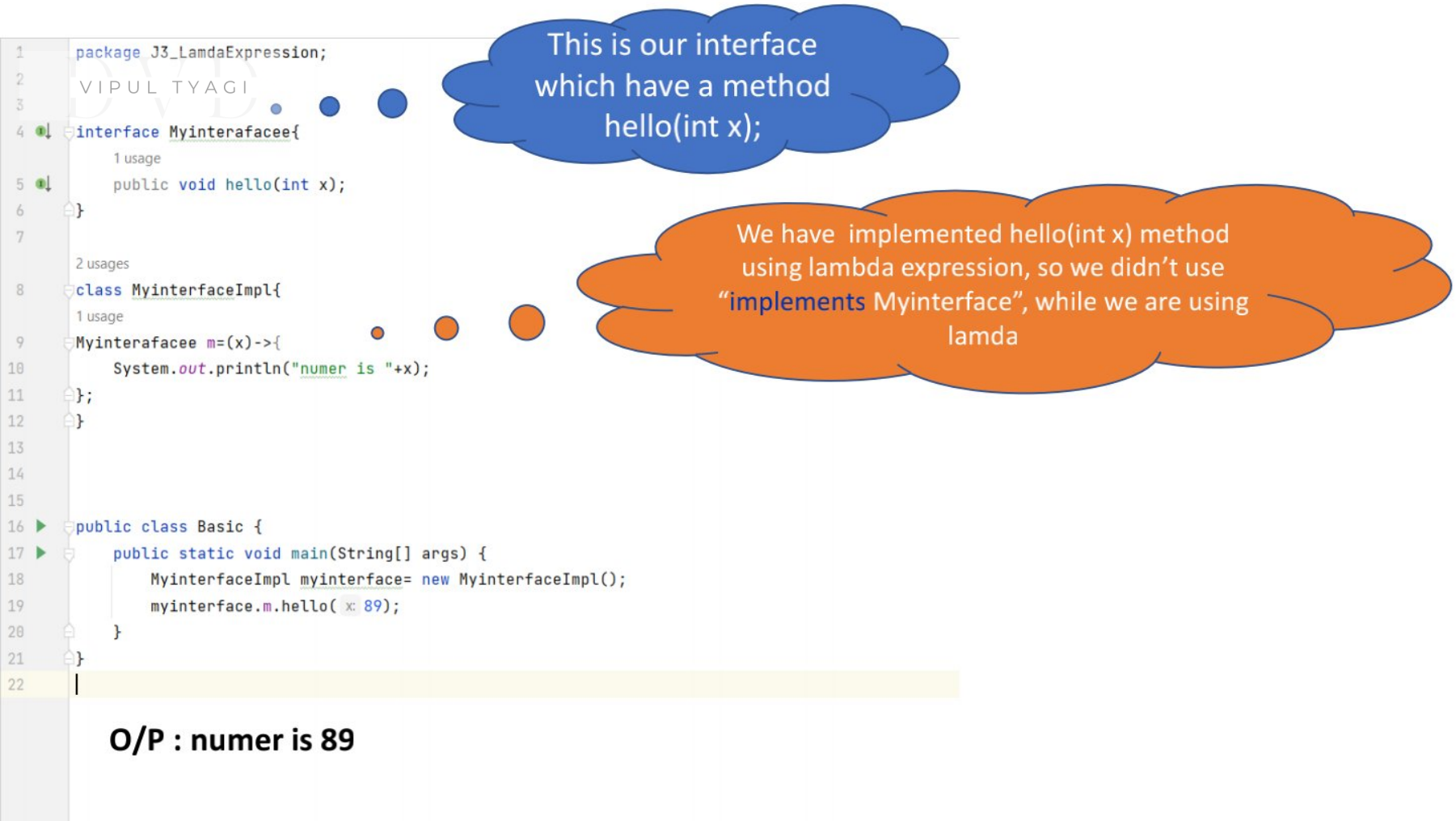
JAVA 8

Java 8 features

1. **Functional interfaces**: Understanding functional interfaces is crucial to understanding lambda expressions and the Stream API.
2. **forEach :**It provides an easy and concise way to iterate over a collection of elements and perform a specified action on each element
3. **Lambda expressions**: Learning lambda expressions is the next step as they are widely used in conjunction with functional interfaces.
4. **Stream API**: After understanding lambda expressions, the Stream API can be learned, which provides a functional way to process collections of data.
5. **Optional class**: The Optional class can be learned next as it is frequently used with the Stream API and lambda expressions.
6. **Method references**: Method references can be learned next as they provide a way to write more concise and readable code.
7. **Default methods**: Finally, default methods can be learned, as they allow interfaces to have concrete methods, which makes it easier to evolve interfaces without breaking existing code.

