

# About the Course

# About the Instructor

# Course Objectives

# Targeted Audience

**Code for this course**

**Thank you !  
Hope to see you in the  
coursre.**

# Why Java 8 ?

- Most popular and widely accepted language in the world.
- Java creators wanted to introduce the Functional features such as:
  - Lambdas
  - Streams
  - Optional and etc.,
- Technological advancements with the mobile/laptops/systems.
- New **Java 8** features simplify the concurrency operations.

# Functional Programming:

- Embraces creating Immutable objects.
- More concise and readable code.
- Using functions/methods as first class citizens.

## Example:

```
Function<String,String> addSomeString = (name) ->  
name.toUpperCase().concat("default");
```

- Write code using **Declarative** approach.



# **Imperative vs Declarative Programming**

# Imperative Style of Programming

- Focuses on how to perform the operations.
- Embraces Object mutability.
- This style of programming lists the step by step of instructions on how to achieve an objective.
- We write the code on what needs to be done in each step.
- Imperative style is used with classic Object Oriented Programming.

# Declarative Style of Programming

- Focuses on what is the result you want.
- Embraces Object immutability.
- Analogous to **SQL** (Structured Query Language).
- Use the functions that are already part of the library to achieve an objective.
- Functional Programming uses the concept of declarative programming.

# Imperative vs Declarative Programming

## Example 1

Sum of 100 numbers from 0 to 100

# Imperative vs Declarative Programming

## Example 2

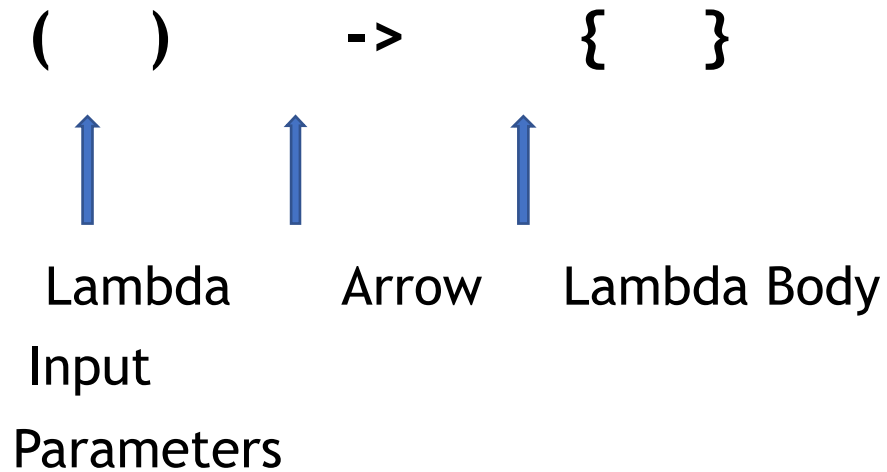
Removing duplicates from a list of integers

# What is Lambda Expression?

- **Lambda** is equivalent to a function (method) without a name.
- Lambda's are also referred as **Anonymous** functions.
  - Method parameters
  - Method Body
  - Return Type
- Lambdas are not tied to any class like a regular method.
- Lambda can also be assigned to variable and passed around.

# Syntax of the Lambda Expression

Lambda Expression:



# Usages of Lambda

- Lambda is mainly used to implement Functional Interfaces(SAM).

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```



# **Lets code our first Lambda!**

Implement Runnable using Lambda

# Lambda in Practice ( Things to keep in Mind)

( ) -> Single Statement or Expression; // curly braces are not needed.

( )-> { <Multiple Statements> }; // curly braces are needed for multiple statements

# Lambdas vs Legacy Java(until Java7)

## Legacy:

```
Runnable runnable = new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Inside Runnable 1");  
    }  
};
```

## Java 8:

```
Runnable runnableLambda = () -> {System.out.println("Inside Runnable  
2");};
```

# Functional Interfaces

- Exists since Java 1.0

## Definition:

- A Functional Interface(SAM) is an interface that has exactly one abstract method.

## @FunctionalInterface:

- This annotation is introduced as part of the JDK 1.8.
- Optional annotation to signify an interface as Functional Interface.

