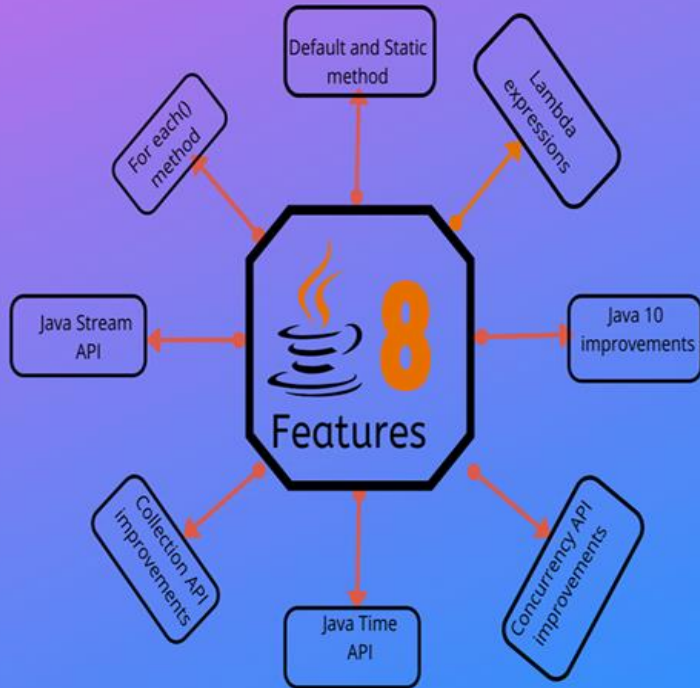
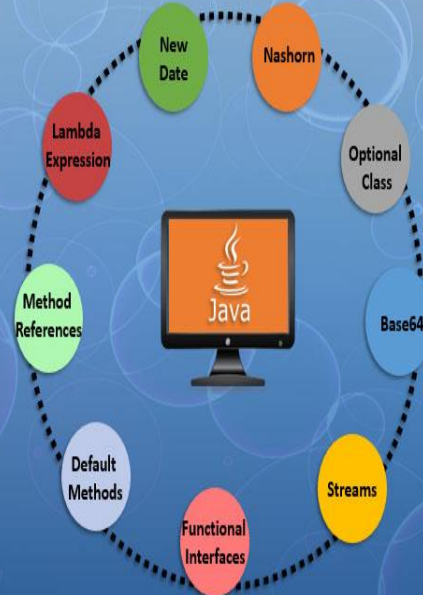


Java 8 new features

Rehab Academy For Programming



Java 8 Features



Agenda

- Session1:
 - Default and static methods in Interfaces.
 - ForEach() method in Iterable interface.
 - Functional Interfaces and Lambda Expressions.

Session2:

Lambda Expressions .

Type interfering.

Session 3:

- Java Stream API for Bulk Data Operations on Collections.
- Optional class.

Agenda

Session 4:

- Java Time API.
 - Java Time API-examples.
 - Java Time API-utilities.
 - Java Time API-parsing and formatting.

•Session 5:

- Collection API improvements.
- Java IO improvements.
- Miscellaneous Core API improvements.
- Concurrency API improvements.

Interface Default and Static Methods

- Prior to java 8, interface in java can only have abstract methods(interfaces can't have method body). All the methods of interfaces are public & abstract by default.
- Java 8 allows the interfaces to have **default** and **static** methods(method with implementation).
- The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

Interface Default and Static Methods

- **Why default method?**
 - For example, if several classes such as A, B, C and D implements an interface XYZInterface then if we add a new method to the XYZInterface, we have to change the code in all the classes(A, B, C and D) that implements this interface. imagine if there are hundreds of classes implementing an interface then it would be almost impossible to change the code in all those classes. This is why in java 8, we have a new concept “default methods”.
 - These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.
- **Static methods** in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces.
- **Example** : forEach method defined in Iterable interface as a default method.

ForEach() method in Iterable interface

- Whenever we need to traverse through a Collection, we need to create an Iterator whose whole purpose is to iterate over and then we have business logic in a loop for each of the elements in the Collection. We might get ConcurrentModificationException if iterator is not used properly.
- Java 8 has introduced *forEach* method in `java.lang.Iterable` interface so that while writing code we focus on business logic only.
- *ForEach* method helps in having the logic for iteration and business logic at separate place resulting in higher separation of concern and cleaner code.

