JAVA 8 NEW FEATURES

DEFAULT METHODS IN INTERFACES

- When Java team decided to provided lot of features that supports lambda, functional programming by updating the collections framework, they got a problem. If they add new abstract methods inside the interfaces/abstract classes all the classes implementing them has to updated as per the requirements and this will effect all the teams using Java.

- To overcome this and provide backward compatibility even after adding new features inside Interfaces, Java team allowed concrete method implementations inside Interfaces which are called as default methods.

- Default methods are also known as defender methods or virtual extension methods.

- Just like regular interface methods, default methods are also implicitly public.

- Unlike regular interface methods, default methods will be declared with default keyword at the beginning of the method signature along with the implementation code.

JAVA 8 NEW FEATURES

DEFAULT METHODS IN INTERFACES

- Sample default method inside an Interface,

```
public interface ArithmeticOperation {
        default void method1() {
                System.out.println("This is a sample default method inside Interface");
        }
}
```

- Interface default methods are by-default available to all the implementation classes. Based on requirement, implementation class can use these default methods with default behavior or can override them.

- We can't override the methods present inside the Object class as default methods inside a interface. The compiler will throw errors if we do so.

- We can't write default methods inside a class. Even in the scenarios where we are overriding the default keyword should not be used inside the class methods.

JAVA 8 NEW FEATURES

DEFAULT METHODS IN INTERFACES

- The most typical use of default methods in interfaces is to incrementally provide additional features/enhancements to a given type without breaking down the implementing classes.

- What happens when a class implements multiple Interfaces which has similar default methods defined inside them?

```
public interface A {
        default void m1() { // }
}

public interface B {
        default void m1() { // }
}

public class C implements A,B {
        ??????
}
```

✓ *This will create ambiguity to the compiler and it will throw an error. We also call it as Diamond problem.*

✓ *To solve it we must provide implementation of method m1() inside your class either with your own logic or by invoking one of the interface default method.*

```
@Override
public void m1() {
     // Custom implementation OR
    //  A.super.m1() OR
    //  B.super.m1() OR
}
```

JAVA 8 NEW FEATURES

INTERFACE WITH DEFAULT METHODS Vs ABSTRACT CLASS

INTERFACE WITH DEFAULT	ABSTRACT CLASS
Interfaces with default methods can't have a constructor, state and behavior	Abstract classes can have a constructor, state and behavior
There is no chance for creating instance variables	Instance variables can be created inside abstract classes
Static and instance blocks are not allowed	Static and instance blocks are allowed
They can be used to write lambda expressions if they have only abstract method inside them	Can't be leveraged to write lambda expressions
Inside them we can't override Object class methods	Inside them we can override Object class methods

JAVA 8 NEW FEATURES

STATIC METHODS IN INTERFACES

- From Java 8, just like we can write default methods inside interfaces, we can also write static methods inside them to define any utility functionality.

- Since static methods don't belong to a particular object, they are not available to the classes implementing the interface, and they have to be called by using the interface name preceding the method name.

- Defining a static method within an interface is similar to defining one in a class.

- Static methods in interfaces make possible to group related utility methods, without having to create artificial utility classes that are simply placeholders for static methods.

- Since interface static methods by default not available to the implementation class, overriding concept is not applicable.

- Based on our requirement we can define exactly same method in the implementation class, it’s valid but not overriding.

JAVA 8 NEW FEATURES

STATIC METHODS IN INTERFACES

eazy bytes

```java
public interface A {
     public static void sayHello() {
       System.out.println("Hi, This is a static method inside Interfaces");
     }
}

public class B implements A {
    private static void sayHello() {
       System.out.println("Hi, This is a static method inside class");
    }

    public static void main(String[] args) {

        B b = new B();
        b.sayHello();
         B.sayHello();
         A.sayHello();
    }

}
```

- ✓ Since static methods are allowed from Java 8, we can write a main method inside an interface and execute it as well.

JAVA 8 NEW FEATURES

OPTIONAL TO DEAL WITH NULLS

eazy bytes

- If I ask a question to all the developers in the world to raise their hand if they have seen NullPointerException in their code, there might be no one who will not raise their hand ☺

- We as developers spend good amount of time fixing or caring for the NullPointerException in our code and it is a very painful or tedious process always. Below is one of the sample code where we can see an null pointer exception,

- In the above code we may get null pointer exception at any instance like while accessing the order from user/ item from order/name from item in case if they are null values. If we have to handle them we will end up writing code like below,

```java
public String getUserOrderDetails(User user) {
    return user.getOrder().getItem().getName();
}
```

→

```java
public String getUserOrderDetails(User user) {
    if (user != null) {
        Order order = user.getOrder();
        if (order != null) {
            Item item = order.getItem();
            if (item != null) {
                return item.getName();
            }
        }
    }
    return "Not Available";
}
```

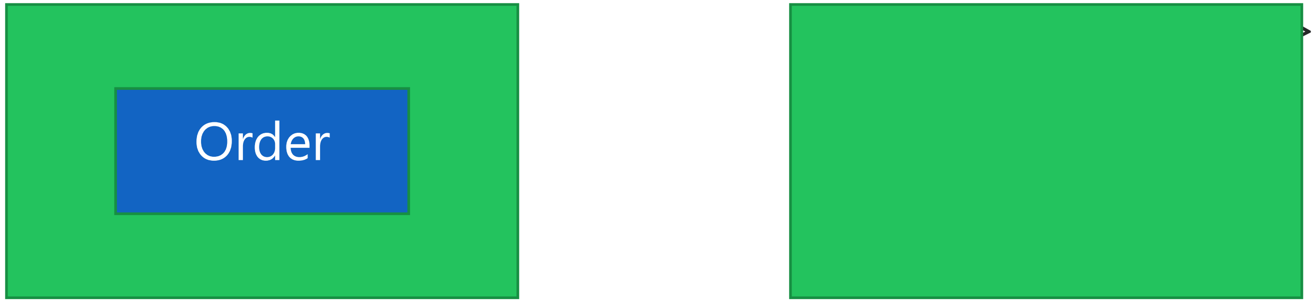
JAVA 8 NEW FEATURES

OPTIONAL TO DEAL WITH NULLS

- Sometimes we may know based on our business domain knowledge that particular objects can never be null but every time you doubt that a variable could be null, you're obliged to add a further nested if block ☹

- To handle the challenges with null values, Java 8 introduces java.util.Optional<T>

- An important, practical semantic difference in using Optionals instead of nulls is that in the first case, declaring a variable of type Optional<Order> instead of Order clearly signals that a missing value is permitted there. Otherwise you always depends on the business domain knowledge of the user.



- ✓ *When a value is present, the Optional class wraps it if not the absence of a value is modeled with an empty optional returned by the method Optional.empty()*

JAVA 8 NEW FEATURES

OPTIONAL TO DEAL WITH NULLS

- You may wonder about the difference between a null reference and Optional.empty(). Semantically, they could be seen as the same thing, but in practice, the difference is huge. Trying to access a null causes a NullPointerException, whereas Optional.empty() is a valid, workable object of type Optional that can be invoked in useful ways.

- It's important to note that the intention of the Optional class isn't to replace every single null reference. Instead, its purpose is to help you design more-comprehensible APIs so that by reading the signature of a method, you can tell whether to expect an optional value.

✓ *Optional<Product> optProd = Optional.empty(); // Creating an empty optional*
✓ *Optional<Product> optProd = Optional.of(product); // Optional from a non-null value*
✓ *Optional<Product> optProd = Optional.ofNullable(product); // If product is null, the resulting Optional object would be empty*

Advantages of Optional

✓ *Null checks are not required.*
✓ *No more NullPointerException at run-time.*
✓ *We can develop clean and neat APIs.*
✓ *No more Boiler plate code*

JAVA 8 NEW FEATURES

OPTIONAL TO DEAL WITH NULLS

eazy bytes

- Important methods provided by Optional in Java 8,

  - ✓ *empty - Returns an empty Optional instance*
  - ✓ *filter - If the value is present and matches the given predicate, returns this Optional; otherwise, returns the empty one*
  - ✓ *isPresent - Returns true if a value is present; otherwise, returns false*
  - ✓ *ifPresent - If a value is present, invokes the specified consumer with the value; otherwise, does nothing*
  - ✓ *get - Returns the value wrapped by this Optional if present; otherwise, throws a NoSuchElementException*
  - ✓ *map - If a value is present, applies the provided mapping function to it*
  - ✓ *orElse - Returns the value if present; otherwise, returns the given default value*
  - ✓ *orElseGet - Returns the value if present; otherwise, returns the one provided by the given Supplier*
  - ✓ *orElseThrow - Returns the value if present; otherwise, throws the exception created by the given Supplier*

LAMBDA (Λ) EXPRESSION

- Java was designed in the 1990s as an object-oriented programming language, when object-oriented programming was the principal paradigm for software development. Recently, functional programming has risen in importance because it is well suited for concurrent and event-driven (or "reactive") programming. That doesn't mean that objects are bad. Instead, the winning strategy is to blend object-oriented and functional programming.