- It's important to note that these Java 8 features are interconnected, and mastering one feature can improve your understanding of the others. Therefore, it's recommended to practice and apply these features in real-world scenarios to reinforce your understanding of each concept.

Let's understand basic **Lambda Expression** first




```java
package J3_LamdaExpression;
VIPUL TYAGI

interface Myinterafacee{
    1 usage
    public void hello(int x);
}

2 usages
class MyinterfaceImpl{
1 usage
Myinterafacee m=(x)->{
    System.out.println("numer is "+x);
};
}



public class Basic {
    public static void main(String[] args) {
        MyinterfaceImpl myinterface= new MyinterfaceImpl();
        myinterface.m.hello( x: 89);
    }
}
```

O/P : numer is 89

Let's understand basic **Lambda Expression** first **CONT..**

Suppose we want to implement a method which sums the two number, Using lambda


```
public int sum(int a, int b)
 { return a + b; }
```

So to implement this method we create an interface which have a method for providing two values and getting the sum

```java
package J3_LamdaExpression;

import java.util.function.IntBinaryOperator;
1 usage
interface SummerClass{
    1 usage
    public int applyAsInt(int a, int b);
}

public class Basic1 {
    public static void main(String[] args) {
        SummerClass sum = (a, b) -> a + b;
        int result = sum.applyAsInt( a: 3,  b: 5);
        System.out.println("Result: " + result);
    }
}
```

Let’s understand basic **Lambda Expression** first **CONT**..

A. Functional Interfaces

- In Java 8, a functional interface is an interface that has only one abstract method. Functional interfaces are used extensively in Java 8's lambda expressions and method references, which provide a more concise and expressive way of writing code.

- Functional interfaces are annotated with the **@FunctionalInterface** annotation to ensure that they have only one abstract method. The following is an example of a functional interface in Java 8:

A. Functional Interfaces CONT..


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

```java
MyFunctionalInterface myLambda = () -> System.out.println("Doing something");
myLambda.doSomething();
```

Here, the lambda expression **() -> System.out.println("Doing something")** implements the **doSomething()** method of the **MyFunctionalInterface** interface. The lambda expression is then assigned to a variable of type **MyFunctionalInterface**, and the **doSomething()** method is called using the **myLambda.doSomething()** syntax.

A. Functional Interfaces CONT....

- TYPES OF FUNCTIONAL INTERFACES

1. Predicate : Represents a function that takes one argument and returns a boolean value.
2. Consumer : Represents a function that takes one argument and returns no result.
3. Supplier : Represents a function that takes no arguments and returns a value.
4. Function : Represents a function that takes one argument and returns a value.

…. Many more

A. Functional Interfaces CONT....

1. Predicate

- In Java 8, the **Predicate** functional interface represents a function that takes one argument and returns a boolean value. The abstract method of the **Predicate** interface is **test**, which takes an object of the type T and returns a boolean.

- The **Predicate** interface is commonly used in Java 8's stream API to filter elements based on a given condition. The **test** method is used to evaluate the condition for each element in the stream, and only those elements that pass the condition are included in the result.

A. Functional Interfaces CONT....

1. Predicate

```java
import java.util.function.Predicate;
// 1). predicate-- boolean result

public class F1_FunctionalInteface_Prediacte {
    public static void main(String[] args) {
        /**
         *
         * This is a functional interface whose functional method is **test**(Object).
         *
         * Generally predicate means a statement that determines whether a value could be true or false.
         * In programming predicates mean functions with one argument that return a boolean value.
         *
         * Functional Interface Predicate was realised in Java 8. "Functional" means that it contains only one abstract method.
         * It accepts an argument and returns a boolean.
         *
         *  In Java, functional interfaces are used for handling Lambda expressions, constructors, and Method references.
         *
         * Usually Java 8 Predicate is used to apply a **filter** for a collection of objects. Let's take a look at its main applications and widely used methods,
         *  as well as solve some practice problems.
         */


        Predicate<String> checklength= (str)->{return str.length()>5;};
        // Predicate<String> checklength= (str)-> str.length()>5; // we can also use this , both are same
        System.out.println(checklength.test( t "vikrant"));

    }
}