# New Functional Interfaces in Java8

- Consumer
- Predicate
- Function
- Supplier

# New Functional Interfaces in Java8

- Consumer - BiConsumer
- Predicate - BiPredicate
- Function - BiFunction, UnaryOperator, BinaryOperator
- Supplier

# New Functional Interfaces in Java8

- **Consumer** - IntConsumer, DoubleConsumer, LongConsumer
- **Predicate** - IntPredicate, BiPredicate, LongPredicate
- **Function** - IntFunction, DoubleFunction, LongFunction, IntToDoubleFunction, IntoLongFunction, DoubleToIntFunction, DoubleToLongFunction, LongToIntFunction, LongToDoubleFunction, ToIntFunction, ToDoubleFunction, ToLongFunction
- **Supplier** - IntSupplier, LongSupplier, DoubleSupplier, BooleanSupplier

# Method Reference

- Introduced as part of Java 8 and its purpose is to simplify the implementation Functional Interfaces.
- Shortcut for writing the **Lambda Expressions**.
- Refer a method in a class.



# Syntax of Method Reference

ClassName::instance-methodName

ClassName::static-methodName

Instance::methodName

# Where to use Method Reference?

- Lambda expressions referring to a method directly.

## Using Lambda:

```
Function<String,String> toUpperCaseLambda = (s)->s.toUpperCase();
```

## Using Method Reference:

```
Function<String,String> toUpperCaseMethodRefernce =  
String::toUpperCase;
```

# Where Method Reference is not Applicable ?

## Example:

```
Predicate<Student> predicateUsingLambda = (s) -> s.getGradeLevel()>=3;
```

# Constructor Reference

- Introduced as part of Java 1.8

## Syntax:

Classname::new

## Example:

```
Supplier<Student> studentSupplier = Student::new;
```

## Invalid:

```
Student student = Student::new; // compilation issue
```

# Lambdas and Local Variables

What is a **Local variable** ?

- Any variable that is declared inside a method is called a local variable.
- Lambdas have some restrictions on using local variables:
  - Not allowed to use the same the local variable name as **lambda parameters** or inside the **lambda body**.
  - Not allowed **re-assign** a value to a local variable.
- No restrictions on **instance** variables.

# Local Variables - Not Allowed

## Repeated Variable Name:

- Variable **i** is declared in the same scope and used as a parameter in Lambda.
- You cannot use the same variable as a lambda parameter or inside the lambda body.

Same Variable as Input:

```
int i=0; //Repeated varibale name not allowed
Consumer<Integer> c1 = (i) -> {
    System.out.println(i+value);
};
```

# Local Variables - Not Allowed

Same Variable as Lambda parameter:

```
int i=0;  
Consumer<Integer> c1 = (i) -> { //Repeated variable name not  
allowed  
    System.out.println(i+value);  
};
```

Same Variable in Lambda Body:

```
int i=0;  
Consumer<Integer> c1 = (a) -> {  
    int i=0; //Repeated variable name not allowed  
    System.out.println(i+value);  
};
```



# Local Variables - Not Allowed

- Not allowed to modify the value inside the lamda

```
int value =4;  
Consumer<Integer> c1 = (a) -> {  
    //value=6; //reassigning not allowed  
    // System.out.println(i+value);  
};
```

# Effectively Final

- Lambda's are allowed to use local variables but not allowed to modify it even though they are not declared final. This concept is called **Effectively Final**.
- Not allowed to modify the value inside the lamda

```
int value =4;
Consumer<Integer> c1 = (a) -> {
    //value=6; //reassigning not allowed
    // System.out.println(i+value);
};
```
- Prior to Java 8 , any variable that's used inside the anonymous class should be declared **final**.

# Advantages of Effectively Final:

- Easy to perform concurrency operations.
- Promotes Functional Programming and demotes the Imperative style programming.

# Introduction to Streams API:

- Introduced as part of **Java8**
- Main purpose is to perform some **Operation on Collections**.
- **Parallel operations** are easy to perform with Streams API without having to spawn a multiple threads.
- Streams API can be also used with arrays or any kind of I/O .

