Java 8 Lambdas & Streams:

Introduction to Functional Programming in Java

What is this Course About

- Introduction To Functional Programming
- Lambda Expressions
- Building new functions and API
- The new collection framework
- Map/ filter/ reduce & collections
- The new Stream API

What you will learn

- To write lambda expressions in Java 8
- To leverage them to create new API
- To efficiently use the new collections
- To avoid useless computation in implementing map/filter/reduce
- To build and use simple stream

Targeted Audience

- This is a Java course
  (write simple Java Code)
- Good knowledge of the language
  (Class, Interface, Simple Methods, Simple Java Structure)
- Basic knowledge of the Java main API
  (Object class , String class, Collection class, I/O class)
- Generics
  (Purpose of generics in Java)
- Collection API
  (What is collection, list, set, map , iterators)

Lambda Expressions in Java 8

- How to write, build and use lambdas

Functional Programming

- ✓ Programming Paradigm
- ✓ Rooted in mathematics.
- ✓ Language independent.
- ✓ Key Principle –
  - ✓ All computation is the execution of mathematical **functions**.

- ✓ Function
  - ✓ Mapping from input to output

- ✓ A programming style that treats computation as the evaluation of mathematical functions avoids changing-state and mutable data.

Functional Programming

- ✓ A programming style that treats computation as the evaluation of mathematical functions avoids changing-state and mutable data.
- ✓ Based on **Lambda Calculus.**
- ✓ **Some inputs are transformed to some output without modifying the input.**

f(x) = x*2

1-> 2

2->4

3->6

……

g(x,y) = f(x) + f(y)

g(20,1) = f(20) + f(1)

40 + 2

42

h(x) = { 1, if xis even

{ 0, if x is odd

Functional Programming

Cannot change the inputs.

For Example: h(x) h(12)

Cannot change variables (Immutable)

`x = x+1` `for(i=0;i<10;i++)`

Programming without variable assignment

Functional Programming

The Guarantee of **immutable**

- ✓ Programming are logical
- ✓ Built for **Distributed Computing.**

blackbox(x)

x = 5;

blackbox(x) == blackbox(x);

x = x+1 x= x+1

6 7

Outcome: True or False or Error

Functional Programming

State: What Is It?

- 1; //a value.
- int x=1; //x is an id that has a value.
- x=x+1; //now x has changed state.

Functional Programming

```
public class Squint {

        public static void main(String[] args) {
        printSquares(20);
        }

        private static void printSquares(int n) {

                if(n>0) {
                        printSquares(n-1);
                        System.out.format("%d, \t%d\n", n, n*n);
                }
        }
}
```

Functional Programming

prominent programming languages which support functional programming such as,

- Common Lisp, Scheme,
- Clojure,
- Wolfram Language[(also known as Mathematica),
- Racket,
- Erlang,
- OCaml,
- Haskell, and
- F#

ANONYMOUS CLASS IN JAVA

- ✓ An *Anonymous class* in Java is a class **not given a name**.
- ✓ *Anonymous classes are* declared and instantiated in a **single statement**.
- ✓ You should consider using an anonymous class whenever you need to create a class that will be instantiated only once
- ✓ An anonymous class must always implement an interface or extend an abstract class
- ✓ An anonymous class must always **implement an interface or extend an abstract class**.
- ✓ However, **you don't use the extends or implements** keyword to create an anonymous class.

```
new interface-or-class-name() { class-body }
```

ANONYMOUS CLASS IN JAVA

Here's an example that implements an interface named runnable, which defines a single method named run

```
Runnable r = new Runnable()
    {
        public void run()
        {
            //code for the run method goes here
        }
    };
```

ANONYMOUS CLASS IN JAVA

Here are a few other important facts concerning anonymous classes:

- ✓ An anonymous class cannot have a constructor. Thus, you cannot pass parameters to an anonymous class when you instantiate it.
- ✓ An anonymous class can access any variables visible to the block within which the anonymous class is declared, including local variables.
- ✓ An anonymous class can also access methods of the class that contains it.

What is lambda expression?

Lambda Expressions: (Functional Programming Style)

- Lambda Expressions are anonymous functions
  (Ways of representing “functions”)

- Instantiate interfaces with a single method
  (Pre-built function building blocks)

- Replace more verbose class declarations

- Set of function composition methods

Implementing an Interface

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Hello!");
    }
}
```

```
MyRunnable r = new MyRunnable();
new Thread(r).start();
```

Using an Anonymous Class

```
new Thread(new Runnable() {
      @Override
      public void run() {
            System.out.println("Hello!");
      }
   }).start();
```

Using a Lambda Expression

- Replace This

```
new Thread(new Runnable() {
      @Override
      public void run() {
            System.out.println("Hello!");
      }
   }).start();
```

- With this

```
(args) -> { body }

Runnable r = () -> System.out.println("Hello!");
new Thread(r).start();
```

Where to Use Lambda Expressions

Important Note:

Lambda expressions can only appear in places where they will be assigned to a variable whose type is a functional interface

Where to Use Lambda Expressions

You get an instance of an inner class that implements Interface

- The expected type must be an interface that has exactly one (abstract) method

Functional Interface

- A functional interface has a single abstract method (SAM)
- Functional interfaces included with Java runtime

  Runnable, Callable, Comparator, TimerTask
- Prior to Java SE 8

  Known as “Single Abstract Method” (SAM) Types.

SAM Interface are now called as Functional Interfaces in Java 8

Lambda Expression Syntax

```
Runnable r = () -> System.out.println("Hello!");
```

Arrow Tokens
(Or)
Lambda Operator

Method Signature

Method Implementation

Lambda Expression Syntax

Lambda Expression With MULTI-LINE Statements

```
Runnable r = () -> {
        System.out.println("Hello Line 1!");
        System.out.println("Hello Line 2!");
};
```

Different Ways of *writing* Lambdas

```
Runnable noArguments = () -> System.out.println("Hello World");

ActionListener oneArgument = event -> System.out.println("button clicked");

Runnable multiStatement = () -> {
System.out.print("Hello");
System.out.println(" World");
};

BinaryOperator<Long> add = (x, y) -> x + y;

BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;
```

Demo 1
Creating Functional Interface

Annotate Functional Interface with @FunctionalInterface Annotation

```
@FunctionalInterface
public interface SimpleInterface {

    public int calculate(int x, int y);

}
```

Should contain only one SAM

Demo 1
One Liner Lambda Expr

```
SimpleInterface si = () -> System.out.println(10*20);
si.calculate();
```

Demo 2
Multi Liner Lambda Expr

```
SimpleInterface si = () -> {
    int data = 10*20;
    System.out.prntln(data);
};

si.calculate();
```

Demo 3
Lambda Expression with Arguments

```
SimpleInterface si = (int x,int y) ->
{
    int data = x*y;
    System.out.println(data);
};

si.calculate(31,23);
```

Demo 3
Lambda Expression with Arguments With Type Inference