```

A. Functional Interfaces CONT....

1. Predicate

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Define a predicate to filter even numbers
Predicate<Integer> isEven = n -> n % 2 == 0;

// Use the predicate to filter even numbers from the list
List<Integer> evenNumbers = numbers.stream()
                                   .filter(isEven)
                                   .collect(Collectors.toList());

// Print the even numbers
System.out.println(evenNumbers); // Output: [2, 4, 6, 8, 10]
```

```java
List<String> words = Arrays.asList("apple", "banana", "cherry", "date", "elderberry

// Define a predicate to check if a string starts with a given prefix
Predicate<String> startsWith = s -> s.startsWith("c");

// Use the predicate to check if any word in the list starts with the prefix
boolean anyWordStartsWithC = words.stream()
                                  .anyMatch(startsWith);

// Print the result
System.out.println(anyWordStartsWithC); // Output: true
```

A. Functional Interfaces CONT....

1. Predicate

- Real life Example :-

The predicate interface is commonly used in programming when you need to check whether an object satisfies a certain condition. It is an interface in Java that represents a function that takes in one input parameter and returns a boolean value.

- **Here are some real-life programming examples where the predicate interface can be useful:**

1. **Filtering data**: Suppose you have a large dataset and you want to filter out some specific records based on certain conditions. In this case, you can use the predicate interface to define the conditions and apply them to the dataset. For example, you can use a predicate to filter out all the records where the age is greater than 30.

2. **Validating input**: When you are building a web application, you need to validate user input before processing it. For instance, you might want to check whether a user's email address is valid or not. Here, you can use a predicate to define the validation condition and apply it to the input data.

3. **Sorting objects**: Suppose you have a list of objects and you want to sort them based on some criteria. In this case, you can use a predicate to define the criteria and apply it to the objects. For example, you can use a predicate to sort a list of employees based on their salary.

A. Functional Interfaces CONT....

1. Predicate

In laymen language, how can I use it,

Try to use the condition in if() statement, in your mind only, it also runs on True or False condition, it acts as a test(T) method of predicate interface


```
25
26          Predicate<String> checklength= (str)->{return str.length()>5;};
27          // Predicate<String> checklength= (str)-> str.length()>5; // we can also use this , both are same
28          System.out.println(checklength.test( t: "vimarsh"));
29
30          ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
31          String vim= "vimarsh";
32          if(vim.length()>5)
33          {
34              System.out.println("true");
35          }
36          else{
37              System.out.println("false");
38          }
39
40      }
41  }
```

Using Predicate

Using if() condition to know predicate interface

```
Debug:  F1_FunctionalInteface_Prediacte
Debugger  Console
"C:\Program Files\Java\jdk-17.0.4.1\bin\java.exe" -agentlib:jdwp=transport=dt_socket,address=127.0.0.1:57446,suspend=y,server=n -ja
Connected to the target VM, address: '127.0.0.1:57446', transport: 'socket'
false
true
```

A. Functional Interfaces CONT....

- TYPES OF FUNCTIONAL INTERFACES

1. Predicate : Represents a function that takes one argument and returns a boolean value.
2. Consumer : Represents a function that takes one argument and returns no result.
3. Supplier : Represents a function that takes no arguments and returns a value.
4. Function : Represents a function that takes one argument and returns a value.

…. Many more

A. Functional Interfaces
2. Consumer

In Java 8, the Consumer interface is a functional interface that represents an operation that takes a single input and returns no result. It is located in the java.util.function package.

The Consumer interface has a single method:

```
void accept(T t);
```

Here, **T** is the type of the input to the operation. The method takes a single parameter of type **T** and returns no result.

A. Functional Interfaces CONT....

2. Consumer

The Consumer interface can be used in many situations, such as iterating over a collection of objects and performing an operation on each one, or processing the elements of a stream.

