Java 8 New Features

Amit Kukadiya

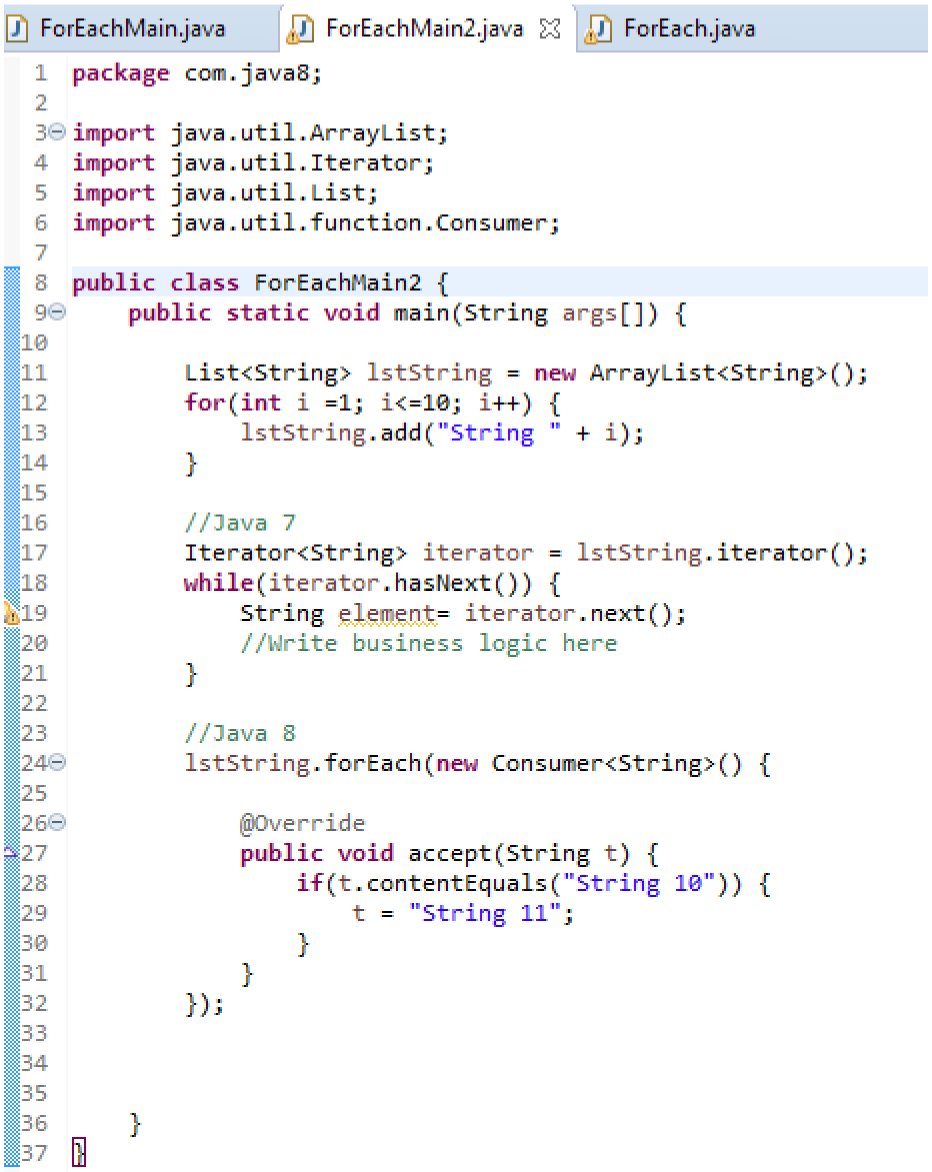
5th December, 2020

Java 8 Features

1. For Each method in iterable interface
2. Interface Change in java 8 , Methods with default & static keywords in java 8 interface
3. Functional interface
4. Lambda Expression in java 8
5. Java Stream API for Collection Bulk operations
6. Java Time API.
7. Collection API improvements
8. Concurrency API Improvements
9. Java IO Improvements
10. Miscellaneous Core API improvements

For each in Iterable interface

- Whenever we need to traverse through a collection then We need to create Iterator in Java 7.
- Purpose of Iterator is to write business logic.
- Java 8 introduced a foreach method with an iterable interface.
- Foreach method takes the one argument java.util.function.Consumer.
- Here Consume is the one type of functional interface. We can focus on writing business logic.