# What is a Stream ?

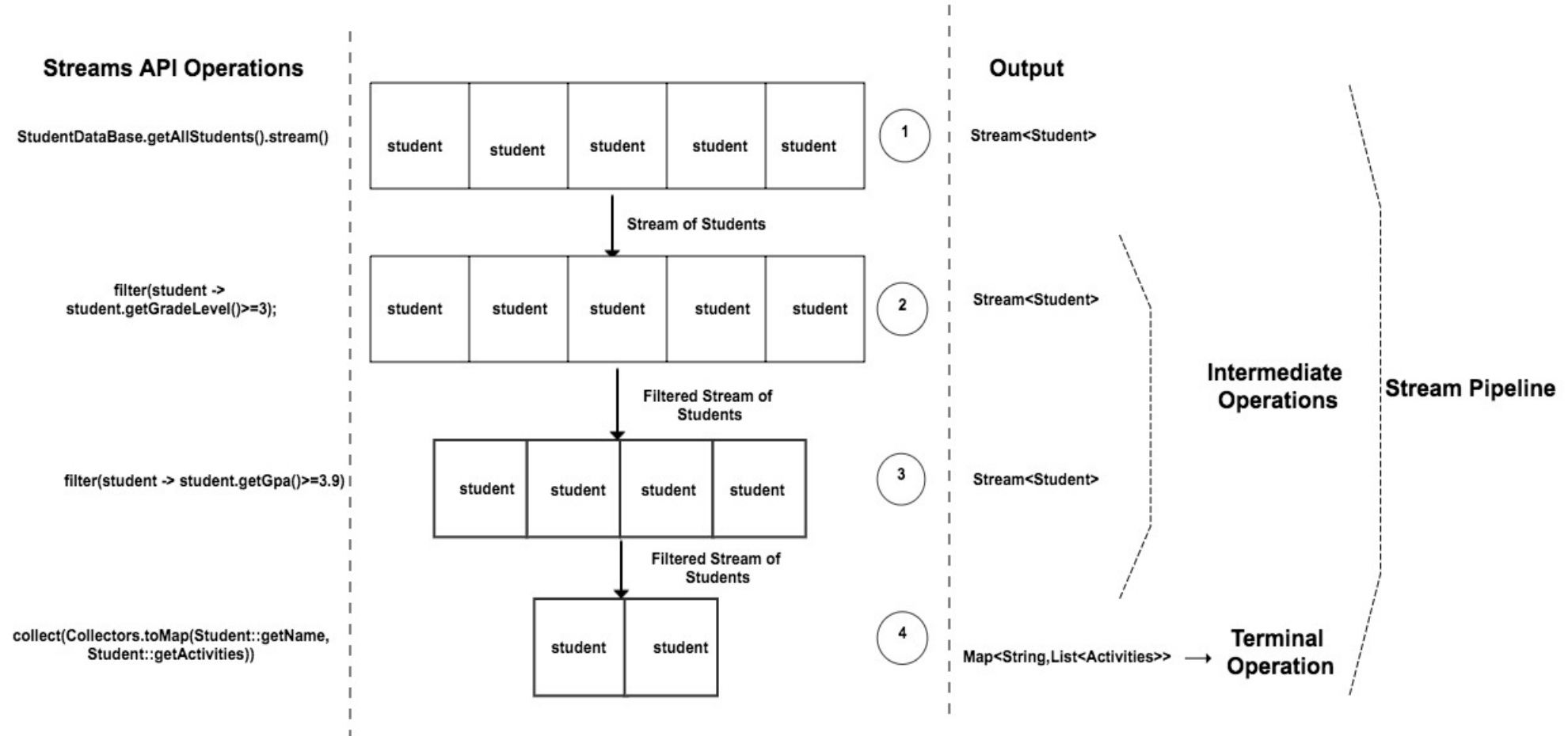
- Stream is a sequence of elements which can be created out of a collections such as **List** or **Arrays** or any kind of **I/O** resources and etc.,

```
List<String> names = Arrays.asList("adam","dan","jenny");  
names.stream(); // creates a stream
```

- Stream operations can be performed either **sequentially** or **parallel**.

```
names.parallelStream(); // creates a parallel stream
```

# How Stream API Works ?



# Collections and Streams

Collections	Streams
Can add or modify elements at any point of time. For Example: <b>List -&gt; list.add(&lt;element&gt;)</b>	Cannot add or modify elements in the stream. It is a fixed data set.
Elements in the collection can be accessed in any order. Use appropriate methods based on the collection. For Example: <b>List -&gt; list.get(4);</b>	Elements in the Stream can be accessed only in sequence.
Collection is eagerly constructed.	Streams are lazily constructed.