```
SimpleInterface si = (x,y) -> {
    int data = x*y;
    System.out.println(data);
};

si.calculate(31,23);
```

Demo 4
Lambda Expression with Arguments & Return Type

```
SimpleInterface si = (x,y) -> {
    int data = x*y;
    return data;
};

si.calculate(24,54);
```

One More Understanding of Functional Interface and Lambda

```
@FunctionalInterface
public interface ITrade {

public boolean check(Trade t);

}


ITrade newTradeChecker = (t) -> t.getTradeStatus().equals("NEW");

Trade t = new Trade();

t.setTradeStatus("OLD");

System.out.println(newTradeChecker.check(t));
```

Built-in Functional Interfaces using Runnable

```
public class UseRunnable {

        public static void main(String[] args){

            Runnable r1 = new Runnable(){

                @Override
                public void run() {
                        System.out.println("Running Thread 1");
                  }
               };

            Runnable r2 = new Runnable(){

                @Override
                public void run() {
                        System.out.println("Running Thread 2");
                   }
                };

                new Thread(r1).start();
                new Thread(r2).start();

        }
}
```

Built-in Functional Interfaces using Runnable

```
Runnable rx1 = () -> System.out.println("Running Thread 1");

Runnable rx2 = () -> System.out.println("Running Thread 2");

new Thread(rx1).start();

new Thread(rx2).start();
```

Built-in Functional Interfaces using Runnable

```
Runnable rx1 = () -> {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Running Thread 1");
};

Runnable rx2 = () -> System.out.println("Running Thread 2");

new Thread(rx1).start();

new Thread(rx2).start();
```

Built-In Functional Interface Comparator Without Lambdas

```
List<String> strings = new ArrayList<>();

strings.add("AAA");
strings.add("EEE");
strings.add("ggg");
strings.add("CCC");
strings.add("ddd");
strings.add("bbb");

//Simple Case-sensitive sort operation
Collections.sort(strings);
System.out.println("Simple Sort:");
for(String str: strings){
        System.out.println(str);
}
```

Code Continued….

Built-In Functional Interface Comparator

```
//Case-sensitive sort with an anonymous class
Collections.sort(strings,new Comparator<String>() {

@Override
public int compare(String s1, String s2) {

        return s1.compareToIgnoreCase(s2);
}
});

System.out.println("Sort with Comparator");

for(String str: strings){
        System.out.println(str);
}
```

Built-In Functional Interface Comparator With Lambdas

```
//Case-sensitive sort with an anonymous class using Lambdas

Comparator<String> comp = (s1,s2) ->  s1.compareToIgnoreCase(s2);

Collections.sort(strings,comp);
```

List of Built-In Functional Interface

- Runnable
- Callable
- Comparator
- ActionListener
- Etc…

Example Domain

Artist

An individual or group who creates music.

- ***name***: The name of the artist (e.g., “The Beatles”)
- ***members***: A set of other artists who comprise this group (e.g., “John Lennon”); this field might be empty
- ***origin***: The primary location of origin of the group (e.g., “Liverpool”).

Track

A single piece of music.

- ***name***: The name of the track (e.g., “Yellow Submarine”)

Album

A single release of music, comprising several tracks.

- ***name***: The name of the album (e.g., “Revolver”)
- ***tracks***: A list of tracks
- ***musicians***: A list of artists who helped create the music on this album

Exercise

- Create Album (POJO) Class as per the usecase
- Create a 5 Album class object with relevant album information
- Add the 5 Album objects into the ArrayList
- Create a Functional Interface named IAlbum and create SAM named sortAlbumByName
- Use Lambda expression to pass the behavior for sortAlbumByName
- Hint: Use In-Built Comparator functional interface for sorting logic

Lambda Expression – Target Typing

Target Typing is achieved by assigning the Lambda Expression in,

- Variable `//Email email = (name)-> name+"@techmahindra.com"`
- Function Parameter

getEmail("SabariBalaji", (name)-> name+"@techmahindra.com"))

Lambda Expression – Target Typing

```
public class TargetType {

    @FunctionalInterface
    public interface Email{
        String constructEmail(String name);
    }

    //Email email = (name)-> name+"@techmahindra.com"

    public String getEmail(String name, Email email){
        return email.constructEmail(name);
    }

    public static void main(String[] args){
        System.out.println("Hi");
        System.out.println(new TargetType().getEmail("SabariBalaji", (name)->
        name+"@techmahindra.com"));

    }
}
```

Capturing in Lambda Expression

- Capturing the variable (which is an argument of the method in which Lambdas are enclosed) inside the Lambda Codes.
- Before Java 8 release. The argument variables used in anonymous classes need to be declared final explicitly by developer.
- Now that challenge is relaxed in Java 8 Anonymous class and in Lambda Codes.

Capturing in Lambda Expression

```
Trade trade = new Trade("IBM",20000,"OPEN");
```

```
public boolean checkStatus(String status){

        ITrade simpleTrade = (t) -> t.getTradeStatus().equals(status) ? true:false;

        return simpleTrade.check(trade);
}
```

```
 public boolean checkFunction( String status){

        ITrade t = new ITrade(){

        @Override
        public boolean check(Trade t) {

                return t.getTradeStatus().equals(status);

           }
        };
        return t.check(trade);
 }
```

NOTE: Local variables are alone treated as **FINAL** (by default in java 8) class or instance variables are not final.

TYPE INFERENCE

Type Inference: How Lambdas input parameter can be type inferred.?

TYPE INFERENCE is not new in Java 8, it is used in Java 7 as Diamond operator (<>)

```
List<String> list = new ArrayList<>();
Map<String,Integer> map = new HashMap<>();
```

So,

TYPE INFERENCE Advantages:

- Java 7 diamond operator “<>”
- Reduces typing
- Easy on eye
- Lambdas with type inference
- Lambdas become expressive
- Not mandatory

TYPE INFERENCE

```
/*(List<Trade> trades,PriceSkewer priceSkewer) -> {
//logic goes here returning trade collection
}*/

/*(trades,priceSkewer) -> {
//logic
}*/

public interface Pricer{

        public List<Trade> skew(List<Trade> trades,PriceSkewer priceSkewer);
}

Pricer priser = (trades,priceSkewer) -> {
        //logic
        return new ArrayList<Trade>();
};
```

Java.util.function

The java.util.function

- A new package from java 8, with the most useful functional interfaces
- There are 43 of them!
- Four Catagories
  - The Consumer
  - The Supplier
  - The Functions
  - The Predicates

What are functions package in JAVA8

