

JAVA 8

Agenda

- ❖ New Features in Java language
 - Lambda Expression
 - Functional Interface
 - Interface's default and Static Methods
 - Method References
- ❖ New Features in Java libraries
 - Stream API
 - Date/Time API

What is Lambda Expression?

Lambda Expression

- ❖ Unnamed block of code (or an unnamed function) with a list of formal parameters and a body.
 - ✓ Concise
 - ✓ Anonymous
 - ✓ Function
 - ✓ Passed around

- Why should we care about Lambda Expression?

Lambda Expression

Example 1:

```
Comparator<Person> byAge = new Comparator<Person>() {  
    public int compare(Person p1, Person p2) {  
        return p1.getAge().compareTo(p2.getAge());  
    }  
};
```

Example 2:

```
JButton testButton = new JButton("Test Button");  
testButton.addActionListener(new ActionListener() {  
    @Override public void actionPerformed(ActionEvent ae) {  
        System.out.println("Hello Anonymous inner class");  
    }  
});
```

Lambda Expression

Example 1: with lambda

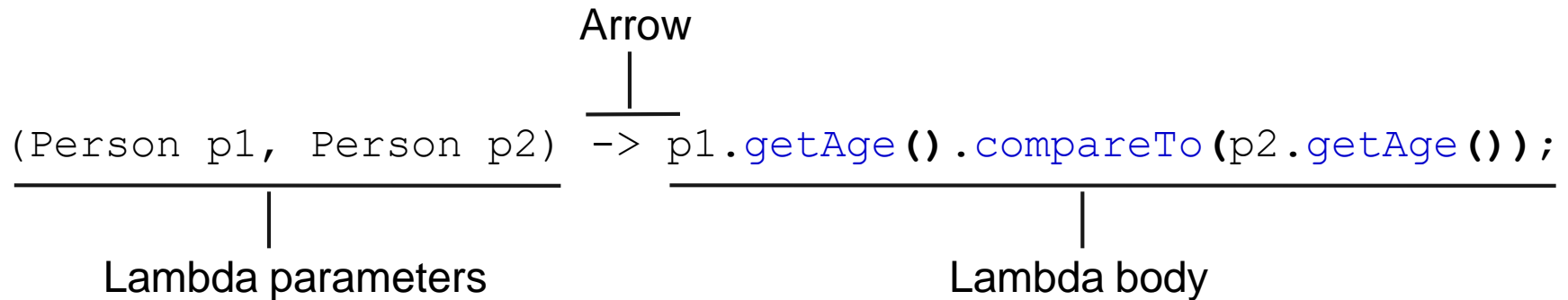
```
Comparator<Person> byAge =  
(Person p1, Person p2) -> p1.getAge().compareTo(p2.getAge());
```

Example 2: with lambda

```
testButton.addActionListener(e -> System.out.println("Hello  
Lambda"));
```

Lambda Syntax

Lambda Expression



The basic syntax of a lambda is either

(parameters) -> expression

or

(parameters) -> { statements; }

Lambda Expression

❖ Lambda does not have:

- ✓ Name

- ✓ Return type

- ✓ Throws clause

- ✓ Type parameters

Lambda Expression

Examples:

1. `(String s) -> s.length()`
2. `(Person p) -> p.getAge() > 20`
3. `() -> 92`
4. `(int x, int y) -> x + y`
5. `(int x, int y) -> {
 System.out.println("Result:");
 System.out.println(x + y);
}`

Where should we use Lambda?

Lambda Expression

Iterator:

```
List<String> features = Arrays.asList("Lambdas", "Default  
Method", "Stream API", "Date and Time API");
```

//Prior to Java 8 :

```
for (String feature : features) {  
    System.out.println(feature);  
}
```

//In Java 8:

```
Consumer<String> con = n -> System.out.println(n)  
features.forEach(con);
```

What is functional interface?