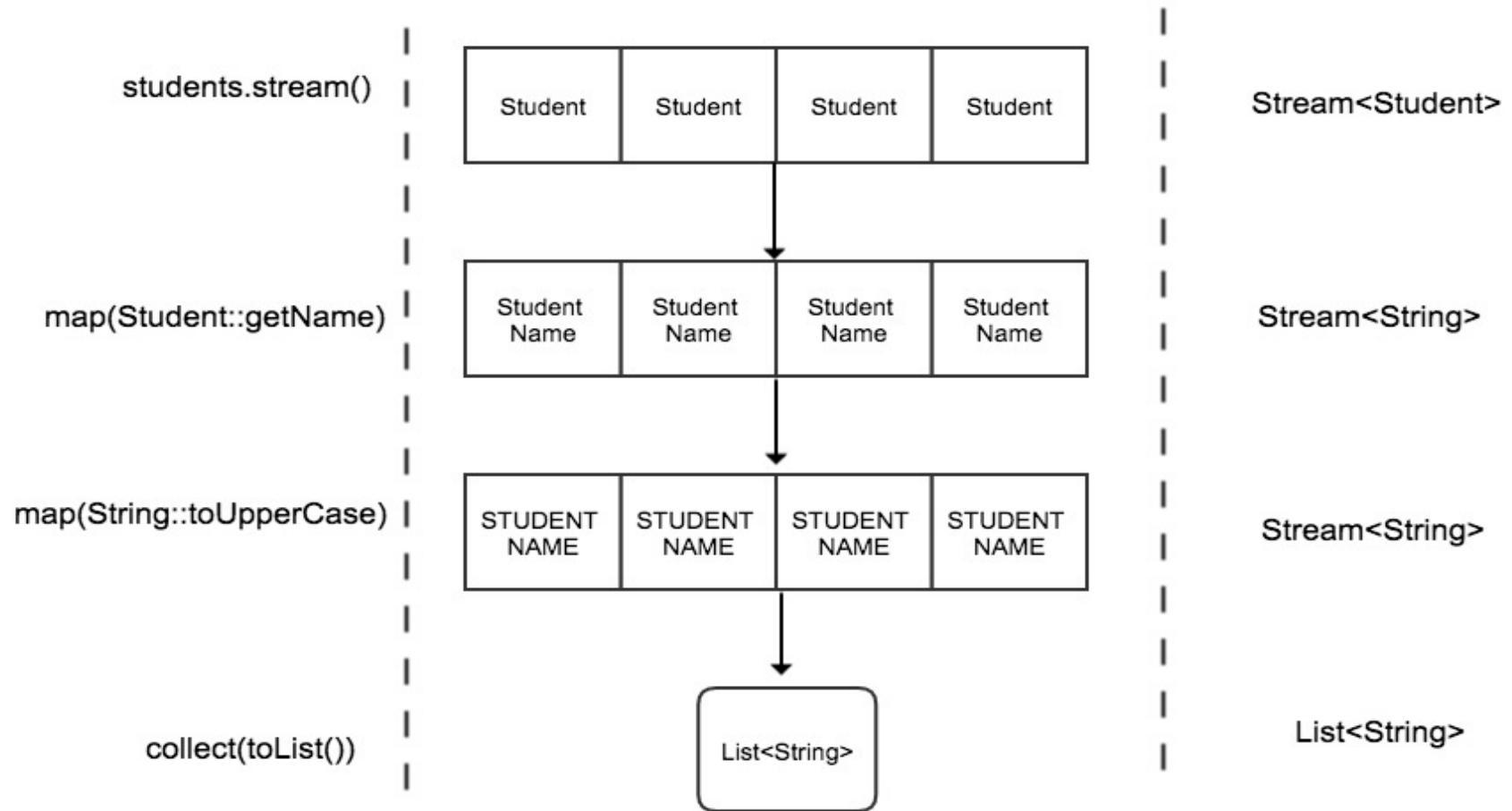
# Collections and Streams

Collections	Streams
Collections can be traversed “n” number of times.	Streams can be traversed only once.
Performs <b>External Iteration</b> to iterate through the elements.	Performs <b>Internal Iteration</b> to iterate through the elements.



# Stream API : map()

- **map** : Convert(transform) one type to another.
- Don't get confused this with **Map** Collection.



# Stream API : flatMap()

- **flatMap** : Converts(Transforms) one type to another as like map() method
- Used in the context of Stream where each element in the stream represents multiple elements.