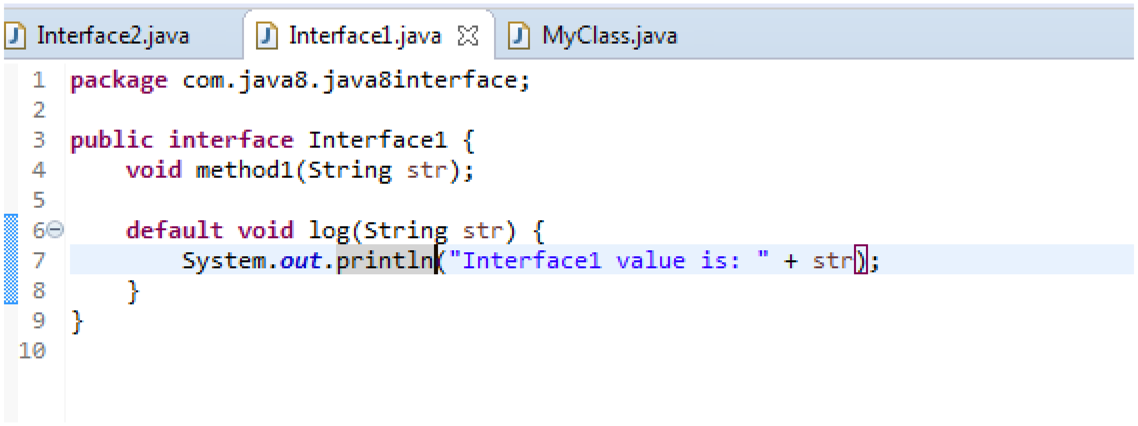


```java
package com.java8;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.function.Consumer;

public class ForEachMain2 {
    public static void main(String args[]) {

        List<String> lstString = new ArrayList<String>();
        for(int i =1; i<=10; i++) {
            lstString.add("String " + i);
        }

        //Java 7
        Iterator<String> iterator = lstString.iterator();
        while(iterator.hasNext()) {
            String element= iterator.next();
            //Write business logic here
        }

        //Java 8
        lstString.forEach(new Consumer<String>() {

            @Override
            public void accept(String t) {
                if(t.contentEquals("String 10")) {
                    t = "String 11";
                }
            }
        });



    }
}
```

Reading Rainbow Tip: https://www.journaldev.com/2389/java-8-features-with-examples#iterable-forEach

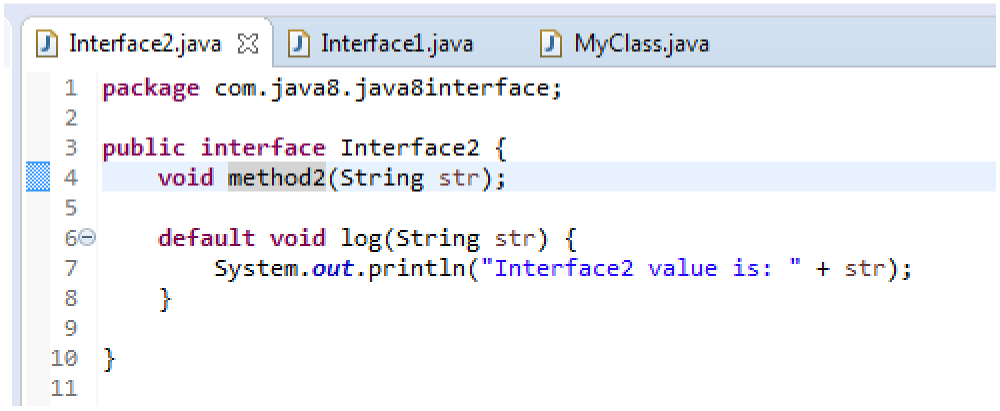
Interface change in java 8

- Default Method implementations
- Designing interfaces has always been a tough job because if there is change in interface then it would be necessary to change the classes in which interface is implemented.
- But Java 8 provides default & static method implementations. Here it would not be necessary to change the classes in which interfaces are implemented.



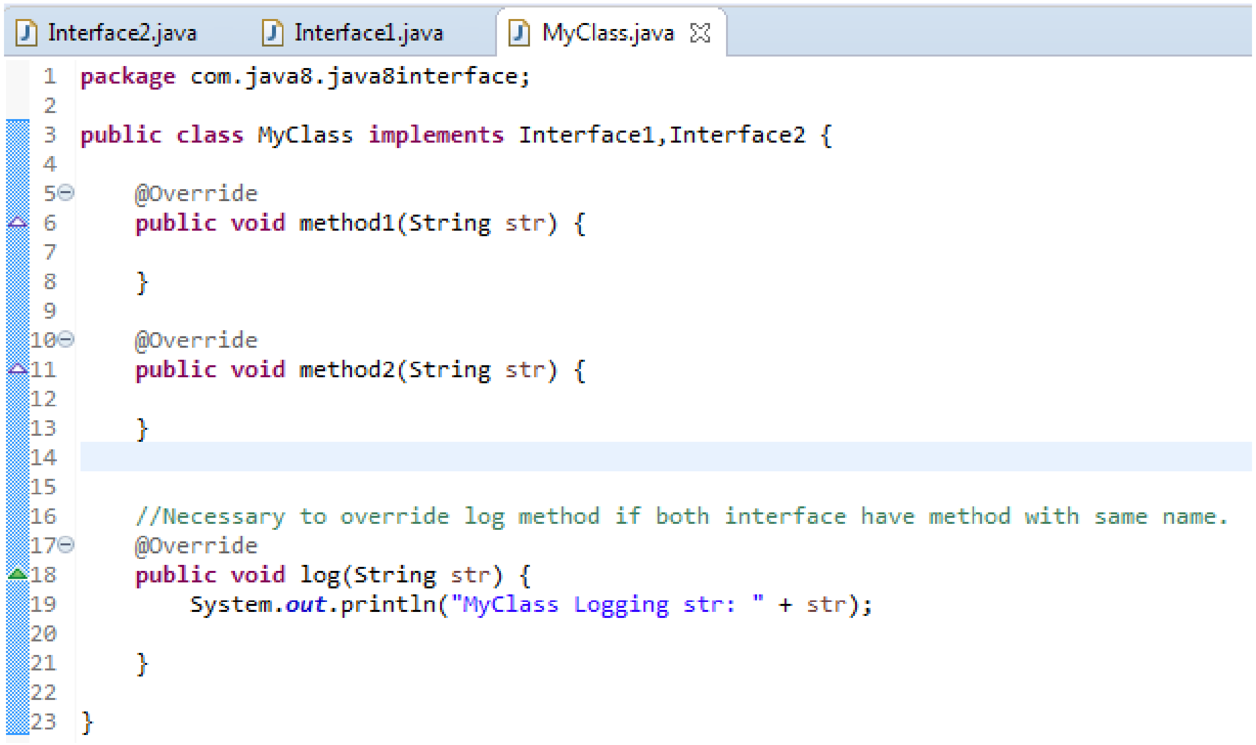
Interface2.java Interface1.java MyClass.java

```java
package com.java8.java8interface;

public interface Interface1 {
    void method1(String str);

    default void log(String str) {
        System.out.println("Interface1 value is: " + str);
    }
}

