**Assignment APIs and Annotation**

Q1.Program to display current date and time in Java

import java.time.LocalDateTime;

import java.time.format.DateTimeFormatter;

public class CurrentDateTime {

public static void main(String[] args) {

// Get current date and time

LocalDateTime currentDateTime = LocalDateTime.now();

// Format the date and time

DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

// Convert to string with formatting

String formattedDateTime = currentDateTime.format(format);

// Display the date and time

System.out.println("Current Date and Time: " + formattedDateTime);

}

}

Q2.Write a program to convert a date to a string in the format "MM/dd/yyyy"

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

public class DateToString {

public static void main(String[] args) {

// Create a LocalDate object (current date)

LocalDate currentDate = LocalDate.now();

// Define the format "MM/dd/yyyy"

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MM/dd/yyyy");

// Convert the date to string using the formatter

String formattedDate = currentDate.format(formatter);

// Display the formatted date

System.out.println("Formatted Date: " + formattedDate);

}

}

Q3. What is the difference between collections and streams? Explain with an Example

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Collections** | **Streams** |
| **Purpose** | Store and manipulate groups of objects. | Process and manipulate data in a functional style. |
| **Mutability** | Can be mutable (add, update, remove elements). | Immutable; doesn't modify the underlying source. |
| **Lazy Evaluation** | Not inherently lazy; operations are eager. | Inherently lazy; operations are evaluated on demand. |
| **Data Source** | Data stored in memory. | Can work with various sources (e.g., collections, arrays, files). |
| **Intermediate Operations** | Transformation operations on data. | Intermediate operations like map, filter, etc. |
| **Terminal Operations** | Actions that produce a result or a side effect. | Final operations like forEach, reduce, collect, etc. |
| **Parallelism** | Limited parallelism support (external iteration). | In-built parallelism support for improved performance. |
| **Memory Efficiency** | May consume more memory due to mutability. | Typically, more memory-efficient due to immutability. |
| **Use Cases** | Data storage, modification, traditional processing. | Data transformation, filtering, parallel processing. |

Examples:

Using **Collections** (Traditional Loop):

import java.util.ArrayList;

import java.util.List;

public class CollectionsExample {

public static void main(String[] args) {

List<Integer> numbers = new ArrayList<>();

numbers.add(3);

numbers.add(7);

numbers.add(2);

numbers.add(8);

numbers.add(5);

// Filtering using a loop (eager execution)

List<Integer> filteredNumbers = new ArrayList<>();

for (Integer number : numbers) {

if (number > 5) {

filteredNumbers.add(number);

}

}

// Display the filtered numbers

System.out.println("Filtered numbers (Collections): " + filteredNumbers);

}

}

Using **Streams**:

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class StreamsExample {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(3, 7, 2, 8, 5);

// Filtering using streams (lazy execution)

List<Integer> filteredNumbers = numbers.stream()

.filter(number -> number > 5) // Intermediate operation (lazy)

.collect(Collectors.toList()); // Terminal operation

// Display the filtered numbers

System.out.println("Filtered numbers (Streams): " + filteredNumbers);

}

}

Q4. What is enums in java? explain with an example

In Java, an **enum** (short for "enumeration") is a special data type that represents a group of constants (unchangeable variables, like final variables). Enums are used when we know all the possible values of a type at compile time, such as days of the week, directions (NORTH, SOUTH, EAST, WEST), months of the year, etc.

**Key Characteristics of Enums:**

* **Type-Safe**: Enums provide type safety. This means you can only assign valid predefined constants to a variable.
* **Fixed Set of Constants**: Enums define a fixed set of constants. You cannot add new values to an enum after it is defined.
* **Methods and Constructors**: Enums can have fields, methods, and constructors. They can also implement interfaces, though they can't extend classes as enums implicitly extend java.lang.Enum.