## Example:

- Each Stream element represents multiple elements.
  - Stream<List>
  - Stream<Arrays>

# Stream API - `distinct()` , `count()` and `sorted()`

- **distinct** - Returns a stream with unique elements
- **count** - Returns a long with the total no of elements in the Stream.
- **sorted** - Sort the elements in the stream

# Stream API - filter()

- **filter** - filters the elements in the stream.

Input to the filter is a **Predicate** Functional Interface.

# Streams API - reduce()

- **reduce** - This is a terminal operation. Used to reduce the contents of a stream to a single value.
- It takes two parameters as an input.
  - **First parameters** - default or initial value
  - **Second Parameter** - BinaryOperator<T>

# Stream API : Max/Min using reduce()

- **max** -> Maximum(largest) element in the stream.
- **min** -> Minimum(smallest) element in the stream.

# Stream API : `limit()` and `skip()`

- These two function helps to create a sub-stream.
- `limit(n)` - limits the “n” numbers of elements to be processed in the stream.
- `skip(n)` - skips the “n” number of elements from the stream.



# Streams API : `anyMatch()`, `allMatch()` , `noneMatch()`

- All these functions takes in a **predicate** as an input and returns a **Boolean** as an output.
- **`anyMatch()`**- Returns **true** if any one of the element matches the predicate, otherwise false.
- **`allMatch()`** - Returns **true** if all the element in the stream matches the predicate, otherwise false.
- **`noneMatch()`** - Just opposite to **`allMatch()`**. Returns **true** if none of the element in the stream matches the predicate, otherwise false.

# Streams API : **findFirst()** and **findAny()**

- Used to find an element in the stream.
- Both the functions returns the result of type **Optional**.
- **findFirst()** - Returns first element in the stream.
- **findAny()** - Returns the first encountered element in the stream.

# **Streams API - Short Circuiting**

# What is Short Circuiting ?

## Examples of Short Circuiting:

### Example 1:

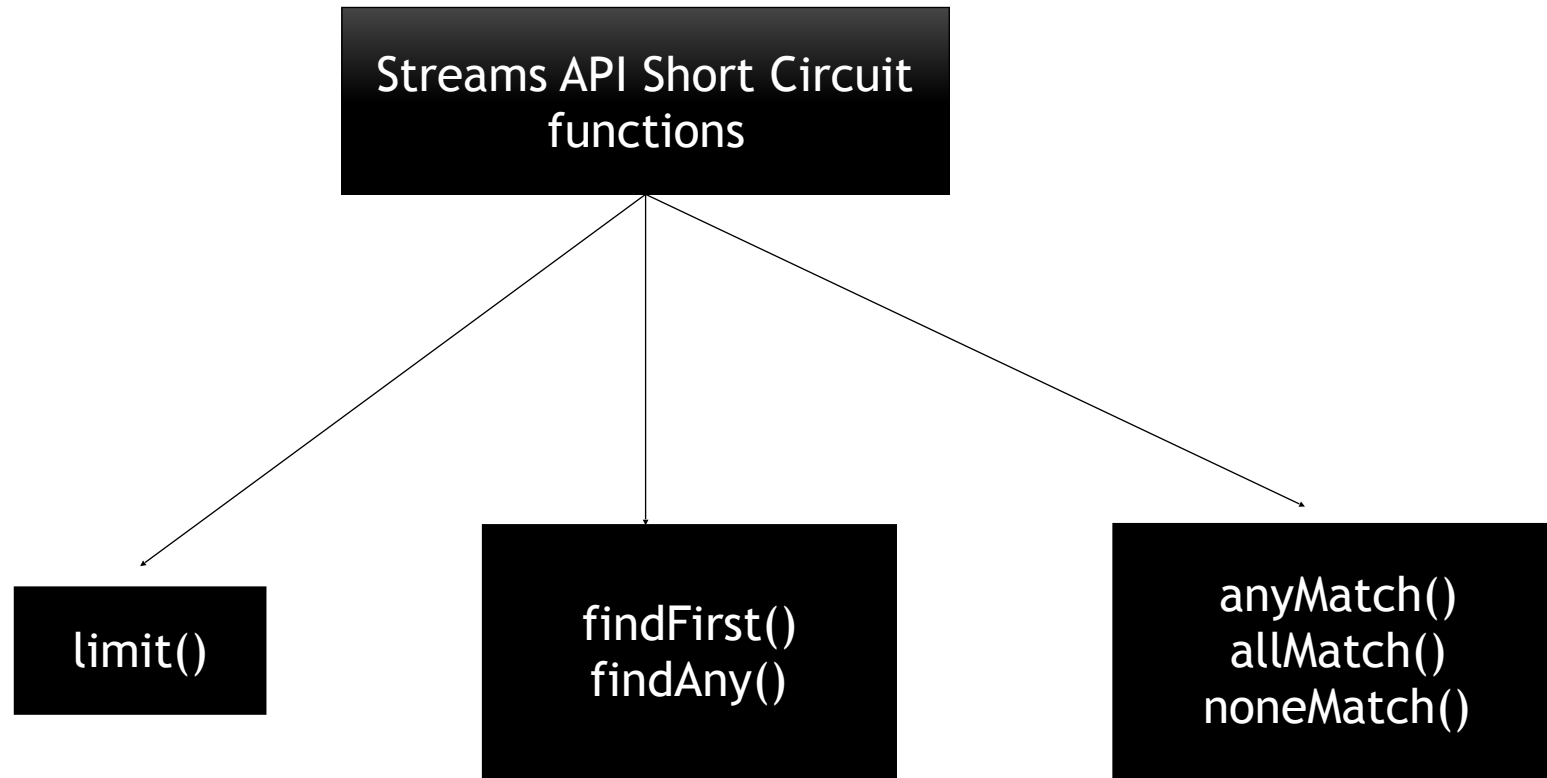
```
if(boolean1 && boolean2){ //AND  
    //body  
}
```