```



Interface2.java Interface1.java MyClass.java

```java
package com.java8.java8interface;

public interface Interface2 {
    void method2(String str);

    default void log(String str) {
        System.out.println("Interface2 value is: " + str);
    }

}

```

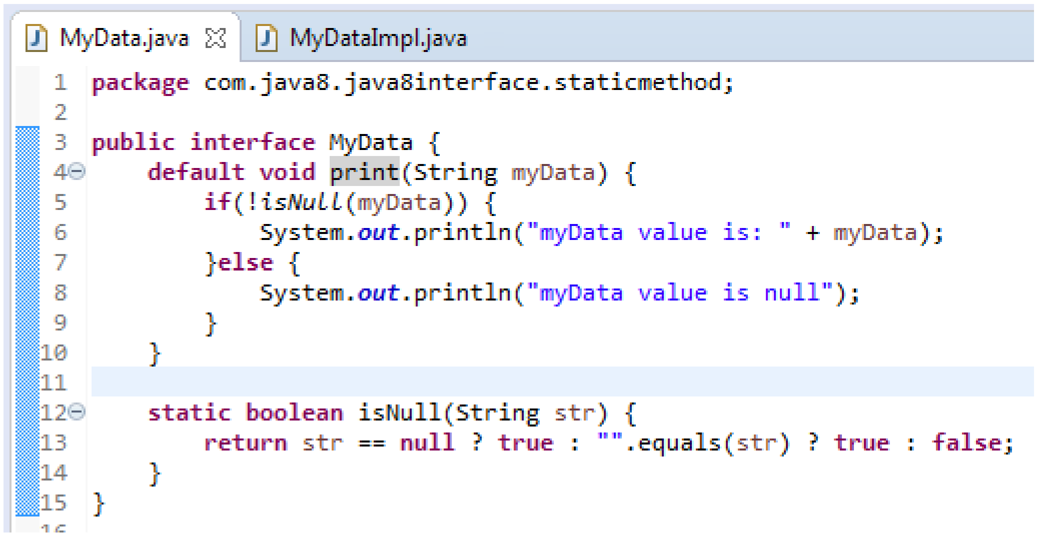


```java
package com.java8.java8interface;

public class MyClass implements Interface1,Interface2 {

    @Override
    public void method1(String str) {

    }

    @Override
    public void method2(String str) {

    }


    //Necessary to override log method if both interface have method with same name.
    @Override
    public void log(String str) {
        System.out.println("MyClass Logging str: " + str);

    }

}
```

Important points about java 8 interfaces:

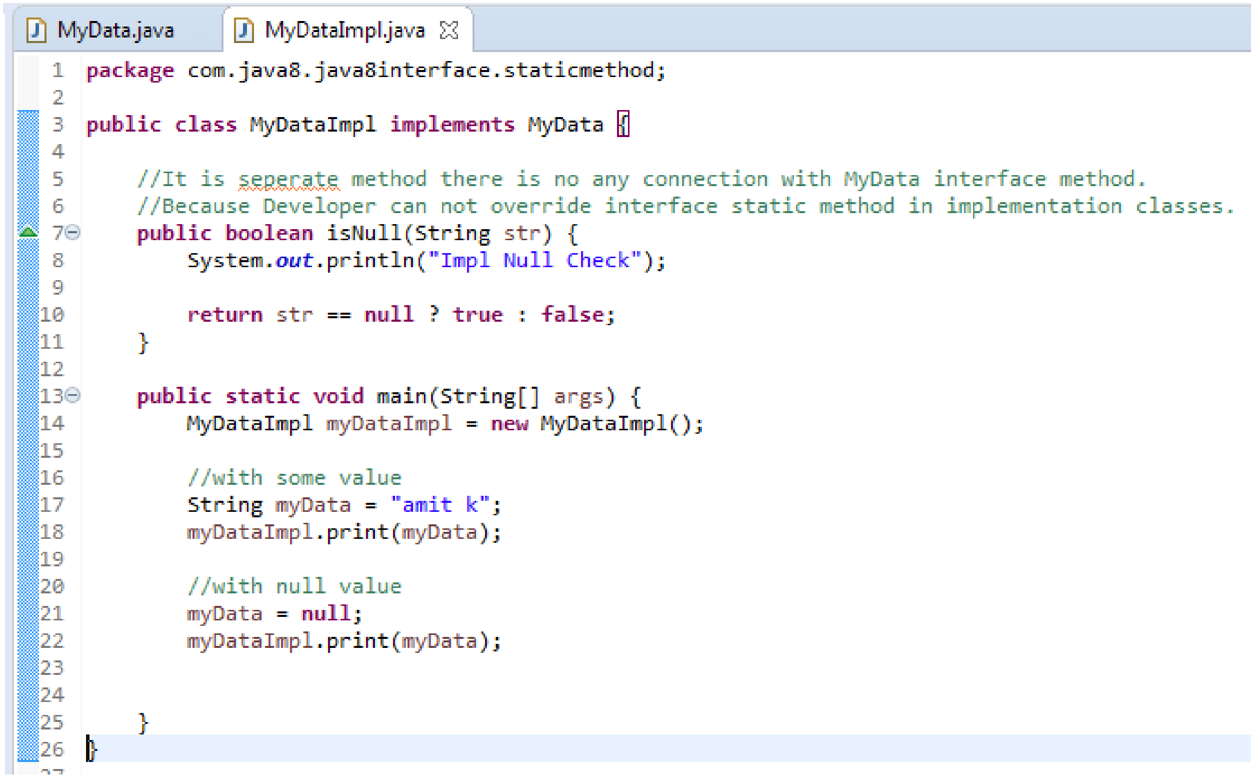
- ➢ Developers can extend interfaces without having the fear about existing classes in which interfaces are implemented.
- ➢ Java collection internally uses java 8 interfaces. Java 8 interface allowed to declare methods with body. So here many utility classes are avoided.
- ➢ One of the major reasons for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
- ➢ If any class hierarchy has the same method signature then the default method becomes irrelavant. See the example MyClass.