- The main objective of Lambda Expression which is introduced in Java 8 is to bring benefits of functional programming into Java.

- A "lambda expression" is a block of code that you can pass around so it can be executed later, once or multiple times. It is an anonymous (nameless) function. That means the function which doesn't have the name, return type and access modifiers.

- It enable to treat functionality as a method argument, or code as data.

- Lambda has been part of Java competitive languages like Python, Ruby already.

eazy bytes

- Examples of Lambda programming

✓
```
public void printHello() {
      System.out.println("Hello");     =====➔     () -> {
}                                                        System.out.println("Hello");
                                                    }
```

✓
```
public void printHello() {
      System.out.println("Hello");     =====➔     () ->  System.out.println("Hello");
}
```

Note: When we have a single line of code inside your lambda method, we can remove the {} surrounding it to make it more simple.

eazy bytes

- Examples of Lambda programming

✓
```
public void printInput(String input) {
      System.out.println(input);          ======➔          (input) -> {
}                                                               System.out.println(input);
                                                            }
```

✓
```
public void printInput(String input) {
      System.out.println(input);          ======➔          input ->  System.out.println(input);
}
```

Note: When we have a single input parameter inside your lambda method, we can remove the () surrounding it to make it more simple.

eazy bytes

- Examples of Lambda programming

✓

```
public void add(int a, int b) {
    int res = a+b;
    System.out.println(res);
}
```

=====➔

```
(int a, int b) -> {
    int res = a+b;
    System.out.println(res);
}
```

✓

```
public void add(int a, int b) {
    int res = a+b;
    System.out.println(res);
}
```

=====➔

```
(a, b) -> {
    int res = a+b;
    System.out.println(res);
}
```

Note: The compiler can detect the input parameters data type based on the abstract method. But you can still mention them inside your lambda code and these are optional in nature.

JAVA 8 NEW FEATURES

LAMBDA (Λ) EXPRESSION

eazy bytes

- Examples of Lambda programming

✓
```
public int add(int a, int b) {
      int res = a+b;
     return  res;
}
```
=====➔ (a, b) -> return a+b;

✓
```
public int add(int a, int b) {
      int res = a+b;
     return  res;
}
```
=====➔ (a, b) -> a+b;

Note: If we have a single line of code inside your lambda method then return keyword is optional. We can remove it and compiler can interpret that the outcome of the statement should be return.

eazy bytes

- Lambda expressions are used heavily inside the Collections, Streams libraries from Java 8. So it is very important to understand them.

- Java Lambda Expression Syntax,

(argument-list) -> {body}

- With the help of Lambda expressions we can reduce length of the code, readability will be improved and the complexity of anonymous inner classes can be avoided.

- We can provide Lambda expressions in the place of object and method arguments.

- In order to write lambda expressions or code we need Functional interfaces.

JAVA 8 NEW FEATURES

LAMBDA (Λ) EXPRESSION

- Lambda expressions can use variables defined in an outer scope. They can capture static variables, instance variables, and local variables, but only local variables must be final or effectively final.

- A variable is final or effectively final when it's initialized once and is never mutated in its owner class; we can't initialize it in loops or inner classes.

```java
public class AnanymousVsLambda {

    int sum = 0;

    public void sum() {
        int tempSum = 0;
        ArithmeticOperation sumOperation = (a,b)-> {
            int sum = 0;
            tempSum = 0; //Compilation error
            this.sum = a+b;
            System.out.println("The value of sum inside lambda is: "+sum);
            return this.sum;
        };
        System.out.println("The sum of given 2 numbers is: "+sumOperation.
                performOperation(10, 20));
    }
```

✓ *Inside lambda code or blocks, we can't modify the local variables present outside the lambda blocks.*

✓ *There is a reason for this restriction. Mutating variables in a lambda expression is not thread safe.*

✓ *But we can update the static or member variables present inside the class. Because they stored on the heap, but local variables are on the stack. Because we're dealing with heap memory, the compiler can guarantee that the lambda will have access to the latest value always.*

JAVA 8 NEW FEATURES

ANANYMOUS INNER CLASS Vs LAMBDA EXPRESSION

ANANYMOUS CLASS	LAMBDA EXPRESSIONS
It is a anonymous inner class with out name	It is a anonymous method with out name
It can implement the interfaces with any number of abstract methods inside them	It can implement the interfaces which has only 1 abstract method called Functional interfaces
They can be instantiated and can extend abstract and concrete classes.	They can't be instantiated and can't extend abstract and concrete classes.
Instance variables can be declared and "this" inside it always refer to the current inner class	Only local variables can be declared and "this" inside it always refer to the outer enclosing class
Memory inside heap will be allocated on demand whenever we create an object for it	Permanent memory (Method area) will be allocated for it

eazy bytes

- Functional interfaces contains only one abstract method. Below are the thumb rules for Functional interfaces,

  - ✓ Only 1 abstract method is allowed (Also called as SAM – Single Abstract method)
  - ✓ Any number of default methods are allowed
  - ✓ Any number of static methods are allowed
  - ✓ Any number of private methods are allowed

- Since functional interfaces has only one abstract method, we can write lambda code/pass the implementation behavior of it using lambda code. Since there is only one unimplemented method inside interface compiler doesn't complain for method name, parameters datatype, return type of the method etc.

- In Java 8, a new annotation @FunctionalInterface is introduced to mark an interface as Functional interface. With this if in future if any one try to add another abstract method, compiler will throw an error.

JAVA 8 NEW FEATURES

FUNCTIONAL INTERFACE

- Java already has some interfaces similar to functional Interfaces even before Java 8. Those are like,
  - ✓ Runnable – It contains only 1 abstract method run()
  - ✓ Comparable : It contains only 1 abstract method compareTo()

@FunctionalInterface
public interface ArithmeticOperation{
public int performOperation (int a, int b); ======➔ *Valid functional interface as it contains SAM*
}

@FunctionalInterface
public interface ArithmeticOperation{
public int performOperation (int a, int b); ======➔ *Invalid functional interface as it contains 2 abstract methods*
public int performOperation (int c);

JAVA 8 NEW FEATURES

FUNCTIONAL INTERFACE

eazy bytes

```
@FunctionalInterface
public interface ArithmeticOperation{

 }
```

=======→ *Invalid functional interface as it doesn't have single abstract method*

```
@FunctionalInterface
public interface ArithmeticOperation{
        public int performOperation (int a, int b);
}
```

```
@FunctionalInterface
public interface SimpleOperation extends ArithmeticOperation {

}
```

SimpleOperation interface is a valid functional interface though it doesn't have any abstract method because it extends another functional interface which has only 1 abstract method.

```
@FunctionalInterface
public interface SameOperation extends ArithmeticOperation {
        public int performOperation (int a, int b);
}
```

SameOperation interface is a valid functional interface though it inherits 1 abstract method from its parent and it also has 1. This is because the abstract method name and signature is same as parent interface.

```
@FunctionalInterface
public interface InvalidOperation extends ArithmeticOperation {
        public int performAnotherOperation (int a, int b);
}
```

InvalidOperation interface is not a valid functional interface because it inherits 1 abstract method from its parent and it also has another abstract method which violates the rule of SAM.

```
public interface NormalOperation extends ArithmeticOperation {
        public int normalOperation (int a, int b);
}
```

Compiler will not have any problem with NormalOperation interface as it is not marked as a Functional interface and it can have any number of abstract methods.

JAVA 8 NEW FEATURES

FUNCTIONAL INTERFACE

- To make developer life easy, Java 8 has provided some pre defined functional interfaces by considering most common requirements during the development. All such interfaces are present inside the ‘java.util.function’ package.

- Below are the most important functional interfaces provided by the Java team,

  - ✓ *java.util.function.Predicate <T>*
  - ✓ *java.util.function.Function<T, R>*
  - ✓ *java.util.function.Consumer <T>*
  - ✓ *java.util.function.Supplier<T>*
  - ✓ *java.util.function.BiPredicate<T, U>*
  - ✓ *java.util.function.BiFunction<T, U, R>*
  - ✓ *java.util.function.BiConsumer<T, U>*
  - ✓ *java.util.function.UnaryOperator<T>*
  - ✓ *java.util.function.BinaryOperator<T>*
  - ✓ *Primitive Functional Interfaces*

eazy bytes

java.util.function.Predicate <T>

- Predicate Functional interface handles the scenarios where we accept a input parameter and return the boolean after processing the input.
  - *@param <T> the type of the input to the function*

✓ *boolean test(T t);* *=== > Single abstract method available*

✓ *static <T> Predicate<T> isEqual(Object targetRef)* *=== > Static method to check equality of 2 objects*

✓ *default Predicate<T> or(Predicate<? super T> other)* *=== > Default method that can be used while joining multiple predicate conditions. This acts like a logical OR condition*

✓ *default Predicate<T> negate()* *=== > Default method that can be used while joining multiple predicate conditions. This acts like a logical NOT condition*

✓ *default Predicate<T> and(Predicate<? super T> other)* *=== > Default method that can be used while joining multiple predicate conditions. This acts like a logical AND condition*

eazy bytes

java.util.function.Function<T, R>

- Function is similar to Predicate except with a change that instead of boolean it can return any datatype as outcome. It represents a function that accepts one argument and produces a result.
  - *@param <T> the type of the input to the function*
  - *@param <R> the type of the result of the function*

✓ *R apply(T t);* === > *Single abstract method available*

✓ *static <T> Function<T, T> identity()* === > *Utility Static method which will return the same input value as output*

✓ *default compose(..)/andThen(..)* === > *Default method that can be used for chaining*

- The difference between andThen() and compose() is that in the andThen first func will be executed followed by second func whereas in compose it is vice versa.

