Oracle released a new version of Java as Java 8 in March 18, 2014. It was a revolutionary release of the Java for software development platform. It includes various upgrades to the Java programming, JVM, Tools and libraries.

Java 8 Programming Language Enhancements

Java 8 provides following features for Java Programming:

- Lambda expressions,
- Functional interfaces,
- Stream API,
- Default methods,
- Static methods in interface,
- Optional class,
- Collectors class,
- ForEach() method,
- …. Many more

Java Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.
The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

Functional Interface

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface.

Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.
Why use Lambda Expression
To provide the implementation of Functional interface.
Less coding.

Java Lambda Expression Syntax
(argument-list) -> {body}
Java lambda expression is consisted of three components.
1) Argument-list: It can be empty or non-empty as well.
2) Arrow-token: It is used to link arguments-list and body of expression.
3) Body: It contains expressions and statements for lambda expression.
No Parameter Syntax

```
() -> {
//Body of no parameter lambda
}
(p1) -> {
//Body of single parameter lambda
}
```

```
interface Drawable{
    public void draw();
}
public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;

        //without lambda, Drawable implementation using anonymous class
        Drawable d=new Drawable(){
            public void draw(){System.out.println("Drawing "+width);}
```

```
        };
        d.draw();
    }  }

@FunctionalInterface  //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}
```

Predicates

A predicate is a function with a single argument and returns boolean value.

```
public class PredicateDemo {

        public static void main(String[] args) {
                Predicate<String> p = (s) -> s.equalsIgnoreCase("up");
                System.out.println(p.test("up"));
        }

}
```

Q)Write a predicate to check the given number even or odd ?

Supplier

Supplier won't take any input and it will always supply objects.
Supplier Functional Interface contains only one method get().

```
public class SupplierDemo {

	public static void main(String[] args) {

		Supplier<Employee> s = () -> {
			System.out.println("Hello");
			return new Employee();
		};
		System.out.println(s.get());
	}

}

class Employee {
	int id;

	@Override
	public String toString() {
		return "Employee [id=" + id + "]";
	}

}
```

Supplier won't take any input and it will always supply objects

```
public class SupplierDemo {

	public static void main(String[] args) {

		Supplier<Integer> s = () -> {
			System.out.println("Hello");
```

```
                return (int)(Math.random()*1000);};
            System.out.println(s.get());
        }

}
```

Random Password generator.

Consumer

```
public class ConsumerDemo {

        public static void main(String[] args) {
            List<String> stringList = Arrays.asList("Hi", "Hello");

            Consumer<List<String>> c = (s) -> {
                s.forEach(k -> System.out.println(k));
                ;
            };

            c.accept(stringList);
            Consumer<String> c1=(s) -> {
                System.out.println(s);
                System.out.println(s.toUpperCase());
            };

            c1.accept("up");
        }

}
```

Functions

Functions are exactly same as predicates except
 that functions can return any type of result but
 function should (can) return only one value
 and that value can be any type as per our requirement.

```
public class FunctionDemo {

	public static void main(String[] args) {

		Function<String, String> f=(s) -> s.toLowerCase();

		System.out.println(f.apply("AB"));

	}

}
```

Method and Constructor references by using ::(double colon) operator

Functional Interface method can be mapped to our specified method by using :: (double colon) operator. This is called method reference

Our specified method can be either static method or instance method. Functional Interface method and our specified method should have same argument types, except this the remaining things like returntype, methodname, modifiersetc are not required to match.

Classname::methodName
if the method is instance method
Objref::methodName

Constructor References
We can use :: (double colon)operator to refer constructors also

Syntax: classname :: new

```
package com;

public class ConstructorReferenceDemo {

	public static void main(String[] args) {
		EmployeeService es=Employee::new;
		Employee employee = es.getEmployee();
		System.out.println(employee.id=100);
	}

}



interface EmployeeService{

	Employee getEmployee();
}
```

Streams

To process objects of the collection, in 1.8 version Streams concept introduced

Java Streams provide a powerful and concise way to work with collections and perform various operations such as filtering, mapping, and reducing. Streams allow you to process data in a functional style, which often leads to more readable and maintainable code.

