# Java 8

Presented by
VAISHALI TAPASWI

**FANDS INFONET Pvt.Ltd.**
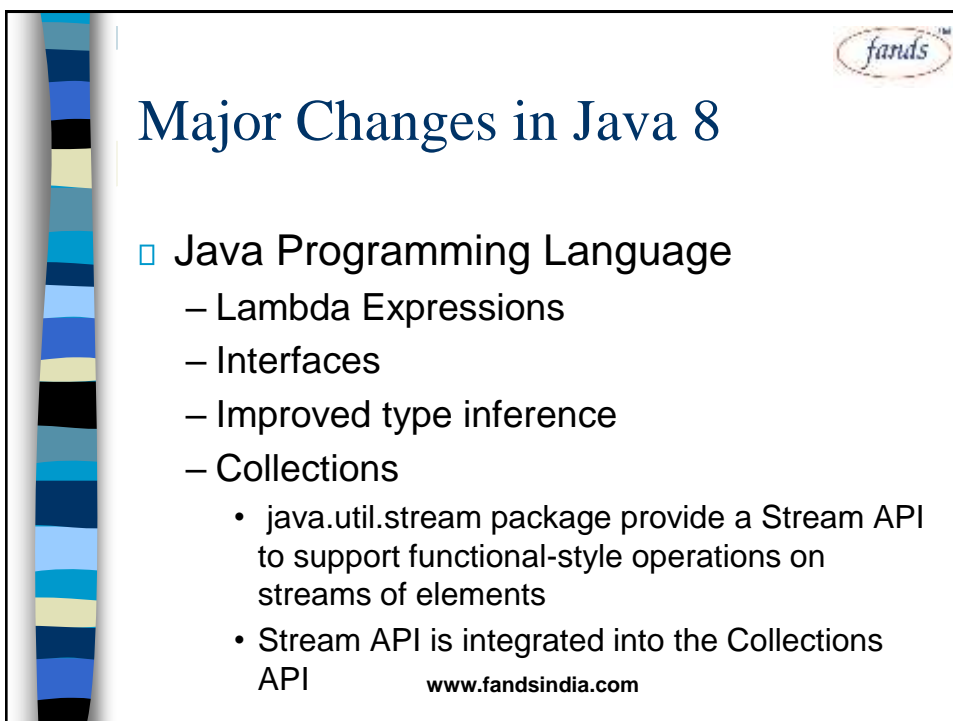**www.fandsindia.com**
**fands@vsnl.com**

# Ground Rules

- **Turn off cell phone. If you cannot, please keep it on silent mode. You can go out and attend your call.**

- **If you have questions or issues, please let me know immediately.**

- **Let us be punctual.**

www.fandsindia.com

*fands*

# Agenda

---

*fands*

# Major Changes in Java 8

- Java Programming Language
  - Lambda Expressions
  - Interfaces
  - Improved type inference
  - Collections
    - java.util.stream package provide a Stream API to support functional-style operations on streams of elements
    - Stream API is integrated into the Collections API **www.fandsindia.com**

# Major Changes in Java 8

- Compact Profiles
  - Contain predefined subsets of the Java SE platform and enable applications that do not require the entire Platform to be deployed and run on small devices.
  - Enable reduced memory footprint for applications that do not require the entire Java platform.
  - The javac compiler has a -profile option, which allows the application to be compiled using one of the supported profiles