```
List<String> list = Arrays.asList("apple", "banana", "orange");

Consumer<String> consumer = (String s) -> System.out.println(s);

list.forEach(consumer);
```

In this example, we create a list of strings and a Consumer that simply prints out each string. We then use the forEach method of the list to iterate over its elements and apply the Consumer to each one. The output of this code would be:

```
apple
banana
orange
```

A. Functional Interfaces CONT....

2. Consumer

Implementation

```java
package J0_FunctionalInterface;

import java.util.List;
import java.util.function.Consumer;

2 usages
class conimpl {

    1 usage
    Consumer<Integer> c=(i)->{
        System.out.println(i*i);
    };


}


public class F2_functionalInterface_Consumer_0 {

    public static void main(String[] args) {
        List<Integer> l= List.of(1,3,4,5,887,4,4321234);

        conimpl nj= new conimpl();


        l.forEach(nj.c);

    }
}

```

A. Functional Interfaces CONT....

2. Consumer

Implementation

```java
public class F2_functionalInterface_Consumer {
    // 2). Consumner-- it modifies the data no output
    public static void main(String[] args) {
        /**
         * Java Consumer is a functional interface which represents an operation that accepts a single input argument and returns no result.
         * Unlike most other functional interfaces, Consumer is expected to operate via side-effects.
         *
         * @FunctionalInterface
         * public interface Consumer<T> {
         *     void accept(T t);
         * }
         * The Consumer's functional method is accept(Object). It can be used as the assignment target for a lambda expression or method reference.
         */
    Consumer<Integer> printMultiplyBy100 = (val) -> System.out.println(val*100);

        printMultiplyBy100.accept( t: 3);
        printMultiplyBy100.accept( t: 4);
        printMultiplyBy100.accept( t: 5);

    IntConsumer printMultipleBy500= a-> System.out.println(a*500);
        printMultipleBy500.accept( value: 1);
        printMultipleBy500.accept( value: 3);
        printMultipleBy500.accept( value: 4);

    }
}
```

A. Functional Interfaces CONT....

2. Consumer

- Some more Examples

1.Consumer<String> - This functional interface accepts a single argument of type String and performs some operation on it.

```
Consumer<String> printUpperCase = str -> System.out.println(str.toUpperCase());
printUpperCase.accept("hello"); // output: "HELLO"
```

A. Functional Interfaces CONT....

2. Consumer

- Some more Examples

2.Consumer<Integer> - This functional interface accepts a single argument of type Integer and performs some operation on it.

```
Consumer<Integer> printSquare = num -> System.out.println(num * num);
printSquare.accept(5); // output: 25
```

A. Functional Interfaces CONT....

2. Consumer

- Some more Examples

3.Consumer<List<String>> - This functional interface accepts a single argument of type List<String> and performs some operation on it.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
Consumer<List<String>> printNames = list -> list.forEach(System.out::println);
printNames.accept(names); // output: "Alice", "Bob", "Charlie"
```

A. Functional Interfaces CONT....

2. Consumer

- Some more Examples

4.Consumer<Throwable> - This functional interface accepts a single argument of type Throwable and performs some operation on it.

```
Consumer<Throwable>  handleException = ex -> System.out.println("An error occurred: " + ex.getMessage());
                                                                try {
                                                                      // some code that may throw an exception
                                                                   } catch (Exception ex) {
                                                                      handleException.accept(ex);
                                                                  }
```

A. Functional Interfaces CONT....

2. Consumer

- method of streams uses consumer functional intreface

1.forEach(Consumer<? super T> action) - This method applies the given **Consumer** function to each element of the stream.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream().forEach(name -> System.out.println("Hello, " + name + "!"));
// output: "Hello, Alice!", "Hello, Bob!", "Hello, Charlie!"
```

A. Functional Interfaces CONT....

2. Consumer

- method of streams uses consumer functional intreface

2.peek(Consumer<? super T> action) - This method returns a stream consisting of the same elements as the original stream, but applies the given **Consumer** function to each element as they are consumed from the resulting stream.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream().peek(name -> System.out.println("Processing " + name)).forEach(System.out::println);
// output: "Processing Alice", "Alice", "Processing Bob", "Bob", "Processing Charlie", "Charlie"
```