JAVA 8 NEW FEATURES

PREDICATE Vs FUNCTION

PREDICATE	FUNCTION
Is used for checking the conditions on given input and return boolean value	Is used for executing business logic for the input and return any type of output.
The single abstract method(SAM) name is test()	The single abstract method(SAM) name is apply()
It takes one type parameter which indicates input parameter and return parameter is always boolean	It takes 2 type parameters which indicates input parameter and return parameter
It has a static method called isEqual() which will check the equality of the 2 values/objects.	It has a static method called identity() which will return the same as input given.
It has 3 default methods for chaining namely and(), or() & negate()	It has 2 default methods for chaining namely andThen() & compose()

eazy bytes

java.util.function.UnaryOperator<T>

- If we have scenarios where both the input and output parameters data type is same, then instead of using Function<T,R> we can use the UnaryOperator<T>
    - *@param <T> the type of the operand and result of the operator*
- It is a child of Function<T,T>. So all the methods apply(), compose(), andThen() are available inside the UnaryOperator interface also.
- In other words we can say that UnaryOperator takes one argument, and returns a result of the same type of its arguments.

java.util.function.Consumer<T>

- As the name indicates Consumer interface will always consumes/accept the given input for processing but not return anything to the invocation method.

    - *@param <T> the type of the input to the function*

✓ *void accept(T t);* === > *Single abstract method available*

✓ *default andThen(..)* === > *Default method that can be used for chaining*

- No static methods are available in Consumer functional interface.

JAVA 8 NEW FEATURES

SUPPLIER FUNCTIONAL INTERFACE

eazy bytes

java.util.function.Supplier<T>

- As the name indicates Supplier interface will always return a value with out accepting any input. Think of the scenarios like generating report or OTP where we don't provide any input.

    - *@param <T> the type of results supplied by this supplier*

✓ *T get();* === > *Single abstract method available*

- There are no static and chaining methods available in Supplier functional interface. The reason is that it will not accept any input so there is no meaning of chaining in it.

JAVA 8 NEW FEATURES

eazy bytes

CONSUMER Vs SUPPLIER

CONSUMER	SUPPLIER
Is used in the scenarios where we send an input but not expect any return value from it	Is used in the scenarios where we don't send any input but expecting return value from it
The single abstract method(SAM) name is accept()	The single abstract method(SAM) name is get()
It takes one type parameter which indicates input parameter and return parameter is always void	It takes one type parameter which indicates output parameter and input parameter is not needed
It has no static methods but has 1 default method andThen() for chaining	It has no static and default methods inside it
It is like setter method inside our POJO classes	It is like getter method inside our POJO classes

JAVA 8 NEW FEATURES

BI FUNCTIONAL INTERFACES

- As of now we have see the functional interfaces which will accept only 1 parameter as input but what if we have a need to send 2 input parameters. To address the same Java has Bi Functional interfaces.

- **java.util.function.BiPredicate<T, U>** – Similar to Predicate but it can accept 2 input parameters and return a boolean value
  - *@param <T> the type of the first argument to the predicate*
  - *@param <U> the type of the second argument the predicate*

- **java.util.function.BiFunction<T, U, R>** – Similar to Function but it can accept 2 input parameters and return a output as per the data type mentioned.
  - *@param <T> the type of the first argument to the function*
  - *@param <U> the type of the second argument to the function*
  - *@param <R> the type of the result of the function*

JAVA 8 NEW FEATURES

BI FUNCTIONAL INTERFACES

- **java.util.function.BiConsumer<T, U>** – Similar to Consumer but it can accept 2 input parameters and no return value same as Consumer
    - *@param <T> the type of the first argument to the operation*
    - *@param <U> the type of the second argument to the operation*
- There is no BiSupplier for Supplier functional interface as it will not accept any input parameters.

JAVA 8 NEW FEATURES

BINARY OPERATOR FUNCTIONAL INTERFACE

java.util.function.BinaryOperator<T>

- BinaryOperator<T> is a child of BiFunction<T,U,R> .We will use this in the scenarios where the 2 input parameters and 1 return parameter data types is same.
  - *@param <T> the type of the operands and result of the operator*
- In other words we can say that BinaryOperator takes two arguments of the same type and returns a result of the same type of its arguments.
- In addition to the methods that it inherits from BiFunction<T,U,R>, it also has 2 utility static methods inside it. They both will be used to identify the minimum or maximum of 2 elements based on the comparator logic that we pass,
  - *static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)*
  - *static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)*

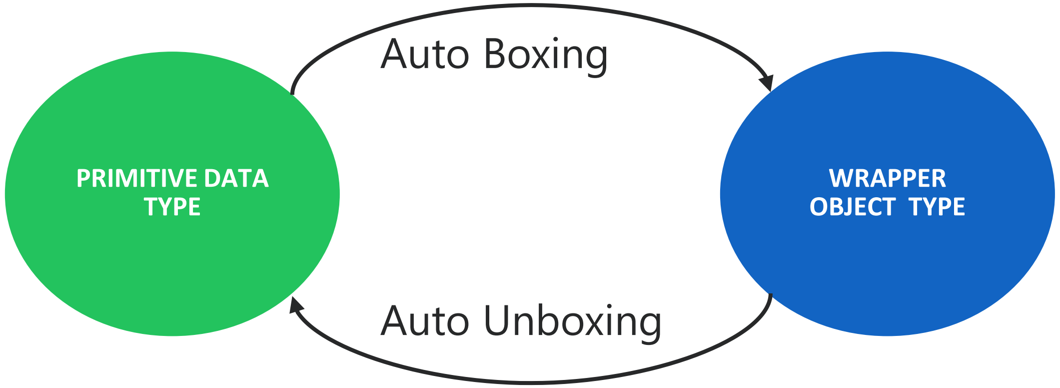
JAVA 8 NEW FEATURES

PRIMITIVE TYPE FUNCTIONAL INTERFACES

- All the functional interfaces like Predicate, Function, Consumer that we discussed previously accepts or returns only Object type values like Integer, Double, Float, Long etc.

- There is a performance problem with this. If we pass any primitive input values like int, long, double, float Java will auto box them in order to convert them into corresponding wrapper objects. Similarly once the logic is being executed Java has to convert them into primitive types using unboxing.

- Since there is lot of auto boxing and unboxing happening it may impact performance for larger values/inputs. To overcome such scenarios Java has primitive type functional interfaces as well.



JAVA 8 NEW FEATURES

PRIMITIVE TYPE FUNCTIONAL INTERFACES

- Below are the most important primitive functional interfaces provided by the Java team,

  ✓ *java.util.function.IntPredicate – Always accepts int as input*

  boolean test(int value);

  ✓ *java.util.function.DoublePredicate – Always accepts double as input*

  boolean test(double value);

  ✓ *java.util.function.LongPredicate – Always accepts long as input*

  boolean test(long value);

  ✓ *java.util.function.IntFunction<R> - Always accepts int as input and return any type as output*

  R apply(int value);

JAVA 8 NEW FEATURES

PRIMITIVE TYPE FUNCTIONAL INTERFACES

eazy bytes

- Below are the most important primitive functional interfaces provided by the Java team,

  ✓ *java.util.function.DoubleFunction<R> - Always accepts double as input and return any type as output*

  R apply(double value);

  ✓ *java.util.function.LongFunction<R> - Always accepts long as input and return any type as output*

  R apply(long value);

  ✓ *java.util.function.ToIntFunction<T> - Always return int value but accepts any type as input*

  int applyAsInt(T value);

  ✓ *java.util.function.ToDoubleFunction<T> - Always return double value but accepts any type as input*

  double applyAsDouble(T value);

JAVA 8 NEW FEATURES

PRIMITIVE TYPE FUNCTIONAL INTERFACES

- Below are the most important primitive functional interfaces provided by the Java team,

  ✓ *java.util.function.ToLongFunction<T> - Always return long value but accepts any type as input*

  long applyAsLong(T value);

  ✓ *java.util.function.IntToLongFunction - Always takes int type as input and return long as return type*

  long applyAsLong(int value);

