication:*
   - Configure in-memory authentication or use a custom user service:
   ```
   java
   @Autowired
   public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
       auth.inMemoryAuthentication()
           .withUser("user").password("{noop}password").roles("USER");
   }
   ```

OAuth 2.0 Tutorial:
==========
1. *Add Dependencies:*
   - Include the following dependencies:
     xml
     <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-oauth2-client</artifactId>
     </dependency>

2. *Configure OAuth 2.0:*
   - In application.properties or application.yml, specify OAuth 2.0 settings:
     properties
     spring.security.oauth2.client.registration.google.client-id=<your-client-id>
     spring.security.oauth2.client.registration.google.client-secret=<your-client-secret>

3. *Secure Endpoints:*
   - Use @EnableOAuth2Client on your @SpringBootApplication class.
   - Secure specific endpoints using @PreAuthorize or configure in SecurityConfig.

4. *Customize Login Page:*
   - Customize the login page by providing your own HTML template in resources/templates/login.html.

Remember to replace placeholder values with your actual configurations. This is a simplified guide, and you should refer to the official Spring Boot and OAuth 2.0 documentation for detailed explanations and best practices.

OAuth 2.0
=========
OAuth 2.0, which stands for "Open Authorization", is a standard designed to allow
a website or application to access resources hosted by other web apps on behalf of a user.

* OAuth 2.0 is an authorization protocol and NOT an authentication protocol.
* It is designed primarily as a means of granting access to a set of resources, for example, remote APIs or user's data.
* OAuth 2.0 uses Access Tokens.

OAuth 2.0 Roles
===============
* Resource Owner
* Client
* Authorization Server
* Resource Server

Resource Owner : The user or system that owns the protected resources and can grant access to them.

Client : The client is the system that requires access to the protected resources.
To access resources, the Client must hold the appropriate Access Token.

Authorization Server : This server receives requests from the Client for Access Token
and issues them upon successful authentication and consent by the Resource Owner.

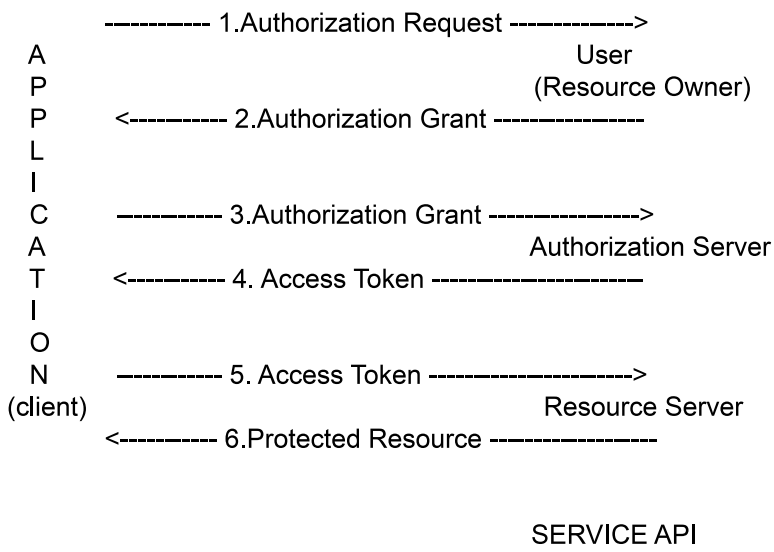
Resource Server : A server that protects the user's resource an receives access requests from the Client.
It accepts and validates an Access Token from the Client and returns the appropriate
resources to it.

OAuth 2.0 Scopes
================

Scopes are an important concept in OAuth 2.0.

They are used to specify exactly the reason for which access to resource may be granted.

Abstract Protocol Flow
==========



SERVICE API

HIBERNATE
==========

Hibernate is an open-source Java framework that provides an object-relational mapping (ORM) solution. It simplifies database access for Java applications by mapping Java objects to database tables, enabling developers to work with databases using Java objects rather than writing raw SQL queries. Here are some key aspects of Hibernate:

Features of Hibernate:

1. **Object-Relational Mapping (ORM)**: Hibernate allows you to map Java objects to database tables, making it easier to work with relational databases.