- If the first expression evaluates to false then the second expression wont even execute.

### Example 2:

```
if(boolean1 || boolean2){ //OR  
    //body  
}
```

- If the first expression evaluates to true then the second expression wont even execute.



- All these functions does not have to iterate the whole stream to evaluate the result.

# Streams API : Stateful vs Stateless

- Does Streams have an internal state?
  - Yes
- Does all the Stream functions maintain an internal state ?
  - No

# What is a State in Streams API ?

Converts a `List<Student>` to `List<String>`

```
private static List<String> namesUpperCase(List<Student> names){  
    List<String> namesUpperCase = names.stream()  
        .map(Student::getName)  
        .map(String::toUpperCase)  
        .collect(toList());  
  
    return namesUpperCase;  
}
```

**(Stream State)**

**(Stream Pipeline)**

# Intermediate Operations

- Stateful functions
  - `distinct()`
  - `sorted()`
  - `skip()`
  - `limit()`
- Stateless functions
  - `map()`
  - `filter()`, etc.,



# Stateful functions:

## Convert List<Student> to List<String>

```
public static List<String> printUniqueStudentActivities() {  
    List<String> studentActivities = StudentDataBase.getAllStudents()  
        .stream()  
        .map(Student::getActivities)  
        .flatMap(List::stream)  
        .distinct() // needs the state of the previously processed elements  
        .sorted() // needs the state of the previously processed elements  
        .collect(toList());  
    return studentActivities;  
}
```

# Stateless Functions:

Convert List<Student> to List<String>

```
private static List<String> namesUpperCase(List<Student> names){  
    List<String> namesUpperCase = names.stream() //Stream<Student>  
        .map(Student::getName) //Stream<String> - stateless  
        .map(String::toUpperCase) // Stream<String> -> UpperCase -  
stateless  
        .collect(toList()); // returns List - stateless  
    return namesUpperCase;  
}
```

# Streams API - Factory methods

- Of()
- generate()
- iterate()

# Streams API - of(), iterate() and generate()

- **Of()** -> Creates a stream of certain values passed to this method.

**Example:**

```
Stream<String> stringStream = Stream.of("adam", "dan", "Julie");
```

**iterate(), generate()** -> Used to create infinite Streams.

**Example:**

```
Stream.iterate(1, x->x*2)
```

**Example:**

```
Stream.generate(<Supplier>)
```

# Numeric Streams

Represents the **primitive values** in a Stream.

- IntStream
- LongStream
- DoubleStream

# Numeric Stream Ranges:

## Int Stream:

`IntStream.range(1,50)` -> Returns an IntStream of 49 elements from 1 to 49.

`IntStream.rangeClosed(1,50)` -> Returns an IntStream of 50 elements from 1 to 50.

