ASSIGNMENT

Q1 ans-

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;


public class CurrentDateTimeExample {
    public static void main(String[] args) {
        // Get the current date and time
        LocalDateTime currentDateTime = LocalDateTime.now();

        // Create a DateTimeFormatter to format the date and time as per your requirement
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

        // Format the current date and time using the formatter
        String formattedDateTime = currentDateTime.format(formatter);

        // Display the current date and time
        System.out.println("Current Date and Time: " + formattedDateTime);
    }
}
```

Q2 ans-

```
import java.text.SimpleDateFormat;
import java.util.Date;
```

```
public class DateToStringExample {
    public static void main(String[] args) {
        // Create a sample date
        Date date = new Date();

        // Define the desired date format
        String pattern = "yyyy-MM-dd HH:mm:ss";

        // Convert the date to string using SimpleDateFormat
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern);
        String formattedDate = simpleDateFormat.format(date);

        // Display the formatted date
        System.out.println("Formatted Date: " + formattedDate);
    }
}
```

Q3 ans-

`Collection` and `Stream` are two different concepts in Java that are related to handling collections of elements, but they serve different purposes and have distinct characteristics:

1. **Purpose**:
   - `Collection`: The `Collection` interface is a part of the Java Collections Framework (`java.util` package) and provides a way to store and manipulate a group of objects (elements) as a single unit. It includes various concrete classes like `ArrayList`, `LinkedList`, `HashSet`, etc., that implement different types of collections.
   - `Stream`: The `Stream` interface is a part of the Java Stream API (`java.util.stream` package) and is used for processing sequences of elements in a functional programming style. Streams allow you to perform operations on collections, such as filtering, mapping, and reducing, using functional programming constructs like lambdas.

2. **Data Handling**:

- `Collection`: Collections are used for storing and managing elements. They allow adding, removing, and accessing elements directly by their index or other methods.

- `Stream`: Streams are used for processing collections and performing aggregate operations on their elements. Streams do not store elements; instead, they provide a pipeline for processing data in a functional way.

3. **Mutability**:

- `Collection`: Collections are mutable, meaning you can add, remove, or modify elements in the collection.

- `Stream`: Streams are generally non-mutable, as they are designed for functional-style operations. Once created, a stream cannot be modified. Instead, you perform operations on a stream to produce a new stream or a terminal operation to get a result.

4. **Eager vs. Lazy Evaluation**:

- `Collection`: Collections follow eager evaluation, meaning elements are computed and stored in the collection at the time of creation or modification.

- `Stream`: Streams follow lazy evaluation, meaning intermediate operations are not executed until a terminal operation is called. This allows for potential optimizations and performance improvements, as only the required elements are processed.

5. **Terminal Operations**:

- `Collection`: Collections do not have special terminal operations. You generally use loops or other mechanisms to perform actions on elements in a collection.

- `Stream`: Streams have various terminal operations like `forEach`, `collect`, `reduce`, `min`, `max`, etc., that produce a result or a side effect after processing the elements.

In summary, `Collection` is used for storing and managing elements, providing direct access to elements and supporting mutation. On the other hand, `Stream` is used for functional-style processing of elements in a collection, providing a declarative and concise way to perform operations on the data. Streams are often used for data processing, filtering, mapping, and aggregating elements, promoting more readable and maintainable code compared to traditional loops over collections.

Q4 ans-

In Java, an `enum` is a special data type that represents a group of constants, also known as an enumeration. An enumeration is a fixed set of predefined named values, and these values are often referred to as "enumerators" or "enums." Enums provide a way to define a set of related symbolic constants, making the code more expressive, type-safe, and maintainable.

Enums were introduced in Java 5 to provide a more robust and type-safe alternative to using integer constants or String literals to represent a fixed set of values.

To define an enum in Java, you use the `enum` keyword followed by the name of the enum and the list of enum constants enclosed in curly braces. Each constant represents a distinct value of the enum, and the constant names are usually written in uppercase by convention.

Here's an example of defining and using an enum in Java:

```java
public class EnumExample {
    // Define an enum named "Color" with three constants: RED, GREEN, and BLUE
    enum Color {
        RED, GREEN, BLUE
    }

    public static void main(String[] args) {
        // Using enum constants
        Color myColor = Color.RED;

        // Switch statement with enum
        switch (myColor) {
            case RED:
                System.out.println("Selected color is RED.");
```

```
            break;
        case GREEN:
            System.out.println("Selected color is GREEN.");
            break;
        case BLUE:
            System.out.println("Selected color is BLUE.");
            break;
    }
  }
}
```

In this example, we define an enum named `Color` with three constants: `RED`, `GREEN`, and `BLUE`. We then create a variable `myColor` of type `Color` and assign it the value `Color.RED`. We use a `switch` statement to demonstrate how to work with the enum constants.

Enums are especially useful when you have a fixed set of related options or choices that you want to represent in your code. They help avoid common issues like typographical errors, improve code readability, and provide a more intuitive and self-documenting way to work with constant values.

Q5 ans -

In Java, there are several built-in annotations provided by the Java Standard Edition (SE) platform. These annotations serve various purposes and help developers communicate important information to the compiler, runtime, or other tools. Some of the commonly used built-in annotations in Java include:

1. **@Override**: This annotation is used to indicate that a method in a subclass is intended to override a method in its superclass. It helps catch errors at compile-time if the method does not actually override a method in the superclass.

2. **@Deprecated**: This annotation is used to mark a method, class, or field as deprecated, indicating that it is no longer recommended for use. It serves as a warning to developers to avoid using the marked element and to consider using an alternative.

3. **@SuppressWarnings**: This annotation is used to suppress specific warnings generated by the Java compiler. It allows developers to hide warnings that might be irrelevant or not critical for the code.

4. **@FunctionalInterface**: This annotation is used to specify that an interface is intended to be a functional interface, meaning it has a single abstract method. It is used for functional programming with lambda expressions and method references.

5. **@SafeVarargs**: This annotation is used to suppress unchecked warnings when using varargs (variable-length argument) methods. It is typically applied to methods that use generics and varargs together.

6. **@SuppressWarnings**: This annotation is used to suppress specific warnings generated by the Java compiler. It allows developers to hide warnings that might be irrelevant or not critical for the code.

7. **@Deprecated**: This annotation is used to mark a method, class, or field as deprecated, indicating that it is no longer recommended for use. It serves as a warning to developers to avoid using the marked element and to consider using an alternative.

8. **@FunctionalInterface**: This annotation is used to specify that an interface is intended to be a functional interface, meaning it has a single abstract method. It is used for functional programming with lambda expressions and method references.

9. **@SafeVarargs**: This annotation is used to suppress unchecked warnings when using varargs (variable-length argument) methods. It is typically applied to methods that use generics and varargs together.

10. **@SuppressWarnings**: This annotation is used to suppress specific warnings generated by the Java compiler. It allows developers to hide warnings that might be irrelevant or not critical for the code.

11. **@Deprecated**: This annotation is used to mark a method, class, or field as deprecated, indicating that it is no longer recommended for use. It serves as a warning to developers to avoid using the marked element and to consider using an alternative.

12. **@FunctionalInterface**: This annotation is used to specify that an interface is intended to be a functional interface, meaning it has a single abstract method. It is used for functional programming with lambda expressions and method references.

13. **@SafeVarargs**: This annotation is used to suppress unchecked warnings when using varargs (variable-length argument) methods. It is typically applied to methods that use generics and varargs together.

These are just a few examples of the built-in annotations available in Java. Java also allows developers to create custom annotations using the `@interface` keyword, providing a powerful mechanism for metadata and additional information in code. Annotations are extensively used in Java frameworks, libraries, and various tools for runtime processing and documentation generation.