A. Functional Interfaces CONT....

2. Consumer

- method of streams uses consumer functional intreface

3.collect(Collector<? super T,A,R> collector) - This method collects the elements of the stream into a result container, which is created and managed by the given **Collector**. The **Collector** often uses a **Consumer** to accumulate elements into the container.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
StringBuilder result = names.stream().collect(StringBuilder::new, (sb, name) ->
sb.append(name.toUpperCase()), StringBuilder::append);
System.out.println(result);
// output: "ALICEBOBCHARLIE"
```

A. Functional Interfaces CONT....

2. Consumer

- Scenario where we can use the Consumer interface in web development

When we are submitting something in a any method we can use it

```
@PostMapping("/register")
public String registerUser(@ModelAttribute("userForm") UserForm userForm) {
    Consumer<User> registrationProcess = user -> {
        userService.registerUser(user);
        emailService.sendConfirmationEmail(user);
    };
    User newUser = new User(userForm.getUsername(), userForm.getPassword(), userFor
    registrationProcess.accept(newUser);
    return "registration-success";
}
```

A. Functional Interfaces CONT....

- TYPES OF FUNCTIONAL INTERFACES

1. Predicate : Represents a function that takes one argument and returns a boolean value.
2. Consumer : Represents a function that takes one argument and returns no result.
3. Supplier : Represents a function that takes no arguments and returns a value.
4. Function : Represents a function that takes one argument and returns a value.

…. Many more

A. Functional Interfaces

3. Supplier

In Java 8, functional interfaces were introduced to support functional programming in Java. A functional interface is an interface that has only one abstract method. Supplier is one of the built-in functional interfaces in Java 8.

The **Supplier** functional interface is a generic interface that represents a supplier of results. It has one method **get()** that takes no arguments and returns a value of type **T**. The purpose of this interface is to provide a way to generate or supply values of a certain type without any input.

A. Functional Interfaces CONT....

3. Supplier

Here is an example of a **Supplier** functional interface:

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

Using this interface, you can create objects that implement the **Supplier** interface and supply values of any type. Here is an example of using a **Supplier** to generate a random number:

```
Supplier<Double> randomGenerator = () -> Math.random();
double randomNumber = randomGenerator.get();
```

A. Functional Interfaces CONT....

3. Supplier

Here are some more detailed explanations of the six real-life scenarios where the **Supplier** functional interface can be used, along with examples:

1.Generating unique IDs:

Suppose we want to generate unique IDs for users in our system. We can use a **Supplier** to generate these IDs as follows:

```
import java.util.UUID;
import java.util.function.Supplier;

public class User {
    private String id;
    private String name;

    public User(String name) {
        this.name = name;
        this.id = generateId();
    }

    private String generateId() {
        Supplier<String> supplier = () -> UUID.randomUUID().toString();
        return supplier.get();
    }

    // Other methods and properties...
}
```

In this example, we create a **Supplier** that generates unique IDs using the **UUID.randomUUID()** method. We then call **supplier.get()** to retrieve a new ID each time a **User** object is created.

A. Functional Interfaces  CONT....

3. Supplier

Here are some more detailed explanations of the six real-life scenarios where the **Supplier** functional interface can be used, along with examples:

2.Lazy initialization:

Lazy initialization is a design pattern that defers the creation of an object until the point when it is actually needed. This pattern is often used when creating an object is an expensive operation or when an object may not be needed during the entire lifetime of the application.

In lazy initialization, the object is created only when it is first requested, rather than being created when the application starts up. This approach can save time and resources, especially if the object is never actually used.

To implement lazy initialization, a **Supplier** can be used to provide a single instance of the object on-demand. The **Supplier** is initialized with a method that creates the object, but the method is not called until the first time the **Supplier** is asked to provide an instance of the object.

Let's implement it in the next slide

A. Functional Interfaces  CONT....

3. Supplier

Here are some more detailed explanations of the six real-life scenarios where the **Supplier** functional interface can be used, along with examples:

2.Lazy initialization:

```
import java.util.function.Supplier;
 public class ExpensiveObject {
 private Supplier<ExpensiveObject> lazyInstance = () -> createExpensiveObject();

 public ExpensiveObject() {
 // Constructor
}
 private ExpensiveObject createExpensiveObject() {
 // Code to create the expensive object goes here...
try {
      Thread.sleep(5000); // Simulate an expensive operation that takes 5 seconds   } catch
(InterruptedException e) {
e.printStackTrace(); }
  return new ExpensiveObject();