```
//What Are Functions
//Check if a movie is classic

interface Movie{
        boolean isClassic(int movieId);
}

interface Person{
        boolean isEmployee(int empId);
}

interface Hospital{
        void admit(Patient patient);
}

interface Tester<T>{
        boolean test(T t);
}

 class WhatAreFunctions {

}
```

Important functional interfaces in Java

Interface name	Arguments	Returns	Example
`Predicate<T>`	`T`	`boolean`	Has this album been released yet?
`Consumer<T>`	`T`	`void`	Printing out a value
`Function<T,R>`	`T`	`R`	Get the name from an `Artist` object
`Supplier<T>`	None	`T`	A factory method
`UnaryOperator<T>`	`T`	`T`	Logical not (`!`)
`BinaryOperator<T>`	`(T, T)`	`T`	Multiplying two numbers (`*`)

Consumer<t> Functional Interface

@FunctionalInterface
public interface Consumer<T>

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side-effects.
This is a functional interface whose functional method is accept(Object).

Since:
1.8

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
void	accept(T t) Performs this operation on the given argument.
default Consumer<T>	andThen(Consumer<? super T> after) Returns a composed Consumer that performs, in sequence, this operation followed by the after operation.

Consumer<t> Functional Interface

```
List<String> strings = new ArrayList<>();

strings.add("AAA");
strings.add("EEE");
strings.add("ggg");
strings.add("CCC");
strings.add("ddd");
strings.add("bbb");

/*for(String str: strings){
System.out.println(str);
}*/

Consumer<String> consumer = str -> System.out.println(str);
strings.forEach(consumer);
```

Predicate<t> Functional Interface

@FunctionalInterface
public interface Predicate<T>

Represents a predicate (boolean-valued function) of one argument.

This is a functional interface whose functional method is test(Object).

Since:
1.8

All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods

Modifier and Type	Method and Description
default Predicate<T>	**and**(Predicate<? super T> other) Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
static <T> Predicate<T>	**isEqual**(Object targetRef) Returns a predicate that tests if two arguments are equal according to **Objects.equals(Object, Object)**.
default Predicate<T>	**negate**() Returns a predicate that represents the logical negation of this predicate.
default Predicate<T>	**or**(Predicate<? super T> other) Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
boolean	**test**(T t) Evaluates this predicate on the given argument.

Predicate<t> Functional Interface

```
List<Person> people = new ArrayList<>();

people.add(new Person("Joe",48));
people.add(new Person("Mary",72));
people.add(new Person("Mike",23));
people.add(new Person("Alvin",76));

        Predicate<Person> pred = new Predicate<Person>(){
                @Override
                public boolean test(Person p) {
                return (p.getPersonAge()>=65);
            }
        };

        Predicate<Person> predOlder = (p) -> p.getPersonAge()>=65;
        Predicate<Person> predYounger = (p) -> p.getPersonAge()<=40;

        people.forEach(p -> {
                if(pred.test(p)){
                System.out.println(p);
            }
        });
```

Method References in Java8

A method reference is the shorthand syntax for a lambda expression that executes just **ONE** method. Here's the general syntax of a method reference:

Object :: methodName

We know that we can use lambda expressions instead of using an anonymous class. But sometimes, the lambda expression is really just a call to some method,

for example:>

```
Consumer<String> c = s -> System.out.println(s);
```

To make the code clearer, you can turn that lambda expression into a method reference:

```
Consumer<String> c = System.out::println;
```

Method References in Java8

In a method reference, you place the object (or class) that contains the method before the :: operator and the name of the method after it without arguments.

First of all, a method reference can't be used for any method. **They can only be used to replace a single-method lambda expression.**

So to use a method reference, you first need a lambda expression with one method.

And to use a lambda expression, you first need a functional interface, an interface with just one abstract method.

In other words:

Instead of using
AN ANONYMOUS CLASS

you can use
A LAMBDA EXPRESSION

And if this just calls one method, you can use
A METHOD REFERENCE

Method References in Java8

Types of Method References

Type	Syntax	Method Reference	Lambda expression
Reference to a static method	*Class::staticMethod*	*String::valueOf*	*s -> String.valueOf(s)*
Reference to an instance method of a particular object	*instance::instanceMethod*	*s:toString*	*() -> “string”.toString()*
Reference to an instance method of an arbitrary object of a particular type	*Class:instanceMethod*	*String::toString*	*s -> s.toString()*
Reference to a constructor	*Class::new*	*String::new*	*() -> new String()*

Method References in Java8

```
@FunctionalInterface
interface IMovie{

      public boolean check(int id);
}

public class MethodRef {

      public static boolean isClassic(int id){
            return true;
      }

      public static void main(String[] args) {
            IMovie movie =  MethodRef :: isClassic;
      }
```

Method References in Java8

```
public class Person {

private String personName;
private int personAge;

}

/*Collections.sort(people, Person :: compareAges);*/
Collections.sort(people, this :: compareAges);
people.forEach((p) -> System.out.println(p));

public  int compareAges(Person p1,Person p2){
        Integer age1 = p1.getPersonAge();
        return age1.compareTo(p2.getPersonAge());
}


public static int compareAges(Person p1,Person p2){
        Integer age1 = p1.getPersonAge();
        return age1.compareTo(p2.getPersonAge());
}
```

Default Method in Interface – JAVA8

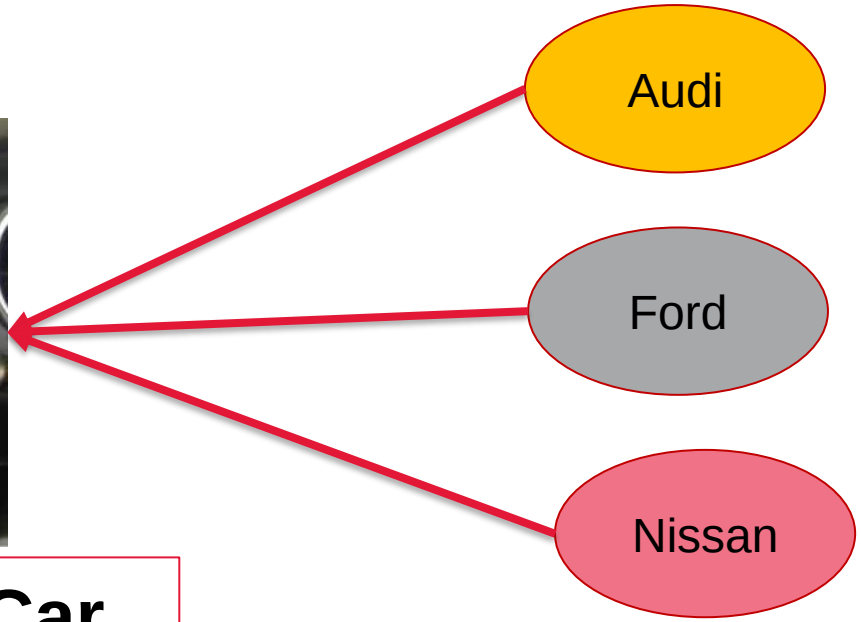
Default Method in Interface – Inheritance