ForEach() method in Iterable interface

- *It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.so we can iterate on any type of collections like maps,lists,...and so on*
- *This method takes a single parameter which is a functional interface(Consumer). So, you can pass **lambda expression** as an argument.*

ForEach() examples-iterator:

Java 8 forEach() example 1

```
import java.util.ArrayList;
import java.util.List;
public class ForEachExample {
    public static void main(String[] args) {
        List<String> gamesList = new ArrayList<String>();
        gamesList.add("Football");
        gamesList.add("Cricket");
        gamesList.add("Chess");
        gamesList.add("Hockey");
        System.out.println("-----Iterating by passing lambda expression-----");
        gamesList.forEach/games -> System.out.println/games));

    }
}
```

ForEach() examples-stream:

2. Java 8 stream if logic – lambda conditional filter

If we intend to apply **only 'if' logic** then we can pass the condition directly to the `filter()` function.

In given example, we are checking *if* a number is even then print a message.

Main.java

```
ArrayList<Integer> numberList = new ArrayList<>(Arrays.asList(1,2,3,4,5,6));

numberList.stream()
    .filter(i -> i % 2 == 0)
    .forEach(System.out::println);
```

Functional Interfaces and Lambda Expressions

Functional interfaces:

- Functional interfaces are new concept introduced in Java 8. An interface with exactly one abstract method becomes Functional Interface.
- We don't need to use `@FunctionalInterface` annotation to mark an interface as Functional Interface. `@FunctionalInterface` annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces, and it's best practice to use it.
- `java.lang Runnable` with single abstract method `run()` is a great example of functional interface.
- One of the major benefits of functional interface is the possibility to use **lambda expressions** to instantiate them. We can instantiate an interface with **anonymous class** but the code looks bulky.

Lambda Expressions

- **Lambda expressions:**

The major benefit of java 8 functional interfaces is that we can use **lambda expressions** to instantiate them and avoid using bulky anonymous class implementation (like next runnable class example).

Java 8 Functional Interfaces and Lambda Expressions help us in writing smaller and cleaner code.

Java 8 Collections API has been rewritten and new Stream API is introduced that uses a lot of functional interfaces.

Lambda Expressions

- Lambda Expressions syntax is **(argument) -> (body)**.
- Just like if-else blocks, we can avoid curly braces ({}) since we have a single statement in the method body. For multiple statements, we would have to use curly braces like any other methods.
- Type is interfered which mean no need to specify the type of object we use in lambda expressions.
- For runnable class we can use it as :
 - **Runnable r1 = () -> System.out.println("My Runnable");**
 - Since run() method takes no argument, our lambda expression also have no argument.

What is the meaning of type interfering?

It means the data type of any expression(method return type or parameter type)can be deduced automatically by the compiler.

Lambda Expressions

```
Runnable r = new Runnable(){  
    @Override  
    public void run() {  
        System.out.println("My Runnable");  
    }  
};
```

Since functional interfaces have only one method, lambda expressions can easily provide the method implementation. We just need to provide method arguments and business logic. For example, we can write above implementation using lambda expression as:

```
Runnable r1 = () -> {  
    System.out.println("My Runnable");  
};
```

Lambda Expressions

- **Why do we need Lambda Expression?**

- Reduced Lines of Code.
- Sequential and Parallel Execution Support(streams).
Passing Behaviors into methods(can send lambda expression to a method as an argument).
- Higher Efficiency with Laziness.

Lambda Expressions

Delayed Initialization

In object-oriented programming, we ensure that objects are well constructed before any method calls. We encapsulate, ensure proper state transitions, and preserve the object's invariants. This works well most of the time, but when parts of an object's internals are heavyweight resources, we'll benefit if we postpone creating them. This can speed up object creation, and the program doesn't expend any effort creating things that may not be used.

Lazy Evaluations

Java already uses lazy execution when evaluating logical operations. For example, in `fn1() || fn2()`, the call `fn2()` is never performed if `fn1()` returns a boolean `true`. While Java uses lazy or normal order when evaluating logical operators, it uses an eager or applicative order when evaluating method arguments. All the arguments to methods are fully evaluated before a method is invoked. If the method doesn't use all of the passed arguments, the program has wasted time and effort executing them. We can use lambda expressions to postpone the execution of select arguments.

Let's start with a method `evaluate()` that takes quite a bit of time and resources to run.