 }

public ExpensiveObject getInstance()
{ return lazyInstance.get(); }
 }
```

A. Functional Interfaces  CONT....

3. Supplier

Here are some more detailed explanations of the six real-life scenarios where the **Supplier** functional interface can be used, along with examples:

2.Lazy initialization:

Lazy initialization is a common design pattern used in many real-life applications. Here are some examples:

1.Database Connections: When developing an application that uses a database, it is common to use lazy initialization to establish a connection to the database. Establishing a connection to the database can be an expensive operation, and creating a connection for each request would be inefficient. Instead, the application can use a connection pool and lazily initialize a connection only when it is needed.

2.Dependency Injection: In software development, dependency injection frameworks often use lazy initialization to create objects only when they are actually needed. This can help to improve performance and reduce resource usage by avoiding the creation of objects that are never used.

3.Configuration Settings: In applications that use configuration settings, lazy initialization can be used to load settings only when they are needed. For example, an application might have many configuration settings, but not all of them will be used at once. By using lazy initialization, the application can load settings only when they are requested, rather than loading all the settings at once.

A. Functional Interfaces  CONT….

3. Supplier

Here are some more detailed explanations of the six real-life scenarios where the **Supplier** functional interface can be used, along with examples:

2.Lazy initialization:

Lazy initialization is a common design pattern used in many real-life applications. Here are some examples:

2.Dependency Injection: In software development, dependency injection frameworks often use lazy initialization to create objects only when they are actually needed. This can help to improve performance and reduce resource usage by avoiding the creation of objects that are never used.

Let’s discuss this

A. Functional Interfaces  CONT....

3. Supplier

Here are some more detailed explanations of the six real-life scenarios where the **Supplier** functional interface can be used, along with examples:

2.Lazy initialization:

Lazy initialization is a common design pattern used in many real-life applications. Here are some examples:

What is DI(Dependency Injection **)?**

In software development, Dependency Injection (DI) is a design pattern in which an object is passed its dependencies (i.e., other objects that it needs to function) rather than creating them internally. Dependency injection frameworks are tools that help implement this pattern by managing the dependencies of an application and providing them to the objects that need them.

Lazy initialization is often used in dependency injection frameworks to create objects only when they are actually needed. This can help to improve performance and reduce resource usage by avoiding the creation of objects that are never used.

A. Functional Interfaces  CONT….

3. Supplier

Here are some more detailed explanations of the six real-life scenarios where the **Supplier** functional interface can be used, along with examples:

2.Lazy initialization:

Lazy initialization is a common design pattern used in many real-life applications. Here are some examples:

How to use this interface in DI?

For example, imagine an application that uses an object called **DataService** to retrieve data from a database. The **DataService** object depends on a **DatabaseConnection** object to function. In a dependency injection framework, the **DatabaseConnection** object can be created lazily, only when the **DataService** object needs it to retrieve data from the database.

If the **DatabaseConnection** object were created eagerly (i.e., at the start of the application), it would consume resources even if the **DataService** object never actually used it. By using lazy initialization, the **DatabaseConnection** object is created only when it is actually needed, which can help to improve performance and reduce resource usage.

Let’s implement it in the code in next slide

```
public class DataService {
 private final Supplier<DatabaseConnection> connectionSupplier;

 public DataService(Supplier<DatabaseConnection> connectionSupplier) {
this.connectionSupplier = connectionSupplier;
 }
 public void fetchData() {
// Lazily initialize the DatabaseConnection object
DatabaseConnection connection = connectionSupplier.get();

 // Use the connection to fetch data from the database // ...
   }
}
```


Explanation is on next page

```
public class DatabaseConnection {
      public DatabaseConnection() {
          // Code to establish a database connection goes here...
    System.out.println("Establishing database connection...");

