1) what is the lambda expression of Java 8?

Ans:
1. Lambda expressions are anonymous functions used to create instances of functional interfaces.

   Syntax: `(parameters) -> { body }`

2. Parameters represent the input arguments to the lambda expression. They can be empty or contain one or more parameters.

   Syntax examples:

   - Empty parameters: `() -> { // body }`

   - Single parameter: `x -> { // body }`

   - Multiple parameters: `(x, y) -> { // body }`

3. The arrow operator (`->`) separates the parameter list from the body of the lambda expression.

4. The body represents the implementation of the lambda expression. It can be a single expression or a block of code enclosed in curly braces.

   Syntax examples:

   - Single expression: `x -> x * x`

   - Block of code: `(x, y) -> { int sum = x + y; return sum; }`

5. Lambda expressions are commonly used with functional interfaces, which have exactly one abstract method.

6. Lambda expressions allow for more concise and expressive code, enabling functional programming concepts such as higher-order functions.

2) Can you pass lambda expressions to a method? When?

Ans:

Yes, in Java 8 and later versions, we can pass lambda expressions as arguments to methods. This is possible because lambda expressions can be used to create instances of functional interfaces.

Functional interfaces are interfaces that have a single abstract method. They serve as the type for lambda expressions and provide the target type for lambda expressions when passing them as method arguments.

Here are a few scenarios where you can pass lambda expressions to methods:

1. Callbacks: You can pass lambda expressions to methods that expect a functional interface as a callback. The lambda expression defines the behavior or action to be executed when a certain event occurs.

Example:

```
public void performAction(Runnable action) {
    // Perform some actions before invoking the callback
    action.run();
    // Perform some actions after invoking the callback
}

// Usage:
performAction(() -> System.out.println("Callback executed"));
```

2. Filters and Predicates: You can pass lambda expressions as predicates or filters to methods that perform filtering operations.

Example:

```
public List<Integer> filterList(List<Integer> numbers, Predicate<Integer> filter) {
    List<Integer> filteredList = new ArrayList<>();
    for (Integer number : numbers) {
        if (filter.test(number)) {
            filteredList.add(number);
        }
    }
    return filteredList;
```

```
}

// Usage:
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> evenNumbers = filterList(numbers, n -> n % 2 == 0);
```

3. Iterations and Actions: You can pass lambda expressions as actions or behaviors to methods that perform iterations or actions on a collection.

Example:

```
public void forEachElement(List<String> elements, Consumer<String> action) {
    for (String element : elements) {
        action.accept(element);
    }
}

// Usage:
List<String> fruits = Arrays.asList("Apple", "Banana", "Orange");
forEachElement(fruits, fruit -> System.out.println("Fruit: " + fruit));
```

3) What is a functional interface in Java 8?

Ans:

1. Definition: A functional interface in Java is an interface that has only one abstract method.

   - It can have any number of default or static methods.

   - The single abstract method represents the primary behavior of the interface.

2. Purpose: Functional interfaces serve as a foundation for functional programming in Java.

   - They enable the use of lambda expressions and method references to represent behavior as data.

   - They provide a way to treat functionality as a first-class citizen in the Java programming language.

3. Lambda Expressions: Functional interfaces are the target type for lambda expressions.

   - Lambda expressions can be used to create instances of functional interfaces concisely.

   - The lambda expression's parameter list and body must match the signature of the abstract method.

4. Built-in Functional Interfaces: Java 8 introduced several built-in functional interfaces in the `java.util.function` package.

   - These interfaces cover common use cases, such as predicates, consumers, suppliers, and functions.

   - Examples include `Predicate`, `Consumer`, `Supplier`, `Function`, and more.

5. Functional Programming Paradigm: Functional interfaces enable functional programming concepts.

   - They allow for higher-order functions, where functions can accept other functions as arguments or return functions as results.

   - Functional interfaces promote immutability, declarative style, and composability of functions.

4) Why do we use lambda expressions in Java ?

Ans:

Lambda expressions in Java are used for several reasons, as they bring significant benefits to the language. Here are some of the key reasons why lambda expressions are used in Java:

1. Concise and Readable Code: Lambda expressions provide a concise and expressive way to write code. They allow you to represent functionality as a compact and readable piece of code, reducing boilerplate and improving code clarity.

2. Functional Programming: Lambda expressions enable functional programming concepts in Java. They allow you to treat behavior as data, supporting higher-order functions, function composition, and functional programming patterns.

3. Enhanced API Support: Many APIs in Java 8 and later versions are designed to work seamlessly with lambda expressions. They provide functional interfaces as method parameters, allowing developers to pass lambda expressions as arguments, which results in more flexible and customizable behavior.

4. Improved Collections Framework: Lambda expressions greatly enhance the Collections Framework in Java. They provide a concise syntax for iterating, filtering, and transforming collections, making code involving collections more expressive and readable.

5. Concurrency and Parallelism: Lambda expressions facilitate concurrent and parallel programming in Java. They are often used with functional interfaces in Java's concurrency utilities, such as `java.util.concurrent` package, to write cleaner and more maintainable concurrent code.

6. Event Handling and Callbacks: Lambda expressions are commonly used for event handling and callbacks. They allow you to define the behavior or action to be executed when an event occurs, resulting in cleaner and more manageable event-driven code.

7. Improved Developer Productivity: By enabling shorter and more expressive code, lambda expressions can enhance developer productivity. They reduce the need for writing repetitive code, leading to quicker development cycles and easier code maintenance.

5) Is it mandatory for a lambda expression to have parameters?

Ans:

No, it is not mandatory for a lambda expression to have parameters. The presence or absence of parameters depends on the functional interface that the lambda expression is targeting.

If the functional interface represents a function that takes no arguments, the lambda expression can have empty parentheses. Here's an example:

```
Runnable myLambda = () -> {

    // Code to be executed

};
```

In this example, the `Runnable` interface represents a function that takes no arguments and returns no result. The lambda expression `() -> { // Code }` matches the signature of the `run()` method in the `Runnable` interface.