Lambda Expressions

Lambda Expression Examples

Below I am providing some code snippets for lambda expressions with small comments explaining them.

```
() -> {} // no parameters; void result

() -> 42 // No parameters, expression body
() -> null // No parameters, expression body
() -> { return 42; } // No parameters, block body with return
() -> { System.gc(); } // No parameters, void block body

// Complex block body with multiple returns
() -> {
    if (true) return 10;
    else {
        int result = 15;
        for (int i = 1; i < 10; i++)
            result *= i;
        return result;
    }
}

(int x) -> x+1 // Single declared-type argument
(int x) -> { return x+1; } // same as above
(x) -> x+1 // Single inferred-type argument, same as below
x -> x+1 // Parenthesis optional for single inferred-type case

(String s) -> s.length() // Single declared-type argument
(Thread t) -> { t.start(); } // Single declared-type argument
s -> s.length() // Single inferred-type argument
t -> { t.start(); } // Single inferred-type argument

(int x, int y) -> x+y // Multiple declared-type parameters
(x,y) -> x+y // Multiple inferred-type parameters
(x, final y) -> x+y // Illegal: can't modify inferred-type parameters
(x, int y) -> x+y // Illegal: can't mix inferred and declared types
```

Lambda Expressions

Method and Constructor References

A method reference is used to refer to a method without invoking it; a constructor reference is similarly used to refer to a constructor without creating a new instance of the named class or array type.

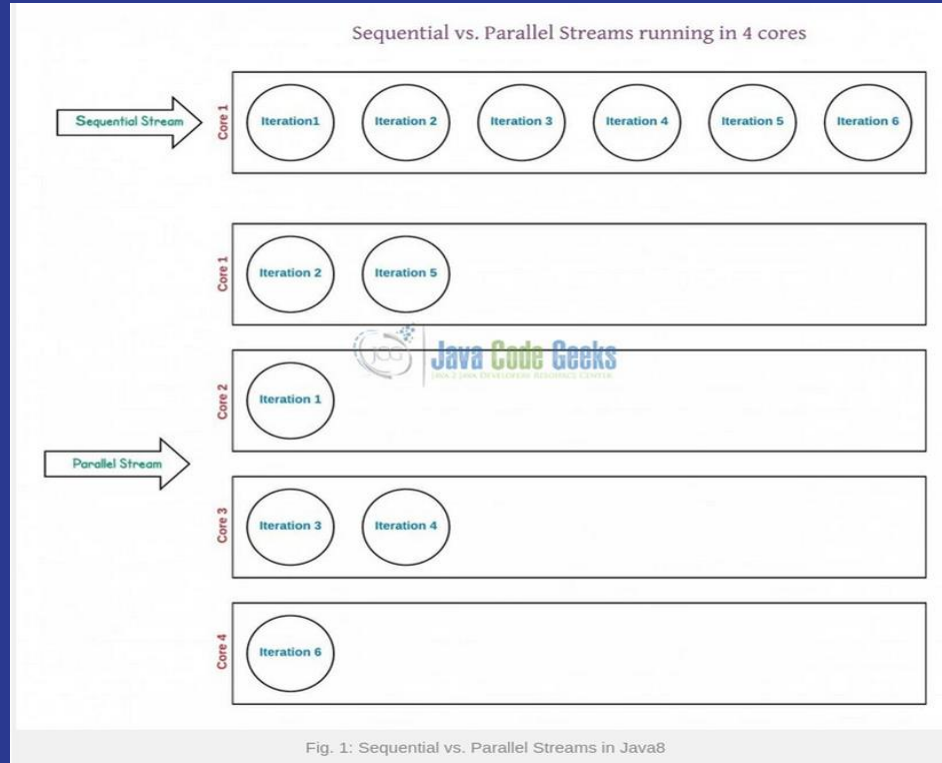
Examples of method and constructor references:

```
System::getProperty  
System.out::println  
"abc"::length  
ArrayList::new  
int[]::new
```


Java Stream API for Bulk Data Operations on Collections

- A new package `java.util.stream` has been added in Java 8 to perform filter/map/reduce like operations with the collection.
- Stream API will allow sequential as well as parallel execution.
- Collection interface has been extended with *stream()* and *parallelStream()* default methods to get the Stream for sequential and parallel execution.
- Parallel streams are efficient for large data collections and time efficient and it uses join -fork mechanism.
- Mainly used to have multiple views of the same collection without accessing database again.
- Can be used for any type of collections (lists-maps-arrays....)

