1) Program to display current date and time in Java?

Ans:

```
import java.util.Date;

public class DateTimeDisplay {
    public static void main(String[] args) {
        // Create a Date object representing the current date and time
        Date currentDate = new Date();

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

2) Write a program to convert a date to a string in the format “MM/dd/yyyy”.

Ans:

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

public class DateToStringConversion {
    public static void main(String[] args) {
        // Create a Date object representing the date you want to convert
        Date date = new Date();  // You can replace this with your desired date

        // Define the desired format
        SimpleDateFormat dateFormat = new SimpleDateFormat("MM/dd/yyyy");

        // Convert the date to the desired string format
        String dateString = dateFormat.format(date);

        // Display the converted date string
        System.out.println("Converted Date String: " + dateString);
    }
}
```

3) What is the difference between collections and streams? Explain with an example.

Ans:

Difference between Collections and Streams:

1. Eager vs. Lazy Evaluation:
   - Collections are evaluated eagerly. When you retrieve elements from a collection, it loads all elements into memory at once.
   - Streams are evaluated lazily. Operations on streams are only performed when needed, potentially allowing for more efficient processing.

2. Mutability:
   - Collections are mutable, meaning you can modify their content directly.
   - Streams are usually not mutable; they represent a sequence of data that you transform without changing the original data source.

3. Parallelism:
   - Streams support parallelism, allowing operations to be executed in parallel threads for better performance.
   - Most traditional collections do not provide built-in support for parallel processing.

4. Examples

Example using a List collection:

```
import java.util.ArrayList;
import java.util.List;

public class CollectionExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        System.out.println("Names in the collection: " + names);
    }
}
```

Example using a Stream to filter and transform data:

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

        List<String> filteredNames = names.stream()
            .filter(name -> name.length() > 4)  // Filter names with length > 4
            .map(name -> name.toUpperCase())    // Convert names to uppercase
            .collect(Collectors.toList());      // Collect the result into a list

        System.out.println("Filtered and transformed names: " + filteredNames);
    }
}
```

4) What is enums in Java? Explain with an example

Ans:

- Enums in Java are a way to define a type that represents a fixed set of constants.
- Enums provide better type safety compared to using plain integers or strings for representing constant values.
- Enum constants are defined within an enum declaration using a comma-separated list.
- Enums are often used to represent things like days of the week, months, states, options, etc.

Example:

```
enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}

public class EnumExample {
    public static void main(String[] args) {
        Day today = Day.WEDNESDAY;

        switch (today) {
            case SUNDAY:
                System.out.println("Today is Sunday!");
```

```
            break;
        case MONDAY:
            System.out.println("Today is Monday.");
            break;
        case WEDNESDAY:
            System.out.println("Today is Wednesday.");
            break;
        default:
            System.out.println("It's some other day.");
    }
  }
}
```

5) What are in-built annotations in Java?

Ans:

In Java, annotations are used to provide metadata about code, classes, methods, variables, and other program elements.

Annotations are used for a variety of purposes, including documentation, code generation, and runtime behaviour specification.

Java provides several built-in annotations that serve specific purposes.

1. @Override:
   - Indicates that a method in a subclass is intended to override a method in its superclass.
   - Helps catch errors at compile-time if the method signature doesn't match the overridden method.

2. @Deprecated:
   - Marks a method, class, or field as deprecated, indicating that it is no longer recommended to use.
   - Helps communicate to developers that certain elements are no longer the preferred way of accomplishing a task.

3. @SuppressWarnings:
   - Suppresses compiler warnings for specific types of warnings.
   - Useful when you know that the code in question is safe despite the warning.

4. @FunctionalInterface:
   - Indicates that an interface is a functional interface, meaning it has exactly one abstract method.
   - Used for interfaces that can be used with lambda expressions or method references.

5. @SafeVarargs:
   - Indicates that a method is safe to call with a varargs parameter (variable-length argument list) without causing unchecked warnings.
   - Helps avoid warnings related to varargs usage.

6. @NonNull, @Nullable:
   - These annotations provide nullability information about method parameters, return values, and fields.
   - Useful for tools that perform static analysis to catch potential null pointer exceptions.

7. @Documented:
   - Indicates that the annotated element should be included in the generated API documentation.
   - Ensures that the annotations are visible in the generated documentation.

8. @SuppressWarnings:
   - Suppresses specific compiler warnings on the annotated element.
   - Useful when you want to suppress warnings for specific situations.

9. @Inherited:
    - Specifies that an annotation declared on a superclass should also be inherited by its subclasses.
    - The annotation is inherited by subclasses unless they override it.