- New term of Java 8
- A functional interface is an interface with only one *abstract* method.

```
public interface Runnable {  
    run();  
};
```

```
public interface Comparator<T> {  
    int compare(T t1, T t2);  
};
```

- Methods from the Object class don't count.

```
public interface MyFunctionalInterface {  
  
    someMethod();  
  
    /**  
     * Some more documentation  
     */  
    equals(Object o);  
};
```


Annotation in Functional Interface

- A functional interface can be annotated.
- It's optional.

```
package com.tctruc.java8;

@FunctionalInterface
public interface MyFunctionalInterface {
    void display(String yourName);
}
```

- Show compile error when define more than one abstract method.

```
@FunctionalInterface  
public interface MyFunctionalInterface {
```


Invalid '@FunctionalInterface' annotation; MyFunctionalInterface is not a functional interface

```
void display(String yourName);  
void anotherDisplay();  
}
```

Interface Default and Static Methods

Interface Default and Static Methods

- Extends interface declarations with two new concepts:
 - Default methods
 - Static methods
- Advantages:
 - No longer need to provide your own companion utility classes. Instead, you can place static methods in the appropriate interfaces

java.util.Collections }
java.util.Collection }  static <T> Collection<T>
synchronizedCollection(Collection<T>)

- Adding methods to an interface without breaking the existing implementation

Interface Default and Static Methods

- Syntax

```
[modifier] default | static returnType nameOfMethod (Parameter List) {  
    // method body  
}
```

```
package java8.interfacemethods;  
  
public interface MyData {  
      
    default void print(String str) {  
        if (!isNull(str))  
            System.out.println("MyData Print::" + str);  
    }  
      
    static boolean isNull(String str) {  
        System.out.println("Interface Null Check");  
          
        return str == null ? true : "".equals(str) ? true : false;  
    }  
}
```

Default methods

- Classes implement interface that contains a default method
 - ❑ Not override the default method and will inherit the default method
 - ❑ Override the default method similar to other methods we override in subclass
 - ❑ Redecclare default method as abstract, which force subclass to override it

```
package java8.interfacemethods;  
  
public interface Defaulable {  
  
    public default String notRequired(){  
        return "Default implementation";  
    }  
}
```

```
package java8.interfacemethods;  
  
public abstract class AbstractOveridbleImpl implements Defaulable {  
  
    public abstract String notRequired();  
}
```

```
public class ForceOveridableImpl extends AbstractOveridbleImpl {  
  
    @Override  
    public String notRequired() {  
        return "Force Overriden implement";  
    }  
}
```

Default methods

- Solve the conflict when the same method declare in interface or class
 - Method of Superclasses, if a superclass provides a concrete method.
 - If a superinterface provides a default method, and another interface supplies a method with the same name and parameter types (default or not), then you must overriding that method.

Static methods

- Similar to default methods except that we can't override them in the implementation classes

```
package java8.interfacemethods;

public interface MyData {

    default void print(String str) {
        if (!isNull(str))
            System.out.println("MyData Print::" + str);
    }

    static boolean isNull(String str) {
        System.out.println("Interface Null Check");

        return str == null ? true : "".equals(str) ? true : false;
    }
}
```

```
package java8.interfacemethods;

public class MyDataImpl implements MyData{

    public boolean isNull(String str) {
        System.out.println("Impl Null Check");
        return str == null ? true : false;
    }

    public static void main(String args[]){
        MyDataImpl obj = new MyDataImpl();
        obj.print("");
        obj.isNull("abc");
    }
}
```

Method References

- Method reference is an important feature related to lambda expressions. In order to that a method reference requires a target type context that consists of a compatible functional interface

```
class PersonAgeComparator implements Comparator<Person> {  
    public int compare(Person a, Person b) {  
        return a.getBirthDay().compareTo(b.getBirthDay());  
    }  
}
```

```
Arrays.sort(rosterAsArray, new PersonAgeComparator());
```



```
Arrays.sort(rosterAsArray, Person::compareByAge);
```

Method References

- There are four kinds of method references:
 - Reference to a static method
 - Reference to an instance method of a particular object
 - Reference to an instance method of an arbitrary object of a particular type
 - Reference to a constructor

Method References

- Reference to a static method

The syntax: **ContainingClass::staticMethodName**

```
package java8.methodreferences;

public class MetRefThread {

    public static void runBody() {
        for (int i = 0; i < 10; i++) {
            System.out.println("square of i is " + (i * i));
        }
    }

    public static void main(String[] args) {
        new Thread(MetRefThread::runBody).start();
    }
}
```