2. **Database Independence**: Hibernate abstracts the underlying database, allowing you to write database-agnostic code. You can switch between different databases without changing your application's code.

3. **Automatic Table Generation**: Hibernate can automatically generate database tables based on your Java entities.

4. **Caching**: Hibernate provides caching mechanisms to improve performance, reducing the number of database queries.

5. **Query Language (HQL)**: Hibernate offers its query language called HQL (Hibernate Query Language) for querying the database. HQL is similar to SQL but operates on Java objects.

6. **Lazy Loading**: Hibernate supports lazy loading, which means that it loads related data from the database only when it is explicitly requested.

7. **Transaction Management**: It provides built-in support for managing database transactions.

8. **Association Mapping**: Hibernate supports various types of associations between entities, such as one-to-one, one-to-many, and many-to-many.

Differences between Hibernate and JDBC:

1. **Abstraction Level**: Hibernate provides a higher-level abstraction for database access, while JDBC is a lower-level API for interacting with databases. With Hibernate, you work with Java objects, whereas with JDBC, you write SQL queries and work with result sets directly.

2. **SQL vs. HQL**: With JDBC, you write SQL queries explicitly. In Hibernate, you can use HQL, which is a higher-level query language that operates on Java objects.

3. **Mapping**: In Hibernate, you define entity classes to map to database tables, while in JDBC, you need to write code to map Java objects to SQL and vice versa.

4. **Portability**: Hibernate provides database independence, allowing you to switch between different databases with minimal code changes. JDBC code may be tightly coupled to a specific database, making it less portable.

Examples:

JDBC Example:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class JDBCDemo {
    public static void main(String[] args) {
        try {
            // Load the JDBC driver
            Class.forName("com.mysql.jdbc.Driver");

            // Establish a connection to the database
            Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "username",
"password");

            // Create a SQL statement
            Statement statement = connection.createStatement();

            // Execute a query
            ResultSet resultSet = statement.executeQuery("SELECT * FROM employees");

            // Process the result set
            while (resultSet.next()) {
                System.out.println(resultSet.getString("name"));
            }

            // Close resources
            resultSet.close();
            statement.close();
            connection.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Hibernate Example:

```java
@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

    // Getters and setters
}

public class HibernateDemo {
    public static void main(String[] args) {
        SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();

        // Create a new employee
        Employee employee = new Employee();
        employee.setName("John Doe");

        // Save the employee to the database
        session.beginTransaction();
        session.save(employee);
        session.getTransaction().commit();

        // Retrieve an employee
        Employee retrievedEmployee = session.get(Employee.class, 1);
        System.out.println(retrievedEmployee.getName());

        session.close();
        sessionFactory.close();
    }
}
```

In the Hibernate example, you define an `Employee` entity, and Hibernate takes care of the database interaction for you. This code is more abstract and portable compared to the low-level JDBC code.

Dependency Injection in Spring

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application.

Dependency Injection makes our programming code loosely coupled.

Two ways to perform Dependency Injection in Spring framework

Spring framework provides two ways to inject dependency

1. By Constructor
2. By Setter method

Dependency Injection (DI) is a software design pattern that promotes the separation of concerns in an application by allowing components to request their dependencies rather than creating them directly. In DI, a component's dependencies are "injected" into it from the outside, typically via constructor parameters or setter methods. This approach decouples components, making them more modular, reusable, and easier to test.

The key concepts of Dependency Injection are as follows:

Dependency: A dependency is an external component or service that a class or function relies on to perform its tasks. These can include database connections, configuration settings, logging services, and more.

Injection: Injection refers to the process of providing a component's dependencies from the outside, typically through constructor arguments or setter methods. Dependencies are "injected" into the component rather than being created or managed by the component itself.

