Topics Included

Java Util Function

- Java Predicate Example
- Java Supplier Example
- Java Custom Functional Interface
- Function Examples
- UnaryOperator and BinaryOperator Example
- BiConsumer, BiFunction and BiPredicate Example
- Consumer Interface and forEach Loop
- Function.apply
- Java DoubleSupplier Example
- Java LongSupplier Example
- Java IntSupplier Example
- Java BooleanSupplier Example
- Java Consumer Example

Lambda expressions

Lamda Expression help us to implement SAM interface and we can write code in functional style

What is SAM : Single Abstract Method

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

By Bhanu Pratap Singh

Lambda expressions

- Java lambda expression is used to provide the implementation of functional interface.
  Function interface means Interface with one method.
- Lambda expression saves lot of code lines.
- Lambda expressions are very effective when we have to iterate, filter and extract data from collection.

By Bhanu Pratap Singh

Lambda expressions Syntax

```
() -> { }
(p) -> {  }
(p1,p2) -> {  }

{} = body of lambda expression
() = argument to lambda expression
```

By Bhanu Pratap Singh

Lambda expressions Example

```
package lamda;

@FunctionalInterface
public interface FunctionalInterfaceExample2 {

void sayMessage(String message);
}
```

```
FunctionalInterfaceExample2 message1 = (a) -> {
a.toUpperCase();
};


FunctionalInterfaceExample2 message2 = (a) -> {
System.out.println(a.toUpperCase());
};
```

By Bhanu Pratap Singh

Lambda expressions Example

```
FunctionalInterfaceExample2 message = (a) -> { a.toUpperCase() };

FunctionalInterfaceExample2 message = (a) -> {
System.out.println(a.toUpperCase())};
```

```
message = (a) -> a.toUpperCase();

message = (a) -> System.out.println(a.toUpperCase());
```

By Bhanu Pratap Singh

Lambda expressions Example

```
package lamda;


@FunctionalInterface
public interface FunctionalInterfaceExample {

int operation(int a, int b);
}
```

By Bhanu Pratap Singh

Lambda expressions Example

```
FunctionalInterfaceExample add1 = (a, b) -> {
return a + b;
};

FunctionalInterfaceExample myltiply1 = (a, b) -> {
return a * b;
};

FunctionalInterfaceExample substract1 = (a, b) -> {
return a - b;
};
```

By Bhanu Pratap Singh

Lambda expressions Example

```
FunctionalInterfaceExample add = (a, b) -> a + b;

FunctionalInterfaceExample myltiply = (a, b) -> a * b;

FunctionalInterfaceExample substract = (a, b) -> a - b;
```

By Bhanu Pratap Singh

Lambda expressions Example

```
package function;

import java.util.function.Function;

public class UpperCaseFunction {

    Function<String, String> upperCase = (name) -> name.toUpperCase();

    public static void main(String[] args) {

    UpperCaseFunction upperCaseFunction = new UpperCaseFunction();

        System.out.println(upperCaseFunction.upperCase.apply("test"));
    }
}
```

By Bhanu Pratap Singh

Lambda expressions Example

```
package function;
import java.util.function.Function;

public class AddStringFunction {

    static Function<String, String> addString = (name) -> name.toUpperCase().concat("Ram");

    public static void main(String[] args) {

    String value = AddStringFunction.addString.apply("test");
        System.out.println(value); // TESTRam
    }

}
```

By Bhanu Pratap Singh

Lambda expressions Example

```
package function;

import java.util.function.Function;

public class StringLengthFunction {

    static Function<String, Integer> length = (name) -> name.length();

    public static void main(String[] args) {

        int value = StringLengthFunction.length.apply("test");

        System.out.println(value); // 4
    }
}
```

By Bhanu Pratap Singh

Lambda expressions Example

```
package function;

import java.util.function.Function;

    public class StringContainsFunction {

    static Function<String, Boolean> contains = (name) -> name.contains("Test");

    public static void main(String[] args) {
        System.out.println(StringContainsFunction.contains.apply("t")); // false
        System.out.println(StringContainsFunction.contains.apply("Test")); //
        false
    }
}
```

By Bhanu Pratap Singh

Lambda expressions Example

```
package function;

import java.util.function.Function;

public class FindGraterFunction {

    static Function<Integer, Boolean> grater = (age) -> (age > 30);

    public static void main(String[] args) {
        System.out.println(grater.apply(90)); // true
        System.out.println(grater.apply(10)); // false
    }
}
```

By Bhanu Pratap Singh

Lambda expressions Example

```
package function;


import java.util.function.Function;


public class FindPrimeFunction {

    static Function<Integer, Integer> isPrimeNumber = (number) -> (number % 2);
    public static void main(String[] args) {
        System.out.println(isPrimeNumber.apply(90)); //0
        System.out.println(isPrimeNumber.apply(11)); //1

    }
}
```

By Bhanu Pratap Singh

DoubleFunction

```
/**
 * Represents a function that accepts a double-valued argument and produces a result.
 * @param <R> the type of the result of the function
 * @see Function
 * @since 1.8
 */
@FunctionalInterface
public interface DoubleFunction<R> {


    /**
     * Applies this function to the given argument.
     *
     * @param value the function argument
     * @return the function result
     */
    R apply(double value);
}
```

By Bhanu Pratap Singh

DoubleFunction Example

```
package function;

import java.util.function.DoubleFunction;

public class DoubleFunctionExample {

    static DoubleFunction<Integer> doubleFunction = d -> (int)(d * 10);

    public static void main(String[] args) {

        System.out.println(doubleFunction.apply(20.90)); // 290
    }
}
```

By Bhanu Pratap Singh

DoubleToIntFunction

```
/**
 * Represents a function that accepts a double-valued argument and produces
an
 * int-valued result.
 *
 * @see Function
 * @since 1.8
 */
@FunctionalInterface
public interface DoubleToIntFunction {

    /**
     * Applies this function to the given argument.
     *
     * @param value the function argument
     * @return the function result
     */
    int applyAsInt(double value);
}
```

By Bhanu Pratap Singh

DoubleToIntFunction Example

```
package function;

import java.util.function.DoubleToIntFunction;

public class DoubleToIntFunctionExample1 {

    static DoubleToIntFunction doubleToIntFunction = fun -> (int) fun;

    public static void main(String[] args) {

       System.out.println(doubleToIntFunction.applyAsInt(90.47565476534));//
       90
     }
}
```

By Bhanu Pratap Singh

DoubleToLongFunction

```
package java.util.function;
/**
 * Represents a function that accepts a double-valued argument and produces
a
 * long-valued result.
 * @see Function
 * @since 1.8
 */
@FunctionalInterface
public interface DoubleToLongFunction {
    /**
     * Applies this function to the given argument.
     *
     * @param value the function argument
     * @return the function result
     */
    long applyAsLong(double value);
}
```

By Bhanu Pratap Singh

DoubleToLongFunction Example

```
package function;

import java.util.function.DoubleToLongFunction;

public class DoubleToLongFunctionExample {

    // long rang -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

    static DoubleToLongFunction doubleToLongFunction = fun -> (long) fun;

    public static void main(String[] args) {
        System.out.println(doubleToLongFunction.applyAsLong(98744434333232233232.647487
        0898666));
        // 9223372036854775807
        }
}
```

By Bhanu Pratap Singh

IntFunction

```
/**
 * Represents a function that accepts an int-valued argument and produces a
 * result.
 * @param <R> the type of the result of the function
 * @see Function
 * @since 1.8
 */
@FunctionalInterface
public interface IntFunction<R> {

    /**
     * Applies this function to the given argument.
     *
     * @param value the function argument
     * @return the function result
     */
    R apply(int value);
}
```

By Bhanu Pratap Singh

IntFunction Example

```java
package function;

import java.util.function.IntFunction;

public class IntFunctionExample {

    static IntFunction<Integer> intFunction = fun -> fun * fun;

    public static void main(String[] args) {

        System.out.println(intFunction.apply(90));// 8100
    }
}
```

By Bhanu Pratap Singh

IntToDoubleFunction

```java
package java.util.function;

/**
 * Represents a function that accepts an int-valued argument and produces a
 * double-valued result
 * @see Function
 * @since 1.8
 */
@FunctionalInterface
public interface IntToDoubleFunction {

    /**
     * Applies this function to the given argument.
     *
     * @param value the function argument
     * @return the function result
     */
    double applyAsDouble(int value);
}
```

By Bhanu Pratap Singh

IntToDoubleFunction Example

```java
package function;

import java.util.function.IntToDoubleFunction;

public class IntToDoubleFunctionExample {

    static IntToDoubleFunction intToDoubleFunction = fun -> fun * fun;

    public static void main(String[] args) {
        System.out.println(intToDoubleFunction.applyAsDouble(7648));//
        5.8491904E7
        System.out.println(intToDoubleFunction.applyAsDouble(7));// 49.0
    }
}
```

By Bhanu Pratap Singh

LongFunction

```java
package java.util.function;

/**
 * Represents a function that accepts a long-valued argument and produces a
 * result.
 * @param <R> the type of the result of the function
 * @see Function
 * @since 1.8
 */
@FunctionalInterface
public interface LongFunction<R> {
    /**
     * Applies this function to the given argument.
     *
     * @param value the function argument
     * @return the function result
     */
    R apply(long value);
}
```

By Bhanu Pratap Singh

LongFunction Example

```java
package function;

import java.util.function.LongFunction;

public class LongFunctionExample {

    static LongFunction<String> longFunction = fun -> String.valueOf(fun * 2);

    public static void main(String[] args) {
        String value = longFunction.apply(567544l);
        System.out.println(value);// 1135088
    }
}
```

By Bhanu Pratap Singh

LongToDoubleFunction

```java
package java.util.function;
/**
 * Represents a function that accepts a long-valued argument and produces a
 * double-valued result.
 * @see Function
 * @since 1.8
 */
@FunctionalInterface
public interface LongToDoubleFunction {
    /**
     * Applies this function to the given argument.
     *
     * @param value the function argument
     * @return the function result
     */
    double applyAsDouble(long value);
}
```

By Bhanu Pratap Singh

LongToDoubleFunction Example

```
package function;

import java.util.function.LongToDoubleFunction;

public class LongToDoubleFunctionExample {

    static LongToDoubleFunction longToDoubleFunction = fun -> fun/9.0;
    public static void main(String[] args) {
        System.out.println(longToDoubleFunction.applyAsDouble(Long.MAX_VALUE));//
        1.0248191152060861 4E18
        System.out.println(longToDoubleFunction.applyAsDouble(71l));//7.888888888888889
    }
}
```

By Bhanu Pratap Singh

LongToIntFunction

```
package java.util.function;

/**
 * Represents a function that accepts a long-valued argument and produces an
 * int-valued result.
 * @see Function
 * @since 1.8
 */
@FunctionalInterface
public interface LongToIntFunction {
    /**
     * Applies this function to the given argument.
     *
     * @param value the function argument
     * @return the function result
     */
    int applyAsInt(long value);
}
```

By Bhanu Pratap Singh

LongToIntFunction Example

```java
package function;

import java.util.function.LongToIntFunction;

public class LongToIntFunctionExample {

    static LongToIntFunction longToIntFunction = fun -> (int) (fun);
    public static void main(String[] args) {
        System.out.println(longToIntFunction.applyAsInt(440L)); //440
    }
}
```

