**1.Lambda expression :**

The Lambda expression is used to provide the implementation of an interface that has a functional interface. It saves a lot of code. In the case of the lambda expression, we don't need to define the method again for providing the implementation. Here, we write the implementation code.

Lambda expression provides implementation of *functional interface*. An interface which has only one abstract method is called functional interface. Java provides an annotation **@*FunctionalInterface***, which is used to declare an interface as functional interface.

**Functional Interface**

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

**2.Java Method References :**

In Java 8 interfaces underwent significant changes with the introduction of default and static methods. These changes were made to enhance the flexibility and functionality of interfaces without breaking existing code.

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

For information <https://medium.com/@JavaFusion/exploring-java-8s-interface-revolution-default-methods-and-static-methods-2ff8df284128>

**Default Method**

A default method is declared in an interface using the default keyword followed by the method signature and implementation.

**Static Methods**

Static methods in interfaces can be called directly on the interface without requiring an instance of a class implementing the interface. These methods are commonly used for utility functions related to the interface.

Syntax : ContainingClass::staticMethodName

**Reference to an Instance Method**

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax : containingObject::instanceMethodName

**Reference to a Constructor**

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax : ClassName::**new**

**3.Stream API**

the Stream API is a powerful and flexible way to process collections of data. It allows you to perform operations on a sequence of elements, such as**filtering, mapping, sorting, and aggregating***,* without directly modifying the underlying collection.

**1.Intermediate Operations:** These operations transform a stream into another stream. Examples are **filter, map, sorted, distinct, limit, and skip**.

* **map(Function<T, R>):** Transforms each element of the Stream into another form using the provided function.
* **filter(Predicate<T>):** Selects elements from the Stream based on a specified condition.
* **flatMap(Function<T, Stream<R>>):** Transforms each element into zero or more elements by applying a function that returns a stream for each element.
* **distinct():** Removes duplicate elements from the Stream.
* **sorted():** Sorts the elements of the Stream.
* **limit(long n):** Truncates the Stream to be no longer than the specified size.
* **skip(long n):** Skips the first n elements of the Stream.
* **peek(Consumer<T>):** Performs a specified action on each element of the Stream without consuming the elements.

**2.Terminal Operations**:

These operations produce a result or side effect and terminate the stream. Examples **forEach, collect, reduce, count, anyMatch, allMatch, and noneMatch**.

* **forEach(Consumer<T>):** Acts as each element of the Stream.
* **collect(Collector<T, A, R>):** Reduces the elements of the Stream into a mutable result container, such as a list or a map.
* **reduce(BinaryOperator<T>):** Reduces the elements of the Stream to a single value using an associative accumulation function.
* **count():** Returns the count of elements in the Stream.
* **anyMatch(Predicate<T>):** Returns true if any element of the Stream matches the given predicate.
* **allMatch(Predicate<T>):** Returns true if all elements of the Stream match the given predicate.
* **noneMatch(Predicate<T>):** Returns true if no elements of the Stream match the given predicate.
* **findFirst():** Returns an Optional describing the first element of the Stream, or an empty Optional if the Stream is empty.
* **findAny():** Returns an Optional describing some element of the Stream, or an empty Optional if the Stream is empty.

**Functional Programming**: Streams encourage a functional programming style, where you chain together operations to process data without mutable state or side effects.

**Parallel Execution:** Streams can be processed in parallel using the parallel() method, which can significantly improve performance for operations that can be parallelized.

**4.Short-Circuit Operations**

Short-circuit operations are a subset of terminal operations that do not need to process the entire Stream to produce a result. They can provide an early exit from the stream processing pipeline, potentially saving computation time.

* **anyMatch(Predicate<T>):** Stops processing and returns true if any element matches the given predicate.
* **allMatch(Predicate<T>):** Stops processing and returns false if any element does not match the given predicate.
* **noneMatch(Predicate<T>):** Stops processing and returns true if no elements match the given predicate.
* **findFirst():** Returns the first element encountered in the Stream and then stops processing.
* **findAny():** Returns any element encountered in the Stream and then stops processing.