Method References

- Reference to an instance method of a particular object

The syntax: **containingObject::instanceMethodName**

```
public class ObjectMetRef {  
    void startsWith(String s, String b) {  
        System.out.println(String.valueOf(s.charAt(0)));  
        System.out.println(String.valueOf(b.charAt(0)));  
    }  
    public static void main(String[] args) {  
        ObjectMetRef something = new ObjectMetRef();  
        Converter<String, String> converter = something::startsWith;  
        converter.convert("Java", "8");    // "J" and "8"  
    }  
}
```

Method References

- Reference to an instance method of an arbitrary object of a particular type

The syntax: **ContainingType::methodName**

```
package java8.methodreferences;

import java.util.Arrays;

public class ArraySort {
    public static void main(String[] args) {
        String[] strArray = { "abe", "adb", "deb", "abc", "ghi", "acd", "acg", "acb" };
        Arrays.sort(strArray, String::compareToIgnoreCase);
        for (String str : strArray) {
            System.out.print(str + " ");
        }
    }
}
```

Method References

- Reference to a constructor

The syntax: **ClassName::new**

```
public class ConstructorReference {  
    private String content;  
  
    public ConstructorReference() {  
        this.content = "created by constructor reference";  
    }  
  
    public String getContent() {  
        return content;  
    }  
  
    public static void main(String... args) {  
        ConstructorReferenceArg<ConstructorReference>  
            constructorSam = ConstructorReference::new;  
        System.out.println("\n\n\tcontent = "  
            + constructorSam.whatEverMethodName().getContent());  
    }  
}
```

```
public interface ConstructorReferenceArg<T> {  
    T whatEverMethodName();  
}
```

Method References

- Method references as Lambdas Expressions

Syntax	Example	As Lambda
ClassName::new	String::new	() -> new String()
Class::staticMethodName	String::valueOf	(s) -> String.valueOf(s)
object::instanceMethodName	x::toString	() -> "hello".toString()
Class::instanceMethodName	String::toString	(s) -> s.toString()

Stream

Before we look into Java Stream API Examples, let's see why it was required. Suppose we want to iterate over a list of integers and find out sum of all the integers greater than 10.

Prior to Java 8, the approach to do it would be:

```
private static int sumIterator(List<Integer> list) {  
    Iterator<Integer> it = list.iterator();  
    int sum = 0;  
    while (it.hasNext()) {  
        int num = it.next();  
        if (num > 10) {  
            sum += num;  
        }  
    }  
    return sum;  
}
```

Stream

There are three major problems with the previous approach:

1. We just want to know the sum of integers but we would also have to provide how the iteration will take place, this is also called external iteration because client program is handling the algorithm to iterate over the list.
2. The program is sequential in nature, there is no way we can do this in parallel easily.
3. There is a lot of code to do even a simple task.

Stream

1. To overcome all the above shortcomings, Java 8 Stream API was introduced. We can use Java Stream API to implement internal iteration, that is better because java framework is in control of the iteration.
2. Internal iteration provides several features such as sequential and parallel execution, filtering based on the given criteria, mapping etc.
3. Most of the Java 8 Stream API method arguments are functional interfaces, so lambda expressions work very well with them. Let's see how can we write above logic in a single line statement using Java Streams.

```
private static int sumStream(List<Integer> list) {  
    return list.stream().filter(i -> i > 10).mapToInt(i -> i).sum();  
}
```

What exactly is Stream ?