## Long Stream:

`LongStream.range(1,50)` -> Returns a LongStream of 49 elements from 1 to 49.

`LongStream.rangeClosed(1,50)` -> Returns a LongStream of 50 elements from 1 to 50.

## DoubleStream:

- It does not support the `range ()` and `rangeClosed()`.

# Numeric Stream - Aggregate Functions

- `sum()`
- `max()`
- `min()`
- `average()`

# Numeric Streams : Boxing() and UnBoxing()

## Boxing():

- Converting a primitive type to Wrapper Class type

## Example:

- Converting an int (primitive) to Integer(wrapper).

## UnBoxing():

- Converting a Wrapper Class type to primitive type.

## Example:

- Converting an Integer(wrapper) to int(primitive).



## Numeric Streams - `mapToObj()`, `mapToLong()`, `mapToDouble()`

- **`mapToObj`** -> Convert a each element numeric stream to some Object.
- **`mapToLong`** -> Convert a numeric stream to a Long Stream.
- **`mapToDouble`** -> Convert a numeric stream to a Double Stream.

# Stream Terminal Operations

- Terminal Operations collects the data for you.
- Terminal Operations starts the whole stream pipeline.
- Terminal Operations:
  - `forEach()`
  - `min()`
  - `max()`
  - `reduce()`
  - `collect()` and etc.

# Terminal Operation - collect()

- The **collect()** method takes in an input of type Collector.
- Produces the result as per the input passed to the collect() method.

# Terminal Operations - `joining()`

- `joining()` Collector performs the String concatenation on the elements in the stream.
- `joining()` has three different overloaded versions.

# Terminal Operations - counting()

- **counting()** Collector returns the total number of elements as a result.

# Terminal Operation - mapping()

- **mapping()** collector applies a transformation function first and then collects the data in a collection( could be any type of collection)

# Terminal Operations - `maxBy()` , `minBy()`

- **Comparator** as an input parameter and **Optional** as an output.
- **`maxBy()`**
  - This collector is used in conjunction with comparator. Returns the max element based on the property passed to the comparator.
- **`minBy()`**
  - This collector is used in conjunction with comparator. Returns the smallest element based on the property passed to the comparator.

# Terminal Operations - `summingInt()`, `averagingInt()`

- `summingInt()` - this collector returns the sum as a result.
- `averagingInt()` - this collector returns the average as a result.



# Terminal Operations - groupingBy()

- `groupingBy()` collector is equivalent to the `groupBy()` operation in SQL.
- Used to group the elements based on a property.
- The output of the `groupingBy()` is going to be a `Map<K,V>`
- There are three different versions of `groupingBy()`.
  - `groupingBy(classifier)`
  - `groupingBy(classifier,downstream)`
  - `groupingBy(classifier,supplier,downstream)`

# Terminal Operations - partitioningBy()

- `partitioningBy()` collector is also a kind of `groupingBy()`.
- `partitioningBy()` accepts a predicate as an input.
- Return type of the collector is going to be `Map<K,V>`
  - The key of the return type is going to be a Boolean.
- There are two different versions of `partitioningBy()`
  - `partitioningBy(predicate)`
  - `partitioningBy(predicate,downstream)` // downstream -> could be of any collector

# **Introduction to Parallel Streams**

# What is a Parallel Stream ?

- Splits the source of data in to multiple parts.
- Process them parallelly.
- Combine the result.

# How to Create a Parallel Stream ?

## Sequential Stream:

```
IntStream.rangeClosed(1,1000)  
    .sum();
```

## Parallel Stream:

```
IntStream.rangeClosed(1,1000)  
    .parallel()  
    .sum();
```

# How Parallel Stream works ?

- Parallel Stream uses the **Fork/Join framework** that got introduced in Java 7.

## How many Threads are created ?

- Number of threads created == number of processors available in the machine.