```
interface Engine{
	default String make(){
	return "Default Engine";
	}
}

interface Vehicle{
	default String model(){
	return "Default Vehicle";
	}
}

class car implements Engine,Vehicle{
String makeAndModel(){
return Engine.super.make()+Vehicle.super.model();
}
}
```

Default Method in Interface – JAVA8

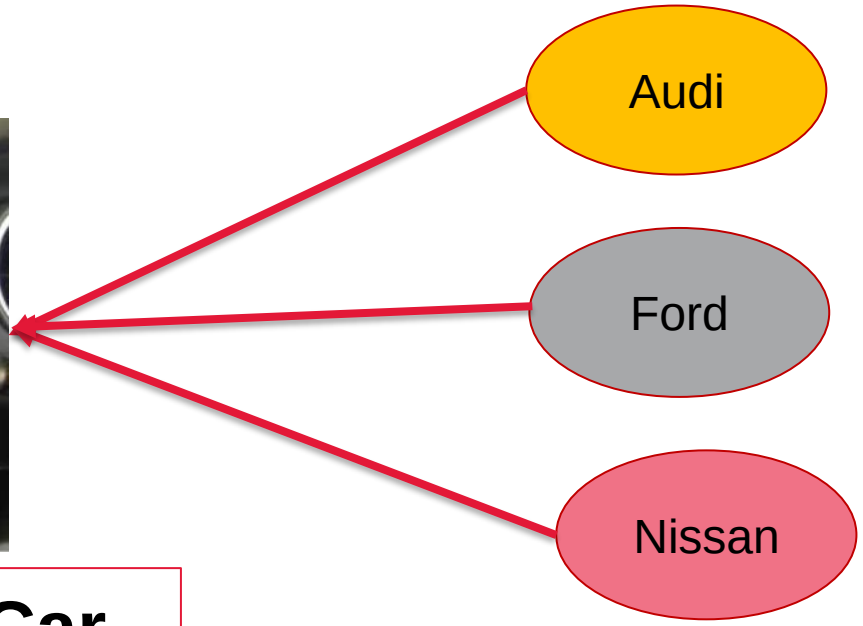
Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

An example that involves manufacturers of computer-controlled cars who publish industry-standard interfaces that describe which methods can be invoked to operate their cars.



Computer Controller Car Interface Manufacturer

Default Method in Interface – JAVA8

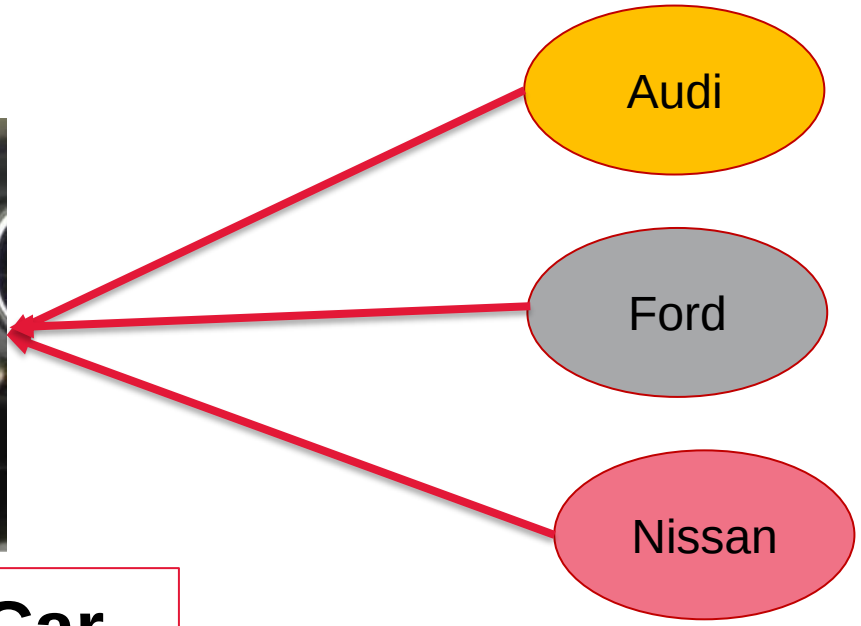


Computer Controller Car Interface Manufacturer

```
int turn(Direction direction,double radius,double);
int changeLanes(Direction direction,double startSpeed,double endSpeed);
int getRadarFront(double distanceToCar,double speedOfCar);
```

What if those computer-controlled car manufacturers add new functionality, **such as flight**, to their cars?

Default Method in Interface – JAVA8



Computer Controller Car Interface Manufacturer

```
int turn(double radius,double startSpeed, double endSpeed);
int changeLanes(Direction direction,double startSpeed,double endSpeed);
int getRadarFront(double distanceToCar,double speedOfCar);
```

int flight(double altitude,double speedOfCar);

Default Method in Interface – JAVA8

These manufacturers would need to specify new methods to enable other companies (such as electronic guidance instrument manufacturers) to adapt their software to flying cars.

Audi

Ford

Nissan

Where would these car manufacturers declare these new flight-related methods?

If they add them to their original interfaces, then programmers who have implemented those interfaces would have to rewrite their implementations. If they add them as static methods, then programmers would regard them as utility methods, not as essential, core methods.

Default Method in Interface – JAVA8

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

Default Method in Interface – JAVA8

```
public interface IComputerInterfaceCar {

int turn(double radius,double startSpeed, double endSpeed);
int changeLanes(String direction,double startSpeed,double endSpeed);
int getRadarFront(double distanceToCar,double speedOfCar);

        default String flight(double altitude, double speedOfCar){
                return "Car Can Implement Flight";
        }
}
  public class Audi implements IComputerInterfaceCar {
  }