Without Dependency Injection:

In this example, the Car class creates its own Engine:

```
class Engine {
    public void start() {
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine;

    public Car() {
        engine = new Engine(); // Creating the engine internally
    }

    public void start() {
        engine.start();
        System.out.println("Car started");
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car(); // Car creates its own Engine
        car.start();
    }
}
```

With Dependency Injection:

With Dependency Injection, you can provide the Engine to the Car from the outside:

```
class Engine {
    public void start() {
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine;

    public Car(Engine engine) {
        this.engine = engine; // Injecting the engine from the outside
    }

    public void start() {
```

```
        engine.start();
        System.out.println("Car started");
    }
}

public class Main {
    public static void main(String[] args) {
        Engine carEngine = new Engine(); // Engine created externally
        Car car = new Car(carEngine); // Car receives the Engine through injection
        car.start();
    }
}
```

SPRING
==========
Spring is a widely used framework for building enterprise-level Java applications. It provides comprehensive infrastructure support for developing Java-based applications, making it easier to create robust, maintainable, and scalable software. Spring offers various modules for different purposes, such as dependency injection, aspect-oriented programming, data access, messaging, and more. In a Spring application, you can develop components as POJOs (Plain Old Java Objects) and use Spring to manage these components, their dependencies, and other aspects of the application.

Here's a simplified example of a traditional Spring application (non-Spring Boot) to help you understand the fundamental concepts. We'll create a simple Spring application that manages a list of users.

Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

1) Predefined Templates
Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

2) Loose Coupling
The Spring applications are loosely coupled because of dependency injection.

3) Easy to test
The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

4) Lightweight
Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

5) Fast Development
The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

6) Powerful abstraction
It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

7) Declarative support

It provides declarative support for caching, validation, transactions and formatting.

Step 1: Set Up the Project

Create a Maven project and add the necessary dependencies to your `pom.xml`.

pom.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>spring-user-app</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.10.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

Step 2: Create a User Entity

Create a `User` entity class to represent users:

```java
public class User {
    private long id;
    private String username;
    private String email;

    // Getters and setters
}
```

Step 3: Create a UserRepository

Create a repository class to manage user data:

```java
public class UserRepository {
    private List<User> users = new ArrayList<>();

    public void addUser(User user) {
        users.add(user);
    }

    public List<User> getUsers() {
        return users;
    }
}
```

Step 4: Create a Controller

Create a controller class to handle user-related operations:

```java
public class UserController {
    private UserRepository userRepository;

    public UserController(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public void addUser(User user) {
        userRepository.addUser(user);
    }

    public List<User> getUsers() {
        return userRepository.getUsers();
    }
}
```

Step 5: Create the Spring Configuration

Create a Spring configuration XML file to define Spring beans and wire them together. Let's call it `applicationContext.xml`:

applicationContext.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="userRepository" class="com.example.UserRepository" />
    <bean id="userController" class="com.example.UserController">
        <constructor-arg ref="userRepository" />
    </bean>
</beans>
```

Step 6: Create the Main Application

Create a Java class to load the Spring application context and use the defined beans:

```java
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

        UserController userController = context.getBean(UserController.class);