Machine has 8 cores

**Sequential  
Stream**

element 1	element 2	element 3	.....	element n
--------------	--------------	--------------	-------	--------------

**Processor1**

**Parallel  
Stream**

element 1	element 2	element 3	.....	element n
--------------	--------------	--------------	-------	--------------

**Processor  
1**

element 1	element 2	element 3	.....	element n
--------------	--------------	--------------	-------	--------------

**Processor  
2**

element 1	element 2	element 3	.....	element n
--------------	--------------	--------------	-------	--------------

**Processor  
3**

|

|

|

|

element 1	element 2	element 3	.....	element n
--------------	--------------	--------------	-------	--------------

**Processor n**

# Introduction to Optional

- Introduced as part of Java 8 to represent a Non-Null value
- Avoids **Null Pointer Exception** and **Unnecessary Null Checks**.
- Inspired from the new languages such as scala , groovy etc.,



# **Default and Static Methods in Interfaces**

# Interfaces in Java - Prior Java 8:

- Define the contract.
- Only allowed to declare the method. Not allowed to implement a method in Interface.
- Implementation is only allowed in the Implementation class.
- Not easy for an interface to evolve.

# Default Methods - Java 8

- **default** keyword is used to identify a default method in an interface.

## Example from List Interface:

```
default void sort(Comparator<? super E> c) {  
    Object[] a = this.toArray();  
    Arrays.sort(a, (Comparator) c);  
    ListIterator<E> i = this.listIterator();  
    for (Object e : a) {  
        i.next();  
        i.set((E) e);  
    }  
}
```

- Prior to Java 8 we normally use `Collections.sort()` to perform the similar operation.
- Can be overridden in the Implementation class.
- Used to evolve the Interfaces in Java.

# Static Methods - Java 8

- Similar to **default** methods.
- This cannot be overridden by the implementation classes.

# Abstract Classes vs Interfaces in Java 8

- Instance variables are not allowed in Interfaces.
- A class can extend only one class but a class can implement multiple interfaces.

# Does this enable Multiple Inheritance in Java?

- Yes
- This was never possible before Java 8.

# Introduction to New Date/Time Libraries

- `LocalDate`, `LocalTime` and `LocalDateTime` and part of the **`java.time`** package.
- These new classes are created with the inspiration from the **Joda-Time** library.
- All the new time libraries are **Immutable**.
- Supporting classes like **Instant**, **Duration**, **Period** and etc.
- `Date`, `Calendar` prior to Java 8.

**LocalDate:** Used to represent the date.

**LocalTime:** Used to represent the time.

**LocalDateTime:** Used to represent the date and time.



# Period:

- Period is a date-based representation of time in **Days , Months and Years** and is part of the **java.time** package.
- Compatible with **LocalDate**.
- It represents a **Period of Time** not just a specific date and time.

## Example:

Period period1 = Period.ofDays(10); // **represents a Period of 10 days**

Period period2 = Period.ofYears(20); // **represents a Period of 20 years**

# Period : Use-Case

- Mainly used calculate the difference between the two dates.

## Example:

```
LocalDate localDate = LocalDate.of(2018,01,01);  
LocalDate localDate1 = LocalDate.of(2018,01,31);
```

```
Period period = Period.between(localDate,localDate1); // calculates the difference  
between the two dates
```

# Duration

- A time based representation of time in hours , minutes, seconds and nanoseconds.
- Compatible with **LocalTime** and **LocalDateTime**
- It represents a duration of time not just a specific time.

## Example:

`Duration duration1 = Duration.ofHours(3);` // represents the duration of 3 hours

`Duration duration1 = Duration.ofMinutes(3);` // represents the duration of 3 minutes

# Duration : Use-Case

- It can be used to calculate the difference between the time objects such as **LocalTime** and **LocalDateTime**.

## Example:

```
LocalTime localTime = LocalTime.of(7,20);
```

```
LocalTime localTime1 = LocalTime.of(8,20);
```

```
Duration duration = Duration.between(localTime,localTime1);
```

# Instant:

- Represent the time in a machine readable format.

## Example:

```
Instant ins = Instant.now();
```

- Represents the time in seconds from January 01,1970(**EPOCH**) to current time as a huge number.

# Time Zones

- `ZonedDateTime`, `ZoneId`, `ZoneOffset`
- **`ZonedDateTime`** - Represents the date/time with its time zone.

**Example:**

`2018-07-18T08:04:14.541-05:00[America/Chicago]`

**`ZoneOffset`** -> `-05:00`

**`ZoneId`** -> `America/Chicago`

# DateTimeFormatter

- Introduced in **Java 8** and part of the **java.time.format** package.
- Used to parse and format the **LocalDate**, **LocalTime** and **LocalDateTime**.

# Parse and Format

- **parse** - Converting a String to a LocalDate/LocalTime/LocalDateTime.
- **format** - Converting a LocalDate/LocalTime/LocalDateTime to a String.