  }
    }
```

```
public class Main {
public static void main(String[] args) {
// Create a lazy-initialized supplier for the DatabaseConnection object
Supplier<DatabaseConnection> connectionSupplier = () -> {
    return new DatabaseConnection();
 };
 // Create a DataService object using the supplier
DataService service = new DataService(connectionSupplier);
// Call the fetchData() method (the DatabaseConnection object will be lazily initialized)

 service.fetchData(); } }
```

A. Functional Interfaces  CONT….

3. Supplier

Here are some more detailed explanations of the six real-life scenarios where the **Supplier** functional interface can be used, along with examples:

2.Lazy initialization:

Lazy initialization is a common design pattern used in many real-life applications. Here are some examples:

In this example, the **DataService** class takes a **Supplier<DatabaseConnection>** as a constructor parameter. The **Supplier** interface is used to lazily initialize the **DatabaseConnection** object, so it is only created when it is actually needed.

The **Main** class creates a **Supplier** object that creates a new **DatabaseConnection** object when the **get()** method is called. This supplier is then passed to the **DataService** object.

When the **fetchData()** method of the **DataService** object is called, the **DatabaseConnection** object is lazily initialized by calling the **get()** method of the **Supplier** object. If the **fetchData()** method is never called, the **DatabaseConnection** object is never created, which can help to reduce resource usage in the application.

Method of streams uses supplier functional intreface


```java
import java.util.function.Supplier;
import java.util.stream.Stream;

public class StreamExample {
    public static void main(String[] args) {
        Supplier<Stream<String>> supplier = () -> Stream.of("apple", "banana", "or
        supplier.get().filter(fruit -> fruit.length() > 5).forEach(System.out::pri
    }
}
```

In this example, we define a **Supplier** that generates a **Stream** of strings containing the values "apple", "banana", and "orange". We then use the **filter** method to select only the fruits with a length greater than 5, and the **forEach** method to print them out.
Using a **Supplier** allows you to delay the generation of a **Stream** until it is actually needed, which can improve performance and memory usage in some cases. It also allows you to reuse the same **Stream** multiple times, which can be useful when working with large data sets.

A. Functional Interfaces CONT....

- TYPES OF FUNCTIONAL INTERFACES

1. Predicate : Represents a function that takes one argument and returns a boolean value.
2. Consumer : Represents a function that takes one argument and returns no result.
3. Supplier : Represents a function that takes no arguments and returns a value.
4. Function : Represents a function that takes one argument and returns a value.

…. Many more

A. Functional Interfaces
4. Function

A function is a type of functional interface that takes an argument and returns a result. It is represented by the **java.util.function.Function** interface, which has one abstract method: **apply()**. The **apply()** method takes one argument of a certain type and returns a result of another type.

```
Function<String, Integer> stringLength = str -> str.length();
int length = stringLength.apply("Hello"); // length is 5
```

In this example, **stringLength** is a function that takes a **String** argument and returns an **Integer** result. The lambda expression **str -> str.length()** is the implementation of the **apply()** method, which returns the length of the input string. The **apply()** method is called on the **stringLength** object with the input string "Hello", and the result is stored in the **length** variable.

A. Functional Interfaces CONT...

4. Function

```java
import java.util.function.Function;

public class F3_FunctionalInterface_Function {
    public static void main(String[] args) {

        /*
        Represents a function that accepts one argument and produces a result.
            This is a functional interface whose functional method is ** apply(Object)  **.
            Since:
            1.8
            Type parameters:
            <T> - the type of the input to the function <R> - the type of the result of the function
         */
        //One of the usages of the Function type in the standard library is the Map
        Function<Integer,Integer> getInt= t->{
            return t*10;
        };
        // System.out.println(getInt.apply(2));
        //The Function interface also has a default compose method that allows us to combine several functions into one and execute them sequentially:

        Function<Integer, String> intToString = Object::toString;
        Function<String, String> quote = s -> "'" + s + "'";

        Function<Integer, String> quoteIntToString = quote.compose(intToString);

        System.out.println(quoteIntToString.apply( t: 5));
    }
}
```

OUTPUT: '5'

VIPUL TYAGI

A. Functional Interfaces CONT...

4. Function

Scenario where we can use 'function' functional interface

1.Mapping: The **Function** interface can be used to map one object to another. For example, you can define a function that takes a **Person** object and returns their name as a **String**.

```
Function<Person, String> getName = (person) -> person.getName();
```

2.Conversion: The **Function** interface can be used to convert an object from one type to another. For example, you can define a function that takes a **String** and converts it to an integer.