**Enum Syntax:**

enum EnumName {

CONSTANT1, CONSTANT2, CONSTANT3;

}

**Example of an Enum:**

Let's consider an example where we define an enum for **Days of the Week**:

// Defining an enum

enum Day {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;

}

public class EnumExample {

public static void main(String[] args) {

// Accessing enum constants

Day today = Day.WEDNESDAY;

// Using enum in switch-case

switch (today) {

case MONDAY:

System.out.println("Start of the work week.");

break;

case WEDNESDAY:

System.out.println("Midweek day!");

break;

case FRIDAY:

System.out.println("Almost the weekend!");

break;

case SUNDAY:

System.out.println("Relax, it's Sunday.");

break;

default:

System.out.println("Regular day.");

break;

}

// Looping through all enum constants

for (Day day : Day.values()) {

System.out.println(day);

}

}

}

Q5. What are in built annotations in java?

ava provides several **built-in annotations** that are used to provide metadata for code and influence the behavior of certain operations, such as compile-time checks, runtime behavior, or framework integration. These annotations can be categorized into different groups, such as those used for compiler instructions, runtime, and others.

**Key Built-in Annotations in Java:**

1. **@Override**
   * **Purpose**: Indicates that a method is intended to override a method declared in a superclass.
   * **Usage**: Used during compile-time to ensure that a method overrides a method in a superclass. If it does not, the compiler will show an error.

Ex: class Parent {

void display() {

System.out.println("Parent class method");

}

}

class Child extends Parent {

@Override

void display() {

System.out.println("Child class method");

}

}

**@Deprecated**

* **Purpose**: Marks a method, class, or field as deprecated, meaning it should no longer be used, and there's likely a better alternative available.
* **Usage**: It generates a warning when the annotated element is used in code. It is often used when methods are obsolete but remain in the codebase for backward compatibility.

@Deprecated

public void oldMethod() {

System.out.println("This method is deprecated.");

}

**@SuppressWarnings**

* **Purpose**: Instructs the compiler to suppress specific warnings that it would normally generate.
* **Usage**: Commonly used to suppress unchecked warnings, deprecated API warnings, or unused variable warnings.

@SuppressWarnings("unchecked")

public void someMethod() {

List rawList = new ArrayList(); // This will generate an unchecked warning, but it's suppressed

}

**@SafeVarargs**

* **Purpose**: Suppresses warnings about possible unsafe operations when working with generic types and variable arguments (varargs).
* **Usage**: Applicable to methods that use varargs with generics to avoid warnings about type safety. It can be applied to static methods, final methods, and constructors.

@SafeVarargs

public static <T> void printArray(T... elements) {

for (T element : elements) {

System.out.println(element);

}

}

**@FunctionalInterface**

* **Purpose**: Marks an interface as a functional interface. A functional interface has exactly one abstract method and can be used in lambda expressions or method references.
* **Usage**: The compiler will enforce that the interface only contains one abstract method.

@FunctionalInterface

interface MyFunctionalInterface {

void doSomething();

}

**@Inherited**

* **Purpose**: Indicates that an annotation type is automatically inherited by subclasses of a class. It applies only to annotations that are applied to class declarations.
* **Usage**: If a superclass is annotated with an annotation marked with @Inherited, the subclass will inherit that annotation.

import java.lang.annotation.Inherited;

@Inherited

@interface MyInheritedAnnotation {

}

@MyInheritedAnnotation

class ParentClass {

}

class ChildClass extends ParentClass {

}

In short :  **@Override**: Ensures a method is overriding a superclass method.

 **@Deprecated**: Marks a method or class as deprecated.

 **@SuppressWarnings**: Suppresses specific compiler warnings.

 **@SafeVarargs**: Suppresses warnings for unsafe generic varargs.

 **@FunctionalInterface**: Marks an interface as a functional interface.

 **@Retention**: Controls the retention policy of an annotation.

 **@Target**: Specifies where an annotation can be applied.

 **@Inherited**: Allows annotations to be inherited by subclasses.

 **@Documented**: Ensures the annotation appears in Javadoc.