  ✓ *java.util.function.IntToDoubleFunction - Always takes int type as input and return double as return type*

  double applyAsDouble(int value);

  ✓ *java.util.function.LongToIntFunction - Always takes long type as input and return int as return type*

  int applyAsInt(long value);

JAVA 8 NEW FEATURES

PRIMITIVE TYPE FUNCTIONAL INTERFACES

eazy bytes

- Below are the most important primitive functional interfaces provided by the Java team,

  ✓ *java.util.function.LongToDoubleFunction - Always takes long type as input and return double as return type*

  double applyAsDouble(long value);

  ✓ *java.util.function.DoubleToIntFunction - Always takes double as input and return int as return type*

  int applyAsInt(double value);

  ✓ *java.util.function.DoubleToLongFunction - Always takes double type as input and return long as return type*

  long applyAsLong(double value);

  ✓ *java.util.function.ToIntBiFunction - Always takes 2 input parameters of any type and return int as return type*

  int applyAsInt(T t, U u);

PRIMITIVE TYPE FUNCTIONAL INTERFACES

eazy bytes

- Below are the most important primitive functional interfaces provided by the Java team,

  - ✓ *java.util.function.ToLongBiFunction - Always takes 2 input parameters of any type and return long as return type*

    ```
    long applyAsLong(T t, U u);
    ```

  - ✓ *java.util.function.ToDoubleBiFunction - Always takes 2 input parameters of any type and return double as return type*

    ```
    double applyAsDouble(T t, U u);
    ```

  - ✓ *java.util.function.IntConsumer - Always accepts int value as input*

    ```
    void accept(int value);
    ```

  - ✓ *java.util.function.LongConsumer - Always accepts long value as input*

    ```
    void accept(long value);
    ```

- Below are the most important primitive functional interfaces provided by the Java team,

  - ✓ *java.util.function.DoubleConsumer - Always accepts double value as input*

    void accept(double value);

  - ✓ *java.util.function.ObjIntConsumer<T> - Always accepts 2 inputs of type int and any data type*

    void accept(T t, int value);

  - ✓ *java.util.function.ObjLongConsumer<T> - Always accepts 2 inputs of type long and any data type*

    void accept(T t, long value);

  - ✓ *java.util.function.ObjDoubleConsumer<T> - Always accepts 2 inputs of type double and any data type*

    void accept(T t, double value);

JAVA 8 NEW FEATURES

PRIMITIVE TYPE FUNCTIONAL INTERFACES

eazy bytes

- Below are the most important primitive functional interfaces provided by the Java team,

  ✓ *java.util.function.IntSupplier - Always return int value as output*

  int getAsInt();

  ✓ *java.util.function.LongSupplier - Always return long value as output*

  long getAsLong();

  ✓ *java.util.function.DoubleSupplier - Always return double value as output*

  double getAsDouble();

  ✓ *java.util.function.BooleanSupplier - Always return boolean value as output*

  boolean getAsBoolean();

PRIMITIVE TYPE FUNCTIONAL INTERFACES

- Below are the most important primitive functional interfaces provided by the Java team,

  ✓ *java.util.function.IntUnaryOperator - Always accept and return int values*

  int applyAsInt(int operand);

  ✓ *java.util.function.LongUnaryOperator - Always accept and return long values*

  long applyAsLong(long operand);

  ✓ *java.util.function.DoubleUnaryOperator - Always accept and return double values*

  double applyAsDouble(double operand);

JAVA 8 NEW FEATURES

PRIMITIVE TYPE FUNCTIONAL INTERFACES

eazy bytes

- Below are the most important primitive functional interfaces provided by the Java team,

  ✓ *java.util.function.IntBinaryOperator - Always accept 2 input parameters and return in int values*

  int applyAsInt(int left, int right);

  ✓ *java.util.function.LongBinaryOperator - Always accept 2 input parameters and return in long values*

  long applyAsLong(long left, long right);

  ✓ *java.util.function.DoubleBinaryOperator - Always accept 2 input parameters and return in double values*

  double applyAsDouble(double left, double right);

JAVA 8 NEW FEATURES

METHOD REFERENCES

eazy bytes

- Sometimes, there is already a method that carries out exactly the action that you'd like to pass on inside lambda code. In such cases it would be nicer to pass on the method instead of duplicating the code again.

- This problem is solved using method references in Java 8. Method references are a special type of lambda expressions which are often used to create simple lambda expressions by referencing existing methods.

- There are 4 types of method references introduced in Java 8,

  - ✓ *Static Method Reference (Class::staticMethod)*
  - ✓ *Reference to instance method from instance (objRef::instanceMethod)*
  - ✓ *Reference to instance method from class type (Class::instanceMethod)*
  - ✓ *Constructor Reference (Class::new)*

- As you might have observed, Java has introduced a new operator :: (double colon) called as Method Reference Delimiter.. In general, one don't have to pass arguments to method references.

Static Method Reference (Class::staticMethod)

```
public class MethodReference {

    /**
     * @param args
     */
    public static void main(String[] args) {

        ArithmeticOperation operation = (a,b)-> {
            int sum = a+b;
            System.out.println("The sum of given input values using lambda is: "+sum);
            return sum;
        };
        operation.performOperation(2, 3);

        ArithmeticOperation methodRef = MethodReference::performAddition;
        methodRef.performOperation(2, 3);
    }

    public static int performAddition(int a, int b) {
        int sum = a+b;
        System.out.println("The sum of given input values using method reference is: "+sum);
        return sum;
    }

}
```

- ✓ *As you can see first we have written a lambda expression code for the Functional interface 'ArithmeticOperation' to calculate the sum of given 2 integers.*
- ✓ *Later since we have same logic of code present inside a static method we used static method reference as highlighted.*
- ✓ *This way we can leverage the existing static methods code to pass the behavior, instead of writing lambda code again.*
- ✓ *Using method references is mostly advised if you already have code written inside a method or if your code inside lambda is large(we can put in separate method).*

Reference to instance method from instance (objRef::instanceMethod)

```
public class MethodReference {

    public static void main(String[] args) {
        ArithmeticOperation operation = (a, b) -> {
            int sum = a + b;
            System.out.println("The sum of given input values "
                    + "using lambda is: " + sum);
            return sum;
        };
        operation.performOperation(2, 3);
        MethodReference methodRef = new MethodReference();
        ArithmeticOperation instanceMethod = methodRef::performInstanceAddition;
        instanceMethod.performOperation(2, 3);
    }

    public int performInstanceAddition(int a, int b) {
        int sum = a + b;
        System.out.println("The sum of given input values using instance "
                + "method reference is: " + sum);
        return sum;
    }
```

- ✓ *As you can see first we have written a lambda expression code for the Functional interface 'ArithmeticOperation' to calculate the sum of given 2 integers.*
- ✓ *Later since we have same logic of code present inside a method we used instance method reference as highlighted using an object of the class.*
- ✓ *This way we can leverage the existing instance methods code to pass the behavior, instead of writing lambda code again.*

eazy bytes

Reference to instance method from class type (Class::instanceMethod)

```java
public class MethodReference {

    public static void main(String[] args) {

        List<String> departmentList = new ArrayList<>();
        departmentList.add("Supply");
        departmentList.add("HR");
        departmentList.add("Sales");
        departmentList.add("Marketing");
        departmentList.forEach(s -> System.out.println(s));

        departmentList.forEach(System.out::println);

    }

}
```

- ✓ *This type of method reference is similar to the previous example, but without having to create a custom object*
- ✓ *As you can see first we have written a lambda expression code inside forEach method to print all the list elements.*
- ✓ *Later since we have same logic of code present inside a method of System.out, we used instance method reference as highlighted using class type itself.*
- ✓ *This way we can leverage the existing methods code to pass the behavior, instead of writing lambda code again.*

eazy bytes

Constructor Reference (Class::new)

```java
@FunctionalInterface
public interface ProductInterface {

    Product getProduct(String name, int price);

}
```

```java
public class Product {

    String name;
    int price;

    public Product(String name, int price) {
        this.name = name;
        this.price = price;
    }
```

```java
public class MethodReference {

    public static void main(String[] args) {

        //ProductInterface productInterface =
        // (name,price)-> new Product(name,price);
        ProductInterface productInterface = Product::new;
        Product prod =productInterface.getProduct("Apple IPhone", 1500);
        System.out.println(prod.getName());
        System.out.println(prod.getPrice());

    }
```

- ✓ *Constructor references are just like method references, except that the name of the method is new. For example, Product::new is a reference to a Product constructor.*
- ✓ *As you can see we have used constructor reference in the place of lambda code to create a new product details using functional interface 'ProductInterface'.*
- ✓ *Few other examples of Constructor reference are,*
    - *String::new;*
    - *Integer::new;*
    - *ArrayList::new;*
    - *UserDetail::new;*

JAVA 8 NEW FEATURES

STREAMS API

eazy bytes

- Java 8 introduced java.util.stream API which has classes for processing sequence of objects that we usually stored inside the collections. The central API class is the Stream<T>.