Let's consider an example where we have a list of integers and we want to perform some operations on them using Java Streams.

```java
```

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

public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Example 1: Filter and Print
        System.out.println("Even numbers:");
        numbers.stream()
               .filter(n -> n % 2 == 0)
               .forEach(System.out::println);

        // Example 2: Map and Collect
        List<String> squaredStrings = numbers.stream()
                                             .map(n -> n * n)
                                             .map(Object::toString)
                                             .collect(Collectors.toList());

        System.out.println("Squared numbers as strings: " +
squaredStrings);

        // Example 3: Reduce
        int sum = numbers.stream()
                         .reduce(0, Integer::sum);

        System.out.println("Sum of numbers: " + sum);
    }
}
```

Explanation of the examples:

1. **Filter and Print:**
   - We start by creating a stream from the list of integers using `numbers.stream()`.

- We then use the `filter` operation to keep only the even numbers (`n % 2 == 0`).
- Finally, we use the `forEach` terminal operation to print each even number.

2. **Map and Collect:**
- Here, we create a stream and use the `map` operation twice:
  - First, we square each number with `n -> n * n`.
  - Then, we convert each squared number to its string representation using `Object::toString`.
- The `collect` terminal operation gathers the mapped elements into a new list.

3. **Reduce:**
- The `reduce` operation allows us to perform a reduction on the elements of the stream.
- In this example, we calculate the sum of all numbers using `Integer::sum` as the binary operator and starting with an initial value of 0.

Java Streams provide a declarative and functional way to process data, making code more concise and easier to read. Keep in mind that streams also support parallel processing, which can lead to performance improvements for large datasets when used correctly.

Java 8 introduced a new and improved Date and Time API that addresses the shortcomings of the older `java.util.Date` and `java.util.Calendar` classes. The new API is more flexible, easier to use, and provides better support for handling date and time-related operations. Here's a detailed explanation of the Java 8 Date and Time API:

1. **Key Classes:**
- `LocalDate`: Represents a date without a time component, such as "2023-08-16".

- `LocalTime`: Represents a time without a date component, such as "15:30:00".
- `LocalDateTime`: Represents a date and time without time zone information.
- `ZonedDateTime`: Represents a date and time along with time zone information.
- `Duration`: Represents a duration of time.
- `Period`: Represents a period of time in terms of years, months, and days.

2. **Instantiation and Usage:**
- You can create instances of `LocalDate`, `LocalTime`, and `LocalDateTime` using static factory methods or by parsing strings.
- `ZonedDateTime` can be created using a `ZoneId` and `Instant`.
- You can perform arithmetic operations like adding or subtracting time using methods like `plus`, `minus`, and `plusDays`.
- `Duration` and `Period` can be used for more precise time and date arithmetic.

3. **Formatting and Parsing:**
- You can format date and time objects using the `DateTimeFormatter` class.
- `DateTimeFormatter` provides patterns for formatting and parsing, and you can create custom formats.

4. **Time Zones:**
- The `ZoneId` class represents a time zone.
- `ZonedDateTime` allows you to work with date and time in specific time zones.

5. **Comparisons and Adjustments:**
- Comparisons and adjustments of date and time are straightforward using the new API.
- Methods like `isBefore`, `isAfter`, and `equals` are available for comparing dates and times.
- Adjustments like setting specific fields or truncating to a specific unit are supported.

6. **Immutable:**
   - The new Date and Time API is designed to be immutable, which helps prevent unexpected modifications.

7. **Backward Compatibility:**
   - While the new API is more powerful, the old `java.util.Date` and `java.util.Calendar` classes can be converted to the new API using utility methods.

Here's an example demonstrating the usage of the Java 8 Date and Time API:

```java
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.ZonedDateTime;
import java.time.Duration;
import java.time.Period;
import java.time.format.DateTimeFormatter;

public class DateTimeExample {
    public static void main(String[] args) {
        // LocalDate
        LocalDate date = LocalDate.now();
        System.out.println("Today's date: " + date);

        // LocalTime
        LocalTime time = LocalTime.now();
        System.out.println("Current time: " + time);

        // LocalDateTime
        LocalDateTime dateTime = LocalDateTime.now();
        System.out.println("Current date and time: " + dateTime);

        // Formatting and Parsing
```

```
        DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        String formattedDateTime = dateTime.format(formatter);
        System.out.println("Formatted date and time: " +
formattedDateTime);

        LocalDateTime parsedDateTime =
LocalDateTime.parse("2023-08-16 15:30:00", formatter);
        System.out.println("Parsed date and time: " + parsedDateTime);

        // Duration
        Duration duration = Duration.ofHours(2);
        System.out.println("Duration: " + duration);

        // Period
        Period period = Period.ofMonths(3);
        System.out.println("Period: " + period);

        // ZonedDateTime
        ZonedDateTime zonedDateTime = ZonedDateTime.now();
        System.out.println("Current date and time with time zone: " +
zonedDateTime);
    }
}
```

The Java 8 Date and Time API simplifies many common date and time-related tasks and provides a more intuitive and consistent way of working with temporal data. It's recommended to use this API for any new Java projects that involve date and time operations.