  public class Ford implements IComputerInterfaceCar {
  }

  public class Nissan implements IComputerInterfaceCar {
  }
```

Collections– Java 8

Collections is the most heavily used API in Java.

What would you do without collections?

> Collections are fundamental to many programming tasks: they let you group and process data.

To illustrate collections in action,

- ✓ Imagine you want to create a collection of dishes to represent a menu and then iterate through it to sum the calories of each dish.
- ✓ You may want to process the collection to select only low-calorie dishes for a special healthy menu.

Collections– Java 8

How would you process a large collection of elements?

- ✓ To gain performance you'd need to process it in parallel
- ✓ And leverage multicore architectures.
- ✓ But writing parallel code is complicated in comparison to working with iterators
- ✓ In addition, it's no fun to debug!

So, what could the Java language designers do to save your precious time and **make your life easier as programmers?** You may have guessed: the answer is ***streams***.

Working With Streams – Java 8

Java SE 8 Stream API is designed to help you manage collections of data i.e objects that are members of the collection framework.

such as,

ArrayList
HashMap
List
etc...

Package for Streams : java.util.stream

Collections vs Streams – Java 8

A collection is an in-memory data structure to hold values and before we start using collection, all the values should have been populated.

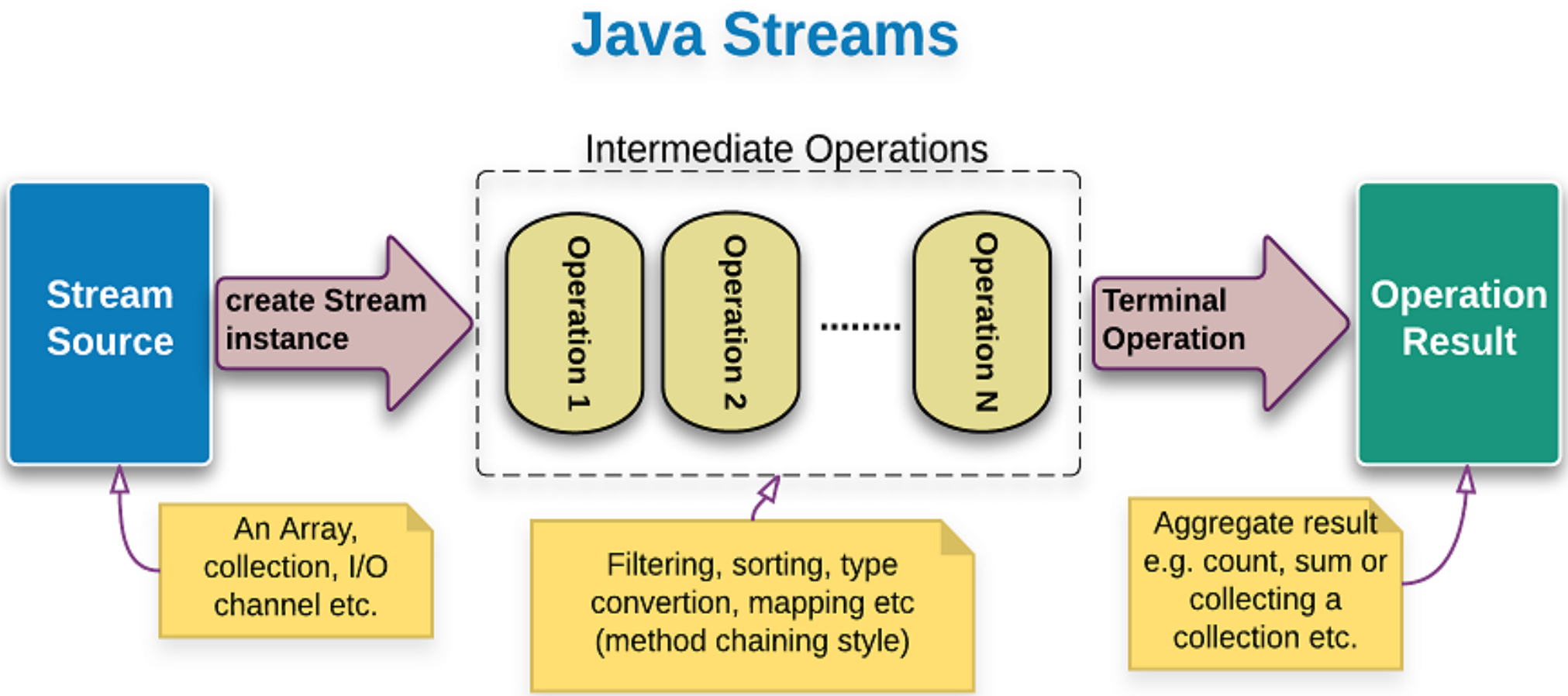
Whereas a java Stream is a data structure that is computed on-demand.

Java Stream doesn’t store data, it operates on the source data structure (collection and array) and produce pipelined data that we can use and perform specific operations.

Such as we can create a stream from the list and filter it based on a condition

Java Stream operations use functional interfaces, that makes it a very good fit for functional programming using lambda expression

Working With Streams – Java 8

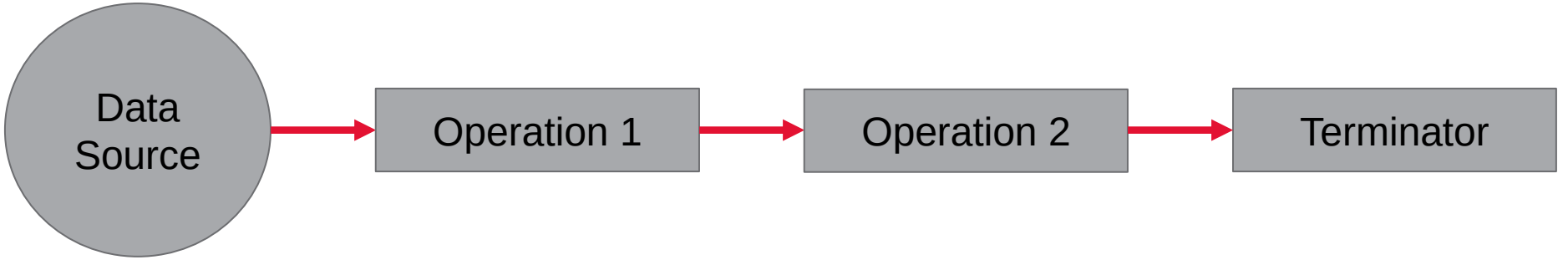


Stream Operation – Java 8

A stream life cycle can be divided into three types of operation:

- Obtaining the instance of Stream from a source. A source might be an array, a collection, a generator function, an I/O channel, etc
- Zero or more **intermediate operations** which transform a stream into another stream, such as filtering, sorting, element transformation (mapping)
- A **terminal operation** which produces a result, such as count, sum or a new collection.

What is Stream Pipeline?



- A stream pipeline is nothing but combined **intermediate and terminal operations**
- Many stream operations **return a stream** themselves
- This allows operations to be chained to form a larger pipeline.

Definitions

- ☑ A stream **is** a pipeline of functions that can be evaluated.
- ☑ Streams **can** transform data.
- ☒ A stream **is not** a data structure.
- ☒ Streams **cannot** mutate data.

Intermediate operations

- Always return streams.
- Lazily executed.

Common examples include:

Function	Preserves count	Preserves type	Preserves order
map	✓	✗	✓
filter	✗	✓	✓
distinct	✗	✓	✓
sorted	✓	✓	✗
peek	✓	✓	✓

Stream examples

Get the unique surnames in uppercase of the first 15 book authors that are 50 years old or over.