        // Use userController to handle user operations
    }
}
```

In this example, we manually configure Spring beans and their dependencies using an XML configuration file. This

demonstrates the basics of a traditional Spring application. You can expand this foundation to build more complex applications.

SPRING AND SPRINGBOOT
==========

Spring and **Spring Boot** are two frameworks in the Spring ecosystem for building Java applications. Here's an overview of each and the key differences between them:

Spring:

1. **Core Container:** The Spring Framework, often referred to as "Spring," provides a comprehensive programming and configuration model for building enterprise applications. It includes modules for managing application configuration, handling dependency injection, and providing core features like AOP (Aspect-Oriented Programming) and transactions.

2. **XML Configuration:** Spring applications are often configured using XML files or Java-based configurations. Configuration details, such as bean definitions and dependency injection, are explicitly defined in these configuration files.

3. **Fine-Grained Control:** Spring offers a high degree of customization and fine-grained control over application components and their behavior. Developers have the flexibility to configure and define application components in a highly customizable way.

4. **Requires Boilerplate Code:** Developing Spring applications typically involves writing more boilerplate code, such as XML configuration or Java-based configuration classes, to set up the application context and configure beans.

5. **Flexible for Complex Applications:** Spring is well-suited for complex enterprise applications where custom configuration and extensive control are required.

Spring Boot:

1. **Opinionated Framework:** Spring Boot is an opinionated framework built on top of the Spring Framework. It simplifies the setup and development of Spring applications by providing a set of defaults and conventions.

2. **Auto-Configuration:** Spring Boot introduces auto-configuration, which automatically configures many aspects of the application based on classpath dependencies. Developers can override these defaults when needed.

3. **Annotations and Defaults:** Spring Boot encourages the use of annotations and sensible defaults. This leads to reduced configuration and less boilerplate code.

4. **Microservices-Ready:** Spring Boot is popular for building microservices due to its ease of development and deployment. It's optimized for creating stand-alone, production-ready Spring-based applications with minimal effort.

5. **Quick Start:** Spring Boot provides a quick and easy way to start new projects. Developers can create a new Spring Boot application with minimal configuration and immediately start building features.

Key Differences:

1. **Configuration:** Spring applications typically require explicit configuration through XML files or Java-based configuration classes, while Spring Boot encourages convention over configuration and relies on auto-configuration.

2. **Boilerplate Code:** Spring applications often involve writing more boilerplate code for setting up the application context and defining beans. Spring Boot reduces the need for boilerplate code.

3. **Complexity:** Spring offers a high degree of customization and is suitable for complex, custom enterprise

applications. Spring Boot is designed to simplify development and is well-suited for microservices and rapid application development.

4. **Default Settings:** Spring Boot provides default settings and opinions to get started quickly, while Spring leaves many configuration decisions to the developers.

In summary, Spring is a comprehensive framework that offers fine-grained control and flexibility, while Spring Boot is an opinionated framework that simplifies Spring application development by providing defaults and conventions. The choice between Spring and Spring Boot depends on the specific project requirements and the development team's preferences. Spring Boot is often preferred for new projects, prototypes, and microservices.

SPRING-BOOT
==========

application.properties file
==========
#changing port number
server.port=8081

spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/practice
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

#Logging the sql queries
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type=TRACE
spring.jpa.show-sql=true

#Write a sql query beautiful on console
spring.jpa.properties.hibernate.format_sql=true

WHAT IS SPRING-BOOT
==========

Spring Boot is a Java-based framework that simplifies the development of standalone, production-ready applications. It is part of the larger Spring ecosystem, but it focuses on making it easy to create Spring applications with minimal configuration.
Spring Boot provides a set of conventions and defaults that enable developers to build applications quickly and efficiently.
1. Simple web application.
==========

```
String data = "Welcome To Plant %s!";

    @GetMapping("/")
public String message(@RequestParam(value = "name", defaultValue = "Nani") String name){
    return String.format(data,name);
}
```

--> @RequestParam : it is telling Spring to expect a name value in the request, but if it’s not there, it will use the word "Nani" by default.

http://localhost:8081/?name=Rajesh

output: Welcome To Plant Rajesh!

2. Spring Boot Restful Services:
==========

1. Spring Boot is often used to create RESTful web services, which are web services that use HTTP methods (GET, POST, PUT, DELETE) to perform CRUD (Create, Read, Update, Delete) operations on resources.
2. Spring Boot simplifies the creation of RESTful services by providing annotations like @RestController to define REST endpoints, and it includes libraries to handle HTTP requests and responses.
3. You can easily build RESTful APIs to expose data or services over HTTP.
4. These services are typically used for communication between systems, mobile apps, web applications, and other clients

3. SpringBoot Data JPA
==========

Spring Data JPA is a part of the larger Spring Data project.
It is a framework that simplifies data access in Java applications that use the Java Persistence API (JPA) to interact with relational databases.
JPA is a Java specification for working with databases in an object-oriented way, allowing developers to work with databases using Java classes and objects instead of SQL queries.

1. Automatic Repository Creation: Spring Boot Data JPA can automatically create repository interfaces for your domain models, which significantly reduces the amount of boilerplate code you need to write for basic CRUD operations.

2. Query Methods: It allows you to create custom query methods by simply defining method names following a specific naming convention. Spring Data JPA translates these method names into SQL queries automatically.

3. Pagination and Sorting: Built-in support for paging and sorting of query results.

4. Transactional Support: It simplifies transaction management, ensuring that database operations are executed within a single transaction context.

5. Custom Queries: You can write native SQL queries, JPQL (Java Persistence Query Language) queries, or use the Criteria API to create complex queries.

RequestBod for List<User>
==========