- ➢ Static Method implementations
- ➢ Java interface can declare method as static.
- ➢ We can not override them in implementation classes.
- ➢ Java interface static method is visible within the interface.
- ➢ Java interface static method is used to create utility methods like check null , blank etc..



MyData.java | MyDataImpl.java

```java
package com.java8.java8interface.staticmethod;

public interface MyData {
    default void print(String myData) {
        if(!isNull(myData)) {
            System.out.println("myData value is: " + myData);
        }else {
            System.out.println("myData value is null");
        }
    }

    static boolean isNull(String str) {
        return str == null ? true : "".equals(str) ? true : false;
    }
}
```



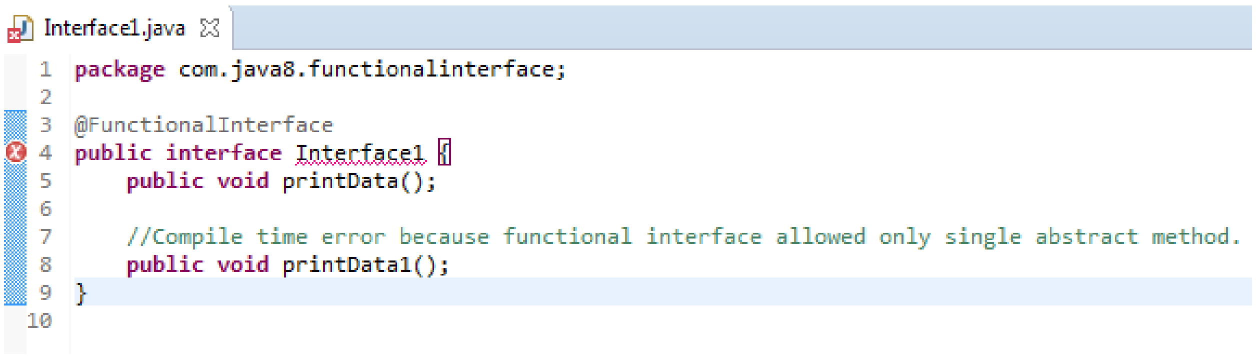
MyData.java | MyDataImpl.java

```java
package com.java8.java8interface.staticmethod;

public class MyDataImpl implements MyData {

    //It is seperate method there is no any connection with MyData interface method.
    //Because Developer can not override interface static method in implementation classes.
    public boolean isNull(String str) {
        System.out.println("Impl Null Check");

        return str == null ? true : false;
    }

    public static void main(String[] args) {
        MyDataImpl myDataImpl = new MyDataImpl();

        //with some value
        String myData = "amit k";
        myDataImpl.print(myData);

        //with null value
        myData = null;
        myDataImpl.print(myData);


    }
}
```

Reference: https://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method

Functional Interface

➢ Functional Interface is the new concept introduced in java 8.
➢ Interface with a single abstract method is called functional interface.
➢ @FunctionalInterface annotation is used to declare an interface as Functional.
➢ @FunctionalInterface annotation avoids some accidental addition of abstract methods.
➢ **It is the best practice to use a functional interface.**
➢ Example of functional interface: java.lang.Runnable with single run() method is an example of functional interface.