**www.fandsindia.com**

# Major Changes in Java 8

- Date-Time Package
  - a new set of packages that provide a comprehensive date-time model.
- IO and NIO
- java.lang and java.util Packages
  - Parallel Array Sorting
  - Standard Encoding and Decoding Base64
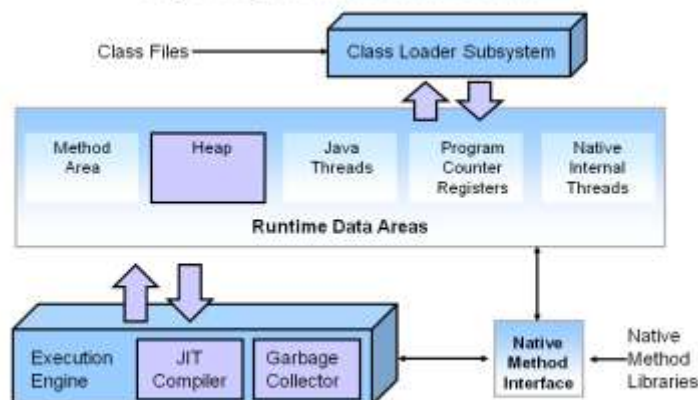  - Unsigned Arithmetic Support

**www.fandsindia.com**

## JRE



☐ Metaspace: A new memory space is born

☐ The JDK 8 HotSpot JVM is now using native memory for the representation of class metadata and is called Metaspace; similar to the Oracle JRockit and IBM JVM's.

**www.fandsindia.com**

# Lambda Expressions

www.fandsindia.com

# Why Lambdas?

- Enables functional programming
  - Object Oriented to Functional Programming
- Readable and Concise Code
- Easier to use APIs and Libraries
- Enables support for parallel processing

www.fandsindia.com

# Functional Vs Object Oriented

- Do we really need functional programming?
  - write better/maintainable code
- Elegant Code
- Passing Behavior in OOP

www.fandsindia.com

# Coding Style in OOP

- Everything is an object
- All code blocks are "associated" with classes and objects
- Function as Values is not possible

# Functional Interfaces and Lambda Expressions

- A functional interface is an interface with a single abstract method.
  - an interface that requires implementation of only a single method in order to satisfy its requirements
  - Before JDK 8 this was obvious – interface had only abstract methods
  - JDK 8 introduces default methods
    - Allows multiple inheritance of behavior for Java
  - JDK 8 also now allows static methods in interfaces

# @FunctionalInterface

- Java 8 has a new annotation, @FunctionalInterface, that can be used for functional interfaces.
- The annotation is not required, but if used,
    - it provides an extra check that everything is consistent (similar to the @Override annotation in earlier versions of Java)
    - the javadoc page for the interface includes a statement that the interface is a functional interface.

**www.fandsindia.com**

# Syntax

- A list of parameters enclosed in parentheses
    - Parameter types can be declared, or can be inferred
    - Empty parentheses indicate - no parameters
    - Parentheses can be omitted for a single parameter whose type can be inferred
- The lambda operator (->)
- A function body, which can be either of the following:
    - a statement block enclosed in braces
    - a single expression (return type is that of the expression) **www.fandsindia.com**

fands

# Example 1:

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> System.out.println(x));
```

- x -> System.out.println(x) is a lambda expression that defines an anonymous function with one parameter named x of type Integer

fands

# Example 2

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> {
    x += 2;
    System.out.println(x);
});
```

- Braces are needed to enclose a multiline body in a lambda expression.

# Example 3:

*fands*

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> {
    int y = x * 2;
    System.out.println(y);
});
```

☐ Just as with ordinary functions, you can define local variables inside the body of a lambda expression

# Example 4:

*fands*

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach((Integer x -> {
    x += 2;
    System.out.println(x);
});
```

☐ You can, if you wish, specify the parameter type.

# Implementation of Java 8 Lambdas

- The Java 8 compiler first converts a lambda expression into a function
- It then calls the generated function
- For example, `x -> System.out.println(x)` could be converted into a generated static function
  ```
  public static void genName(Integer x) {
      System.out.println(x);
  }
  ```
- But what type should be generated for this function? How should it be called? What class should it go in?

# Functional Interfaces

- Design decision: Java 8 lambdas are assigned to functional interfaces.
- A functional interface is a Java interface with exactly one non-default method. E.g.,

  ```
  public interface Consumer<T> {
      void accept(T t);
  }
  ```

- The package `java.util.function` defines many new useful functional interfaces.