```
[
   {
      "firstName":"NaniBabu",
      "lastName":"Pallapu",
      "loginName":"NPallapu",
      "password":"Hyderabad@369",
      "email":"nanipallapu369@mail.com",
      "phone":"9392590089"
   },

      {
      "firstName":"Priyanka",
      "lastName":"Bandi",
      "loginName":"PBandi",
      "password":"Priya@369",
      "email":"pryanka369@mail.com",
```

```
    "phone":"8008142536"
  },

    {
    "firstName":"Divya",
    "lastName":"Kommirisetti",
    "loginName":"DKommirisetti",
    "password":"Divya@2002",
    "email":"divya2002@mail.com",
    "phone":"9567892345"
  }
]
```

@RequestBody for Single User Object
==========

```
    {
    "firstName":"NaniBabu",
    "lastName":"Pallapu",
    "loginName":"NPallapu",
    "password":"Hyderabad@369",
    "email":"nanipallapu369@mail.com",
    "phone":"9392590089"
  }
```

Dependency Injection (DI)
==========
In Spring Boot is a design pattern and framework feature that helps manage the dependencies between various components in a Spring application.
The primary goal of DI is to achieve the Inversion of Control (IoC) principle, where the control of creating and managing objects is shifted from the components themselves to an external entity or container.
In the context of Spring Boot, this container is the Spring IoC container.

Here's why Dependency Injection is used and its advantages:

1. Loose Coupling:** DI promotes loose coupling between components, making it easier to maintain, extend, and test the application. Components don't need to know how their dependencies are created or configured; they simply rely on the provided interfaces.

2. Reusability:** Components can be reused across different parts of the application, as they are not tightly bound to specific implementations of their dependencies.

3. Testability:** DI allows for easier unit testing. You can inject mock or stub dependencies during testing to isolate and verify the behavior of individual components.

4. Configurability:** DI enables you to configure the application's components and dependencies, often through configuration files or annotations, without changing the code. This makes the application more adaptable to different environments or configurations.

Here's a simple example of Dependency Injection in a Spring Boot application:

UserService.java:
```java
@Service
public class UserService {
    private final UserRepository userRepository;
```

```
    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User getUserById(Long id) {
        return userRepository.findById(id).orElse(null);
    }
}
```

UserRepository.java:
```java
public interface UserRepository extends JpaRepository<User, Long> {
}
```

In this example:

- `UserService` is a Spring service class that depends on `UserRepository` for fetching user data.

- `UserRepository` is an interface that extends `JpaRepository`. Spring Data JPA will provide the actual implementation of this interface.

- Constructor injection is used in `UserService` to receive an instance of `UserRepository`. The `@Autowired` annotation tells Spring to inject the dependency.

- By using DI, `UserService` is loosely coupled with the concrete implementation of `UserRepository`. The Spring IoC container takes care of creating and providing instances of `UserRepository` to `UserService`.

In your Spring Boot application, you can configure and define beans in various ways, such as through annotations or XML configuration files. Spring Boot's auto-configuration and component scanning capabilities help simplify the DI process, allowing you to focus on your application's logic rather than the details of object creation and wiring.

Spring Boot MVC (Model-View-Controller)
==========
Spring Boot MVC is an architectural pattern and framework for building web applications in Spring Boot. It provides a structured approach to handling web requests and responses, separating an application into three interconnected components:

1. Model: Represents the application's data and business logic. It contains the information that the application operates on.

2. View: Represents the presentation layer of the application, responsible for rendering the user interface and displaying data from the Model.

3. Controller: Acts as an intermediary between the Model and View. It processes incoming HTTP requests, interacts with the Model to retrieve data, and selects the appropriate View for rendering the response.

Uses of Spring Boot MVC:

Spring Boot MVC is widely used for building web applications, both simple and complex. Some common use cases include:

1. Web Applications: Building web applications with dynamic content and user interaction.

2. RESTful APIs: Creating RESTful web services to expose data and functionality to clients.

3. Authentication and Authorization: Implementing user authentication and authorization for web applications.

4. Form Handling: Handling forms and user input in web applications.

5. View Templating: Rendering HTML or other view templates to generate dynamic web pages.

Here's an example of a simple Spring Boot MVC application to demonstrate the use of the Model, View, and Controller components:

Model: User.java
```java
public class User {
    private String username;
    private String email;