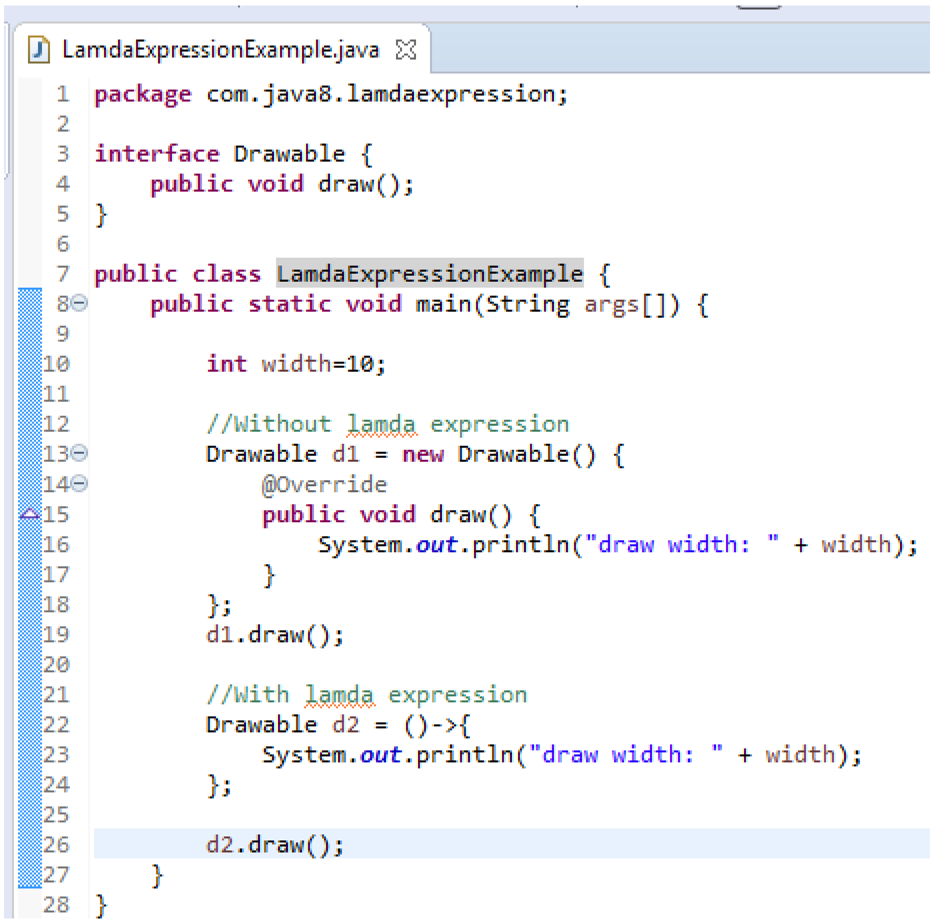


```java
package com.java8.functionalinterface;

@FunctionalInterface
public interface Interface1 {
    public void printData();

    //Compile time error because functional interface allowed only single abstract method.
    public void printData1();
}
```

Reference: https://www.journaldev.com/2389/java-8-features-with-examples#functional-interface-lambdas

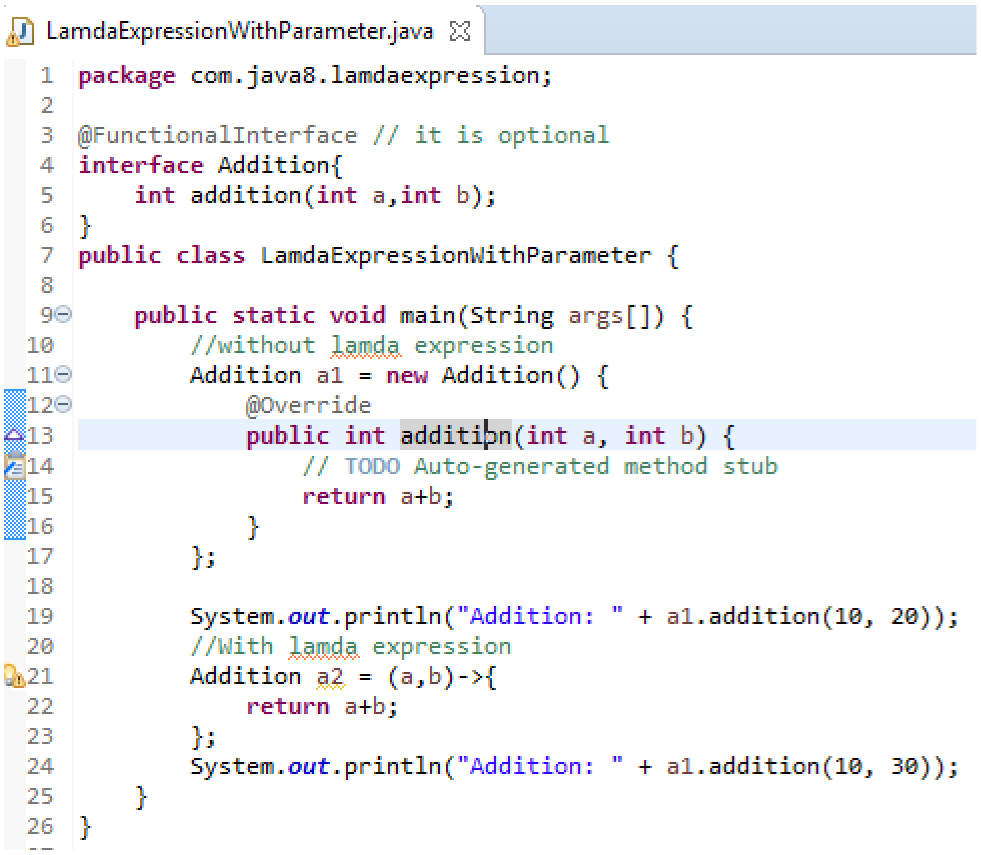
Lamda Expression

- Lamda expression is an implementation of functional interface.
- Why use lamda expression:
  - To provide implementations of functional interface.
  - Write less code.
- Different example of lamda expression are below.