**4.Java Stream.filter() Method**

The filter() method is an intermediate operation that takes a Predicate (a functional interface) as an argument. A Predicate is a functional interface with a single abstract method, test, which returns a boolean value. The filter() method processes each element of the stream and includes it in the result only if the Predicate evaluates to true.

**5. Optional Class**

The **Optional** class in Java 8 is a container class introduced to represent a value that might or might not be present. It is a part of the java.util package and provides a way to avoid null pointer exceptions by explicitly handling the absence of a value.

Optional acts as a wrapper for an object that can either contain a non-null value or be empty. It provides methods to check, retrieve, or process the contained value without directly working with null values.

Moreinfo :<https://medium.com/@ucgorai/java-8-features-that-every-programmer-must-know-39e73a1daafb>

**Benefits of the**Optional**Class**

1. **Avoids NullPointerExceptions**:  
   Reduces the likelihood of runtime exceptions by ensuring null checks are explicitly handled.
2. **Improves Code Readability**:  
   Makes the intent clear by explicitly signaling the possibility of an absent value.
3. **Encourages Functional Programming**:  
   Works well with Java 8’s functional constructs like streams and lambda expressions.
4. **Reduces Boilerplate Code**:  
   Simplifies common patterns like checking for null values and returning defaults.
5. **Makes APIs More Robust**:  
   Enables clear and explicit contracts in APIs about when values might be optional.

**Key Methods of the**Optional**Class**

* **of(value)**: Creates an Optional containing the specified non-null value.
* **ofNullable(value)**: Creates an Optional that can contain a value or be empty if the value is null.
* **empty()**: Returns an empty Optional.
* **isPresent()**: Returns true if the value is present, otherwise false.
* **ifPresent(Consumer)**: Executes a consumer if the value is present.
* **orElse(value)**: Returns the value if present; otherwise, returns a default value.
* **orElseGet(Supplier)**: Returns the value if present; otherwise, invokes a supplier to provide a value.
* **orElseThrow(Supplier)**: Returns the value if present; otherwise, throws an exception.
* **map(Function)**: Applies a function to the value if present and returns a new Optional.
* **flatMap(Function)**: Similar to map, but avoids nesting of Optional.
* **filter(Predicate)**: Returns an Optional if the value satisfies the predicate.

**7. Date and Time API**

Java 8 introduced a modern **Date and Time API** in the java.time package to address the shortcomings of the previous java.util.Date and java.util.Calendar classes. The new API is inspired by the **Joda-Time library** and provides a comprehensive, robust, and user-friendly approach to handle date and time.

Moreinfo : <https://medium.com/@ucgorai/java-8-features-that-every-programmer-must-know-39e73a1daafb>

The key classes of the java.time package are:

* **LocalDate**: Represents a date without time (e.g., 2024-12-03).
* **LocalTime**: Represents a time without a date (e.g., 14:30:15).
* **LocalDateTime**: Represents a combination of date and time without timezone.
* **ZonedDateTime**: Represents a date-time with a timezone.
* **Instant**: Represents a specific point on the timeline (typically used for timestamps).
* **Duration** and **Period**: Represent time-based and date-based amounts, respectively.
* **DateTimeFormatter**: Used for formatting and parsing dates and times.

**Benefits of the Date and Time API in Java 8**

1. **Immutability**:  
   All classes in the new API are immutable and thread-safe, ensuring consistency and preventing unexpected behavior in concurrent environments.
2. **Clearer API Design**:  
   The new API offers intuitive and clear methods for manipulating dates and times, avoiding the confusion of java.util.Date and java.util.Calendar.
3. **Seamless Timezone Handling**:  
   The ZonedDateTime class simplifies handling of time zones.
4. **Fluent API**:  
   Provides a fluent and chainable API for performing operations like adding, subtracting, or formatting dates and times.
5. **Separation of Concerns**:  
   Different classes for date (LocalDate), time (LocalTime), and date-time (LocalDateTime) improve clarity.
6. **Built-in Support for ISO Standards**:  
   Default formats align with ISO-8601 standards, reducing errors in date-time parsing.
7. **Easy Conversion**:  
   Provides methods to convert between legacy and modern date-time classes.