- Don't get confused with the java.io streams which are meant for processing the binary data to/from the files.

  java.io streams != java.util streams

- Collections like List, Set will be used if we want to represent the group of similar objects as a single entity where as Streams will be used to process a group of objects present inside a collection.

- You can create streams from collections, arrays or iterators.

- In Streams the code is written in a declarative way: you specify what you want to achieve like in query style as opposed to specifying how to implement an operation.

eazy bytes

Creating a Stream from collection or list of elements

```java
/**
 * @param args
 */
public static void main(String[] args) {

    List<String> departmentList = new ArrayList<>();
    departmentList.add("Supply");
    departmentList.add("HR");
    departmentList.add("Sales");
    departmentList.add("Marketing");

    Stream<String> depStream= departmentList.stream();
    depStream.forEach(System.out::println);

    Stream<String> inStream = Stream.of("Eazy", "Bytes", "Java");
    inStream.forEach(System.out::println);

    Stream<String> parallelStream= departmentList.parallelStream();
    parallelStream.forEach(System.out::println);

}
```

- ✓ *We can create a stream using either by calling stream() default method introduced in all the collections to support streams or with the help of Stream.of()*
- ✓ *Processing the elements inside streams parallelly is very simple. We just need to call parallelStream() default method instead of stream()*
- ✓ *A stream does not store its elements. They may be stored in an underlying collection or generated on demand.*
- ✓ *Stream operations don't mutate their source. Instead, they return new streams that hold the result.*

JAVA 8 NEW FEATURES

STREAMS API

eazy bytes

- When we work with streams, we set up a pipeline of operations in different stages as mentioned below.

  1. *Creating a Stream using stream(), parallelStream() or Streams.of().*
  2. *One or more intermediate operations for transforming the initial stream into others or filtering etc.*
  3. *Applying a terminal operation to produce a result.*

- Inside java.util.Arrays new static methods were added to convert an array into a stream,

  ✓ *Arrays.stream(array) – to create a stream from an array*
  ✓ *Arrays.stream(array, from, to) – to create a stream from a part of an array*

- To make an empty stream with no elements, we can use empty() method inside Stream class.

  ✓ *Stream<String> emptyStream = Stream.empty();*

- To generate an infinite stream of elements which is suitable for stream of random elements, Stream has 2 static methods called Stream.generate() & Stream.iterate()

eazy bytes

map method (Intermediate Operation)

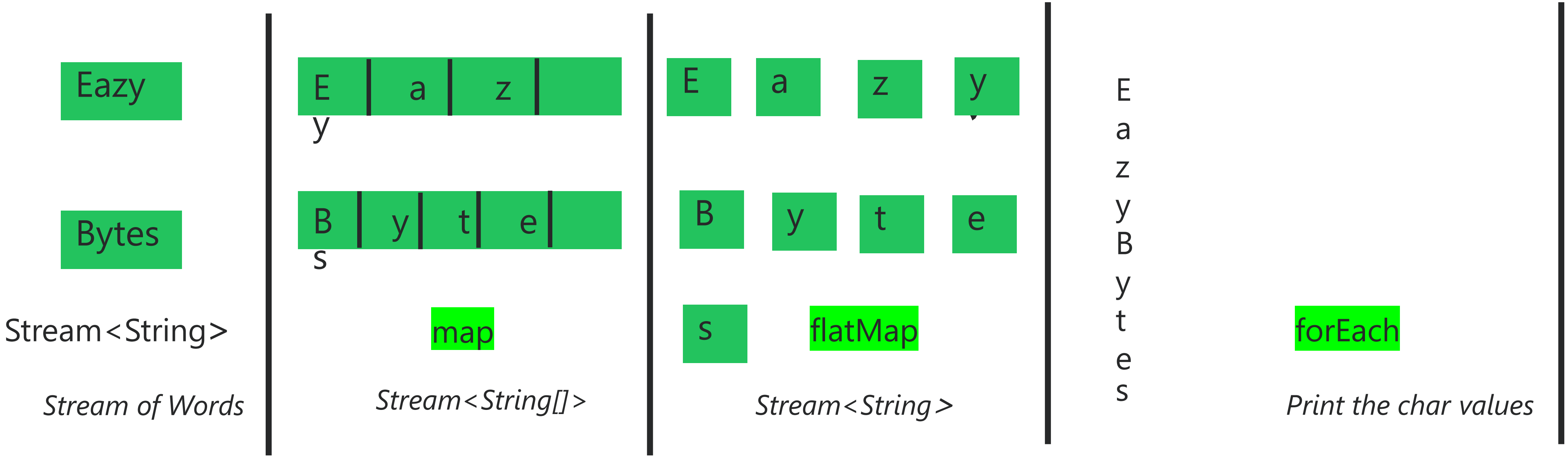
```java
public static void main(String[] args) {
    List<String> departmentList = new ArrayList<>();
    departmentList.add("Supply");
    departmentList.add("HR");
    departmentList.add("Sales");
    departmentList.add("Marketing");

    Stream<String> depStream = departmentList.stream();
    depStream.map(word-> word.toUpperCase()).
            forEach(System.out::println);

}
```

- ✓ *If we have a scenario where we need to apply a business logic or transform each element inside a collection, we use map() method inside streams to process them*
- ✓ *In simple words, the map() is used to transform one object into other by applying a function.*
- ✓ *Here in our example for each element inside our list, we need to transform them into uppercase letters before printing them on the console.*
- ✓ *Stream map method takes Function as argument that is a functional interface.*
- ✓ *Stream map(Function mapper) is an intermediate operation and it returns a new Stream as return value. These operations are always lazy.*
- ✓ *Stream operations don't mutate their source. Instead, they return new streams that hold the result.*

flatMap method (Intermediate Operation)

```
public static void main(String[] args) {
    String[] arrayOfWords = {"Eazy", "Bytes"};
    Stream<String> streamOfwords = Arrays.stream(arrayOfWords);
    streamOfwords
    .map(word -> word.split(""))
    .flatMap(Arrays::stream)
    .forEach(System.out::println);
}
```

- ✓ *Some times based on the input elements we may ended up with multiple streams post map method and if we try to collect them we will get a list of streams instead of a single stream.*
- ✓ *For such cases we can use flatMap. It is the combination of a map and a flat operation i.e., it applies a function to elements as well as flatten them.*



eazy bytes

filter method (Intermediate Operation)

```
public static void main(String[] args) {
    List<String> departmentList = new ArrayList<>();
    departmentList.add("Supply");
    departmentList.add("HR");
    departmentList.add("Sales");
    departmentList.add("Marketing");

    Stream<String> depStream = departmentList.stream();
    depStream.filter(word -> word.startsWith("S"))
                    .forEach(System.out::println);

}
```

- ✓ *If we have a scenario where we need to exclude certain elements inside a collection based on a condition, we can use filter() method inside streams to process them.*
- ✓ *Here in our example our requirement is to filter the departments name that starts with 'S' and print them on to the console*
- ✓ *Stream filter method takes Predicate as argument that is a functional interface which can act has a boolean to filter the elements based on condition defined.*
- ✓ *Stream filter(Predicate<T>) is an intermediate operation and it returns a new Stream as return value. These operations are always lazy.*

limit method (Intermediate Operation)

```java
public static void limitInStreams() {
    Stream.generate(new Random()::nextInt).
        limit(10).forEach(System.out::println);
}
```

- ✓ *If we have a scenario where we need to limit the number of elements inside a stream, we can use limit(n) method inside streams.*
- ✓ *Here in our example we used generate method to provide random integer numbers. But since generate will provide infinite stream of numbers, we limit it to only first 10 elements.*
- ✓ *Stream limit method takes a number which indicates the size of the elements we want to limit but this limit number should not be greater than the size of elements inside the stream.*
- ✓ *Note that limit also works on unordered streams (for example, if the source is a Set).In this case we shouldn't assume any order on the result produced by limit.*

eazy bytes

skip method (Intermediate Operation)

```
public static void skipInStreams() {
    Stream.iterate(1, n -> n + 1).skip(10).limit(20).
    forEach(System.out::println);
}
```

- ✓ *Streams support the skip(n) method to return a stream that discards the first n elements.*
- ✓ *Here in our example we used iterate method to provide integer numbers from 1. But since iterate will provide infinite stream of numbers, we skipped first 10 numbers and limit it to only 20 numbers. The output of this method will be the numbers from 11….30*
- ✓ *If the stream has fewer than n elements, an empty stream is returned.*
- ✓ *Note that limit(n) and skip(n) are complementary.*

eazy bytes

Streams are traversable only once

```java
public static void main(String[] args) {
    List<String> departmentList = new ArrayList<>();
    departmentList.add("Supply");
    departmentList.add("HR");
    departmentList.add("Sales");
    departmentList.add("Marketing");
    Stream<String> depStream = departmentList.stream();
    depStream.forEach(System.out::println);
    depStream.forEach(System.out::println);
}
```