LamdaExpressionExample.java

```java
package com.java8.lamdaexpression;

interface Drawable {
    public void draw();
}

public class LamdaExpressionExample {
    public static void main(String args[]) {

        int width=10;

        //Without lamda expression
        Drawable d1 = new Drawable() {
            @Override
            public void draw() {
                System.out.println("draw width: " + width);
            }
        };
        d1.draw();

        //With lamda expression
        Drawable d2 = ()->{
            System.out.println("draw width: " + width);
        };

        d2.draw();
    }
}
```

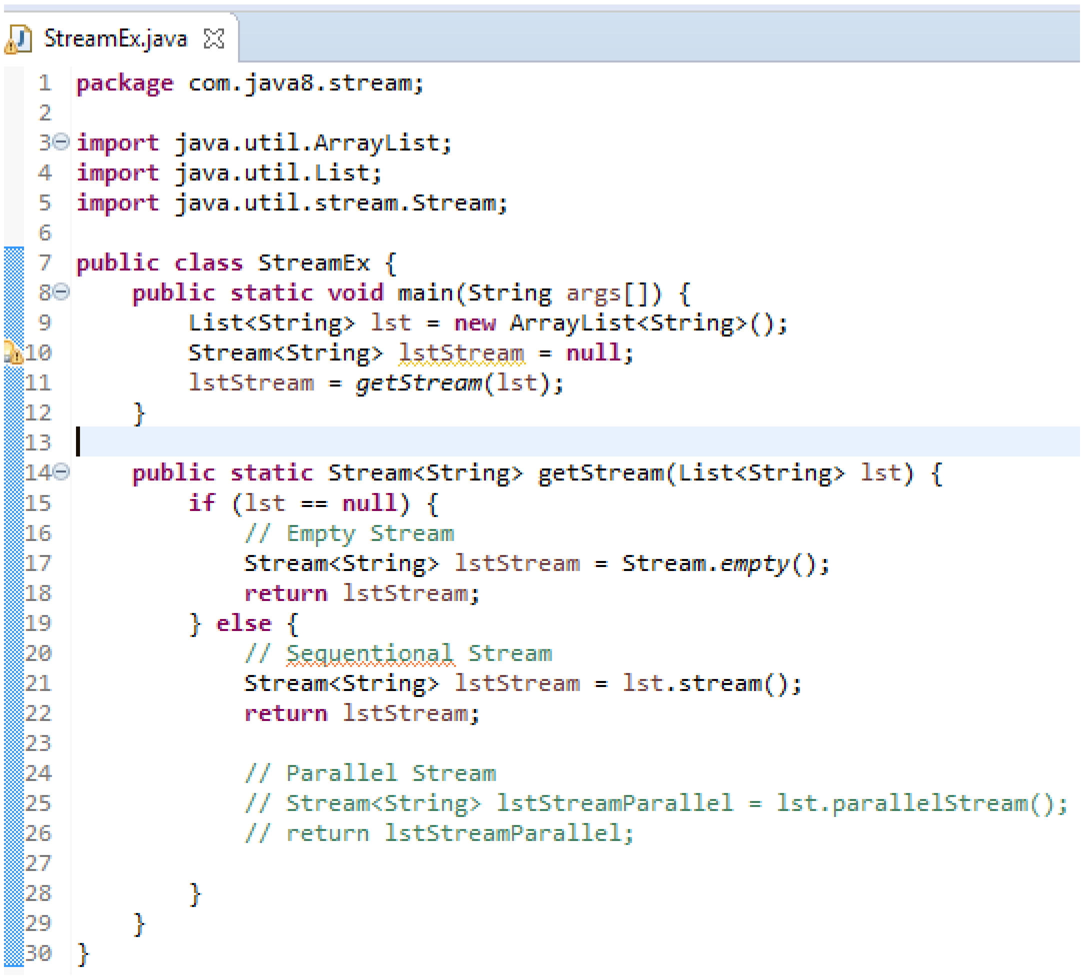

LamdaExpressionWithParameter.java

```
package com.java8.lamdaexpression;

@FunctionalInterface // it is optional
interface Addition{
    int addition(int a,int b);
}
public class LamdaExpressionWithParameter {

    public static void main(String args[]) {
        //without lamda expression
        Addition a1 = new Addition() {
            @Override
            public int addition(int a, int b) {
                // TODO Auto-generated method stub
                return a+b;
            }
        };

        System.out.println("Addition: " + a1.addition(10, 20));
        //With lamda expression
        Addition a2 = (a,b)->{
            return a+b;
        };
        System.out.println("Addition: " + a1.addition(10, 30));
    }
}
```