“A sequence of elements from a source that supports data processing operations”

- **Sequence of elements**— Collections are data structures, they're mostly about storing and accessing elements with specific time/space complexities. But streams are about expressing computations such as filter, sorted, and map. Collections are about data; streams are about computations.
- **Source**— Streams consume from a data-providing source such as collections, arrays, or I/O resources. Note that generating a stream from an ordered collection preserves the ordering. The elements of a stream coming from a list will have the same order as the list.
- **Data processing operations**— Streams support database-like operations and common operations from functional programming languages to manipulate data, such as filter, map, reduce, find, match, sort, and so on. Stream operations can be executed either sequentially or in parallel

Collections and java Stream

- Java Stream is a data structure that is computed on-demand.
- Java Stream doesn't store data,.
- Can easily be outputted as arrays or lists
- Java Stream operations use functional interfaces.
- Java 8 Stream internal iteration principle helps in achieving lazy-seeking in some of the stream operations.
- Java Streams are consumable.
- Java 8 Stream support sequential as well as parallel processing,

Stream-Build Streams

- **Build streams from collections**

```
List<Dish> dishes = new ArrayList<Dish>();
```

```
....(add some Dishes)....
```

```
Stream<Dish> stream = dishes.stream();
```

Stream-Build Streams

- **Build streams from arrays**

```
Integer[] integers =
```

```
{1,2, 3, 4, 5, 6, 7, 8, 9};
```

```
Stream<Integer> stream =Stream.of(integers);
```

Function Package In Java 8

- Function package contains Functional interfaces.
- Some of the main functional interfaces are :-
 - Function interface
 - Predicate interface
 - Consumer interface
 - Supplier interface

Function And BiFunction interface

- Function represents a function that takes one type of argument and returns another type of argument. `Function<T, R>` is the generic form where T is the type of the input to the function and R is the type of the result of the function.
- For handling primitive types, there are specific Function interfaces – `ToIntFunction`, `ToLongFunction`, `ToDoubleFunction`, `ToLongBiFunction`, `ToDoubleBiFunction`, `LongToIntFunction`, `LongToDoubleFunction`, `IntToLongFunction`, `IntToDoubleFunction` etc.
- We can apply our business logic with that value and return the result. Function has function method as *`apply(T t)`* which accepts one argument.

```
import java.util.function.Function;
public class FunctionDemo {
    public static void main(String[] args) {
        Function<Integer, String> function = (num1) -> "Result is: " + num1;
        System.out.println(function.apply(20));
    }
}
```

Function And BiFunction interface

- Some of the Stream methods where Function or it's primitive specialization is used are:
 - `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
 - `IntStream mapToInt(ToIntFunction<? super T> mapper)` – similarly double returning primitive specific stream.
 - `IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper)` similarly for long and double
 - `<A> A[] toArray(IntFunction<A[]> generator)`
 - `<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)`

Predicate And BiPredicate Interface

- It represents a predicate against which elements of the stream are tested. This is used to filter elements from the java stream.
- Just like `Function`, there are primitive specific interfaces for int, long and double.
- Predicate functional method is *test(Object)*. Find the simple example for how to use Predicate.

```
public class BiPredicateDemo {  
    public static void main(String[] args){  
        BiPredicate<Integer, String> condition = (i,s)-> i>20 && s.startsWith("R");  
        System.out.println(condition.test(10,"Ram"));  
        System.out.println(condition.test(30,"Shyam"));  
        System.out.println(condition.test(30,"Ram"));  
    }  
}
```

Predicate And BiPredicate Interface

- Some of the Stream methods where Predicate or BiPredicate specializations are used are:
 - `Stream<T> filter(Predicate<? super T> predicate)`
 - `boolean anyMatch(Predicate<? super T> predicate)`
 - `boolean allMatch(Predicate<? super T> predicate)`
 - `boolean noneMatch(Predicate<? super T> predicate)`

Consumer And BiConsumer Interface

- It represents an operation that accepts a single input argument and returns no result.
- It can be used to perform some action on all the elements of the java stream.
- Consumer functional method is *accept(Object, Object)*. This methods performs the operation defined by BiConsumer.

```
public class ConsumerExample {  
  
    static void print(String name){  
        System.out.println("Hello "+name);  
    }  
  
}
```

```
public static void main(String[] args) {  
    // Referring method to String type Consumer interface  
    Consumer<String>consumer1=ConsumerExample::print;  
    consumer1.accept("John");  
}
```

Consumer And BiConsumer Interface

- Some of the Java 8 Stream methods where `Consumer`, `BiConsumer` or it's primitive specialization interfaces are used are
 - `Stream<T> peek(Consumer<? super T> action)`
 - `void forEach(Consumer<? super T> action)`
 - `void forEachOrdered(Consumer<? super T> action)`