By Bhanu Pratap Singh

ToDoubleBiFunction

```java
package java.util.function;
/**
 * Represents a function that accepts two arguments and produces a double-
valued
 * result.
 * @param <T> the type of the first argument to the function
 * @param <U> the type of the second argument to the function
 * @see BiFunction
 * @since 1.8
 */
@FunctionalInterface
public interface ToDoubleBiFunction<T, U> {
    /**
     * Applies this function to the given arguments.
     * @param t the first function argument
     * @param u the second function argument
     * @return the function result
     */
    double applyAsDouble(T t, U u);
}
```

By Bhanu Pratap Singh

ToDoubleBiFunction Example

```
package function;

import java.util.function.ToDoubleBiFunction;

public class ToDoubleBiFunctionExample {

    static ToDoubleBiFunction<Integer, Double> toDoubleBiFunction = (fun1, fun2) -> fun1 + fun2;
    static ToDoubleBiFunction<Integer, Integer> toDoubleBiFunction1 = (fun1, fun2) -> fun1*fun2;
    public static void main(String[] args) {
        System.out.println(toDoubleBiFunction.applyAsDouble(20, 70.678686));
        System.out.println(toDoubleBiFunction1.applyAsDouble(20, 70));
   }
}
90.678686
1400.0
```

By Bhanu Pratap Singh

ToDoubleFunction

```
package java.util.function;
/**
 * Represents a function that produces a double-valued result.
 * @param <T> the type of the input to the function
 * @see Function
 * @since 1.8
 */
@FunctionalInterface
public interface ToDoubleFunction<T> {
    /**
     * Applies this function to the given argument.
     *
     * @param value the function argument
     * @return the function result
     */
    double applyAsDouble(T value);
}
```

By Bhanu Pratap Singh

ToDoubleFunction Example

```
package function;

import java.util.function.ToDoubleFunction;

public class ToDoubleFunctionExample {

    static ToDoubleFunction<Integer> toDoubleFunction = fun1 -> fun1 * 301;

    public static void main(String[] args) {
        System.out.println(toDoubleFunction.applyAsDouble(90000)); // 2.709E7
        System.out.println(toDoubleFunction.applyAsDouble(900)); //270900.0
    }
}
```

By Bhanu Pratap Singh

ToIntBiFunction

```
package java.util.function;
/**
 * Represents a function that accepts two arguments and produces an int-
valued result
 * @param <T> the type of the first argument to the function
 * @param <U> the type of the second argument to the function
 * @see BiFunction
 * @since 1.8
 */
@FunctionalInterface
public interface ToIntBiFunction<T, U> {
    /**
     * Applies this function to the given arguments.
     * @param t the first function argument
     * @param u the second function argument
     * @return the function result
     */
    int applyAsInt(T t, U u);
}
```

By Bhanu Pratap Singh

ToIntBiFunction Example

```
package function;

import java.util.function.ToIntBiFunction;

public class ToIntBiFunctionExample {

     static ToIntBiFunction<String,String> toIntBiFunction  = (x,y)-> Integer.parseInt(x) +Integer.parseInt(4);

     public static void main(String[] args) {

          System.out.println(toIntBiFunction.applyAsInt("44","80")); // 124

          }

}
```

By Bhanu Pratap Singh

ToIntFunction

```
package java.util.function;
/**
 * Represents a function that produces an int-valued result.
 * @param <T> the type of the input to the function
 * @see Function
 * @since 1.8
 */
@FunctionalInterface
public interface ToIntFunction<T> {
    /**
     * Applies this function to the given argument.
     * @param value the function argument
     * @return the function result
     */
    int applyAsInt(T value);
}
```

By Bhanu Pratap Singh

ToLongBiFunction

```
package java.util.function;
/**
 * Represents a function that accepts two arguments and produces a long-
valued result.
 * @param <T> the type of the first argument to the function
 * @param <U> the type of the second argument to the function
 * @see BiFunction
 * @since 1.8
 */
@FunctionalInterface
public interface ToLongBiFunction<T, U> {
    /**
     * Applies this function to the given arguments.
     * @param t the first function argument
     * @param u the second function argument
     * @return the function result
     */
    long applyAsLong(T t, U u);
}
```

By Bhanu Pratap Singh

Function Example

```
ToIntFunction<Integer> toIntFunction = fun -> fun * 1000;

ToLongBiFunction<Integer, Integer> toLongBiFunction = (fun1, fun2) -> fun1 - fun2;

ToLongFunction<Integer> toLongFunction = fun1 -> fun1 + 5;
```

By Bhanu Pratap Singh

Consumer

Consumer is interface which came as part of java8. It is present in java.utill package.
It represents a function which takes in one argument and produces a result. However these kind of functions don't return any value.

```
package java.util.function;

import java.util.Objects;

/**
```

```
 * Represents an operation that accepts a single input argument and returns no
 * result.
 * @param <T> the type of the input to the operation
 *
 * @since 1.8
 */
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);

    /**
     * Returns a composed {@code Consumer} that performs, in sequence, this
     * operation followed by the {@code after} operation. If performing either
     * operation throws an exception, it is relayed to the caller of the
     * composed operation.  If performing this operation throws an exception,
     * the {@code after} operation will not be performed.
     *
     * @param after the operation to perform after this operation
     * @return a composed {@code Consumer} that performs in sequence this
     * operation followed by the {@code after} operation
     * @throws NullPointerException if {@code after} is null
     */
    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

```
package consumer;

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

public class ConsumerExample1 {

	public static void main(String[] args) {

		Consumer<Integer> con = p -> System.out.println(p);
```

```
		List<Integer> list = new ArrayList<Integer>();
		list.add(100);
		list.add(200);
		list.add(300);
		list.forEach(con);

	}
}
package consumer;

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

public class ConsumerExample2 {

	public static void main(String[] args) {

		Consumer<Integer> con = p -> System.out.println(p*p);

		List<Integer> list = new ArrayList<Integer>();
		list.add(10);
		list.add(20);
		list.add(30);

		list.forEach(con);

	}
}
package consumer;

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

public class ConsumerExample3 {

	public static void main(String[] args) {

		List<Integer> newList = new ArrayList<Integer>();

		Consumer<Integer> con = p -> {
			newList.add(p * 2);
		};
```

```
		Consumer<Integer> con1 = p -> System.out.println(p);

		List<Integer> list = new ArrayList<Integer>();
		list.add(10);
		list.add(20);
		list.add(30);

		list.forEach(con);

		newList.forEach(con1);

	}
}
package consumer;

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

public class ConsumerExample4 {

	Consumer<Person> nameConsumer = person -> {
		if (person.getName().contains("Ram")) {
			person.setName(person.getName().toUpperCase());
			System.out.println(person);
		}
		else {
			System.out.println(person);
		}
	};

	Consumer<Person> ageConsumer = person -> {
		if (person.getAge() == 20) {
			person.setAge(person.getAge()+20);
			System.out.println(person);
		}
		else {
			System.out.println(person);
		}
	};

	public static void main(String[] args) {

		ConsumerExample4 example4 = new ConsumerExample4();
```

```
		List<Person> list = new ArrayList<Person>();
		list.add(new Person("Ram", 10));
		list.add(new Person("Mohan", 20));
		list.add(new Person("Sohan", 30));

		list.forEach(example4.nameConsumer);

		System.out.println("----------");
		list.forEach(example4.ageConsumer);

	}
}

class Person {

	private String name;
	private int age;

	public Person(String name, int age) {
		super();
		this.name = name;
		this.age = age;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public int getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}

	@Override
	public String toString() {
		return "Person [name=" + name + ", age=" + age + "]";
	}
```

```
}
package consumer;

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

public class ConsumerExample5 {

	Consumer<List<Integer>> consumer = (list) -> {
		for (Integer li : list) {
			System.out.println(li);
		}
	};

	public static void main(String[] args) {

		ConsumerExample5 example4 = new ConsumerExample5();

		List<Integer> list = new ArrayList<Integer>();
		list.add(10);
		list.add(20);
		list.add(30);

		example4.consumer.accept(list);

	}

}
package consumer;

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

public class ConsumerExample6 {

	Consumer<List<Employee>> consumer = (list) -> {
		for (Employee li : list) {
			if (li.getAge() > 19) {
				System.out.println(li);
			}
		}
	};
```

```
	public static void main(String[] args) {

		ConsumerExample6 example4 = new ConsumerExample6();

		List<Employee> list = new ArrayList<Employee>();
		list.add(new Employee("Test1", 10));
		list.add(new Employee("Test1", 20));
		list.add(new Employee("Test1", 30));

		example4.consumer.accept(list);

	}
}

class Employee {

	private String name;
	private int age;

	public Employee(String name, int age) {
		super();
		this.name = name;
		this.age = age;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public int getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}

	@Override
	public String toString() {
		return "Person [name=" + name + ", age=" + age + "]";
```

```
	}
}
package consumer;

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

public class ConsumerExample7 {

	static List<Integer> odd = new ArrayList<>();
	static List<Integer> even = new ArrayList<>();

	static Consumer<Integer> number = n -> {
		if (n % 2 == 0) {
			even.add(n);
		} else {
			odd.add(n);
		}
	};

	public static void main(String[] args) {

		Consumer<List<Integer>> printList = list -> list.forEach(n ->
System.out.println(n));

		number.accept(10);
		number.accept(15);
		number.accept(25);
		number.accept(30);

		printList.accept(odd);

		printList.accept(even);
	}
}
package consumer;

import java.util.function.Consumer;

public class ConsumerExample8 {

	Consumer<Citizen> consumer = c -> {
		if (c.getAge() < 18) {
			System.out.println(c.getName() + " not eligible to vote.");
```

```
                } else {
                        System.out.println(c.getName() + " eligible to vote.");
                }
        };


        public static void main(String[] args) {
                ConsumerExample8 electionConsumer = new ConsumerExample8();

                electionConsumer.consumer.accept(new Citizen("Ram", 15));

                electionConsumer.consumer.accept(new Citizen("Mohan", 20));
        }
}

class Citizen {
        private String name;
        private int age;

        public Citizen(String name, int age) {
                this.name = name;
                this.age = age;
        }

        public String getName() {
                return name;
        }

        public int getAge() {
                return age;
        }
}

package consumer;

import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class ConsumerAndThen1 {

        public static void main(String[] args) {

                List<Integer> list = Arrays.asList(300, 400, 500, 600);
```

```
		Consumer<List<Integer>> consumer = listConsumer -> {
			for (int i = 0; i < list.size(); i++) {
				list.set(i, list.get(i) * list.get(i));
			}
		};

		Consumer<List<Integer>> printConsumer = listConsumer -> list.forEach(n ->
System.out.println(n));

		consumer.andThen(printConsumer).accept(list);
	}
}
package consumer;

import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class ConsumerAndThen2 {

	public static void main(String[] args) {

		MyNumber myNumber = new MyNumber();

		List<Integer> list = Arrays.asList(8, 7, 12, 17, 14, 13);

		Consumer<List<Integer>> oddNumConsumer = myNumber::printOddNum;

		Consumer<List<Integer>> evenNumConsumer = myNumber::printEvenNum;