Java Stream API for Bulk Data Operations on Collections



Java Stream API for Bulk Data Operations on Collections

Should i always use Parallel streams ?

- No just use it in the following cases as the cost of join-fork will be expensive in cases which is not suitable (data sets is not large - also order factor is needed).
- we should not casually use the `Stream.parallel()` every time and there assuming that it may make things faster. In fact, invoking `parallel()` may bring down the performance for small streams.

Java Stream API for Bulk Data Operations on Collections

When to use Parallel Streams?

- They should be used when the output of the operation is **not** needed to be **dependent** on the **order** of elements present in source collection (i.e. on which the stream is created)
- Parallel Streams can be used in case of **aggregate** functions
- Parallel Streams quickly iterate over the **large-sized** collections
- Parallel Streams can be used if developers have **performance** implications with the Sequential Streams
- If the environment is not multi-threaded, then Parallel Stream **creates thread** and can affect the new requests coming in.

Java Stream API -examples

```
package com.journaldev.java8.stream;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class StreamExample {

    public static void main(String[] args) {

        List<Integer> myList = new ArrayList<>();
        for(int i=0; i<100; i++) myList.add(i);

        //sequential stream
        Stream<Integer> sequentialStream = myList.stream();

        //parallel stream
        Stream<Integer> parallelStream = myList.parallelStream();

        //using lambda with Stream API, filter example
        Stream<Integer> highNums = parallelStream.filter(p -> p > 90);
        //using lambda in forEach
        highNums.forEach(p -> System.out.println("High Nums parallel="+p));

        Stream<Integer> highNumsSeq = sequentialStream.filter(p -> p > 90);
        highNumsSeq.forEach(p -> System.out.println("High Nums sequential="+p));

    }
}
```

```
High Nums parallel=91
High Nums parallel=96
High Nums parallel=93
High Nums parallel=98
High Nums parallel=94
High Nums parallel=95
High Nums parallel=97
High Nums parallel=92
High Nums parallel=99
High Nums sequential=91
High Nums sequential=92
High Nums sequential=93
High Nums sequential=94
High Nums sequential=95
High Nums sequential=96
High Nums sequential=97
High Nums sequential=98
High Nums sequential=99
```


Java Stream API -examples

Convert a list of object to a list of another type:

Convert a list of object to a list of integers contains id only of the object:

```
List<Integer> idsList = usersList.stream() .map(User::getId) .collect(Collectors.toList());
```

Create list of Strings from list of objects:

```
List<String> names = personList.stream() .map(Person::getName) .collect(Collectors.toList());
```

Convert a list of object to a list of another object:

Consider we have employee object which contain id,salary attributes
EmployeeRaise object that have emp_id,raise which is salary*100

```
List<Emp> employee = Arrays.asList(new Emp(1, 100), new Emp(2, 200), new Emp(3, 300));  
List<EmployeeRaise> employeeRaise = employee.stream()  
    .map(emp -> new EmployeeRaise(emp.getId(), emp.getSalary * 100))  
    .collect(Collectors.toList());  
employeeRaise.stream()  
    .forEach(s -> System.out.println("Id ." + s.getId() + " Salary ." + s.getSalary()));
```

In Java 8, stream().map() lets you convert an object to something else.

Java Stream API -examples

Get the sum of a list.

List of double:(using Method reference ::)

```
Double total= accountList.stream()
    .filter(account ->account!=null&&
        account.getAmount()!=null).mapToDouble(Account::getAmount)
    .sum();
```

List of integers:

```
int sum = lst.stream().filter(o -> o.getField() > 10).mapToInt(o -> o.getField()).sum();
```

Java Stream anyMatch()

```
Stream<String> stream = Stream.of("one", "two", "three", "four");
```

```
boolean match = stream.anyMatch(s -> s.contains("four"));
```

```
System.out.println(match); output true
```

Optional class

The purpose of the class is to provide a type-level solution for representing optional values instead of null references.