```
library.stream()
     .map(book -> book.getAuthor())
     .filter(author -> author.getAge() >= 50)
     .distinct()
     .limit(15)
     .map(Author::getSurname)
     .map(String::toUpperCase)
     .collect(toList());
```

Compute the sum of ages of all female authors younger than 25.

```
library.stream()
     .map(Book::getAuthor)
     .filter(a -> a.getGender() == Gender.FEMALE)
     .map(Author::getAge)
     .filter(age -> age < 25)
     .reduce(0, Integer::sum);
```

Terminal operations

- Return concrete types or produce a side effect.
- Eagerly executed.

Common examples include:

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach	side effect	to perform a side effect on elements

Parallel streams

Parallel streams use the common ForkJoinPool for threading.

```
library.parallelStream()...
```

or intermediate operation:

```
IntStream.range(1, 10).parallel()...
```

Useful operations

Grouping:

```
library.stream().collect(
    groupingBy(Book::getGenre));
```

Stream ranges:

```
IntStream.range(0, 20)...
```

Infinite streams:

```
IntStream.iterate(0, e -> e + 1)...
```

Max/Min:

```
IntStream.range(1, 10).max();
```

FlatMap:

```
twitterList.stream()
     .map(member -> member.getFollowers())
     .flatMap(followers -> followers.stream())
     .collect(toList());
```

Pitfalls

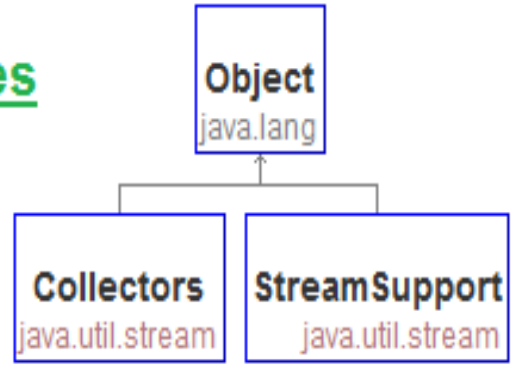
☒ Don't update shared mutable variables i.e.

```
List<Book> myList =
new ArrayList<>();
library.stream().forEach
(e -> myList.add(e));
```

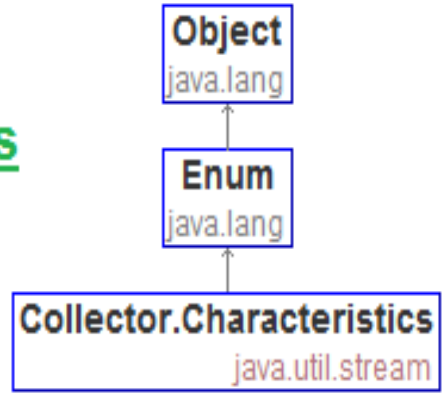
☒ Avoid blocking operations when using parallel streams.

java.util.stream

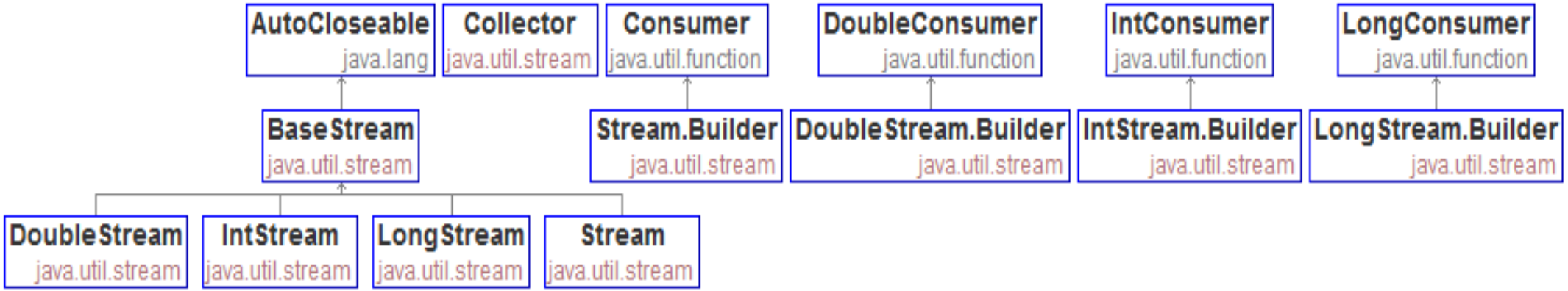
Classes



Enums



Interfaces



Obtaining stream instances from the source ?

Java 8 has modified the existing collection and other data structures API to create/generate stream instances

Collection interface has added new default methods like stream() and parallelStream() which return a Stream instance.

```
List<String> list = Arrays.asList("1", "2", "3");
Stream<String> stream = list.stream();
-----------------------------------------
Stream<Person> streams = Stream.of(people);
-----------------------------------------
String[] strs = {"1", "2", "3"};
Stream<String> stream = Arrays.stream(strs);
```

Working With Streams – Java 8

```
Person[] people = {

    new Person("Mike",12),
    new Person("Mary",34),
    new Person("Joe",54),
    new Person("Philip",22),
};

Stream<Person> stream = Stream.of(people);

stream.filter(p -> p.getPersonAge()>30)
      .forEach(p -> System.out.println(p));
```

Imperative vs Declarative styles of coding

In **imperative programming**, we have to write code line by line to give instructions to the computer about what we want to do to achieve a result.

For example iterating through a collection of integer using 'for loop' to calculate sum is an imperative style.

In **declarative programming**, we just have to focus on what we want to achieve without repeating the low level logic (like looping) every time.

Stream API achieve this by using internal iterator constructs along with lambda expressions.

Examples

In following examples we will compare the old imperative style with new declarative style.

Old Imperative Style	New Declarative Style

`List<String> list = Arrays.asList("Apple","Orange","Banana");`
 	
`for (String s : list) {`
`System.out.println(s);`
`}`
	`List<String> list = Arrays.asList("Apple","Orange", "Banana");`
	
	`//using lambda expression`
	
	`list.forEach(s -> System.out.println(s));`
	
	`//or using method reference on System.out instance`
	`list.forEach(System.out::println);`

Examples

In following examples we will compare the old imperative style with new declarative style.

Counting even numbers in a list, using Collection.stream() and java.util.stream.Stream

Old Imperative Style

```
List<Integer> list;
List = Arrays.asList(3, 2, 12);

int count = 0;

for (Integer i : list) {
    if (i % 2 == 0) {
        count++;
    }
}
System.out.println(count);
```

New Declarative Style