		Consumer<List<Integer>> doneConsumer = myNumber::done;


	oddNumConsumer.andThen(evenNumConsumer).andThen(doneConsumer).accept(li
st);
	}
}

class MyNumber {
	void printOddNum(List<Integer> myNumbers) {
		System.out.println("--printOddNum--");
		myNumbers.forEach(n -> {
			if (n % 2 == 1) {
				System.out.println(n + " ");
			}
```

```
		});
	}

	void printEvenNum(List<Integer> myNumbers) {
		System.out.println("--printEvenNum--");
		myNumbers.forEach(n -> {
			if (n % 2 == 0) {
				System.out.println(n + " ");
			}
		});
	}

	void done(List<Integer> myNumbers) {
		System.out.println("processign done");
	}
}
```

```
package consumer;

import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class ConsumerAndThen4 {

	public static void main(String[] args) {

		List<Integer> list = Arrays.asList(300, 400, 500, 600);

		Consumer<List<Integer>> printConsumer =
				listConsumer -> list.forEach(System.out::println);

		printConsumer.accept(list);
	}
}
```

```
package consumer;

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

public class ConsumerAndThen5 {

	public static void main(String[] args) {
```

```
		List<Crona> list = new ArrayList<Crona>();
		list.add(new Crona("US", 1244302));
		list.add(new Crona("India", 50000));

		Consumer<List<Crona>> printCountryConsumer = listConsumer ->
list.forEach(Crona::getCountry);

		Consumer<List<Crona>> printCasesConsumer = listConsumer ->
list.forEach(Crona::getTotalCases);

		printCountryConsumer.andThen(printCasesConsumer).accept(list);

	}
}

class Crona {

	String country;
	Integer totalCases;

	public Crona(String country, Integer totalCases) {
		super();
		this.country = country;
		this.totalCases = totalCases;
	}

	public String getCountry() {
		System.out.println(country);
		return country;
	}

	public void setCountry(String country) {
		this.country = country;
	}

	public Integer getTotalCases() {
		System.out.println(totalCases);
		return totalCases;
	}

	public void setTotalCases(Integer totalCases) {
		this.totalCases = totalCases;
	}
```

```
}
package consumer;

import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class ConsumerAndThen6 {

	static Consumer<List<Integer>> consumer1 = new Consumer<List<Integer>>() {
		@Override
		public void accept(List<Integer> t) {
			System.out.println("in consumer1");
			for (int i = 0; i < t.size(); i++) {
				if (t.get(i) % 2 == 0) {
					System.out.println(t.get(i));
				}
			}

		}
	};

	static Consumer<Integer> consumer2 = (Integer x) -> System.out.println(x);

	public static void main(String[] args) {

		List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 8, 10);

		forEach(list, consumer2);
		System.out.println("==consumer2 ends==");
		forEach(list, (Integer x) -> System.out.println(x));

		consumer1.accept(list);

	}

	static <T> void forEach(List<T> list, Consumer<T> consumer) {
		for (T t : list) {
			consumer.accept(t);
		}
	}

}

package java.util.function;
```

```java
import java.util.Objects;

/**
 * Represents an operation that accepts two input arguments and returns no
 * result.
* @param <T> the type of the first argument to the operation
 * @param <U> the type of the second argument to the operation
 * @see Consumer
 * @since 1.8
 */
@FunctionalInterface
public interface BiConsumer<T, U> {

    /**
     * Performs this operation on the given arguments.
     *
     * @param t the first input argument
     * @param u the second input argument
     */
    void accept(T t, U u);

    /**
     * Returns a composed {@code BiConsumer} that performs, in sequence, this
     * operation followed by the {@code after} operation. If performing either
     * operation throws an exception, it is relayed to the caller of the
     * composed operation.  If performing this operation throws an exception,
     * the {@code after} operation will not be performed.
     *
     * @param after the operation to perform after this operation
     * @return a composed {@code BiConsumer} that performs in sequence this
     * operation followed by the {@code after} operation
     * @throws NullPointerException if {@code after} is null
     */
    default BiConsumer<T, U> andThen(BiConsumer<? super T, ? super U> after) {
        Objects.requireNonNull(after);

        return (l, r) -> {
            accept(l, r);
            after.accept(l, r);
        };
    }
}
```

```java
package biConsumer;
```

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

public class BiConsumerExample3 {

	static Map<Integer, String> newMap = new HashMap<Integer, String>();

	static BiConsumer<Map<Integer, String>, Map<Integer, String>> mapConsumer =
(map1, map2) -> {
		for (Map.Entry<Integer, String> map : map1.entrySet()) {
			map2.put(map.getKey() * 3, map.getValue());
		}
	};

	public static void main(String args[]) {
		Map<Integer, String> map = new HashMap<>();

		map.put(1, "Java");
		map.put(2, "Go");
		map.put(3, "C");
		map.put(4, "JavaScript");
		map.put(5, "C++");

		mapConsumer.accept(map, newMap);

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

```
package biConsumer;

import java.util.ArrayList;
import java.util.List;
import java.util.function.BiConsumer;

public class BiConsumerExample5 {

	public static void main(String args[]) {

		List<Integer> list1 = new ArrayList<Integer>();
		list1.add(2);
		list1.add(1);
		list1.add(3);
```

```
		list1.add(5);

		List<Integer> list2 = new ArrayList<Integer>();
		list2.add(2);
		list2.add(1);
		list2.add(4);

		BiConsumer<List<Integer>, List<Integer>> compareBiConsumer = (li1, li2) -> {
			for (int i = 0; i < li1.size(); i++) {
				if (li1.get(i) != li2.get(i)) {
					System.out.println(li1.get(i) + " not equal");
				} else {
					System.out.println(li1.get(i) + " equal");
				}
			}

		};

		BiConsumer<List<Integer>, List<Integer>> display = (li1, li2) -> {
			li1.forEach(a -> System.out.print(a + " "));
			System.out.println();
			li2.forEach(a -> System.out.print(a + " "));
			System.out.println();
		};

		try {
			compareBiConsumer.andThen(display).accept(list1, list2);
		} catch (Exception e) {
			System.out.println("Exceptiion : " + e);
		}
	}
}
```

BiPredicate

```
package java.util.function;

import java.util.Objects;

/**
 * Represents a predicate (boolean-valued function) of two arguments.  This is
 * the two-arity specialization of {@link Predicate}.
 *
 * <p>This is a <a href="package-summary.html">functional interface</a>
 * whose functional method is {@link #test(Object, Object)}.
 *
 * @param <T> the type of the first argument to the predicate
```

```
 * @param <U> the type of the second argument the predicate
 *
 * @see Predicate
 * @since 1.8
 */
@FunctionalInterface
public interface BiPredicate<T, U> {

    /**
     * Evaluates this predicate on the given arguments.
     *
     * @param t the first input argument
     * @param u the second input argument
     * @return {@code true} if the input arguments match the predicate,
     * otherwise {@code false}
     */
    boolean test(T t, U u);

    /**
     * Returns a composed predicate that represents a short-circuiting logical
     * AND of this predicate and another.  When evaluating the composed
     * predicate, if this predicate is {@code false}, then the {@code other}
     * predicate is not evaluated.
     *
     * <p>Any exceptions thrown during evaluation of either predicate are relayed
     * to the caller; if evaluation of this predicate throws an exception, the
     * {@code other} predicate will not be evaluated.
     *
     * @param other a predicate that will be logically-ANDed with this
     *              predicate
     * @return a composed predicate that represents the short-circuiting logical
     * AND of this predicate and the {@code other} predicate
     * @throws NullPointerException if other is null
     */
    default BiPredicate<T, U> and(BiPredicate<? super T, ? super U> other) {
        Objects.requireNonNull(other);
        return (T t, U u) -> test(t, u) && other.test(t, u);
    }

    /**
     * Returns a predicate that represents the logical negation of this
     * predicate.
     *
     * @return a predicate that represents the logical negation of this
     * predicate
```

```
     */
    default BiPredicate<T, U> negate() {
        return (T t, U u) -> !test(t, u);
    }

    /**
     * Returns a composed predicate that represents a short-circuiting logical
     * OR of this predicate and another.  When evaluating the composed
     * predicate, if this predicate is {@code true}, then the {@code other}
     * predicate is not evaluated.
     *
     * <p>Any exceptions thrown during evaluation of either predicate are relayed
     * to the caller; if evaluation of this predicate throws an exception, the
     * {@code other} predicate will not be evaluated.
     *
     * @param other a predicate that will be logically-ORed with this
     *              predicate
     * @return a composed predicate that represents the short-circuiting logical
     * OR of this predicate and the {@code other} predicate
     * @throws NullPointerException if other is null
     */
    default BiPredicate<T, U> or(BiPredicate<? super T, ? super U> other) {
        Objects.requireNonNull(other);
        return (T t, U u) -> test(t, u) || other.test(t, u);
    }
}
```

```
package biPredicate;

import java.util.function.BiPredicate;

public class BiPredicateExample1 {

    public static void main(String[] args) {

        BiPredicate<String, Integer> filter = (x, y) -> {
            return x.length() == y;
        };

        boolean result = filter.test("hellojava", 9);
        System.out.println(result);  // true

        boolean result2 = filter.test("hellojava", 10);
        System.out.println(result2); // false
    }
```

```
}
package biPredicate;

import java.util.Arrays;
import java.util.List;
import java.util.function.BiPredicate;

public class BiPredicateExample2 {

	static BiPredicate<String, Integer> biPredicate = (name, score) -> {
		if (score > 80) {
			System.out.println(name + " is Topper");
			return true;
		}
		System.out.println(name + " is Not Topper");
		return false;
	};

	public static void main(String[] args) {

		List<Student> students = Arrays.asList(new Student("Test1", 90), new
Student("Test2", 94),
				new Student("Test3", 33), new Student("Test4", 2));

		biPredicate.test("Ram", 90);
		biPredicate.test("Mohan", 10);

		for (Student student : students) {
			biPredicate.test(student.getName(), student.getMarks());
		}

	}

}

class Student {

	String name;
	Integer marks;

	public Student(String name, int marks) {
		super();
		this.name = name;
		this.marks = marks;
```

```
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	@Override
	public String toString() {
		return "Student [name=" + name + ", marks=" + marks + "]";
	}

	public Integer getMarks() {
		return marks;
	}

	public void setMarks(int marks) {
		this.marks = marks;
	}

}
package biPredicate;

import java.util.function.BiPredicate;

public class BiPredicateExample3 {

	public static void main(String[] args) {

		BiPredicate<Integer, String> condition = (i, s) -> i > 30 && s.startsWith("R");

		BiPredicate<Integer, String> condition1 = (i, s) -> i < 30 && s.startsWith("T");

		System.out.println(condition.test(20, "Test1"));
		System.out.println(condition.test(30, "RTest2"));
		System.out.println(condition.test(50, "Test3"));
		System.out.println("--------");
		System.out.println(condition.negate().test(20, "Test1"));
		System.out.println(condition.negate().test(30, "RTest2"));
		System.out.println(condition.negate().test(50, "Test3"));

		System.out.println("--------");
```

```
                System.out.println(condition.or(condition1).test(20, "Test1"));
                System.out.println(condition.or(condition1).test(30, "RTest2"));
                System.out.println(condition.or(condition1).test(50, "Test3"));

                System.out.println("--------");
                System.out.println(condition.and(condition1).test(20, "Test1"));
                System.out.println(condition.and(condition1).test(30, "RTest2"));
                System.out.println(condition.and(condition1).test(50, "Test3"));