    // Getters and setters
}
```

View: userForm.html
```html
<!DOCTYPE html>
<html>
<head>
    <title>User Registration</title>
</head>
<body>
    <h1>User Registration</h1>
    <form method="post" action="/register">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required><br>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required><br>
        <input type="submit" value="Register">
    </form>
</body>
</html>
```

Controller: UserController.java
```java
@Controller
public class UserController {

    @GetMapping("/registration")
    public String showRegistrationForm(Model model) {
        model.addAttribute("user", new User());
        return "userForm";
    }

    @PostMapping("/register")
    public String registerUser(@ModelAttribute("user") User user) {
        // Process and store the user data
        return "registrationSuccess";
    }
}
```

In this example:

- `User` represents the Model, holding user data.

- The `userForm.html` file is the View, displaying a user registration form.

- `UserController` acts as the Controller, handling the HTTP requests. The `showRegistrationForm` method displays the registration form, and the `registerUser` method processes the submitted form.

This is a simplified example, but it demonstrates the essential components of a Spring Boot MVC application. When a user accesses the `/registration` URL, the registration form is displayed. When the form is submitted, the `registerUser` method processes the user's input, and the response is rendered using a view template.

Spring Boot simplifies the setup and configuration of such applications, making it easier to develop web-based systems.

Microservice Architecture with Spring Boot:
==========

Microservices is an architectural approach where a software application is divided into a collection of small, independent, and loosely coupled services, each responsible for specific functionality. Spring Boot, a popular Java framework, is often used to build microservices due to its ease of development and the tools it provides.

Microservice in Spring Boot:

A **microservice in Spring Boot** is a standalone, independently deployable application that performs a specific function within a larger software system. Each microservice typically has its own database and communicates with other microservices through well-defined APIs (often using HTTP/REST).

Uses of Microservices with Spring Boot:

- **Scalability:** Microservices can be independently scaled, allowing you to allocate more resources to specific services that need it, while leaving others unaffected.

- **Maintainability:** Smaller codebases are easier to manage and update. Changes in one service don't affect others, making maintenance more manageable.

- **Faster Development:** Smaller teams can develop and deploy microservices independently, accelerating development cycles.

- **Resilience:** Isolating services reduces the impact of failures. If one service fails, it doesn't bring down the entire system.

- **Technology Diversity:** Different microservices can use different technologies and databases to choose the right tools for each task.

Example of a Microservice in Spring Boot:

Consider an e-commerce platform. You might have separate microservices for user authentication, product catalog, order processing, and payment handling.

Here's a simplified example of a product catalog microservice using Spring Boot:

ProductService.java:
```java
@RestController
@RequestMapping("/products")
public class ProductService {

    @GetMapping("/{productId}")
    public Product getProductById(@PathVariable Long productId) {
        // Retrieve product information from the database and return it
    }

    @PostMapping
    public Product addProduct(@RequestBody Product product) {
        // Save the new product to the database and return it
    }