When we have an Optional object returned from a method or created by us, we can check if there is a value in it or not with the `isPresent()` method:

example:

```
List<String> filteredList = listOfOptionals.stream()  
    .filter(Optional::isPresent)  
    .map(Optional::get)  
    .collect(Collectors.toList());
```

Java Time API

- It has always been hard to work with Date, Time and Time Zones in java. There was no standard approach or API in java for date and time in Java. One of the nice addition in Java 8 is the java.time package that will streamline the process of working with time in java(as if joda library added to standers java).
- Just by looking at Java Time API packages, I can sense that it will be very easy to use. It has some sub-packages java.time.format that provides classes to **print** and **parse** dates and times and java.time.zone provides support for **time-zones** and their rules.
- Java 8 Date Time API Examples:**LocalDate,LocalTime,LocalDateTime,Instant.**
- For more information you can visit:

<https://www.journaldev.com/2800/java-8-date-localdate-localdatetime-instant>

Java Time API

LocalDate:

- is an immutable class that represents Date with default format of yyyy-MM-dd. We can use now() method to get the current date. We can also provide input arguments for year, month and date to create LocalDate instance.
- We can pass ZoneId for getting date in specific time zone.

```
import java.time.LocalDate;
import java.time.Month;
import java.time.ZoneId;

/**
 * LocalDate Examples
 * @author pankaj
 */
public class LocalDateExample {

    public static void main(String[] args) {

        //Current Date
        LocalDate today = LocalDate.now();
        System.out.println("Current Date="+today);

        //Creating LocalDate by providing input arguments
        LocalDate firstDay_2014 = LocalDate.of(2014, Month.JANUARY, 1);
        System.out.println("Specific Date="+firstDay_2014);

        //Try creating date by providing invalid inputs
        //LocalDate feb29_2014 = LocalDate.of(2014, Month.FEBRUARY, 29);
        //Exception in thread "main" java.time.DateTimeException:
        //Invalid date 'February 29' as '2014' is not a leap year

        //Current date in "Asia/Kolkata", you can get it from ZoneId javadoc
        LocalDate todayKolkata = LocalDate.now(ZoneId.of("Asia/Kolkata"));
        System.out.println("Current Date in IST="+todayKolkata);
    }
}
```

Java Time API

LocalTime

- LocalTime is an immutable class whose instance represents a time in the human readable format. It's default format is hh:mm:ss.zzz. Just like LocalDate, this class provides time zone support and creating instance by passing hour, minute and second as input arguments

```
import java.time.LocalTime;
import java.time.ZoneId;

/**
 * LocalTime Examples
 * @author pankaj
 */
public class LocalTimeExample {

    public static void main(String[] args) {

        //Current Time
        LocalTime time = LocalTime.now();
        System.out.println("Current Time="+time);

        //Creating LocalTime by providing input arguments
        LocalTime specificTime = LocalTime.of(12,20,25,40);
        System.out.println("Specific Time of Day="+specificTime);

        //Try creating time by providing invalid inputs
        //LocalTime invalidTime = LocalTime.of(25,20);
        //Exception in thread "main" java.time.DateTimeException:
        //Invalid value for HourOfDay (valid values 0 - 23): 25

        //Current date in "Asia/Kolkata", you can get it from ZoneId javadoc
        LocalTime timeKolkata = LocalTime.now(ZoneId.of("Asia/Kolkata"));
        System.out.println("Current Time in IST="+timeKolkata);
    }
}
```

Java Time API

LocalDateTime

- LocalDateTime is an immutable date-time object that represents a date-time, with default format as yyyy-MM-dd-HH-mm-ss.zzz. It provides a factory method that takes LocalDate and LocalTime input arguments to create LocalDateTime instance.

```
import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZoneOffset;

public class LocalDateTimeExample {

    public static void main(String[] args) {

        //Current Date
        LocalDateTime today = LocalDateTime.now();
        System.out.println("Current DateTime="+today);

        //Current Date using LocalDate and LocalTime
        today = LocalDateTime.of(LocalDate.now(), LocalTime.now());
        System.out.println("Current DateTime="+today);

        //Creating LocalDateTime by providing input arguments
        LocalDateTime specificDate = LocalDateTime.of(2014, Month.JANUARY, 1, 10, 10, 30);
        System.out.println("Specific Date="+specificDate);

        //Try creating date by providing invalid inputs
        //LocalDateTime feb29_2014 = LocalDateTime.of(2014, Month.FEBRUARY, 28, 25,1,1);
        //Exception in thread "main" java.time.DateTimeException:
        //Invalid value for HourOfDay (valid values 0 - 23): 25

        //Current date in "Asia/Kolkata", you can get it from ZoneId javadoc
        LocalDateTime todayKolkata = LocalDateTime.now(ZoneId.of("Asia/Kolkata"));
        System.out.println("Current Date in IST="+todayKolkata);
    }
}
```