        }
}
}
false
false
false
--------
true
true
true
--------
true
false
false
--------
false
false
false
```

BiFunction

```
package java.util.function;

import java.util.Objects;

/**
 * Represents a function that accepts two arguments and produces a result.
 * This is the two-arity specialization of {@link Function}.
 *
 * <p>This is a <a href="package-summary.html">functional interface</a>
 * whose functional method is {@link #apply(Object, Object)}.
 *
 * @param <T> the type of the first argument to the function
 * @param <U> the type of the second argument to the function
 * @param <R> the type of the result of the function
 *
 * @see Function
 * @since 1.8
```

```
 */
@FunctionalInterface
public interface BiFunction<T, U, R> {

    /**
     * Applies this function to the given arguments.
     *
     * @param t the first function argument
     * @param u the second function argument
     * @return the function result
     */
    R apply(T t, U u);

    /**
     * Returns a composed function that first applies this function to
     * its input, and then applies the {@code after} function to the result.
     * If evaluation of either function throws an exception, it is relayed to
     * the caller of the composed function.
     *
     * @param <V> the type of output of the {@code after} function, and of the
     *           composed function
     * @param after the function to apply after this function is applied
     * @return a composed function that first applies this function and then
     * applies the {@code after} function
     * @throws NullPointerException if after is null
     */
    default <V> BiFunction<T, U, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t, U u) -> after.apply(apply(t, u));
    }
}

package biFunction;

import java.util.function.BiFunction;
import java.util.function.Function;

public class Example1 {

        static BiFunction<Integer, Integer, Integer> addFunction = (num1, num2) -> (num1 +
num2);

        static Function<Integer, Integer> multipleFunction = (num1) -> (num1 * 5);

        public static void main(String[] args) {
```

```
		System.out.println(addFunction.apply(10, 25));

		Integer status = addFunction.andThen(multipleFunction).apply(10, 25);
		System.out.println(status);
		status = addFunction.andThen(result -> result * 5).andThen(result -> result %
2).apply(10, 25);
		System.out.println(status);
	}
}
```

```
package biFunction;

import java.util.function.BiFunction;

public class Example2 {

	public static void main(String args[]) {
		BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;

		try {
			add = add.andThen(null);
			System.out.println("Add = " + add.apply(2, 3));
		} catch (Exception e) {
			System.out.println("Exception: " + e);
		}
	}

}
```

```
package biFunction;

import java.util.function.BiFunction;

public class Example3 {

	public static void main(String[] args) {

		BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;

		add = add.andThen(a -> a / 0);

		try {
			System.out.println("add = " + add.apply(2, 3));
		} catch (Exception e) {
```

```
                System.out.println("Exception: " + e);
            }
        }
}
```

BiPredicate

```
package java.util.function;

import java.util.Objects;

/**
 * Represents a predicate (boolean-valued function) of two arguments.  This is
 * the two-arity specialization of {@link Predicate}.*
 * @param <T> the type of the first argument to the predicate
 * @param <U> the type of the second argument the predicate
 *
 * @see Predicate
 * @since 1.8
 */
@FunctionalInterface
public interface BiPredicate<T, U> {

    /**
     * Evaluates this predicate on the given arguments.
     *
     * @param t the first input argument
     * @param u the second input argument
     * @return {@code true} if the input arguments match the predicate,
     * otherwise {@code false}
     */
    boolean test(T t, U u);

    /**
     * Returns a composed predicate that represents a short-circuiting logical
     * AND of this predicate and another.  When evaluating the composed
     * predicate, if this predicate is {@code false}, then the {@code other}
     * predicate is not evaluated.
     *
     * <p>Any exceptions thrown during evaluation of either predicate are relayed
     * to the caller; if evaluation of this predicate throws an exception, the
     * {@code other} predicate will not be evaluated.
     *
     * @param other a predicate that will be logically-ANDed with this
     *              predicate
     * @return a composed predicate that represents the short-circuiting logical
     * AND of this predicate and the {@code other} predicate
```

```
     * @throws NullPointerException if other is null
     */
    default BiPredicate<T, U> and(BiPredicate<? super T, ? super U> other) {
        Objects.requireNonNull(other);
        return (T t, U u) -> test(t, u) && other.test(t, u);
    }

    /**
     * Returns a predicate that represents the logical negation of this
     * predicate.
     *
     * @return a predicate that represents the logical negation of this
     * predicate
     */
    default BiPredicate<T, U> negate() {
        return (T t, U u) -> !test(t, u);
    }

    /**
     * Returns a composed predicate that represents a short-circuiting logical
     * OR of this predicate and another.  When evaluating the composed
     * predicate, if this predicate is {@code true}, then the {@code other}
     * predicate is not evaluated.
     *
     * <p>Any exceptions thrown during evaluation of either predicate are relayed
     * to the caller; if evaluation of this predicate throws an exception, the
     * {@code other} predicate will not be evaluated.
     *
     * @param other a predicate that will be logically-ORed with this
     *              predicate
     * @return a composed predicate that represents the short-circuiting logical
     * OR of this predicate and the {@code other} predicate
     * @throws NullPointerException if other is null
     */
    default BiPredicate<T, U> or(BiPredicate<? super T, ? super U> other) {
        Objects.requireNonNull(other);
        return (T t, U u) -> test(t, u) || other.test(t, u);
    }
}
package biPredicate1;

import java.util.function.BiPredicate;

/**
 * https://www.youtube.com/user/MrBhanupratap29/playlists?
```

```
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example1 {

	static BiPredicate<String, Integer> biPredicate = (a, b) -> {
		return a.length() == b;
	};

	public static void main(String[] args) {
		System.out.println(biPredicate.test("hellojava8", 10));
		System.out.println(biPredicate.test("hellojava8", 9));

	}

}
package biPredicate1;

import java.util.Arrays;
import java.util.List;
import java.util.function.BiPredicate;

/**
 * https://www.youtube.com/user/MrBhanupratap29/playlists?
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example2 {

	static BiPredicate<String, Integer> biPredicate = (name, score) -> {
		if (score > 80) {
			System.out.println(name + " is Topper");
			return true;
		} else {
			System.out.println(name + " is Not Topper");
			return false;
		}
	};
```

```
	public static void main(String[] args) {
		List<Student> students = Arrays.asList(new Student("Test1", 90), new
Student("Test2", 94),
				new Student("Test3", 33), new Student("Test4", 2));

		biPredicate.test("Ram", 90);
		biPredicate.test("Mohan", 34);

		for (Student student : students) {
			biPredicate.test(student.getName(), student.getScore());
		}
	}
}

class Student {

	private String name;
	private int score;

	public Student(String name, int score) {
		super();
		this.name = name;
		this.score = score;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public int getScore() {
		return score;
	}

	public void setScore(int score) {
		this.score = score;
	}

	@Override
	public String toString() {
		return "Student [name=" + name + ", score=" + score + "]";
	}
```

```
}
package biPredicate1;

import java.util.function.BiPredicate;

/**
 * https://www.youtube.com/user/MrBhanupratap29/playlists?
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example3 {



	public static void main(String[] args) {
		BiPredicate<Integer, String> conditions = (a, b) -> a > 30 &&
b.startsWith("R");
		BiPredicate<Integer, String> conditions1 = (a, b) -> a < 30 &&
b.startsWith("T");

		System.out.println(conditions.test(20, "Test1"));
		System.out.println(conditions.test(30, "RTest2"));
		System.out.println(conditions.test(50, "Test3"));
		System.out.println("------");
		System.out.println(conditions.negate().test(20, "Test1"));
		System.out.println(conditions.negate().test(30, "RTest2"));
		System.out.println(conditions.negate().test(50, "Test3"));

		System.out.println("------");
		System.out.println(conditions.or(conditions1).test(20, "Test1"));
		System.out.println(conditions.or(conditions1).test(30, "RTest2"));
		System.out.println(conditions.or(conditions1).test(50, "Test3"));

		System.out.println("------");
		System.out.println(conditions.and(conditions1).test(20, "Test1"));
		System.out.println(conditions.and(conditions1).test(30, "RTest2"));
		System.out.println(conditions.and(conditions1).test(50, "Test3"));
	}
}
package java.util.function;
```

```
/**
 * Represents a supplier of results.
 *
 * <p>There is no requirement that a new or distinct result be returned each
 * time the supplier is invoked.
 *
 * <p>This is a <a href="package-summary.html">functional interface</a>
 * whose functional method is {@link #get()}.
 *
 * @param <T> the type of results supplied by this supplier
 *
 * @since 1.8
 */
@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
    T get();
}
package supplier;

import java.util.function.Supplier;

/**
 * https://www.youtube.com/user/MrBhanupratap29/playlists?
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example1 {

        static double a = 80;
        static double b = 70;

        static Supplier<Double> ds = () -> 345.198;

        static Supplier<Double> ds1 = () -> Double.valueOf(345);

        static Supplier<Float> ds2 = () -> 345.198f;
```

```
	static Supplier<Double> ds3 = () -> Math.random() * 10000;

	static Supplier<Integer> ds4 = () -> "test".length();

	static Supplier<String> ds5 = () -> {
		return "test".toUpperCase().concat("Ram");
	};

	static Supplier<String> ds6 = () -> "test".toUpperCase().concat("Ram");



	public static void main(String[] args) {
		System.out.println(ds.get());
		System.out.println(ds1.get());
		System.out.println(ds1.get());
		System.out.println(ds3.get());
		System.out.println(ds4.get());
		System.out.println(ds5.get());
		System.out.println();
	}
}
package supplier;

import java.util.Random;
import java.util.function.Supplier;
/**
 * https://www.youtube.com/user/MrBhanupratap29/playlists?
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example2 {

	public static void main(String[] args) {

		Example2 example2 = new Example2();

		Random random = new Random();

		Supplier<Double> supplier = random::nextDouble;
```

```
		Supplier<Double> supplier2 = Example2::getDouble1;

		Supplier<Double> supplier3 = example2::getDouble2;

		System.out.println(supplier.get());

		System.out.println(supplier2.get());