    // Other product-related endpoints
}
```

In this example:

- `ProductService` is a Spring Boot application serving as a microservice for product-related operations.

- It defines endpoints to retrieve a product by ID and add a new product.

- Each operation is independent, and the service can be deployed and scaled on its own.

Differences Between a Normal Spring Boot Application and a Microservices Application:

1. **Monolith vs. Distributed:** A normal Spring Boot application is often a monolith, where all functionalities are tightly integrated into a single codebase. In contrast, a microservices application is distributed, with functionalities divided into separate services.

2. **Size:** Normal Spring Boot applications tend to be larger, with all functionalities in one place. Microservices are smaller, focused on specific tasks.

3. **Complexity:** Microservices applications can be more complex to manage due to the need for service coordination, inter-service communication, and distributed data.

4. **Scalability:** Microservices allow for fine-grained scalability, while a monolith typically scales as a single unit.

5. **Independent Deployment:** Microservices can be deployed independently. In a monolith, changes may require redeploying the entire application.

6. **Technology Stack:** In a monolith, the entire application uses the same technology stack. In microservices, different services can use different stacks.

7. **Maintenance:** Microservices can be easier to maintain as changes in one service don't affect others. In a monolith, changes can be riskier.

The choice between a normal Spring Boot application and a microservices architecture depends on the specific project

requirements, scalability needs, development team size, and other factors. Microservices offer advantages in terms of scalability and maintainability but introduce complexity in managing the interactions between services.

Normal Spring Boot Monolithic Application:

In a monolithic Spring Boot application, all functionalities are bundled together in a single application. Here's an example of a simplified monolithic application for a basic e-commerce platform:

MonolithicEcommerceApplication.java:

```java
@SpringBootApplication
public class MonolithicEcommerceApplication {
    public static void main(String[] args) {
        SpringApplication.run(MonolithicEcommerceApplication.class, args);
    }
}
```

ProductController.java:

```java
@RestController
@RequestMapping("/products")
public class ProductController {
    @GetMapping("/{productId}")
    public Product getProduct(@PathVariable Long productId) {
        // Retrieve and return the product from the database
    }

    @PostMapping
    public Product addProduct(@RequestBody Product product) {
        // Save the new product to the database and return it
    }

    // Other product-related endpoints
}
```

In this monolithic example:

- There's a single Spring Boot application that handles all aspects of the e-commerce platform, including product management.

- The `ProductController` defines endpoints for retrieving and adding products.

Microservices Architecture:

Now, let's look at a simplified microservices architecture for the same e-commerce platform. We'll split the product management into a separate microservice.

ProductService.java:

```java
@SpringBootApplication
```

```
@EnableEurekaClient
public class ProductServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProductServiceApplication.class, args);
    }
}
```

ProductController.java (Microservice):

```java
@RestController
@RequestMapping("/products")
public class ProductController {
    @GetMapping("/{productId}")
    public Product getProduct(@PathVariable Long productId) {
        // Retrieve and return the product from the microservice's database
    }

    @PostMapping
    public Product addProduct(@RequestBody Product product) {
        // Save the new product to the microservice's database and return it
    }