```
List<Integer> list =
        Arrays.asList(3, 2, 12);

long count = list.stream()
                 .filter(i -> i % 2 == 0)
                 .count();
System.out.println(count);
```

Sequential vs Parallel streams

Parallel streams divide the provided task into many and run them in different threads, utilizing multiple cores of the computer.

On the other hand sequential streams work just like for-loop using a single core.

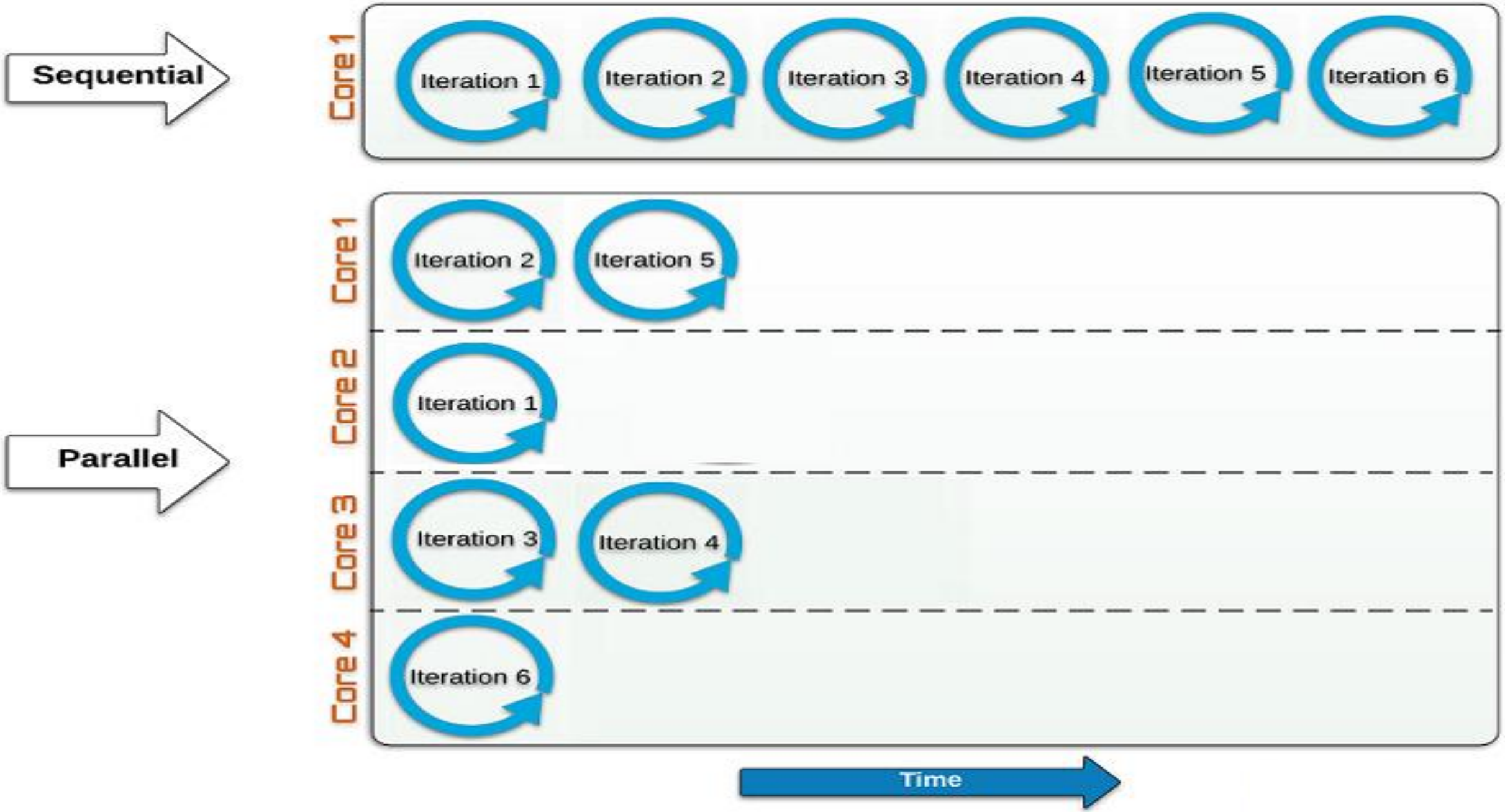
The tasks provided to the streams are typically the iterative operations performed on the elements of a collection or array or from other dynamic sources

Parallel execution of streams run multiple iterations simultaniously in different available cores.

In parallel execution, if number of tasks are more than available cores at a given time, the remaining tasks are queued waiting for currently running task to finish.

NOTE: It is also important to know that iterations are only performed at a terminal operation, that's because streams are **designed to be lazy**.

Sequential vs parallel streams running in 4 cores

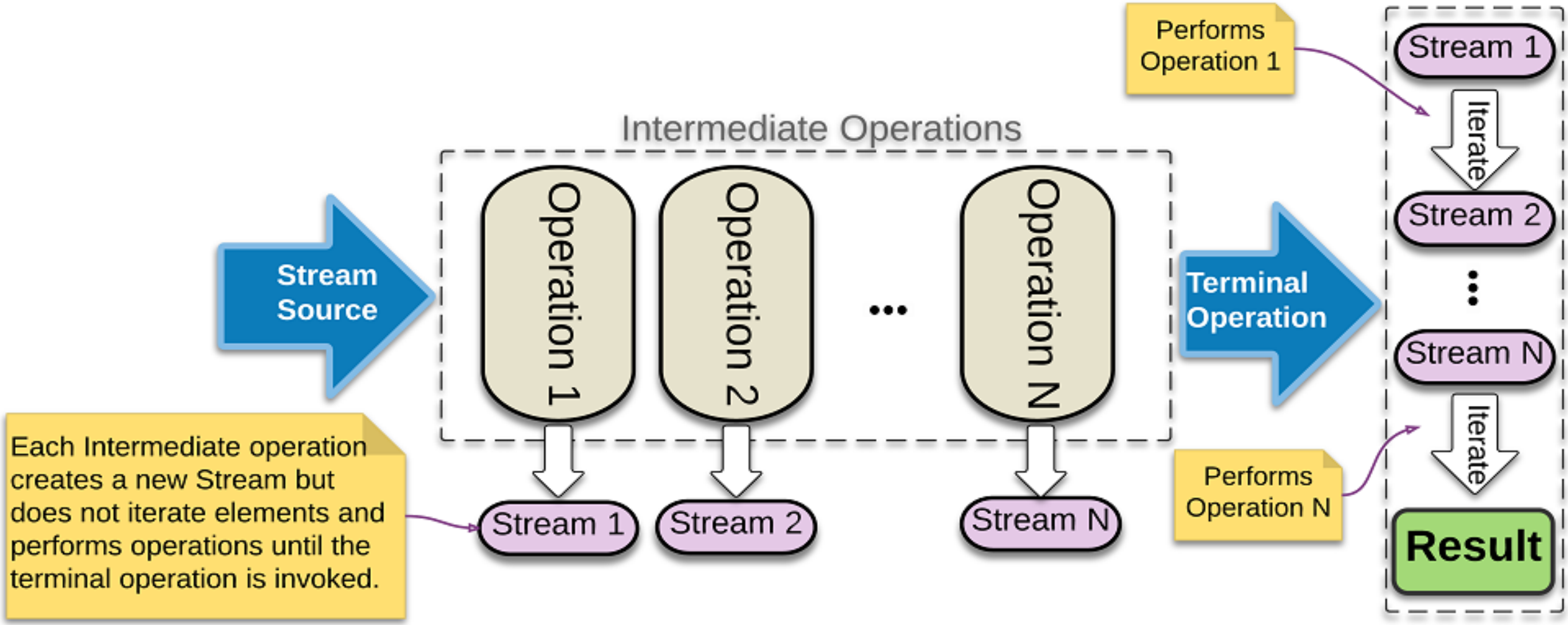


Java 8 Streams - Lazy evaluation

- Streams are lazy because intermediate operations are not evaluated until terminal operation is invoked.
- Each intermediate operation creates a new stream, stores the provided operation/function and return the new stream.
- The pipeline accumulates these newly created streams.
- The time when terminal operation is called, traversal of streams begins and the associated function is performed one by one.

Java 8 Streams - Lazy evaluation

Stream Lazy Evaluation



Stream operations

Obtaining a Stream ⇨ **Intermediate Operation** ⇨ **Terminal Operations**

Collection
stream()
parallelStream()

Stream
IntStream
LongStream
DoubleStream
static generate()Unordered
static of(..)
static empty()
static iterate(..)
static concat(..)
static builder()

BaseStream
sequential()
parallel()
unordered()
onClose(..)

Stream
filter(..)
map(..)
mapToInt(..)
mapToLong(..)
mapToDouble(..)
flatMap(..)
flatMapToInt(..)
flatMapToLong(..)
flatMapToDouble(..)

BaseStream
iterator()
spliterator()

Stream
forEach(..)
forEachOrdered(..)
toArray(..)
reduce(..)
collect(..)
min(..)
max(..)
count()
anyMatch(..)short-circuiting
allMatch(..)short-circuiting
noneMatch(..)short-circuiting
findFirst()short-circuiting
findAny()short-circuiting,

Streams Java7 Vs Java 8

Streams Java7

Scenario: Return the names of Dishes that are Low in calories

- ✓ Create a Dish Class (POJO - Class)
- ✓ Create a MenuApplication Class.
- ✓ Create List<Dish> of dishes.
- ✓ Filter the element using Accumulator
- ✓ Sort the dishes with an Anonymous class
- ✓ Process the sorted list to select the names of dishes
- ✓ And print the names of dishes which has low calories

Streams Java7