		System.out.println(supplier3.get());
	}

	static double getDouble1() {
		return 19.78;
	}

	double getDouble2() {
		return 19.78;
	}
}
```

Comparator

Interface Comparator<T>

- **Type Parameters:**
  T - the type of objects that may be compared by this comparator

All Known Implementing Classes:

Collator, RuleBasedCollator

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

Modifier and Type	Method and Description
int	**compare**(**T** o1, **T** o2) Compares its two arguments for order.
static <T,U extends **Comparable**<? super U>> **Comparator**<T>	**comparing**(**Function**<? super T,? extends U> keyExtractor) Accepts a function that extracts a **Comparable** sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T,U> **Comparator**<T>	**comparing**(**Function**<? super T,? extends U> keyExtractor, **Comparator**<? super U> keyComparator) Accepts a function that extracts a sort key from a type T, and returns a Comparator<T> that compares by that sort key using the specified **Comparator**.

static <T> **Comparator**<T>	**comparingDouble**(**ToDoubleFunction**<? super T> keyExtractor) Accepts a function that extracts a double sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T> **Comparator**<T>	**comparingInt**(**ToIntFunction**<? super T> keyExtractor) Accepts a function that extracts an int sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T> **Comparator**<T>	**comparingLong**(**ToLongFunction**<? super T> keyExtractor) Accepts a function that extracts a long sort key from a type T, and returns a Comparator<T> that compares by that sort key.
boolean	**equals**(**Object** obj) Indicates whether some other object is "equal to" this comparator.
static <T extends **Comparable**<? super T>> **Comparator**<T>	**naturalOrder**() Returns a comparator that compares **Comparable** objects in natural order.
static <T> **Comparator**<T>	**nullsFirst**(**Comparator**<? super T> comparator) Returns a null-friendly comparator that considers null to be less than non-null.
static <T> **Comparator**<T>	**nullsLast**(**Comparator**<? super T> comparator) Returns a null-friendly comparator that considers null to be greater than non-null.
default **Comparator**<**T**>	**reversed**() Returns a comparator that imposes the reverse ordering of this comparator.
static <T extends **Comparable**<? super T>> **Comparator**<T>	**reverseOrder**() Returns a comparator that imposes the reverse of the *natural ordering*.
default **Comparator**<**T**>	**thenComparing**(**Comparator**<? super **T**> other) Returns a lexicographic-order comparator with another comparator.
default <U extends **Comparable**<? super U>> **Comparator**<**T**>	**thenComparing**(**Function**<? super **T**,? extends U> keyExtractor) Returns a lexicographic-order comparator with a function that extracts a Comparable sort key.

default <U> Comparator<T>	thenComparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator) Returns a lexicographic-order comparator with a function that extracts a key to be compared with the given Comparator.
default Comparator<T>	thenComparingDouble(ToDoubleFunction<? super T> keyExtractor) Returns a lexicographic-order comparator with a function that extracts a double sort key.
default Comparator<T>	thenComparingInt(ToIntFunction<? super T> keyExtractor) Returns a lexicographic-order comparator with a function that extracts a int sort key.
default Comparator<T>	thenComparingLong(ToLongFunction<? super T> keyExtractor) Returns a lexicographic-order comparator with a function that extracts a long sort key.

```
package comparator;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

/**
 * https://www.youtube.com/user/MrBhanupratap29/playlists?
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example1 {

	// old way
	static Comparator<Person> comparator = new Comparator<Person>() {

		@Override
		public int compare(Person o1, Person o2) {

			return o1.getFirstName().compareTo(o2.getFirstName());
		}
	};

	static Comparator<Person> nameComparator = (Person o1, Person o2) ->
o1.getFirstName().compareTo(o2.getFirstName());
```

```
	static Comparator<Person> nameComparator1 = (o1, o2) ->
o1.getFirstName().compareTo(o2.getFirstName());

	static Comparator<Person> nameComparator2 = Comparator.comparing(e ->
e.getFirstName());

	static Comparator<Person> comparatorByLastname = Comparator.comparing(e ->
e.getLastName());

	static Comparator<Person> comparatorByWeight = Comparator.comparingDouble(e
-> e.getWeight());

	static Comparator<Person> comparatorByAge = Comparator.comparingInt(e ->
e.getAge());

	static Comparator<Person> comparatorByLong = Comparator.comparingLong(e ->
e.getAge());

	static Comparator<Person> employeeSalaryComparator = Comparator.comparing(e -
> e.getEmployee(),
				(o1, o2) -> o1.getSalary().compareTo(o2.getSalary()));

	static Comparator<Person> employeeSalaryComparator1 =
Comparator.comparingInt(e -> e.getEmployee().getSalary());

	static Comparator<Person> employeeSalaryComparator2 =
Comparator.comparingDouble(e -> e.getEmployee().getSalary());

	static Comparator<Person> employeeSalaryComparator3 =
Comparator.comparingLong(e -> e.getEmployee().getSalary());

	static Comparator<Person> employeeDesignationComparator = Comparator
			.comparing(e -> e.getEmployee().getDesignation());

	static Comparator<Person> employeeDesignationComparator1 =
Comparator.comparing(e -> e.getEmployee(), (o1, o2) -> {
		return o1.getDesignation().compareTo(o2.getDesignation());
	});

	static Comparator<Person> employeeDesignationComparator2 =
				Comparator.comparing(e -> e.getEmployee(), (o1, o2) ->
		o1.getDesignation().compareTo(o2.getDesignation()));

	public static void main(String[] args) {
```

```
		ArrayList<Person> arrayList = new ArrayList<Person>();
		arrayList.add(new Person("CC", "ZZ", 200, 65.9, new Employee(900, "HR")));
		arrayList.add(new Person("AA", "BB", 10, 9.9, new Employee(100,
"Admin")));
		arrayList.add(new Person("BB", "FF", 90, 7.0, new Employee(1000,
"Engineer")));

		Collections.sort(arrayList, nameComparator);
		System.out.println(arrayList);

		Collections.sort(arrayList, (o1, o2) ->
o1.getFirstName().compareTo(o2.getFirstName()));

		Collections.sort(arrayList, employeeSalaryComparator);

		System.out.println(arrayList);

	}

}

class Person {

	private String firstName;

	private Integer age;

	private String lastName;

	private Employee employee;

	private Double weight;

	public Person(String firstName, String lastName, int age, Double weight, Employee
employee) {
		super();
		this.firstName = firstName;
		this.lastName = lastName;
		this.age = age;
		this.weight = weight;
		this.employee = employee;
	}

	public Integer getAge() {
```

```
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}

	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	public Employee getEmployee() {
		return employee;
	}

	public void setEmployee(Employee employee) {
		this.employee = employee;
	}

	public Double getWeight() {
		return weight;
	}

	public void setWeight(Double weight) {
		this.weight = weight;
	}

	@Override
	public String toString() {
		return "Person [firstName=" + firstName + ", age=" + age + ", lastName=" +
lastName + ", employee=" + employee
				+ ", weight=" + weight + "]";
	}
```

```
}

class Employee {

	private Integer salary;
	private String designation;

	public Integer getSalary() {
		return salary;
	}

	public void setSalary(int salary) {
		this.salary = salary;
	}

	public String getDesignation() {
		return designation;
	}

	public void setDesignation(String designation) {
		this.designation = designation;
	}

	public Employee(int salary, String designation) {
		super();
		this.salary = salary;
		this.designation = designation;
	}

	@Override
	public String toString() {
		return "Employee [salary=" + salary + ", designation=" + designation + "]";
	}
}
package comparator;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

/**
 * https://www.youtube.com/user/MrBhanupratap29/playlists?
```

```
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example2 {

	static Comparator<Person4> nameComparator1 = (o1, o2) ->
o1.getFirstName().compareTo(o2.getFirstName());

	static Comparator<Person4> nameComparator2 = Comparator.comparing(e ->
e.getFirstName());

	public static void main(String[] args) {

		ArrayList<Person4> arrayList = new ArrayList<Person4>();

		arrayList.add(new Person4("CC", "ZZ", 200, 65.9, new Employee4(900,
"HR")));
		arrayList.add(new Person4("AA", "BB", 10, 9.9, new Employee4(100,
"Admin")));
		arrayList.add(null);
		arrayList.add(new Person4("BB", "FF", 90, 7.0, new Employee4(1000,
"Engineer")));
		arrayList.add(null);



		Collections.sort(arrayList, nameComparator2);
		for (Person4 str : arrayList) {
			System.out.println(str.getFirstName());
		}
		System.out.println("-------");
		Collections.sort(arrayList, nameComparator2.reversed());

		for (Person4 str : arrayList) {
			System.out.println(str.getFirstName());
		}


		List<String> strList = Arrays.asList("AA", "BB", "CC", "DD");

		Collections.sort(strList, Comparator.reverseOrder());
```

```
                System.out.println(strList);


                Collections.sort(arrayList, Comparator.nullsFirst(nameComparator2));

                System.out.println(arrayList);

                Collections.sort(arrayList, Comparator.nullsLast(nameComparator2));

                System.out.println(arrayList);

                Collections.sort(strList, Comparator.naturalOrder());
                System.out.println(strList);


        }

}

class Person4 {

        private String firstName;

        private Integer age;

        private String lastName;

        private Employee4 Employee4;

        private Double weight;

        Person4(){

        }

        public Person4(String firstName, String lastName, int age, Double weight,
Employee4 Employee4) {
                super();
                this.firstName = firstName;
                this.lastName = lastName;
                this.age = age;
                this.weight = weight;
                this.Employee4 = Employee4;
        }
```

```
	public Integer getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}

	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	public Employee4 getEmployee() {
		return Employee4;
	}

	public void setEmployee(Employee4 Employee4) {
		this.Employee4 = Employee4;
	}

	public Double getWeight() {
		return weight;
	}

	public void setWeight(Double weight) {
		this.weight = weight;
	}

	@Override
	public String toString() {
		return "Person4 [firstName=" + firstName + ", age=" + age + ", lastName=" +
lastName + ", Employee4="
				+ Employee4 + ", weight=" + weight + "]";
```

```
	}

}

class Employee4  {

	private Integer salary;
	private String designation;

	public Integer getSalary() {
		return salary;
	}

	public void setSalary(int salary) {
		this.salary = salary;
	}

	public String getDesignation() {
		return designation;
	}

	public void setDesignation(String designation) {
		this.designation = designation;
	}

	public Employee4(int salary, String designation) {
		super();
		this.salary = salary;
		this.designation = designation;
	}

	@Override
	public String toString() {
		return "Employee4 [salary=" + salary + ", designation=" + designation + "]";
	}

}

package comparator;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

/**
```

```
 * https://www.youtube.com/user/MrBhanupratap29/playlists?
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example3 {

	static Comparator<Person2> ComparatorByFirstName =
Comparator.comparing(Person2::getFirstName);

	static Comparator<Person2> comparatorByLastname =
Comparator.comparing(Person2::getLastName);

	static Comparator<Person2> comparingDouble =
Comparator.comparingDouble(Person2::getWeight);

	static Comparator<Person2> comparatorByAge =
Comparator.comparingInt(Person2::getAge);

	static Comparator<Person2> comparatorByLong = Comparator.comparingLong(e ->
e.getAge());

	static Comparator<Person2> employeeSalaryComparator =
Comparator.comparing(Person2::getEmployee,
			(o1, o2) -> o1.getSalary().compareTo(o2.getSalary()));


	static Comparator<Person2> employeeSalaryComparator1 =
Comparator.comparingInt(e -> e.getEmployee().getSalary());

	static Comparator<Person2> employeeSalaryComparator10 =
Comparator.comparing(Person2::getLastName)
			.thenComparing(Person2::getFirstName);

	static Comparator<Person2> employeeSalaryComparator2 =
Comparator.comparingDouble(e -> e.getEmployee().getSalary());

	static Comparator<Person2> comparingLong = Comparator.comparingLong(e ->
e.getEmployee().getSalary());

	static Comparator<Person2> employeeDesignationComparator = Comparator
			.comparing(e -> e.getEmployee().getDesignation());
```