Java Time API-Utilities

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

//Get the Year, check if it's leap year
System.out.println("Year "+today.getYear()+" is Leap Year? "+today.isLeapYear());

//Compare two LocalDate for before and after
System.out.println("Today is before 01/01/2015? "+today.isBefore(LocalDate.of(2015,1,1)));

//Create LocalDateTime from LocalDate
System.out.println("Current Time="+today.atTime(LocalTime.now()));

//plus and minus operations
System.out.println("10 days after today will be "+today.plusDays(10));
System.out.println("3 weeks after today will be "+today.plusWeeks(3));
System.out.println("20 months after today will be "+today.plusMonths(20));

System.out.println("10 days before today will be "+today.minusDays(10));
System.out.println("3 weeks before today will be "+today.minusWeeks(3));
System.out.println("20 months before today will be "+today.minusMonths(20));

//Temporal adjusters for adjusting the dates
System.out.println("First date of this month=
"+today.with(TemporalAdjusters.firstDayOfMonth()));
LocalDate lastDayOfYear = today.with(TemporalAdjusters.lastDayOfYear());
System.out.println("Last date of this year= "+lastDayOfYear);

Period period = today.until(lastDayOfYear);
System.out.println("Period Format= "+period);
```


Java Time API-parsing and formatting

It's very common to format date into different formats and then parse a String to get the Date Time objects. Let's see it with simple examples.



```
package com.journaldev.java8.time;

import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class DateParseFormatExample {

    public static void main(String[] args) {

        //Format examples
        LocalDate date = LocalDate.now();
        //default format
        System.out.println("Default format of LocalDate="+date);
        //specific format
        System.out.println(date.format(DateTimeFormatter.ofPattern("d::MMM::uuuu")));
        System.out.println(date.format(DateTimeFormatter.BASIC_ISO_DATE));

        LocalDateTime dateTime = LocalDateTime.now();
        //default format
        System.out.println("Default format of LocalDateTime="+dateTime);
        //specific format
        System.out.println(dateTime.format(DateTimeFormatter.ofPattern("d::MMM::uuuu HH::mm::ss")));
        System.out.println(dateTime.format(DateTimeFormatter.BASIC_ISO_DATE));
```

Java Time API

Parsing :

```
//Parse examples
LocalDateTime dt = LocalDateTime.parse("27::Apr::2014 21::39::48",
    DateTimeFormatter.ofPattern("d::MMM::uuuu HH::mm::ss"));
System.out.println("Default format after parsing = "+dt);
}
}
```

When we run above program, we get following output.

```
Default format of LocalDate=2014-04-28
28::Apr::2014
20140428
Default format of LocalDateTime=2014-04-28T16:25:49.341
28::Apr::2014 16::25::49
20140428
Default format of Instant=2014-04-28T23:25:49.342Z
Default format after parsing = 2014-04-27T21:39:48
```

Collection API improvements

- We have already seen **forEach()** method and Stream API for collections.
(The `forEach()` method is a default method introduced by the **Iterable** interface at 1.8..)
Some new methods added in Collection API are:
- Iterator default method **forEachRemaining()**(Consumer action) to perform the given action for each remaining element until all elements have been processed or the action throws an exception(Before Java 8, the `forEachRemaining()` method did not exist).
(`forEachRemaining()` is a default method introduced by the **Iterator** interface at 1.8.)

Differences between Iterator and Iterable in Java:

Both Iterator and Iterable are interfaces in Java which looks very .if any class implements the Iterable interface, it gains the ability to iterate over an object of that class using an Iterator.

Collection API improvements

- **forEachRemaining** example:

Below is a simple program that shows how you would iterate a list using an iterator before Java 8.

```
1 import java.util.ArrayList;
2 import java.util.Iterator;
3 import java.util.List;
4
5 public class IteratorDemo {
6
7     public static void main(String args[]) {
8
9         List<String> fruits = new ArrayList<>();
10        fruits.add("Apple");
11        fruits.add("Banana");
12        fruits.add("Grapes");
13        fruits.add("Orange");
14
15        Iterator<String> iterator = fruits.iterator();
16
17        while (iterator.hasNext()) {
18            System.out.println(iterator.next());
19        }
20    }
21 }
22
```

Below is the same example shown above, but this time, we are using the `forEachRemaining()` method.