Stream-java.util.optional

- Java Optional is a container object which may or may not contain a non-null value. If a value is present, `isPresent()` will return true and `get()` will return the value. Stream terminal operations return Optional object. Some of these methods are:
 - `Optional<T> reduce(BinaryOperator<T> accumulator)`
 - `Optional<T> min(Comparator<? super T> comparator)`
 - `Optional<T> max(Comparator<? super T> comparator)`
 - `Optional<T> findFirst()`
 - `Optional<T> findAny()`

Stream-operations

- **Intermediate operations (return Stream)**

Java Stream API operations that returns a new Stream are called intermediate operations. Most of the times, these operations are lazy in nature, so they start producing new stream elements and send it to the next operation. Intermediate operations are never the final result producing operations. Commonly used intermediate operations are filter and map.

Stream-operations

- **Terminal operations**

Java 8 Stream API operations that returns a result or produce a side effect. Once the terminal method is called on a stream, it consumes the stream and after that we can't use stream. Terminal operations are eager in nature i.e they process all the elements in the stream before returning the result. Commonly used terminal methods are `forEach`, `toArray`, `min`, `max`, `findFirst`, `anyMatch`, `allMatch` etc. You can identify terminal methods from the return type, they will never return a Stream.

Stream-operations

- **Short circuiting operations**

An intermediate operation is called short circuiting, if it may produce finite stream for an infinite stream. For example `limit()` and `skip()` are two short circuiting intermediate operations.

A terminal operation is called short circuiting, if it may terminate in finite time for infinite stream. For example `anyMatch`, `allMatch`, `noneMatch`, `findFirst` and `findAny` are short circuiting terminal operations.

Date AND Time API

Why do we need new Java Date Time API?

1. Java Date Time classes are not defined consistently, we have Date Class in both `java.util` as well as `java.sql` packages. Again formatting and parsing classes are defined in `java.text` package.
2. `java.util.Date` contains both date and time, whereas `java.sql.Date` contains only date. Having this in `java.sql` package doesn't make sense. Also both the classes have same name, that is a very bad design itself.
3. There are no clearly defined classes for time, timestamp, formatting and parsing. We have `java.text.DateFormat` abstract class for parsing and formatting need. Usually `SimpleDateFormat` class is used for parsing and formatting.
4. All the Date classes are mutable, so they are not thread safe. It's one of the biggest problem with Java Date and Calendar classes.
5. Date class doesn't provide internationalization, there is no timezone support. So `java.util.Calendar` and `java.util.TimeZone` classes were introduced, but they also have all the problems listed above.

Date AND Time API

Java 8 Date Time API is [JSR-310](#) implementation. It is designed to overcome all the flaws in the legacy date time implementations. Some of the design principles of new Date Time API are:

- 1. Immutability:** All the classes in the new Date Time API are immutable and good for multithreaded environments.
- 2. Separation of Concerns:** The new API separates clearly between human readable date time and machine time (unix timestamp). It defines separate classes for Date, Time, DateTime, Timestamp, Timezone etc.
- 3. Clarity:** The methods are clearly defined and perform the same action in all the classes. For example, to get the current instance we have `now()` method. There are `format()` and `parse()` methods defined in all these classes rather than having a separate class for them.
- 4. Utility operations:** All the new Date Time API classes comes with methods to perform common tasks, such as plus, minus, format, parsing, getting separate part in date/time etc.
- 5. Extendable:** The new Date Time API works on ISO-8601 calendar system but we can use it with other non ISO calendars as well.

Date and Time Packages

Java 8 Date Time API consists of following packages.

1. `java.time` Package: This is the base package of new Java Date Time API. such as `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, `Duration` etc. All of these classes are immutable and thread safe.
2. `java.time.chrono` Package: This package defines generic APIs for non ISO calendar systems. We can extend `AbstractChronology` class to create our own calendar system.
3. `java.time.temporal` Package: This package contains temporal objects and we can use it for find out specific date or time related to date/time object. For example, we can use these to find out the first or last day of the month. You can identify these methods easily because they always have format "withXXX".
4. `java.time.zone` Package: This package contains classes for supporting different time zones and their rules.