```
	static Comparator<Person2> employeeDesignationComparator1 =
Comparator.comparing(e -> e.getEmployee(), (o1, o2) -> {
		return o1.getDesignation().compareTo(o2.getDesignation());
	});

	static Comparator<Person2> employeeDesignationComparator2 =
Comparator.comparing(e -> e.getEmployee(),
			(o1, o2) -> o1.getDesignation().compareTo(o2.getDesignation()));

	public static void main(String[] args) {

		ArrayList<Person2> arrayList = new ArrayList<Person2>();

		arrayList.add(new Person2("CC", "BB", 200, 65.9, new Employee2(900,
"HR")));
		arrayList.add(new Person2("AA", "FF", 10, 9.9, new Employee2(100,
"Admin")));
		arrayList.add(new Person2("BB", "CC", 99, 7.0, new Employee2(1000,
"Engineer")));
		arrayList.add(new Person2("BB", "CC", 90, 7.0, new Employee2(1000,
"Engineer")));
		arrayList.add(new Person2("BB", "AA", 91, 7.0, new Employee2(1000,
"Engineer")));

		Collections.sort(arrayList, Comparator.comparing(Person2::getFirstName));

		Collections.sort(arrayList,
Comparator.comparing(Person2::getFirstName).thenComparing(Person2::getAge));


		Comparator<Person2> employeeSalaryComparator1 =
Comparator.comparing(Person2::getEmployee);

	}

}

class Person2 {

	private String firstName;

	private Integer age;

	private String lastName;
```

```
	private Employee2 Employee2;

	private Double weight;

	Person2(){

	}

	public Person2(String firstName, String lastName, int age, Double weight,
Employee2 Employee2) {
		super();
		this.firstName = firstName;
		this.lastName = lastName;
		this.age = age;
		this.weight = weight;
		this.Employee2 = Employee2;
	}

	public Integer getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}

	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	public Employee2 getEmployee() {
		return Employee2;
	}
```

```
	public void setEmployee(Employee2 Employee2) {
		this.Employee2 = Employee2;
	}

	public Double getWeight() {
		return weight;
	}

	public void setWeight(Double weight) {
		this.weight = weight;
	}

	@Override
	public String toString() {
		return "Person2 [firstName=" + firstName + ", age=" + age + ", lastName=" +
lastName + ", Employee2="
				+ Employee2 + ", weight=" + weight + "]";
	}

}

class Employee2 implements Comparable<Employee2>{

	private Integer salary;
	private String designation;

	public Integer getSalary() {
		return salary;
	}

	public void setSalary(int salary) {
		this.salary = salary;
	}

	public String getDesignation() {
		return designation;
	}

	public void setDesignation(String designation) {
		this.designation = designation;
	}

	public Employee2(int salary, String designation) {
		super();
```

```
		this.salary = salary;
		this.designation = designation;
	}

	@Override
	public String toString() {
		return "Employee2 [salary=" + salary + ", designation=" + designation + "]";
	}

	@Override
	public int compareTo(Employee2 o) {
		// TODO Auto-generated method stub
		return designation.compareTo(o.getDesignation());
	}
}
package comparator;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

/**
 * https://www.youtube.com/user/MrBhanupratap29/playlists?
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example4 {

	static Comparator<Person1> ComparatorByFirstName =
Comparator.comparing(Person1::getFirstName);

	static Comparator<Person1> comparatorByLastname =
Comparator.comparing(Person1::getFirstName);

	static Comparator<Person1> comparingDouble =
Comparator.comparingDouble(Person1::getWeight);

	static Comparator<Person1> comparatorByAge =
Comparator.comparingInt(Person1::getAge);

	static Comparator<Person1> comparatorByLong = Comparator.comparingLong(e ->
e.getAge());
```

```
	static Comparator<Person1> ComparatorByFirstName2 =

	Comparator.comparing(Person1::getFirstName).thenComparing(Person1::getLastNa
me);

	static Comparator<Person1> ComparatorByFirstName3 =

	Comparator.comparing(Person1::getFirstName).thenComparing(Person1::getLastNa
me).thenComparing(Person1::getAge);

	static Comparator<Person1> ComparatorByFirstName4 =

	Comparator.comparing(Person1::getEmployee).thenComparing(Person1::getAge).th
enComparing(Person1::getFirstName);


	public static void main(String[] args) {

		ArrayList<Person1> arrayList = new ArrayList<Person1>();

		arrayList.add(new Person1("CC", "BB", 200, 65.9, new Employee1(900,
"HR")));
		arrayList.add(new Person1("AA", "FF", 10, 9.9, new Employee1(100,
"Admin")));
		arrayList.add(new Person1("BB", "CC", 99, 7.0, new Employee1(1000,
"Engineer")));
		arrayList.add(new Person1("BB", "CC", 90, 7.0, new Employee1(1000,
"Engineer")));
		arrayList.add(new Person1("BB", "AA", 91, 7.0, new Employee1(1000,
"Engineer")));

		Collections.sort(arrayList, Comparator.comparing(Person1::getFirstName));

		Collections.sort(arrayList,
Comparator.comparing(Person1::getFirstName).thenComparing(Person1::getAge));


	}

}

class Person1 {

	private String firstName;
```

```
	private Integer age;

	private String lastName;

	private Employee1 Employee1;

	private Double weight;

	public Person1(String firstName, String lastName, int age, Double weight,
Employee1 Employee1) {
		super();
		this.firstName = firstName;
		this.lastName = lastName;
		this.age = age;
		this.weight = weight;
		this.Employee1 = Employee1;
	}

	public Integer getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}

	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	public Employee1 getEmployee() {
		return Employee1;
```

```
	}

	public void setEmployee(Employee1 Employee1) {
		this.Employee1 = Employee1;
	}

	public Double getWeight() {
		return weight;
	}

	public void setWeight(Double weight) {
		this.weight = weight;
	}

	@Override
	public String toString() {
		return "Person1 [firstName=" + firstName + ", age=" + age + ", lastName=" +
lastName + ", Employee1="
				+ Employee1 + ", weight=" + weight + "]";
	}

}

class Employee1 implements Comparable<Employee1>{

	private Integer salary;
	private String designation;

	public Integer getSalary() {
		return salary;
	}

	public void setSalary(int salary) {
		this.salary = salary;
	}

	public String getDesignation() {
		return designation;
	}

	public void setDesignation(String designation) {
		this.designation = designation;
	}

	public Employee1(int salary, String designation) {
```

```
		super();
		this.salary = salary;
		this.designation = designation;
	}

	@Override
	public String toString() {
		return "Employee1 [salary=" + salary + ", designation=" + designation + "]";
	}

	@Override
	public int compareTo(Employee1 o) {
		return designation.compareTo(o.getDesignation());
	}
}
```

BinaryOperator

Interface BinaryOperator<T>

- **Type Parameters:**
  T - the type of the operands and result of the operator

All Superinterfaces:

BiFunction<T,T,T>

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

@FunctionalInterface
public interface **BinaryOperator<T>**
extends BiFunction<T,T,T>
Represents an operation upon two operands of the same type, producing a result of the same type as the operands. This is a specialization of BiFunction for the case where the operands and the result are all of the same type.

This is a functional interface whose functional method is BiFunction.apply(Object, Object).

Since:

1.8

Modifier and Type	Method and Description
static <T> **BinaryOperator**<T>	**maxBy**(**Comparator**<? super T> comparator)

	Returns a **BinaryOperator** which returns the greater of two elements according to the specified Comparator.
static <T> **BinaryOperator**<T>	**minBy**(**Comparator**<? super T> comparator) Returns a **BinaryOperator** which returns the lesser of two elements according to the specified Comparator.

Methods inherited from interface java.util.function.BiFunction
andThen, apply

```
package operator;

import java.util.function.BinaryOperator;
import java.util.function.Function;

/**
 * https://www.youtube.com/user/MrBhanupratap29/playlists?
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class BinaryOperatorExample1 {

	static BinaryOperator<Integer> binaryOperator = (a, b) -> a * b;

	static Function<Integer, Integer> function = a -> a + 2;

	public static void main(String[] args) {
		//System.out.println(binaryOperator.apply(3, 4));

		System.out.println(binaryOperator.andThen(function).apply(3, 4));
	}

}
```

```
package operator;

import java.util.Comparator;
import java.util.function.BinaryOperator;

/**
 * https://www.youtube.com/user/MrBhanupratap29/playlists?
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
```

```
 *
 * @author Bhanu Pratap Singh
 *
 */
public class BinaryOperatorExample2 {

	static Comparator<Integer> comparator = (o1, o2) -> o1.compareTo(o2);

	static Comparator<Integer> comparator1 = (o1, o2) -> o1.compareTo(o2);

	public static void main(String[] args) {
		BinaryOperator<Integer> maxBy = BinaryOperator.maxBy(comparator);

		BinaryOperator<Integer> minBy = BinaryOperator.minBy(comparator);

		System.out.println(maxBy.apply(80, 30));

		System.out.println(minBy.apply(80, 30));
	}

}
```

UnaryOperator

java.util.function

Interface UnaryOperator<T>

- **Type Parameters:**
  T - the type of the operand and result of the operator

All Superinterfaces:

Function<T,T>

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

@FunctionalInterface
public interface **UnaryOperator<T>**
extends Function<T,T>
Represents an operation on a single operand that produces a result of the same type as its operand. This is a specialization of Function for the case where the operand and result are of the same type.

This is a functional interface whose functional method is Function.apply(Object).

Since:

1.8

Modifier and Type	Method and Description
static <T> UnaryOperator<T>	identity() Returns a unary operator that always returns its input argument.

Methods inherited from interface java.util.function.Function
andThen, apply, compose

```
package operator;

import java.util.function.UnaryOperator;

public class UnaryOperatorExample1 {

	// Exclusive Or
	static UnaryOperator<Integer> operator = a -> a ^ 2;

	// Bitwise And
	static UnaryOperator<Integer> operator1 = a -> a & 2;

	public static void main(String[] args) {

		System.out.println(operator.apply(4));

		System.out.println(operator1.apply(4));
	}

}
```

```
package operator;

import java.util.function.Function;
import java.util.function.UnaryOperator;

/**
 * https://www.youtube.com/user/MrBhanupratap29/playlists?
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class UnaryOperatorExample2 {
	// Exclusive Or
```

```
	static UnaryOperator<Integer> operator = a -> a ^ 2;

	// Bitwise And
	static UnaryOperator<Integer> operator1 = a -> a & 2;

	// Exclusive Or
	static UnaryOperator<Integer> operator2 = a -> a * 2;

	// Bitwise And
	static UnaryOperator<Integer> operator13 = a -> a + 2;

	public static void main(String[] args) {

		Function<Integer, Integer> operation = operator.andThen(operator1);

		System.out.println(operation.apply(4));

		System.out.println(operator13.andThen(operator2).apply(4));

		System.out.println(operator13.compose(operator2).apply(4));
	}
}
package operator;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.UnaryOperator;

public class UnaryOperatorExample3 {

	public static void main(String[] args) {
		List<Integer> list = Arrays.asList(1, 3, 5, 6, 9);

		UnaryOperator<Integer> unaryOperator = a -> a * a;

		list.forEach(x -> System.out.println(x));

		List<Integer> newList = newList(unaryOperator, list);
		System.out.println(newList);

	}

	private static List<Integer> newList(UnaryOperator<Integer> unaryOperator,
List<Integer> li) {
```

```
        List<Integer> newList = new ArrayList<Integer>();
        li.forEach(a -> newList.add(unaryOperator.apply(a)));
        return newList;