```
1 import java.util.ArrayList;
2 import java.util.Iterator;
3 import java.util.List;
4
5 public class IteratorDemo {
6
7     public static void main(String args[]) {
8
9         List<String> fruits = new ArrayList<>();
10        fruits.add("Apple");
11        fruits.add("Banana");
12        fruits.add("Grapes");
13        fruits.add("Orange");
14
15        Iterator<String> iterator = fruits.iterator();
16
17        iterator.forEachRemaining((fruit) -> System.out.println(fruit));
18    }
19 }
20
```


Run

Output

Apple
Banana
Grapes
Orange

Collection API improvements

- Collection default method **removeIf**(Predicate filter) to remove all of the elements of this collection that satisfy the given predicate.



```
public static void main(String[] args)
{
    // create an ArrayList which going to
    // contains a list of Student names which is actually
    // string values
    ArrayList<String> students = new ArrayList<String>();

    // Add Strings to list
    // each string represents student name
    students.add("Ram");
    students.add("Mohan");
    students.add("Sohan");
    students.add("Rabi");
    students.add("Shabbir");

    // apply removeIf() method
    // we are going to remove names
    // start with S
    students.removeIf(n -> (n.charAt(0) == 'S'));

    System.out.println("Students name Does not start with S");
    // print list
    for (String str : students) {
        System.out.println(str);
    }
}
```

Output:

```
Students name Does not start with S
Ram
Mohan
Rabi
```

Collection API improvements

- Map **replaceAll()**, **compute()**, **merge()** methods.
- Performance Improvement for HashMap class with Key Collisions.

```
import java.util.HashMap;
import java.util.Map;

public class ReplaceAll1 {
    public static void main(String[] args) {
        Map<Integer, String> studentMap = new HashMap<>();
        studentMap.put(101, "Mahesh");
        studentMap.put(102, "Suresh");
        studentMap.put(103, "Krishna");

        System.out.println("--- before replaceAll() ---");

        System.out.println(studentMap);

        studentMap.replaceAll((k,v) -> v + "-" + k);

        System.out.println("--- after replaceAll() ---");

        System.out.println(studentMap);
    }
}
```

Output

```
--- before replaceAll() ---
{101=Mahesh, 102=Suresh, 103=Krishna}
--- after replaceAll() ---
{101=Mahesh-101, 102=Suresh-102, 103=Krishna-103}
```

Collection API improvements

- Collection **splitterator()** method returning Spliterator instance that can be used to traverse elements sequentially or parallel.

A Spliterator may traverse elements individually (`tryAdvance()`) or sequentially in bulk (`forEachRemaining()`).

Java Spliterator interface is an internal iterator that breaks the stream into the smaller parts. These smaller parts can be processed in parallel.

can use it for both Collection API and Stream API classes.

.
<https://howtodoinjava.com/java/collections/java-spliterator/>

Java IO improvements

Some IO improvements known to me are:

- `Files.list(Path dir)` that returns a lazily populated Stream, the elements of which are the entries in the **directory**.
- `Files.lines(Path path)` that reads **all lines** from a file as a Stream.
- `Files.find()` that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.
- `BufferedReader.lines()` that return a Stream, the elements of which are lines read from this `BufferedReader`.

Java IO improvements

Can be used also for:

Find files by filename.

Find by file Size.

Find files by last modified time.

What are symbolic links used for?

A symbolic link (or "symlink") is file system feature that can be used to create a link to a specific file or folder. It is similar to a Windows "shortcut" or Mac "alias," but is not an actual file. Instead, a symbolic link is a entry in a file system that points to a directory or file.

<https://mkyong.com/java/java-files-find-examples/>

1. Files.find() Signature

Review the `Files.find()` signature.

Files.java