- ✓ *Note that, similarly to iterators, a stream can be traversed only once. After that a stream is said to be consumed.*
- ✓ *You can get a new stream from the initial data source to traverse it again as you would for an iterator assuming it's a repeatable source like a collection but not an I/O channel.*
- ✓ *For example like in the example, if we try to traverse the stream again after consuming all the elements inside it, we will get an runtime 'java.lang.IllegalStateException' with a message 'stream has already been operated upon or closed'*

eazy bytes

reduce method (terminal Operation)

```java
public static void reduceInStreams() {
    System.out.println(Stream.iterate(1, n -> n + 1).
            limit(20).reduce(0, (a, b) -> a + b));
}
```

- ✓ *The operations which will combine all the elements in the stream repeatedly to produce a single value such as an Integer. These queries can be classified as reduction operations (a stream is reduced to a value).*
- ✓ *Here in our example we used iterate method to provide integer numbers from 1. But since iterate will provide infinite stream of numbers, we limit it to only 20 numbers. Post that using the reduce() method we calculated the sum of all the first 20 numbers.*
- ✓ *Reduce method here accepting 2 parameters. One is the initial value of the sum variable which is 0 and the second one is the operation that we want to use to combine all the elements in this list (which is addition here)*
- ✓ *There's also an overloaded variant of reduce that doesn't take an initial value, but it returns an Optional object considering empty streams scenario.*

eazy bytes

collect method (terminal Operation)

```java
public static void collectStreams() {
    List<String> departmentList = new ArrayList<>();
    departmentList.add("Supply");
    departmentList.add("HR");
    departmentList.add("Sales");
    departmentList.add("Marketing");

    Stream<String> depStream = departmentList.stream();
    List<String> newDepartmentList = depStream.filter(word -> word.startsWith("S"))
            .collect(Collectors.toList());
    newDepartmentList.forEach(System.out::println);
```

- ✓ *Stream.collect() allows us to repackaging elements to some data structures and applying some additional logic etc. on data elements held in a Stream instance.*
- ✓ *Here in our example we first used filter() method to identify the elements that start with 'S', post that we used collect() method to convert the stream into a list of objects.*
- ✓ *java.util.stream.Collector play an important role in Java 8 streams processing with the help of collect() method inside streams*

Other important methods inside Collector class are,

- *toSet() – Convert stream into a set*
- *toCollection() - Convert stream into a collection*
- *toMap() – Convert stream into a Map after applying key/value determination function.*
- *counting() – Counting number of stream elements*
- *joining() - For concatenation of stream elements into a single String*
- *minBy() - To find minimum of all stream elements based on given Comparator*
- *maxBy() - To find maximum of all stream elements based on given Comparator*
- *reducing() - Reducing elements of stream based on BinaryOperator function provided*

eazy bytes

collectingAndThen method (terminal Operation)

- ✓ *collectingAndThen() will be used in the scenarios where the stream elements need to be collected and then the collected object needs to be transformed using a given rule\function. Using the collectingAndThen collector both these tasks of collection and transformation can be specified and executed together.*

- ✓ *It accepts 2 parameters,*
  - *1st input parameter is downstream which is an instance of a Collector<T,A,R> i.e. the standard definition of a collector. In other words, any collector can be used here.*
  - *2nd input parameter is finisher which needs to be an instance of a Function<R,RR> functional interface.*

```java
public static void collectStreams() {
    List<Product> productList = Arrays.asList(new Product("Apple", 1200), new Product("Samsung", 1000),
            new Product("Nokia", 800), new Product("BlackBerry", 1500), new Product("Apple Pro Max", 1800));

    String maxPriceProduct = productList.stream()
            .collect(Collectors.collectingAndThen(Collectors.maxBy(Comparator.comparing(Product::getPrice)),
                    (Optional<Product> product) -> product.isPresent() ? product.get().getName() : "None"));
    System.out.println("The product with max price tag is: " + maxPriceProduct);
}
```

eazy bytes

groupingBy method (terminal Operation)

- ✓ *The groupingBy() method of Collectors class in Java are used for grouping objects by some property and storing results in a Map instance. In order to use it, we always need to specify a property by which the grouping would be performed. This method provides similar functionality to SQL's GROUP BY clause.*

- ✓ *Below is the sample code where we pass to the groupingBy method a Function (expressed in the form of a method reference) extracting the corresponding Product.getPrice for each Product in the stream. We call this Function a classification function specifically because it's used to classify the elements of the stream into different groups.*

```
public static void groupingByStreams() {
    List<Product> productList = Arrays.asList(new Product("Apple", 1200), new Product("Samsung", 1000),
            new Product("Nokia", 800), new Product("BlackBerry", 1000), new Product("Apple Pro Max", 1500),
            new Product("Mi", 800), new Product("OnePlus", 1000));

    Map<Integer, List<Product>> groupByPriceMap = productList.stream()
            .collect(Collectors.groupingBy(Product::getPrice));
    System.out.println("The list of products grouped by price is: " + groupByPriceMap);
}
```

eazy bytes

partitioningBy method (terminal Operation)

- ✓ *Collectors partitioningBy() method is used to partition a stream of objects(or a set of elements) based on a given predicate. The fact that the partitioning function returns a boolean means the resulting grouping Map will have a Boolean as a key type, and therefore,there can be at most two different groups—one for true and one for false.*

- ✓ *Below is the sample code where we pass to the partitioningBy method a predicate function to partition all the products into >$1000 and <=$1000.*

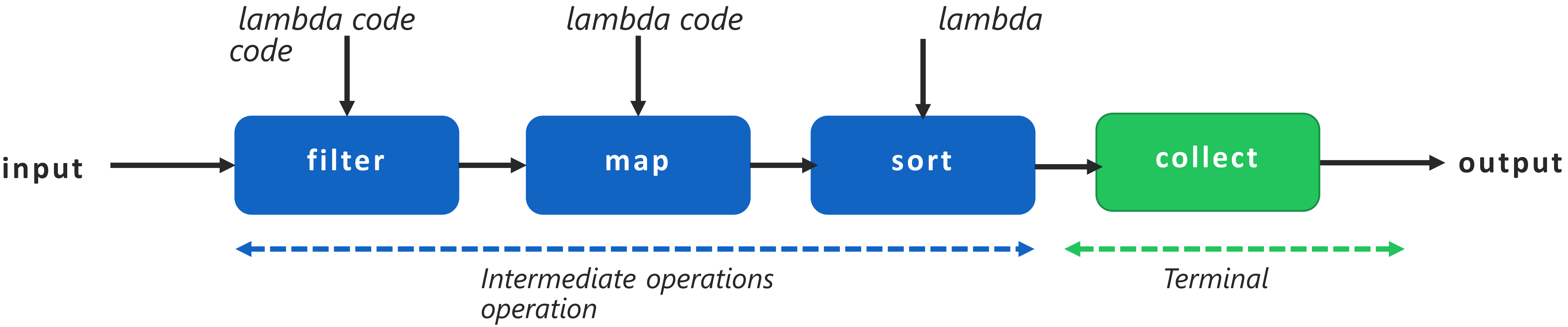
- ✓ *Compared to filters, Partitioning has the advantage of keeping both lists of the stream elements, for which the application of the partitioning function returns true or false.*

```
public static void partitioningByStreams() {
    List<Product> productList = Arrays.asList(new Product("Apple", 1200), new Product("Samsung", 1000),
            new Product("Nokia", 800), new Product("BlackBerry", 1000), new Product("Apple Pro Max", 1500),
            new Product("Mi", 800), new Product("OnePlus", 1000));

     Map<Boolean, List<Product>> costlyProducts  = productList.stream()
            .collect(Collectors.partitioningBy(product->product.getPrice()>1000));
    System.out.println("The list of products partitioned by price is: " + costlyProducts);
}
```

eazy bytes

Chaining stream operations to form a stream pipeline

- ✓ *We can form a chain of stream operations using intermediate and terminal operation to achieve a desire output. This we also call as stream pipeline.*
- ✓ *Suppose think of an example where I have list of int values inside a list where I want to filter all odd numbers, followed by converting the remaining numbers by multiply themselves, sorting and at last display the output in a new list.*



JAVA 8 NEW FEATURES

STREAMS API

eazy bytes

Chaining stream operations to form a stream pipeline

	5 | 2 | 11 | 7 | 4 | 13 | 9	*Forming a stream(Stream<Integers>) from the source*
filter(num->num%2!=0)	5 | 11 | 7 | 13 | 9	*A new stream(Stream<Integers>) which has only Odd numbers is returned*
*map (num-> num*num)*	25 | 121 | 49 | 169 | 81	*A new stream(Stream<Integers>) will be returned after multiplying the same numbers*
sorted ()	25 | 49 | 81 | 121 | 169	*A new stream(Stream<Integers>) will be returned after sorting the number*
collect (toList ())	{ 25, 49, 81, 121, 169 }	*A new list(List<Integers>) will be returned after collecting terminal operation*