```
List<Dish> menu = Arrays.asList(
    new Dish("pork", false, 800, Dish.Type.MEAT)
);

List<Dish> lowCaloriesDishes = new ArrayList<>();
//Filter the element using Accumulator
for(Dish d : menu){
if(d.getCalories()>400){
lowCaloriesDishes.add(d);
}
}
```

Streams Java7

```
//Sort the dishes with an anonymous class
Comparator<Dish> comp = new Comparator<Dish>() {

@Override
public int compare(Dish d1, Dish d2) {
        return Integer.compare(d1.getCalories(), d2.getCalories());
        }
};
Collections.sort(lowCaloriesDishes, comp );


//Process the sorted list to select the names of dishes
List<String> lowCaloriesDishNames = new ArrayList<>();
    for(Dish dish : lowCaloriesDishes){
        lowCaloriesDishNames.add(dish.getName());
    }
```

Streams Java7

```
//Print all the low calories dish names.

for(String name : lowCaloriesDishNames)
System.out.println(name);
```

Streams Java 8

Scenario: Return the names of Dishes that are Low in calories

- ✓ Create a Dish Class (POJO - Class)
- ✓ Create a MenuApplication Class.
- ✓ Create List<Dish> of dishes.
- ✓ Select the dishes that are below 400 calories
- ✓ Sort them by Calories
- ✓ Extract the names of the dishes
- ✓ Store all the names in the list
- ✓ Print the names

Streams Java8

```
menu.stream()
   .filter(d -> d.getCalories()<500)
   .sorted((d1,d2) ->
            Integer.compare(d1.getCalories(),d2.getCalories()))
   .map(Dish::getName)
   .collect(toList())
   .forEach(System.out :: println);
```

Java 8 Date & Time API

- ✓ JDK 8 will have a new Date & Time API
- ✓ Replaces
  - ✓ java.util.Date
  - ✓ java.util.Calendar
  - ✓ java.util.TimeZone
  - ✓ java.util.DateFormat
- ✓ No need to use Joda Time
  - ✓ (**joda**-**Time** provides a quality replacement for the Java date and **time** classes. **Joda**-**Time** is the de facto standard date and **time** library for Java prior to Java SE 8.)
  - ✓ **Joda**-**Time** provides support for multiple calendar systems and the full range of **time**-zones. The Chronology and DateTimeZone classes provide this support. **Joda**-**Time** defaults to using the ISO calendar system, which is the de facto civil calendar used by the world.

Java 8 Date & Time API

- ✓ New API, designed for and integrated into JDK 8
- ✓ Built from Scratch , but inspired by Joda-Time
- ✓ Comprehensive model for Date and Time
- ✓ Supporting commonly used global calendars
- ✓ Immutable to work well with lambdas/functional

Java 8 Date & Time API

- ✓ java.time.*
- ✓ LocalDate

```
LocalDate date = LocalDate.now();

date = date.plusDays(2).plusMonths(2);
date = date.withDayOfMonth(1);
date = date.withMonth(10);
date = date.with(Month.OCTOBER);

System.out.println(date.getDayOfWeek());
```

Java 8 Date & Time API

- ✓ Local Date Adjuster
- ✓ More complex alterations use adjusters
  - ✓ Change date to last day of month
  - ✓ Change date to 3rd Friday of next month

```
LocalDate date = LocalDate.now();


date = date.with(TemporalAdjusters.next(DayOfWeek.TUESDAY));
date = date.with(TemporalAdjusters.firstDayOfNextMonth());

System.out.println(date.getDayOfWeek());
```

Java 8 Date & Time API

- ✓ Java class – LocalTime
  - ✓ Stores hours-minutes-seconds-nanoseconds

```
LocalTime time = LocalTime.now();

System.out.println(time.getHour());
System.out.println(time.getMinute());
System.out.println(time.getSecond());
System.out.println(time.getNano());

time = time.truncatedTo(ChronoUnit.NANOS);

System.out.println(time.toString());
```

Thank you