Reference: https://www.javatpoint.com/java-lambda-expressions

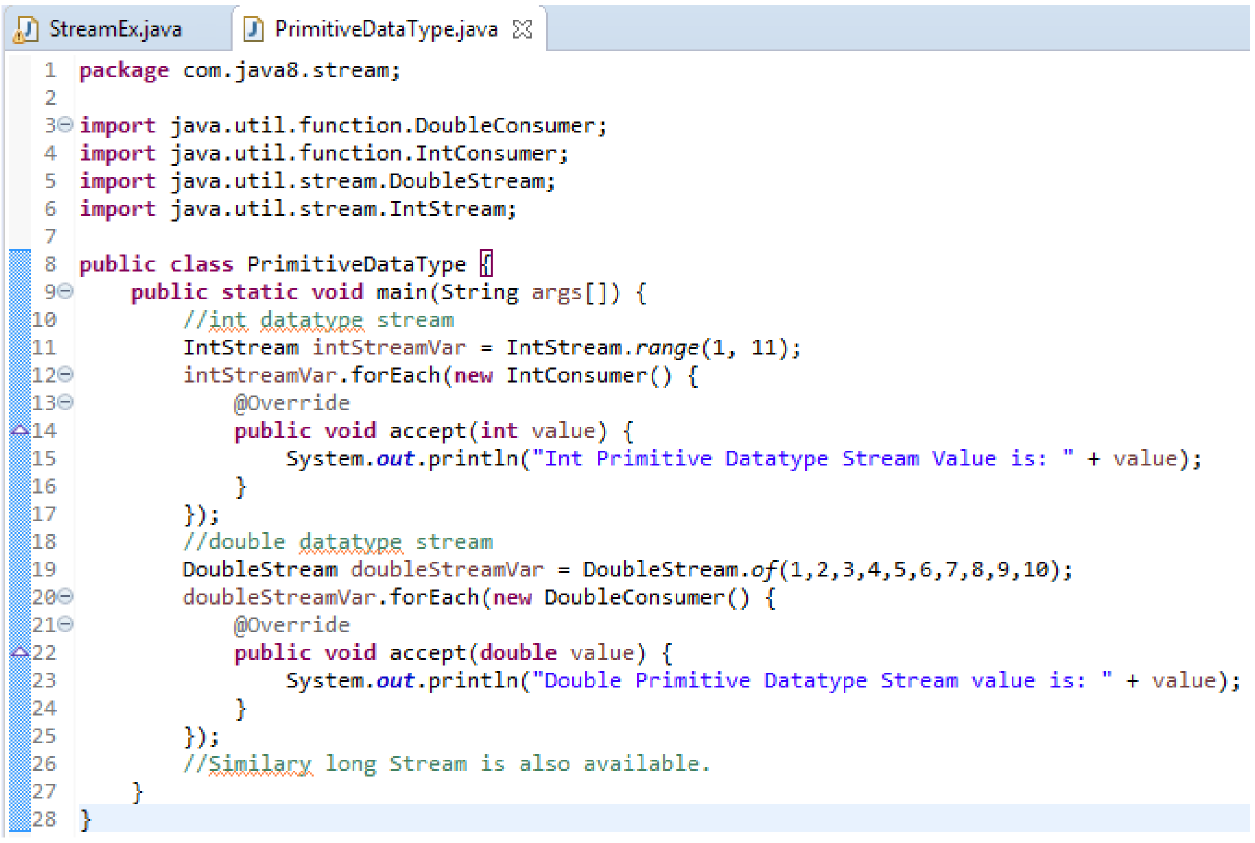
Java Stream API for Collection Bulk Operations

- ➢ Java Stream does not store the data but it operates the data structure of collection/array and produce the pipeline data that can be used to perform specific operations.
- ➢ Java Stream api most of the classes implement function interface and lamda expression.
- ➢ All the Java Stream API interfaces and classes are available in java.util.stream package.
- ➢ Java supported various types of stream. sequential, parallel and empty stream.
- ➢ Java Streams are consumable, so there is no way to create a reference to stream for future usage. since the data is on demand, it is not possible to reuse the same stream multiple times.


StreamEx.java

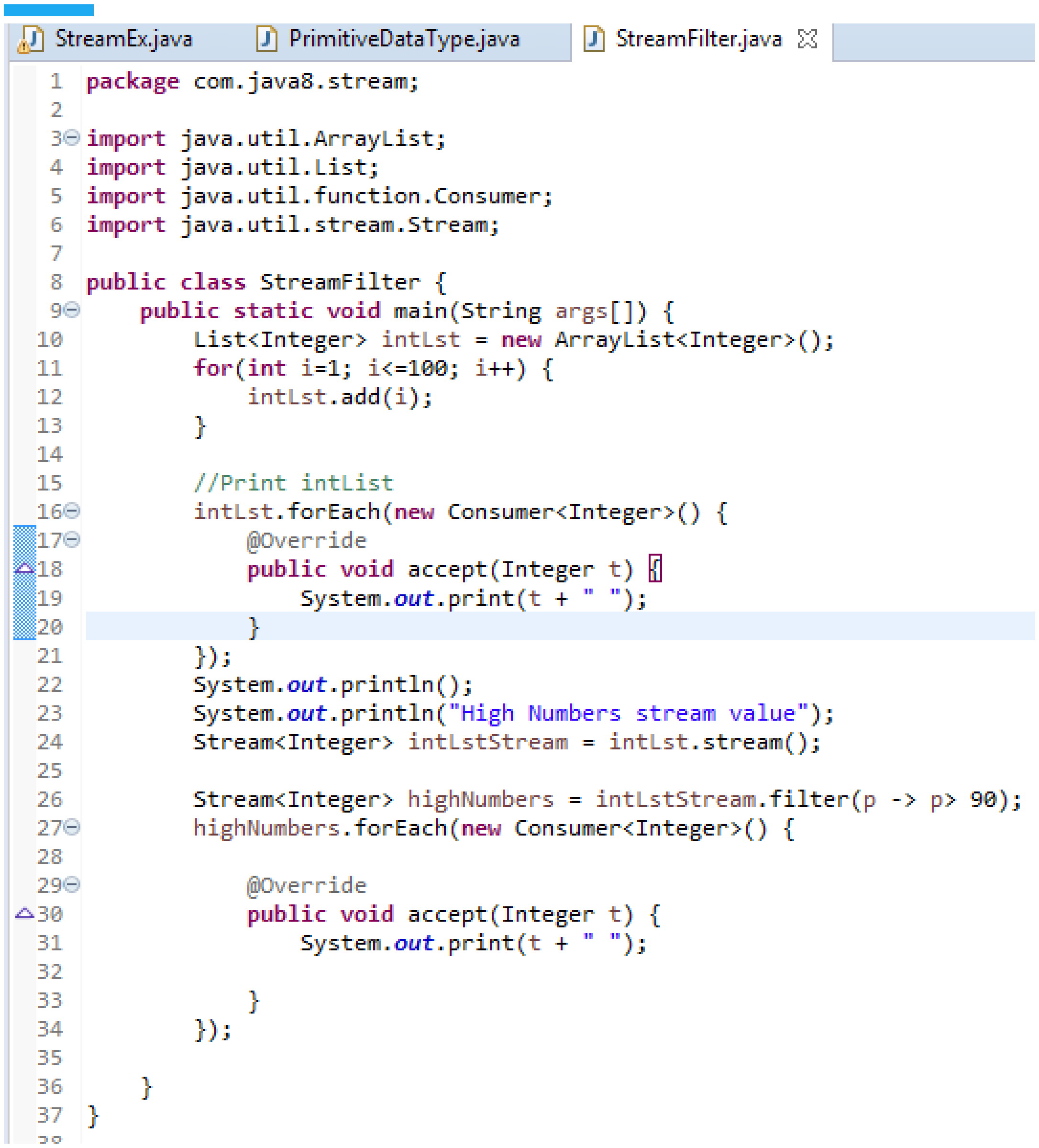
```java
package com.java8.stream;

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

public class StreamEx {
    public static void main(String args[]) {
        List<String> lst = new ArrayList<String>();
        Stream<String> lstStream = null;
        lstStream = getStream(lst);
    }

    public static Stream<String> getStream(List<String> lst) {
        if (lst == null) {
            // Empty Stream
            Stream<String> lstStream = Stream.empty();
            return lstStream;
        } else {
            // Sequentional Stream
            Stream<String> lstStream = lst.stream();
            return lstStream;

            // Parallel Stream
            // Stream<String> lstStreamParallel = lst.parallelStream();
            // return lstStreamParallel;

        }
    }
}
```