Parallel Streams

```java
public static void parallelStreams() {
    List<String> departmentList = new ArrayList<>();
    departmentList.add("Supply");
    departmentList.add("HR");
    departmentList.add("Sales");
    departmentList.add("Marketing");
    departmentList.add("Insurance");
    departmentList.add("Security");
    departmentList.add("Finance");

    departmentList.parallelStream().forEach(System.out::println);

}
```

- ✓ *Streams interface allows you to process its elements in parallel in a convenient way: it's possible to turn a collection into a parallel stream by invoking the method parallelStream() on the collection source.*
- ✓ *A parallel stream is a stream that splits its elements into multiple chunks, processing each chunk with a different thread. Thus, you can automatically partition the workload of a given operation on all the cores of your multicore processor and keep all of them equally busy.*
- ✓ *Here in the example, we have a list of departments which need to be displayed. For the same we took the parallelStream and print each element. This will happen parallelly and the order of elements displayed will not be guaranteed.*
- ✓ *If we want to convert a sequential stream into a parallel one, just call the method parallel() on the stream.*

JAVA 8 NEW FEATURES

COLLECTIONS Vs STREAMS

COLLECTIONS	STREAMS
A collection is an in-memory data structure that holds all the values the data structure currently has	A stream does not store its elements. They may be stored in an underlying collection or generated on demand
A collection is eagerly constructed and elements can be added or removed	Stream is like a lazily constructed collection and elements inside streams can't be added or removed.
Collections doesn't use functional interfaces using lambda expressions	Streams uses lot of functional interfaces using lambda expressions
They are non-consumable i.e. can be traversable multiple times without creating it again.	Streams are consumable i.e. to traverse the stream, it needs to be created every time.
Collections are iterated using external loops like for, while	Streams are iterated internally based on the operation mentioned like map, filter.

JAVA 8 NEW FEATURES

NEW DATE AND TIME API(JODA)

eazy bytes

- The date/time API, before Java 8, has multiple design problems such as java.util.Date and SimpleDateFormatter classes aren't thread-safe. The date class doesn’t represent actual date instead it specifies an instant in time, with millisecond precision.

- The years start from 1900, whereas the months start at index 0. Suppose if you want to represent a date of 21 Sep 2017, you need to create an instance of date using the code below,

Date date = new Date(117, 8, 21); //Not very intuitive

- The problems with the java.util.Date are tried to handled by introducing new methods, deprecating few of the methods inside it and with an alternative class java.util.Calendar. But Calendar also has similar problems and design flaws that lead to error-prone code.

- On top of that the presence of both the Date and Calendar classes increases confusion among developers which one to use. The DateFormat comes with its own set of problems. It isn’t thread-safe and work only with the Date class.

JAVA 8 NEW FEATURES

NEW DATE AND TIME API(JODA)

- With all the limitations that Java.util.Date/Calendar has Developers started using third-party date and time libraries, such as Joda-Time.

- For these reasons, Oracle decided to provide high-quality date and time support in the native Java API. As a result, Java 8 integrates many of the Joda-Time features in the java.time package.

- The new java.time.* package has the below important classes to deal with Date & Time,

  ✓ *java.time.LocalDate*
  ✓ *java.time.LocalDateTime*
  ✓ *java.time.Instant*
  ✓ *java.time.Duration*
  ✓ *java.time.Period*

eazy bytes

java.time.LocalDate

- An instance of this class is an immutable object representing a plain date without the time of day. In other words, it doesn't carry any information about the time/time zone.

```
public static void main(String[] args) {

    LocalDate today = LocalDate.now();
    LocalDate date = LocalDate.of(1989, 6, 16);
    int year = date.getYear();
    Month month = date.getMonth();
    int day = date.getDayOfMonth();
    DayOfWeek dow = date.getDayOfWeek();
    int len = date.lengthOfMonth();
    boolean leap = date.isLeapYear();

    int year1 = date.get(ChronoField.YEAR);
    int month1 = date.get(ChronoField.MONTH_OF_YEAR);
    int day1 = date.get(ChronoField.DAY_OF_MONTH);


}
```

- ✓ *Date can be created using multiple ways like now(), of(), parse()*
- ✓ *As you can see we have many utility methods available to know the details about given date like DayOfMonth, DayOfWeek, isLeapYear etc.*
- ✓ *We can use this in the scenario where we care about only the Date but the time.*

eazy bytes

java.time.LocalTime

- If we have to deal with only time of the day ignoring Date value, then we can use LocalTime class.

```
public static void main(String[] args) {

    LocalTime time = LocalTime.of(12, 30, 10);

    int hour = time.getHour();
    System.out.println("Given Houre is: " + hour);
    int minute = time.getMinute();
    System.out.println("Given minute is: " + minute);
    int second = time.getSecond();
    System.out.println("Given second is: " + second);

    LocalTime parseTime = LocalTime.parse("12:30:10");
    LocalTime currentTime = LocalTime.now();
}
```

- ✓ *Time can be created using multiple ways like now(), of(), parse()*
- ✓ *As you can see we have many utility methods available to know the details about given time like the values of hour, minute, second etc.*
- ✓ *We can use this in the scenario where we care about only the time but not the date.*

java.time.LocalDateTime

- The LocalDateTime is a composite class of both LocalDate and a LocalTime. It represents both a date and a time without a time zone and can be created directly or by combining a date and time, as shown

```
public static void main(String[] args) {

    LocalDate date = LocalDate.of(1989, 6, 16);
    LocalTime time = LocalTime.of(12, 30, 10);
    LocalDateTime dateTime = LocalDateTime.of(1989, Month.JUNE, 16, 12, 30, 10);
    System.out.println("The given Date and Time is: "+dateTime);
    LocalDateTime dateTimeVal = LocalDateTime.of(date, time);

    LocalDate dateLocal = dateTimeVal.toLocalDate();
    System.out.println("The given time value is: "+dateLocal);
    LocalTime timeLocal = dateTimeVal.toLocalTime();
    System.out.println("The given date value is: "+timeLocal);

}
```

eazy bytes

java.time.Instant & java.time.Duration & java.time.Period

```
public static void main(String[] args) {

    Instant instant = Instant.ofEpochSecond(6);
    System.out.println(instant);
    System.out.println(Instant.ofEpochSecond(4, 1_000));
    System.out.println(Instant.ofEpochSecond(4, -1_000));
    Instant instantNow = Instant.now();
    System.out.println(instantNow);

    Duration instantDuration = Duration.between(instant,instantNow);
    System.out.println(instantDuration);

    LocalTime time = LocalTime.of(12, 30, 10);
    LocalTime time1 = LocalTime.of(16, 30, 10);
    Duration timeDuration = Duration.between(time,time1);
    System.out.println(timeDuration);
    LocalDateTime dateTime = LocalDateTime.of(1989, Month.JUNE, 16, 12, 30, 10);
    LocalDateTime dateTime1 = LocalDateTime.of(2000, Month.JUNE, 16, 16, 30, 10);
    Duration dateTimeDuration = Duration.between(dateTime,dateTime1);
    System.out.println(dateTimeDuration);

    LocalDate date = LocalDate.of(1989, 6, 16);
    LocalDate date1 = LocalDate.of(1989, 6, 26);
    Period localDatePeriod = Period.between(date,date1);
    System.out.println(localDatePeriod);

}
```

- ✓ *For humans we use Date and time, but for machines the time is calculated based on number of seconds passed from the Unix epoch time, set by convention to midnight of January 1, 1970 UTC. So to represent them we have Instant class*

- ✓ *Duration can be used to identify the duration of time between two instances, times and date Times. The difference will be given in terms of hours, minutes, seconds and nano seconds.*

- ✓ *Since LocalDate will not have time associated in it, we use Period to find the number of days difference between two local date objects.*

- ✓ *Both Duration & Period has many helper methods to deal with the values inside them*

JAVA 8 NEW FEATURES

NEW DATE AND TIME API(JODA)

eazy bytes

java.time.ZoneId, ZoneOffset, ZonedDateTime, and OffsetDateTime

```
ZoneId india = ZoneId.of("Asia/Kolkata");
ZonedDateTime indiaDateTime = ZonedDateTime.now(india);
System.out.println("Time in India now : " + indiaDateTime);

ZonedDateTime parisZonedDateTime = indiaDateTime.withZoneSameInstant(ZoneId.of("Europe/Paris"));
System.out.println("Time in Paris now : " + parisZonedDateTime);

ZoneOffset zoneOffSet = ZoneOffset.of("+05:30");
OffsetDateTime offsetDateTime = OffsetDateTime.now(zoneOffSet);
System.out.println("Time in India now using offset : " + offsetDateTime);

OffsetDateTime targetOffsetDateTime = offsetDateTime.withOffsetSameInstant(ZoneOffset.of("+01:00"));
System.out.println("Time in Paris now using offset : " + targetOffsetDateTime);
```