    }

}
```

Stream

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream −

Sequence of elements − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.

Source − Stream takes Collections, Arrays, or I/O resources as input source.

Aggregate operations − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.

Pipelining − Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.

Automatic iterations − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

Interface Stream<T>

- **Type Parameters:**
  T - the type of the stream elements

All Superinterfaces:

AutoCloseable, BaseStream<T,Stream<T>>

public interface **Stream<T>**
extends BaseStream<T,Stream<T>>
A sequence of elements supporting sequential and parallel aggregate operations. The following example illustrates an aggregate operation using Stream and IntStream:

```
int sum = widgets.stream().filter(w -> w.getColor() == RED)

.mapToInt(w -> w.getWeight()).sum();
```

Modifier and Type	Method and Description
boolean	allMatch(Predicate<? super T> predicate)

	Returns whether all elements of this stream match the provided predicate.
boolean	**anyMatch**(**Predicate**<? super T> predicate) Returns whether any elements of this stream match the provided predicate.
static <T> **Stream.Builder**<T>	**builder**() Returns a builder for a Stream.
<R,A> R	**collect**(**Collector**<? super T,A,R> collector) Performs a **mutable reduction** operation on the elements of this stream using a Collector.
<R> R	**collect**(**Supplier**<R> supplier, **BiConsumer**<R,? super T> accumulator, **BiConsumer**<R,R> combiner) Performs a **mutable reduction** operation on the elements of this stream.
static <T> **Stream**<T>	**concat**(**Stream**<? extends T> a, **Stream**<? extends T> b) Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.
long	**count**() Returns the count of elements in this stream.
Stream<T>	**distinct**() Returns a stream consisting of the distinct elements (according to **Object.equals(Object)**) of this stream.
static <T> **Stream**<T>	**empty**() Returns an empty sequential Stream.
Stream<T>	**filter**(**Predicate**<? super T> predicate) Returns a stream consisting of the elements of this stream that match the given predicate.
Optional<T>	**findAny**() Returns an **Optional** describing some element of the stream, or an empty Optional if the stream is empty.
Optional<T>	**findFirst**() Returns an **Optional** describing the first element of this stream, or an empty Optional if the stream is empty.
<R> **Stream**<R>	**flatMap**(**Function**<? super T,? extends **Stream**<? extends R>> mapper) Returns a stream consisting of the results of replacing each element of this stream with the contents of a

	mapped stream produced by applying the provided mapping function to each element.
DoubleStream	flatMapToDouble(Function<? super T,? extends DoubleStream> mapper) Returns an DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
IntStream	flatMapToInt(Function<? super T,? extends IntStream> mapper) Returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
LongStream	flatMapToLong(Function<? super T,? extends LongStream> mapper) Returns an LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
void	forEach(Consumer<? super T> action) Performs an action for each element of this stream.
void	forEachOrdered(Consumer<? super T> action) Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
static <T> Stream<T>	generate(Supplier<T> s) Returns an infinite sequential unordered stream where each element is generated by the provided Supplier.
static <T> Stream<T>	iterate(T seed, UnaryOperator<T> f) Returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.
Stream<T>	limit(long maxSize) Returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.
<R> Stream<R>	map(Function<? super T,? extends R> mapper)

	Returns a stream consisting of the results of applying the given function to the elements of this stream.
DoubleStream	mapToDouble(ToDoubleFunction<? super T> mapper) Returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.
IntStream	mapToInt(ToIntFunction<? super T> mapper) Returns an IntStream consisting of the results of applying the given function to the elements of this stream.
LongStream	mapToLong(ToLongFunction<? super T> mapper) Returns a LongStream consisting of the results of applying the given function to the elements of this stream.
Optional<T>	max(Comparator<? super T> comparator) Returns the maximum element of this stream according to the provided Comparator.
Optional<T>	min(Comparator<? super T> comparator) Returns the minimum element of this stream according to the provided Comparator.
boolean	noneMatch(Predicate<? super T> predicate) Returns whether no elements of this stream match the provided predicate.
static <T> Stream<T>	of(T... values) Returns a sequential ordered stream whose elements are the specified values.
static <T> Stream<T>	of(T t) Returns a sequential Stream containing a single element.
Stream<T>	peek(Consumer<? super T> action) Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.
Optional<T>	reduce(BinaryOperator<T> accumulator) Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.
T	reduce(T identity, BinaryOperator<T> accumulator) Performs a reduction on the elements of this stream, using the provided identity value and

	an **associative** accumulation function, and returns the reduced value.
<U> U	**reduce**(U identity, **BiFunction**<U,? super **T**,U> accumulator, **BinaryOperator**<U> combiner) Performs a **reduction** on the elements of this stream, using the provided identity, accumulation and combining functions.
Stream<**T**>	**skip**(long n) Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream.
Stream<**T**>	**sorted**() Returns a stream consisting of the elements of this stream, sorted according to natural order.
Stream<**T**>	**sorted**(**Comparator**<? super **T**> comparator) Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
Object[]	**toArray**() Returns an array containing the elements of this stream.
<A> A[]	**toArray**(**IntFunction**<A[]> generator) Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.

1. **allMatch(Predicate predicate)** returns whether all elements of this stream match the provided predicate.

```
package streamMethods.allMatch1;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
import streamMethods.Student;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
```

```
 *
 */
public class AllMatchAnyMatchMethodExample1 {

	static Predicate<Student> predicate = s -> s.getAge() > 20;
	/**
	 * Important Note: Please watch video in sequence otherwise you will not
understand
	 * because each video required previous concept
	 */
	public static void main(String[] args) {

		List<Student> list = new ArrayList<>();
		list.add(new Student("Test1", 10));
		list.add(new Student("Test2", 20));
		list.add(new Student("Test3", 30));
		list.add(new Student("Test1", 90));
		list.add(new Student("Test2", 7));
		list.add(new Student("Test3", 100));


		boolean allMatch = list.stream().allMatch(predicate);

		System.out.println(allMatch);

		boolean anyMatch = list.stream().anyMatch(predicate);

		System.out.println(anyMatch);

		boolean startsWith = list.stream().anyMatch(s ->
s.getName().contains("Bhanu"));

		System.out.println(startsWith);

		boolean startsWith_1 = list.stream().anyMatch(s ->
s.getName().contains("Test1"));

		System.out.println(startsWith_1);

	}
}
package streamMethods.allMatch1;

import java.util.List;
import java.util.function.Predicate;
```

```
import java.util.stream.Stream;

import streamMethods.Student;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class AllMatchAnyMatchMethodExample2 {

	static Predicate<Student> predicate = s -> s.getAge() > 20;

	static Predicate<Student> predicate2 = s -> s.getAge() > 200;

	static Predicate<Student> predicate3 = e -> e.getAge() < 10 &&
e.getName().startsWith("T");

	static Predicate<Student> predicate4 = e -> e.getAge() < 20 &&
e.getName().startsWith("T");

	/**
	 * Important Note: Please watch video in sequence otherwise you will not
understand
	 * because each video required previous concept
	 */
	public static void main(String[] args) {

		List<Student> list = Student.getListOfStudents();

		Stream<Student> s1 = list.stream();

		System.out.println(list.stream().allMatch(predicate3)); //false

		System.out.println(list.stream().anyMatch(predicate3)); //false

		System.out.println(list.stream().noneMatch(predicate3)); // true

		System.out.println(list.stream().noneMatch(s ->s.getAge() > 20)); // false

		System.out.println(list.stream().noneMatch(predicate2)); //true
```

```
                System.out.println(list.stream().noneMatch(predicate4)); //false

        }
}

/**
   * Returns whether all elements of this stream match the provided predicate.
   * May not evaluate the predicate on all elements if not necessary for
   * determining the result.  If the stream is empty then {@code true} is
   * returned and the predicate is not evaluated.
   * @param predicate
   * @return {@code true} if either all elements of the stream match the
   * provided predicate or the stream is empty, otherwise {@code false}
   */
2.   boolean allMatch(Predicate<? super T> predicate);

package streamMethods.anyMatch2;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

import streamMethods.Student;

public class AllMatchAnyMatchMethodExample1 {

        static Predicate<Student> predicate = s -> s.getAge() > 20;

        public static void main(String[] args) {

                List<Student> list = new ArrayList<>();
                list.add(new Student("Test1", 10));
                list.add(new Student("Test2", 20));
                list.add(new Student("Test3", 30));
                list.add(new Student("Test1", 90));
                list.add(new Student("Test2", 7));
                list.add(new Student("Test3", 100));

                boolean allMatch = list.stream().allMatch(predicate);

                System.out.println(allMatch);

                boolean anyMatch = list.stream().anyMatch(predicate);

                System.out.println(anyMatch);
```

```
		boolean startsWith = list.stream().anyMatch(s ->s.getName().startsWith("Bhanu"));

		System.out.println(startsWith);

	}
}
```

3. <R, A> R collect(Collector<? super T, A, R> collector);
Stream collect(Collector<? super T,A,R> collector) performs a mutable reduction operation on the elements using a Collector.

```
package streamMethods.collect3;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class CollectMethodExample1 {

	public static void main(String[] args) {

		//collect
		Predicate<String> predicate = x -> !x.isEmpty();

		List<String> list = new ArrayList<String>();
		list.add("DD");
		list.add("CC");
		list.add("ADDDA");
		list.add("AA");
		list.add("");
		list.add("");
		list.add("BB");
		//list.add("BB");
```

```
		Stream<String> li = list.stream();

		li.filter(predicate).collect(Collectors.toList());

		System.out.println(list);

		List<String> filteredList = list.stream().filter(x ->
!x.isEmpty()).collect(Collectors.toList());
		System.out.println(filteredList);

		Set<String> filteredSet = list.stream().filter(x ->
!x.isEmpty()).collect(Collectors.toSet());
		System.out.println(filteredSet);

		LinkedList<String> linkedList = list.stream().filter(x -> !x.isEmpty())
				.collect(Collectors.toCollection(LinkedList::new));

		System.out.println(linkedList.getFirst());

		TreeSet<String> treeSet = list.stream().filter(x -> !x.isEmpty())
				.collect(Collectors.toCollection(TreeSet::new));

		Collection<String> treeSet1 = list.stream().filter(x -> !x.isEmpty())
				.collect(Collectors.toCollection(HashSet::new));

		System.out.println(treeSet);

		Function<String, String> function = (a) -> a;

		list.stream().filter(x -> !x.isEmpty()).collect(Collectors.toMap(function,
function));

		Map<Object, Object> map = list.stream().filter(x ->
!x.isEmpty()).collect(Collectors.toMap(k -> k, k -> k));
		System.out.println(map);

	}

}
package streamMethods.collect3;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
```

```
public class CollectMethodExample2 {

	public static void main(String[] args) {

		List<String> list = new ArrayList<String>();
		list.add("DD");
		list.add("CC");
		list.add("ADDDA");
		list.add("AA");
		list.add("");
		list.add("");
		list.add("BB");

		String result = list.stream().collect(Collectors.joining(", "));

		System.out.println(result);

		String result1 = list.stream().collect(Collectors.joining("##"));

		System.out.println(result1);

	}
}
```

```
  /**
   * Returns the count of elements in this stream.
   * @return the count of elements in this stream
   */
4.   long count();
```

```
package streamMethods.count4;

import java.util.List;

import streamMethods.Student;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example1 {
```

```
	public static void main(String[] args) {

		List<Student> list = Student.getListOfStudents();


		System.out.println(list.stream().count());

		System.out.println(list.stream().filter(x -> x.getAge() > 40).count());

		System.out.println(list.stream().filter(x -> x.getName().contains("t") &&
x.getAge() > 40).count());

	}

}
```

Returns a stream consisting of the distinct elements

5. Stream<T> distinct();

```
package streamMethods.distinct5;

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

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example1 {

	public static void main(String[] args) {

		Consumer<String> consumer = a -> System.out.println(a);

		List<String> list = new ArrayList<String>();
		list.add("AA");
```

```
		list.add("CC");
		list.add("DD");
		list.add("EE");
		list.add("BB");
		list.add("EE");
		list.add("BB");

		Stream<String> distinct = list.stream().distinct();

		distinct.forEach(consumer);

		List<String> newList = list.stream().distinct().collect(Collectors.toList());

		System.out.println(newList);

		List<String> list2 = Arrays.asList("AA", "BB", "CC", "BB", "CC", "AA", "AA");

		long count = list2.stream().distinct().count();

		System.out.println("distinct elements:" + count);

		String output = list2.stream().distinct().collect(Collectors.joining(","));

		System.out.println(output);

		String output1 = list2.stream().distinct().collect(Collectors.joining("###"));

		System.out.println(output1);
	}

}
package streamMethods.distinct5;

import streamMethods.Student;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example2 {
```

```
	public static void main(String[] args) {

		Student.getListOfStudents().stream().forEach(x -> {
			System.out.println(x.getName()+" "+x.getAge());
		});

		System.out.println("----");
		Student.getListOfStudents().stream().distinct().forEach(x -> {
			System.out.println(x.getName()+ " "+x.getAge());
		});
	}

}
```

6. Stream<T> filter(Predicate<? super T> predicate);

Returns a stream consisting of the elements of this stream that match the given predicate.