```
Function<String, Integer> stringToInt = (str) -> Integer.parseInt(str);
```

A. Functional Interfaces CONT...

4. Function

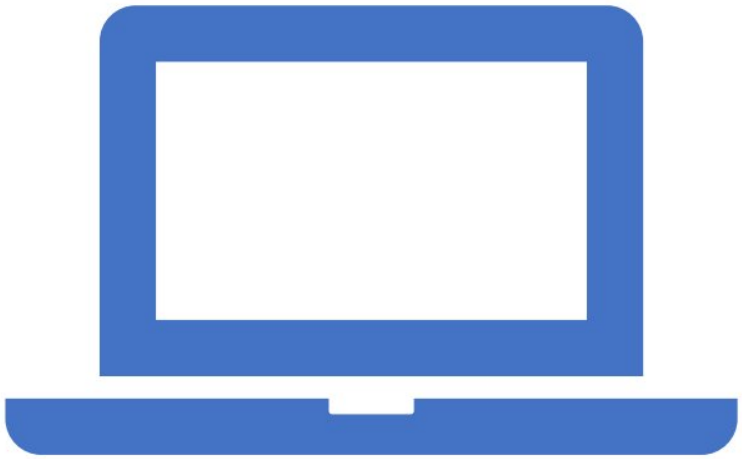
Scenario where we can use 'function' functional interface

3. Composition: The **Function** interface can be used to compose multiple functions together. For example, you can define a function that takes a **String**, converts it to an integer, and then adds 10 to the result.

```
Function<String, Integer> stringToInt = (str) -> Integer.parseInt(str);
Function<Integer, Integer> addTen = (n) -> n + 10;
Function<String, Integer> stringToIntAndAddTen = stringToInt.andThen(addTen);
```

In this example, the **andThen** method is used to compose the **stringToInt** and **addTen** functions together into a single function that takes a **String** as input and returns an **Integer** as output.

Overall, the "Function" functional interface is a powerful tool for representing functions that take one argument and produce a result, and it can be used in a wide range of scenarios in Java development.



Functional Interface :-> END

Java 8 features

1. **Functional interfaces**: Understanding functional interfaces is crucial to understanding lambda expressions and the Stream API.
2. **forEach :**It provides an easy and concise way to iterate over a collection of elements and perform a specified action on each element
3. **Lambda expressions**: Learning lambda expressions is the next step as they are widely used in conjunction with functional interfaces.
4. **Stream API**: After understanding lambda expressions, the Stream API can be learned, which provides a functional way to process collections of data.
5. **Optional class**: The Optional class can be learned next as it is frequently used with the Stream API and lambda expressions.
6. **Method references**: Method references can be learned next as they provide a way to write more concise and readable code.
7. **Default methods**: Finally, default methods can be learned, as they allow interfaces to have concrete methods, which makes it easier to evolve interfaces without breaking existing code.

- It's important to note that these Java 8 features are interconnected, and mastering one feature can improve your understanding of the others. Therefore, it's recommended to practice and apply these features in real-world scenarios to reinforce your understanding of each concept.

B. forEach method Java 8

- The forEach() method is a new feature introduced in Java 8 that provides an easy and concise way to iterate over a collection of elements and perform a specified action on each element.

- The signature of the forEach() method is as follows:

```
void forEach(Consumer<? super T> action)
```

It needs "**consumer**" functional interface in its body

B. forEach method Java 8 CONT....

```
package J2_forEach;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
/*
  For instance, consider a for-loop version of iterating and printing a Collection of Strings:
     List<String> names=List.of("12","1sad","cw","ewdf");
        for (String name : names) {
        System.out.println(name);
        }
     Copy
     We can write this using forEach:
        names.forEach(name -> {
        System.out.println(name);
        });
 */

// Three most popular ways we use the forEach method.
```

```
class consumer{              // 1 Anonymous Consumer Implementation
    Consumer<Integer> printConsumer= new Consumer<Integer>() {
      public void accept(Integer c) {
        System.out.println(c);
      }
    };
}
 class consumer1 {           // 2 Lambda Expression

    Consumer<Integer> c=(i)->{
      System.out.println(i);
    };


}
  public class ForEach {

    public static void main(String[] args) {
      List l=List.of(1,4,2,5,7,6);


      consumer co= new consumer();
      consumer1 con= new consumer1();


      l.forEach(con.c);

      l.forEach(co.printConsumer);

      l.forEach(System.out::println);  // 3 Method Reference



    }

}
```

B. Lambda Expressions