- ➢ Java8 offers int,long and double primitive data types streams. Refers the following example for reference.



```java
package com.java8.stream;

import java.util.function.DoubleConsumer;
import java.util.function.IntConsumer;
import java.util.stream.DoubleStream;
import java.util.stream.IntStream;

public class PrimitiveDataType {
    public static void main(String args[]) {
        //int datatype stream
        IntStream intStreamVar = IntStream.range(1, 11);
        intStreamVar.forEach(new IntConsumer() {
            @Override
            public void accept(int value) {
                System.out.println("Int Primitive Datatype Stream Value is: " + value);
            }
        });
        //double datatype stream
        DoubleStream doubleStreamVar = DoubleStream.of(1,2,3,4,5,6,7,8,9,10);
        doubleStreamVar.forEach(new DoubleConsumer() {
            @Override
            public void accept(double value) {
                System.out.println("Double Primitive Datatype Stream value is: " + value);
            }
        });
        //Similary long Stream is also available.
    }
}
```

- Java8 offers to create stream of file.
- Java8 offers to create stream form String or array.
- We can also convert stream to collection.
- Java Stream also offers to extract elements from the collection baased on filter. We can use stream.filter() method to extract results as per our requirement. Following is an example of filter output of collection.



```java
package com.java8.stream;

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

public class StreamFilter {
    public static void main(String args[]) {
        List<Integer> intLst = new ArrayList<Integer>();
        for(int i=1; i<=100; i++) {
            intLst.add(i);
        }

        //Print intList
        intLst.forEach(new Consumer<Integer>() {
            @Override
            public void accept(Integer t) {
                System.out.print(t + " ");
            }
        });
        System.out.println();
        System.out.println("High Numbers stream value");
        Stream<Integer> intLstStream = intLst.stream();

        Stream<Integer> highNumbers = intLstStream.filter(p -> p> 90);
        highNumbers.forEach(new Consumer<Integer>() {

            @Override
            public void accept(Integer t) {
                System.out.print(t + " ");

            }
        });

    }
}
```

Limitations of Java Stream API:

- ➢ There are a lot of methods in Stream API and the most confusing part is the overloaded methods. It makes the learning curve time taking.
- ➢ Once a Stream is consumed, it can't be used later on. As you can see in above examples that every time I am creating a stream.

Reference: https://www.journaldev.com/2774/java-8-stream

Reference: https://www.baeldung.com/java-8-streams

Java Time API

- **Why we need java time api:**
- Existing Java Date classes are not defined consistency.
- java.util.Date contains both date and time.
- java.sql.Date contains only date.
- Again formatting and parsing classes are defined in java.text package.
- There are no clearly defined classes for date,timestamp and formatting.
- All the date classes are mutable so they are not thread safe. It is one of the biggest problem in existing date and time.
- Existing date and time classes does not provide internationalization. There is no timezone support exist. Jada Time play key role as quality replacement for support time zone in application.
- **Java Date and Time API**
- **Immutable Classes:** All the java date and time api classes are immutable. It means thread safe.
- **Separation:** It provides different classes for date,time, date and time, timestamp, timezone etc..
- **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.
- **Extendable**: The new Date Time API works on ISO-8601 calendar system but we can use it with other non ISO calendars as well.
- https://o7planning.org/13725/java-temporal
- https://javarevisited.blogspot.com/2015/03/20-examples-of-date-and-time-api-from-Java8.html?m=1

What are the predicates in java 8?

Reference: https://www.scaler.com/topics/predicate-in-java-8/