    // Other product-related endpoints
}
```

In this microservices example:

- There are two separate Spring Boot applications: the **ProductService** microservice and the **E-commerceGateway** application (not shown here) that routes requests to various microservices.

- The **ProductService** microservice handles product-related operations, just as in the monolithic example. It runs independently and can be scaled separately.

- Other microservices can handle different parts of the e-commerce platform, and they can use various technologies and databases as needed.

The key difference is that in the microservices architecture, the functionalities are divided into smaller, independently deployable services, whereas the monolithic application handles everything in a single codebase. Microservices offer advantages in terms of scalability, maintainability, and technology stack diversity but introduce complexities in terms of service coordination and inter-service communication.

Spring Cloud is a set of tools and frameworks within the broader Spring ecosystem for building and deploying cloud-native applications. It provides a range of capabilities for building microservices-based systems and addressing common challenges that arise in cloud and distributed architectures. Spring Cloud is primarily designed to work with the Spring Framework, which is a popular framework for building Java-based applications.

Key components and features of Spring Cloud include:

1. **Service Discovery**: Spring Cloud provides tools like Eureka and Consul for service discovery. This allows services to register themselves with a registry and discover other services. This is crucial for building resilient and dynamic microservices architectures.

2. **Load Balancing**: Spring Cloud integrates with load balancers like Ribbon, enabling client-side load balancing for services. This ensures that requests are distributed across instances of a service, improving scalability and fault tolerance.

3. **API Gateway**: Spring Cloud offers tools like Zuul and Spring Cloud Gateway for creating API gateways. These components act as a single entry point to your microservices, providing routing, authentication, and other cross-cutting concerns.

4. **Circuit Breakers**: To deal with failures in distributed systems, Spring Cloud incorporates Hystrix, which provides circuit breaker and fault tolerance patterns. It helps prevent cascading failures in the system.

5. **Configuration Management**: Spring Cloud Config allows you to manage and distribute configuration properties for your services. It supports externalized configuration, versioning, and dynamic property updates without service restarts.

6. **Distributed Tracing**: Tools like Spring Cloud Sleuth and Zipkin provide distributed tracing capabilities, helping you track and diagnose issues across microservices.

7. **Security**: Spring Cloud Security provides features for securing your microservices and managing authentication and authorization in a distributed system.

8. **Messaging**: Spring Cloud Stream simplifies event-driven microservices by providing abstractions for message brokers like Apache Kafka and RabbitMQ.

9. **Distributed Data**: Tools like Spring Cloud Data Flow and Spring Cloud Task help manage and process data across distributed systems.

10. **Monitoring and Metrics**: Spring Cloud integrates with monitoring solutions like Prometheus and Grafana to gather metrics and monitor the health of your microservices.

11. **Batch Processing**: Spring Cloud Data Flow and Spring Batch offer batch processing capabilities, making it easier to manage batch jobs in a microservices architecture.

Spring Cloud promotes the development of scalable, resilient, and maintainable microservices applications by providing pre-built solutions for common distributed system challenges. It's particularly well-suited for Java-based applications and is a popular choice for organizations looking to adopt microservices and cloud-native architectures.

==========
BUILDING APPLICATION :
==========

1. We will build simple ONLINE SHOPPING APPLICATION.

2. We will cover below topics:
    i. Service Discovery
    ii. Centralized Configuration.
    iii. Distributed Tracing.
    iv. Event Driven Architecture.
    v. Centralized Logging.
    vi. Circuit Breaker.
    vii.Secure Microservice Using KeyCloak.

Services We are going to build :
---

1. Product Service : Create and View Products, acts as Product Catalog.

2. Order Service : Can order products.

3. Inventory Service : Can check if product is in stock or not.

4. Notification Service : Can send notifications, after order is placed.

5. Order Service , Inventory Service and Notification Service are going to interact with each other.

6. Synchronous and Asynchronous communication.

1.After creating product-service, order-service and inventory-service, We use WebClient to make Synchronous request from Order-Service to the Inventory-Service, then Inventory-Service will respond with required data and WebClient will take that required data and give it to Order-Service.

2. Discovery-Service is nothing but a server which will store all the information about services(Inventory-Service)) like service-name, IP address.

3. When we are using the Discovery-Service , Our microservices will register in the Discovery-Service by making the request at the starting of the application.

4. Whenever services(Inventory-Service) are making request , Discovery-Service will add the entries of the services into it's local copy. we call it registry, that's why we call it Service Registry.

5. Once all the information about the services(Inventory-Service) is present in Discovery-Service, When our Order-Service wants to call the Inventory-Service, First Order-Service will call Discovery-Service by asking where I can find the Inventory-Service. Then Discovery-Service will respond with particular IP address to call Inventory-Service. Then Order-Service will call to Inventory-Service.

6. In this way, we can avoid the hardcoding URL of the Inventory-Service by making use of Discovery-Service.

7. When we are making initial call to the Discovery-Service, Discovery-Service will sends its registry as the the respnose to the client , the client will store the local copy of the registry in a separate location.

8. In Some reasons, If the Discovery-Service is not available, first Discovery-Service will check the local copy because it already has information about Inventory-Service. It will call Inventory-Service IP address, If the first instance of the Inventory-Service is not available, it will check the next entry of Inventory-Service likewise it goes through all entries of its registry. If all the instances are down, Discovery-Service will fail by saying that Inventory-Service is not available.