- ✓ *The new java.time.ZoneId class is the replacement for the old java.util.TimeZone class which aims to better protect you from the complexities related to time zones, such as dealing with Daylight Saving Time (DST)*
- ✓ *ZoneId describes a time-zone identifier and provides rules for converting between an Instant and a LocalDateTime.*
- ✓ *ZoneOffset describes a time-zone offset, which is the amount of time (typically in hours) by which a time zone differs from UTC/Greenwich.*
- ✓ *ZonedDateTime describes a date-time with a time zone in the ISO-8601 calendar system (such as 2007-12-03T10:15:30+01:00 Europe/Paris).*
- ✓ *OffsetDateTime describes a date-time with an offset from UTC/Greenwich in the ISO-8601 calendar system (such as 2007-12-03T10:15:30+01:00).*

eazy bytes

Calendar systems

✓ *The ISO-8601 calendar system is the default calendar system considered in Java. But four additional calendar systems are provided in Java 8. Each of these calendar systems has a dedicated date class: ThaiBuddhistDate, MinguoDate, JapaneseDate, and HijrahDate (Islamic).*

```
HijrahDate todayIslamic = HijrahDate.now();
System.out.println("Islamic date for today : " + todayIslamic);
```

Formating & Parsing date-time objects

```
public static void main(String[] args) {

    LocalDate date = LocalDate.of(2008, 6, 16);
    String baseISO = date.format(DateTimeFormatter.BASIC_ISO_DATE);
    System.out.println(baseISO); // 20080616
    String localISO = date.format(DateTimeFormatter.ISO_LOCAL_DATE);
    System.out.println(localISO); // 2008-06-16

    LocalDate baseISODate = LocalDate.parse("20080616", DateTimeFormatter.BASIC_ISO_DATE);
    System.out.println(baseISODate);
    LocalDate localISODate = LocalDate.parse("2008-06-16", DateTimeFormatter.ISO_LOCAL_DATE);
    System.out.println(localISODate);

    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MMM/yyyy");
    LocalDate localDate = LocalDate.of(2008, 6, 18);
    String formattedDate = localDate.format(formatter);
    System.out.println(formattedDate);

    DateTimeFormatter germanFormatter = DateTimeFormatter.ofPattern("d. MMMM yyyy", Locale.GERMAN);
    LocalDate date1 = LocalDate.of(2008, 6, 16);
    String formattedDateGer = date1.format(germanFormatter); // 16. Juni 2008
    System.out.println(formattedDateGer);

}
```

- ✓ *The new java.time.format package is introduced in Java for all the formatting and parsing requirements while dealing with Date & Time. The most important class of this package is DateTimeFormatter.*
- ✓ *In comparison with the old java.util.DateFormat class, all the DateTimeFormatter instances are thread-safe. Therefore, you can create singleton formatters like the ones defined by the DateTimeFormatter constants and share them among multiple threads.*
- ✓ *You can also mention a specific pattern and Locale as well using the ofPattern() overloaded methods.*

JAVA 8 NEW FEATURES

COMPLETABLEFUTURE

eazy bytes

- Java 8 introduces new class java.util.concurrent.CompletableFuture focusing asynchronous programming and non-blocking code. It is an extension to Java's Future API which was introduced in Java 5.

- This will run tasks on a separate thread than the main application thread and notifying the main thread about its progress, completion or failure. This way, your main thread does not block/wait for the completion of the task and it can execute other tasks in parallel.

- CompletableFuture has below important methods that can be used for async programming,

  ✓ *runAsync() -> Runs async in the background and will not return anything from the task*
  ✓ *supplyAsync() -> Runs async in the background and will return the values from the task*
  ✓ *get() -> It blocks the execution until the Future is completed*
  ✓ *thenApply()/ thenAccept() -> For attaching a callback to the CompletableFuture*

JAVA 8 NEW FEATURES

MAP ENHANCEMENTS

eazy bytes

- Java 8 team provides several default methods inside Map interface. Below are the few important enhancements happened for Map,

  ✓ *forEach() - > Using this we can iterate the map values easily*
  ✓ *Entry.comparingByValue -> Sorting the map elements based on value*
  ✓ *Entry.comparingByKey -> Sorting the map elements based on key*
  ✓ *getOrDefault() – Can be used to pass a default value instead of null if key is not present*
  ✓ *computeIfAbsent() – Can be used to calculate a value if there is no for given key*
  ✓ *computeIfPresent() – If the specified key is present, calculate a new value for it*
  ✓ *Compute() - Calculates a new value for a given key and stores it in the Map*
  ✓ *remove(key,value) – To remove a map element if both key & value matches*
  ✓ *replace() – For replacement of values if the key is available*
  ✓ *replaceAll() - For replacement of all the values inside the map*

JAVA 8 NEW FEATURES

MAP ENHANCEMENTS

- The internal structure of a HashMap was updated in Java 8 to improve performance. Entries of a map typically are stored in buckets accessed by the generated hashcode of the key. But if many keys return the same hashcode, performance deteriorates because buckets are implemented as LinkedLists with O(n) retrieval. But now if the buckets become too big, they're replaced dynamically with sorted trees,which have O(log(n)) retrieval and improve the lookup of colliding elements.

- The ConcurrentHashMap class was update to improve performance while doing read and write operations

- ConcurrentHashMap supports three new kinds of operations,

  ✓ *forEach,reduce,search – Operates with keys and values of the map*
  ✓ *forEachKey, reduceKeys, searchKeys – Operates with keys*
  ✓ *forEachValue, reduceValues, searchValues – Operates with values*
  ✓ *forEachEntry, reduceEntries, searchEntries - Operates with Map.Entry objects*

JAVA 8 NEW FEATURES

OTHER MISCELLANEOUS UPDATES

- Java 8 team made the most out of the default methods introduced and added several new methods in the collection interfaces & other classes. Below is the snapshot of the same,

  - ✓ *List - > replaceAll(), sort()*
  - ✓ *Iterator -> forEachRemaining()*
  - ✓ *Iterable -> forEach(), spliterator()*
  - ✓ *Collection -> parallelStream(), stream(), removeIf()*
  - ✓ *Comparator->reversed(),thenComparing(),naturalOrder(),reverseOrder(),nullsFirst(),nullsLast()*
  - ✓ *Arrays -> setAll(),parallelSetAll(),parallelSort(),parallelPrefix()*
  - ✓ *String -> join()*
  - ✓ *Math -> [add][subtract][multiply][increment][decrement][negate]Exact, toIntExact,floorMod, floorDiv, and nextDown*
  - ✓ *Number -> sum, min, and max static methods in Short, Integer, Long, Float, and Double. Etc.*
  - ✓ *Boolean -> logicalAnd(), logicalOr(), logicalXor().*
  - ✓ *Objects -> isNull(), nonNull()*

JAVA 8 FEATURES SUMMARY

01

DEFAULT METHODS IN INTERFACES

Concrete implemented methods can be written inside interfaces from Java 8 by using default keyword

02

STATIC METHODS IN INTERFACES

Static methods can be written inside Interfaces from Java 8 which can be leveraged to write lot of utility logic inside them.

03

OPTIONAL TO DEAL WITH NULLS

Using Optional we can write clean APIs in terms of null pointer exceptions and it has many methods that supports null checks

04

LAMBDA (Λ) EXPRESSION

New lambda style programming is introduced in Java 8 which will bring functional style inside our business logic

05

FUNCTIONAL INTERFACE

A new type of interfaces called Functional interfaces are introduced in Java 8. This will have only 1 abstract method and they support lambda programming to a great extent.

JAVA 8 FEATURES SUMMARY

06
METHOD REFERENCES
Method references can be used to create simple lambda expressions by referencing existing methods.

07
CONSTRUCTOR REFERENCES
Constructor reference can be used in the place of lambda code to create a new objects by using new operator

08
STREAMS API
java.util.stream API has classes for processing sequence of objects that we usually stored inside the collections.

09
NEW DATE AND TIME API(JODA)
New Data & Time API (java.time.*) is introduced to overcome the challenges with java.util.Date API and provided new features as well

10
COMPLETABLEFUTURE
CompletableFuture is introduced focusing async and non-blocking programming in Java 8.

JAVA 8 FEATURES SUMMARY

11

MAP ENHANCEMENTS

Multiple default and static methods are introduced inside Map interface

12

OTHER MISCELLANEOUS UPDATES

Multiple other new static and default methods are introduced inside collections, Boolean, Numbers, Math, String etc.

JAVA 8 NEW FEATURES

RELEASE NOTES & GITHUB LINK

Apart from the discussed new features, there are many security, non developer focused features are introduced in Java 8. For more details on them, please visit the link

https://www.oracle.com/java/technologies/javase/8all-relnotes.html

GitHub

https://github.com/eazybytes/Java-New-features/tree/main/Java8

THANK YOU & CONGRATULATIONS

YOU ARE NOW A MASTER OF JAVA 8 NEW FEATURES