```
public static Stream<Path> find(Path start,
                                int maxDepth,
                                BiPredicate<Path, BasicFileAttributes> matcher,
                                FileVisitOption... options)
    throws IOException
```

- The `path`, starting file or folder.
- The `maxDepth` defined the maximum number of directory levels to search. If we put `1`, which means the search for top-level or root folder only, ignore all its subfolders; If we want to search for all folder levels, put `Integer.MAX_VALUE`.
- The `BiPredicate<Path, BasicFileAttributes>` is for condition checking or filtering.
- The `FileVisitOption` tells if we want to follow symbolic links, default is no. We can put `FileVisitOption.FOLLOW_LINKS` to follow symbolic links.

Miscellaneous Core API improvements

- `ThreadLocal` static method with `Initial(Supplier supplier)` to create instance easily.-->The `ThreadLocal` construct allows us to store data that will be accessible only by a specific thread.
- `Comparator` interface has been extended with a lot of default and static methods for natural ordering, reverse order etc.
- `min()`, `max()` and `sum()` methods in `Integer`, `Long` and `Double` wrapper classes.
- `logicalAnd()`, `logicalOr()` and `logicalXor()` methods in `Boolean` class.
- `ZipFile.stream()` method to get an ordered `Stream` over the ZIP file entries. Entries appear in the `Stream` in the order they appear in the central directory of the ZIP file.
- Several utility methods in `Math` class.
- `jjc` command is added to invoke Nashorn Engine(JavaScript *engine* developed in the Java programming language by Oracle).
- `jdeps` command is added to analyze class files
- JDBC-ODBC Bridge has been removed.
- PermGen memory space has been removed→ **enhancement in memory handling.**

Concurrency API improvements

Some important concurrent API enhancements are:

- `ConcurrentHashMap` `compute()`, `forEach()`, `forEachEntry()`, `forEachKey()`, `forEachValue()`, `merge()`, `reduce()` and `search()` methods.
- `CompletableFuture` that may be explicitly completed (setting its value and status)(introduction to reactive programming `Mono-Flux` Datatypes in spring webflux)
- `Executors` `newWorkStealingPool()` method to create a work-stealing thread pool using all available processors as its target parallelism level.

Concurrency API improvements

More about executors :

- The Concurrency API introduces the concept of an ExecutorService as a higher level **replacement** for working with threads **directly**.
- Executors are capable of running asynchronous tasks and typically manage a pool of threads, so we don't have to create new threads **manually**.
- All threads of the internal pool will be **reused** under the hood for reventant tasks, so we can run as many concurrent tasks as we want throughout the life-cycle of our application with a single executor service.

Concurrency API improvements

Threads can be put to sleep for a certain duration. This is quite handy to simulate long running tasks in the subsequent code samples of this article:

```
Runnable runnable = () -> {
    try {
        String name = Thread.currentThread().getName();
        System.out.println("Foo " + name);
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Bar " + name);
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Thread thread = new Thread(runnable);
thread.start();
```

This is how the first thread-example looks like using executors:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});

// => Hello pool-1-thread-1
```

The class `Executors` provides convenient factory methods for creating different kinds of executor services. In this sample we use an executor with a thread pool of size one.

The result looks similar to the above sample but when running the code you'll notice an important difference: the java process never stops! Executors have to be stopped explicitly - otherwise they keep listening for new tasks.

An `ExecutorService` provides two methods for that purpose: `shutdown()` waits for currently running tasks to finish while `shutdownNow()` interrupts all running tasks and shut the executor down immediately.

This is the preferred way how I typically shutdown executors:

```
try {
    System.out.println("attempt to shutdown executor");
    executor.shutdown();
    executor.awaitTermination(5, TimeUnit.SECONDS);
}
catch (InterruptedException e) {
    System.err.println("tasks interrupted");
}
finally {
    if (!executor.isTerminated()) {
        System.err.println("cancel non-finished tasks");
    }
    executor.shutdownNow();
    System.out.println("shutdown finished");
}
```

Concurrency API improvements

Callables and Futures

In addition to `Runnable`, executors support another kind of task named `Callable`. Callables are functional interfaces just like runnables but instead of being `void` they return a value.

This lambda expression defines a callable returning an integer after sleeping for one second:

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
```

----->
Using future

Callables can be submitted to executor services just like runnables. But what about the callables result? Since `submit()` doesn't wait until the task completes, the executor service cannot return the result of the callable directly. Instead the executor returns a special result of type `Future` which can be used to retrieve the actual result at a later point in time.

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());

Integer result = future.get();

System.out.println("future done? " + future.isDone());
System.out.print("result: " + result);
```

After submitting the callable to the executor we first check if the future has already been finished execution via `isDone()`. I'm pretty sure this isn't the case since the above callable sleeps for one second before returning the integer.

Calling the method `get()` blocks the current thread and waits until the callable completes before returning the actual result `123`. Now the future is finally done and we see the following

result on the console:

```
future done? false
future done? true
result: 123
```

<https://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/>

References

<https://www.journaldev.com/2389/java-8-features-with-examples>

<https://www.educative.io/edpresso/introduction-to-foreachremaining-in-javas-iterator>

<https://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/>

<https://mkyong.com/java/java-files-find-examples/>