# Lambda as a Local Variable

```java
public interface Consumer<T> {
  void accept(T t);
}
void forEach(Consumer<Integer> action {
  for (Integer i:items) {
    action.accept(t);
  }
}
List<Integer> intSeq = Arrrays.asList(1,2,3);

Consumer<Integer> cnsmr = x ->
System.out.println(x);
intSeq.forEach(cnsmr);
```

# Properties of the Generated Method

- The method generated from a Java 8 lambda expression has the same signature as the method in the functional interface
- The type is the same as that of the functional interface to which the lambda expression is assigned
- The lambda expression becomes the body of the method in the interface

# Variable Capture

- Lambdas can interact with variables defined outside the body of the lambda
- Using these variables is called variable capture

# Local Variable Capture

```
public class LVCExample {
  public static void main(String[] args) {
    List<Integer> intSeq = Arrays.asList(1,2,3);

    int var = 10;
    intSeq.forEach(x -> System.out.println(x +
var));
  }
}
```

- Note: local variables used inside the body of a lambda must be final or effectively final

# Static Variable Capture

```
public class SVCExample {
  private static int var = 10;
  public static void main(String[] args) {
    List<Integer> intSeq = Arrays.asList(1,2,3);
    intSeq.forEach(x -> System.out.println(x +
var));
  }
}
```

# Method References

- Method references can be used to pass an existing function in places where a lambda is expected
- The signature of the referenced method needs to match the signature of the functional interface method

# Summary of Method References

fands

| Method Reference Type | Syntax | Example |
|---|---|---|
| static | ClassName::StaticMethodName | String::valueOf |
| constructor | ClassName::new | ArrayList::new |
| specific object instance | objectReference::MethodName | x::toString |
| arbitrary object of a given type | ClassName::InstanceMethodName | Object::toString |

---

# Conciseness with Method References

fands

We can rewrite the statement

```
intSeq.forEach(x -> System.out.println(x));
```

more concisely using a method reference

```
intSeq.forEach(System.out::println);
```

# Streams

# Functional Interfaces

- Interfaces with only one abstract method.
- With only one abstract method, these interfaces can be easily represented with lambda expressions
- Example
  ```
  @FunctionalInterface
  public interface SimpleFuncInterface {
      public void doWork();
  }
  ```

# Method References

- Even more brief and clearly expressive way to implement functional interfaces
- Format: <Class or Instance>::<Method>
- Example (Functional Interface)
  ```
  public interface IntPredicates {
       boolean isOdd(Integer n) { return n % 2 != 0; }
  }
  ```
- Example (Call with Lambda Expression)
  ```
  List<Integer> numbers = asList(1,2,3,4,5,6,7,8,9);
  List<Integer> odds = filter(n -> IntPredicates.isOdd(n), numbers);
  ```
- Example (Call with Method Reference)
  ```
  List<Integer> numbers = asList(1,2,3,4,5,6,7,8,9);
  List<Integer> odds = filter(IntPredicates::isOdd,  numbers);
  ```

# Characteristics of Streams

- Streams are not related to InputStreams, OutputStreams, etc.
- Streams are NOT data structures but are wrappers around Collection that carry values from a source through a pipeline of operations.
- Streams are more powerful, faster and more memory efficient than Lists
- Streams are designed for lambdas
- Streams can easily create output as arrays or lists
- Streams employ lazy evaluation
- Streams are parallelizable
- Streams can be "on-the-fly"

# Creating Streams

- From individual values
  - Stream.of(val1, val2, …)
- From array
  - Stream.of(someArray)
  - Arrays.stream(someArray)
- From List (and other Collections)
  - someList.stream()
  - someOtherCollection.stream()

# Common Functional Interfaces Used

- Predicate<T>
  - Represents a predicate (boolean-valued function) of one argument
- Supplier<T>
  - Represents a supplier of results
  - Functional method is T get()
- Function<T,R>
  - Represents a function that accepts one argument and produces a result
- Consumer<T>
  - Represents an operation that accepts a single input and returns no result
  - Functional method is void accept(T t)

# Common Functional Interfaces Used

- UnaryOperator<T>
  - Represents an operation on a single operands that produces a result of the same type as its operand
  - Functional method is R Function.apply(T t)
- BiFunction<T,U,R>
  - Represents an operation that accepts two arguments and produces a result
  - Functional method is R apply(T t, U u)
- BinaryOperator<T>
  - Extends BiFunction<T, U, R>
  - Represents an operation upon two operands of the same type, producing a result of the same type as the operands
- Comparator<T>
  - Compares its two arguments for order.
  - Functional method is int compareTo(T o1, T o2)

# Anatomy of the Stream Pipeline

- A Stream is processed through a pipeline of operations
- A Stream starts with a source data structure
- Intermediate methods are performed on the Stream elements. These methods produce Streams and are not processed until the terminal method is called.
- The Stream is considered consumed when a terminal operation is invoked. No other operation can be performed on the Stream elements afterwards
- A Stream pipeline contains some short-circuit methods (which could be intermediate or terminal methods) that cause the earlier intermediate methods to be processed only until the short-circuit method can be evaluated.

*fands*

# Anatomy of the Stream Pipeline

- Intermediate Methods
  - Map, Filter, distinct, sorted, peek, limit, parallel
- Terminal Methods
  - forEach, toArray, reduce, collect, min, max, count, anyMatch, allMatch, noneMatch, findFirst, findAny, iterator
- Short-circuit Methods
  - anyMatch, allMatch, noneMatch, findFirst, findAny,limit

*fands*

# Optional<T> Class

- A container which may or may not contain a non-null value
- Common methods
  - isPresent() – returns true if value is present
  - Get() – returns value if present
  - orElse(T other) – returns value if present, or other
  - ifPresent(Consumer) – runs the lambda if value is present

# Common Stream API Methods Used

- ## void forEach(Consumer)
  - Easy way to loop over Stream elements
  - You supply a lambda for forEach and that lambda is called on each element of the Stream
  - Related peek method does the exact same thing, but returns the original Stream

- ## Advantages of forEach
  - Designed for lambdas to be marginally more succinct
  - Lambdas are reusable
  - Can be made parallel with minimal effort

# Common Stream API Methods Used

- Stream<T> map(Function)
  - Produces a new Stream that is the result of applying a Function to each element of original Stream
- Stream<T> filter(Predicate)
  - Produces a new Stream that contains only the elements of the original Stream that pass a given test
- Optional<T> findFirst()
  - Returns an Optional for the first entry in the Stream
- Object[] toArray(Supplier)
  - Reads the Stream of elements into a an array
- List<T> collect(Collectors.toList())
  - Reads the Stream of elements into a List or any other collection

# Common Stream API Methods Used

- List<T> collect(Collectors.toList())
  - partitioningBy
    - You provide a Predicate. It builds a Map where true maps to a List of entries that passed the Predicate, and false maps to a List that failed the Predicate.
    - Example
      Map<Boolean,List<Employee>> richTable =
      googlers().collect
      (partitioningBy(e -> e.getSalary() > 1000000));
  - groupingBy
    - You provide a Function. It builds a Map where each output value of the Function maps to a List of entries that gave that value.
    - Example
      Map<Department,List<Employee>> deptTable =
      employeeStream().collect(groupingBy(Employee::getDepartment
      ));

# Common Stream API Methods Used

- T reduce(T identity, BinaryOperator)
  - You start with a seed (identity) value, then combine this value with the first Entry in the Stream, combine the second entry of the Stream, etc.
- Stream<T> limit(long maxSize)
  - Limit(n) returns a stream of the first n elements
- Stream<T> skip(long n)
  - Skip(n) returns a stream starting with element n
- Stream<T> sorted(Comparator)
  - Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator
- Optional<T> min/max(Comparator)
  - Returns the minimum/maximum element in this Stream according to the Comparator
- Stream<T> distinct()
  - Returns a stream consisting of the distinct elements of this stream
- long count()
  - Returns the count of elements in the Stream

---

# Common Stream API Methods Used

- Boolean anyMatch(Predicate), allMatch(Predicate), noneMatch(Predicate)
  - Returns true if Stream passes, false otherwise
  - Lazy Evaluation
    - anyMatch processes elements in the Stream one element at a time until it finds a match according to the Predicate and returns true if it found a match
    - allMatch processes elements in the Stream one element at a time until it fails a match according to the Predicate and returns false if an element failed the Predicate
    - noneMatch processes elements in the Stream one element at a time until it finds a match according to the Predicate and returns false if an element matches the Predicate

# Parallel Streams

# (On The Fly) Streams

- Stream<T> generate(Supplier)
  - The method lets you specify a Supplier
  - This Supplier is invoked each time the system needs a Stream element
  - Example
    ```
    List<Employee> emps =
    Stream.generate(() -> randomEmployee())
    .limit(n)
    .collect(toList());
    ```

- Stream<T> iterate(T seed, UnaryOperator<T> f)
  - The method lets you specify a seed and a UnaryOperator.
  - The seed becomes the first element of the Stream, f(seed) becomes the second element of the Stream, f(second) becomes the third element, etc.
  - Example
    ```
    List<Integer> powersOfTwo =
    Stream.iterate(1, n -> n * 2)
    .limit(n)
    .collect(toList());
    ```
- The values are not calculated until they are needed
- To avoid unterminated processing, you must eventually use a size-limiting method
- This is less of an actual Unbounded Stream and more of an "On The Fly" Stream
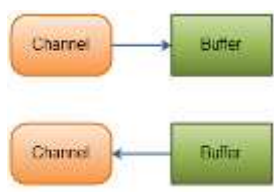
---

# NIO

A Fast AND Steady Approach

# NIO

fands

- Channels and Buffers
- Non-blocking IO
- Selectors

---

# From IO to NIO

fands



- Channels and Buffers
  - IO
    - work with byte streams and character streams
  - NIO
    - work with channels and buffers. Data is always read from a channel into a buffer, or written from a buffer to a channel.
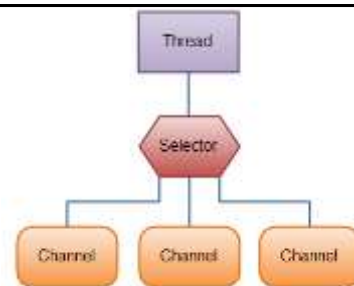
fands

# From IO to NIO

☐ Non-blocking IO
– Java NIO enables you to do non-blocking IO. For instance, a thread can ask a channel to read data into a buffer. While the channel reads data into the buffer, the thread can do something else. Once data is read into the buffer, the thread can then continue processing it. The same is true for writing data to channels.

**www.fandsindia.com**

# From IO to NIO

Thread

Selector

Channel   Channel   Channel

☐ Java NIO: Selectors
– Java NIO contains the concept of "selectors". A selector is an object that can monitor multiple channels for events (like: connection opened, data arrived etc.). Thus, a single thread can monitor multiple channels for data.

**www.fandsindia.com**

# Buffer Capacity, Position and Limit



Buffer capacity, position and limit in write and read mode.
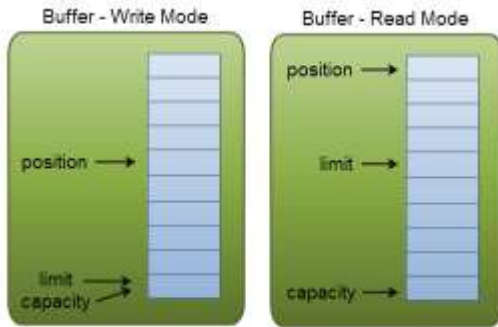
A buffer is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.

**www.fandsindia.com**

# QUESTION / ANSWERS



**www.fandsindia.com**

# THANKING YOU !

**www.fandsindia.com**