```
package streamMethods.filter6;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class FilterMethodExample1 {

	public static void main(String[] args) {

		Predicate<String> predicate = (a) -> a.isEmpty();

		List<String> list = Arrays.asList("Test1", "", "Test3", "", "Test5", "Test3");

		long count = list.stream().filter(x -> x.isEmpty()).count();
		System.out.println(count);

		count = list.stream().filter(x -> !x.isEmpty()).count();
		System.out.println(count);
```

```
	}
}
package streamMethods.filter6;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Stream;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class FilterMethodExample2 {

	public static void main(String[] args) {

		List<String> list = Arrays.asList("Test1", "", "Test3", "", "Test5", "Test3");

		Stream<String> newList = list.stream().filter(x -> x.length() > 3);

		long count = list.stream().filter(x -> x.length() > 3).count();

		System.out.println(count);

		Predicate<String> predicate = (a) -> a.startsWith("t");

		count = list.stream().filter(x -> x.startsWith("t")).count();

		System.out.println(count);

		count = list.stream().filter(x -> x.toLowerCase().startsWith("t")).count();

		System.out.println(count);
	}
}
package streamMethods.filter6;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;
```

```
import streamMethods.Student;
/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class FilterMethodExample3 {

	public static void main(String[] args) {

		List<Integer> list = Arrays.asList(20, 11, 28, 4, 5);

		Stream<Integer> newList = list.stream().filter(x -> x % 2 == 0);

		list.stream().filter(x -> x % 2 == 0).forEach(x -> System.out.println(x));

		System.out.println("-------");
		list.stream().filter(x -> x % 2 != 0).forEach(x -> System.out.println(x));


		System.out.println(Student.getListOfStudents().stream().filter(x -> x.getAge()
> 20).count());
	}
}
```

Returns an {@link Optional} describing some element of the stream, or an empty {@code Optional} if the stream is empty.

7. Optional<T> findAny();

```
package streamMethods.findAny7;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;
import java.util.function.Consumer;
import java.util.stream.Stream;

import streamMethods.Student;

/**
 * https://www.youtube.com/c/learnbybhanu
```

```
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class AllFindMethodsExample1 {

	static Comparator<Student> comparator =
Comparator.comparing(Student::getName);

	static Comparator<Student> comparator1 = Comparator.comparing(x -> x.getAge());

	Comparator<Student> comparator2 = Comparator.comparing(Student::getAge);


	public static void main(String[] args) {

		Consumer<String> consumer = s -> System.out.println(s);

		List<String> list = Arrays.asList("AA", "BB", "CC");

		list.stream().filter(x -> x.contains("CC")).findAny().ifPresent(s ->
System.out.println(s));

		list.stream().findAny().ifPresent(s -> System.out.println(s));


		String get = list.stream().findAny().get();

		Optional<String> findAny = list.stream().filter(x -> x.contains("CC")).findAny();



		System.out.println(list.stream().findFirst().get());

		System.out.println("-----max example--------");

		List<Student> list2 = Student.getListOfStudents();

		Optional<Student> max = list2.stream().max(comparator);

		Student get1 = max.get();

		System.out.println(max.get().getName() + " " + max.get().getAge());
```

```
		System.out.println("-----min example--------");

		list2 = Student.getListOfStudents();

		Optional<Student> min = list2.stream().min(comparator);

		System.out.println(min.get().getName() + " " + min.get().getAge());

		System.out.println("-----limit example--------");

		List<Student> list3 = Student.getListOfStudents();

		Stream<Student> limit = list3.stream().limit(4);

		limit.forEach(x -> System.out.println(x.getName() + " " + x.getAge()));

	}
}
package streamMethods.findAny7;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;
import java.util.function.Predicate;
import java.util.stream.Stream;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class FindAnyMethodExample1 {

	public static void main(String[] args) {

		// findAny()

		Predicate<String> predicate = (a) -> a.contains("AA");
```

```
		List<String> list = Arrays.asList("AAA", "BB", "CC", "AAA","AA");

		Stream<String> stream = list.stream();

		Stream<String> filter = stream.filter(predicate);

		Optional<String> findAny = filter.findAny();

		System.out.println(findAny);

		String output = list.stream().filter(e ->
e.startsWith("EE")).findAny().orElse("null");

		System.out.println(output);
	}

}
package streamMethods.findAny7;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class FindAnyMethodExample2 {

	Predicate<String> predicate = e -> e.startsWith("A");
	Consumer<Integer> consumer = a -> System.out.println(a);

	List<String> list = new ArrayList<String>();

	void print() {

		String output = list.stream().filter(predicate).findAny().orElse("null");

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

```
		List<Integer> numList = Arrays.asList(201, 202, 203, 204);

		numList.stream().filter(n -> n % 2 == 0).findAny().ifPresent(e ->
System.out.println(e));

		//numList.stream().filter(n -> n % 2 == 0).findAny().ifPresent(consumer);

		//boolean isPresent = numList.stream().filter(n -> n % 2 ==
0).findAny().isPresent();
	}

	void add() {
		list.add("AA");
		list.add("CC");
		list.add("ADDDA");
	}

	void add_1() {
		list.add(0, "ADADDA");
	}

	public static void main(String[] args) {

		FindAnyMethodExample2 example2 = new FindAnyMethodExample2();

		example2.add();
		System.out.println(example2.list);
		example2.print();
		example2.add_1();
		System.out.println(example2.list);
		example2.print();
	}

}
package streamMethods.findAny7;

import java.util.Arrays;
import java.util.List;
/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
```

```
 *
 */
public class FindAnyMethodExample3 {

	public static void main(String[] args) {

		List<Integer> numList = Arrays.asList(201, 202, 203, 204);

		numList.stream().filter(n -> n % 2 == 0)

		.findAny().ifPresent(e -> System.out.println(e));
	}

}
package streamMethods.findAny7;

import java.util.function.IntConsumer;
import java.util.stream.DoubleStream;
import java.util.stream.IntStream;
import java.util.stream.LongStream;
/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class FindAnyMethodExample4 {

	public static void main(String[] args) {

		IntConsumer consumer = a ->System.out.println(a);

		IntStream intStream = IntStream.of(10, 20, 30, 40);


		intStream.filter(i -> i > 24).findAny().ifPresent(consumer);

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

		LongStream longStream = LongStream.of(300l, 400l, 500l);

		longStream.filter(l -> l < 400).findAny().ifPresent(l -> System.out.println(l));
```

```
		System.out.println("--------");

		DoubleStream doubleStream = DoubleStream.of(100.52, 200.55, 300.66);

		doubleStream.filter(d -> d > 200).findAny().ifPresent(l ->
System.out.println(l));
	}
}
```

8. Optional<T> findFirst();

```
* Returns an {@link Optional} describing the first element of this stream,
   or an empty {@code Optional} if the stream is empty.  If the stream has
   no encounter order, then any element may be returned.
package streamMethods.findFirst8;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.function.Predicate;

import streamMethods.Student;

public class FindFirstMethodExample1 {

	Predicate<String> predicate = x -> x.contains("B");

	public static void main(String[] args) {

		List<String> list = new ArrayList<String>();
		list.add("DD");
		list.add("CC");
		list.add("ADDDA");
		list.add("AA");
		list.add("");
		list.add("");
		list.add("BB");

		Optional<String> result = list.stream().findFirst();

		System.out.println(result.get()); //DD

		result = list.stream().filter(x -> x.contains("B")).findFirst();

		System.out.println(result.get());
```

```
		Student student = Student.getListOfStudents().stream().filter(x ->
x.getAge()>20).findFirst().get();

		System.out.println(student.getAge());

	}

}
```

9. <R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);

Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

```
package streamMethods.flatMap9;

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

public class Example1 {

	public static void main(String[] args) {

		List<Integer> list1 = Arrays.asList(20, 11, 28, 54, 51);

		list1.stream();

		List<Integer> list2 = Arrays.asList(10, 11, 15, 14, 25);

		List<List<Integer>> list = new ArrayList<List<Integer>>();

		list.add(list1);
		list.add(list2);



		System.out.println(list);

		//list.stream().flatMap(li ->li.stream());

		Stream<Object> flatMap = list.stream().flatMap(li -> li.stream());

		//List<Object> newList = flatMap.collect(Collectors.toList());
```

```
		System.out.println(flatMap.collect(Collectors.toList()));


	}

}
package streamMethods.flatMap9;

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

public class Example2 {

	public static void main(String[] args) {

		String[][] d = { { "A", "Apple" }, { "B", "Banana" }, { "O", "Orange" } };


		Stream<String[]> data =
				Stream.of(new String[][] { { "A", "Apple" }, { "B", "Banana" }, {
"O", "Orange" } });

		Stream<String[]> data2 =
				Stream.of(new String[][] { { "P", "Papaya" }, { "C", "Coconut" },
{ "O", "Orange" } });

		Stream<Stream<String[]>> data3 = Stream.of(data,data2);


		data3.forEach(x->{
			x.forEach(y ->{
				System.out.println(y[0]+" "+y[1]);
			});
		});

		Stream<Stream<String[]>> data4 = data3;

		List<String[]> finalData = data4.flatMap(x ->x).collect(Collectors.toList());

		finalData.forEach(x->{
			System.out.println(x[0]+" "+x[1]);
		});
```

```
	}

}
package streamMethods.flatMap9;

import java.util.function.Supplier;
import java.util.stream.Stream;

public class Example3 {

	public static void main(String[] args) {

		Stream<String> stringStream = Stream.of("A", "B", "C", "D");

		//stringStream.forEach(a -> System.out.println(a));

		//stringStream.forEach(a -> System.out.println(a));

		Supplier<Stream<String>> streamSupplier = () -> Stream.of("A", "B", "C",
"D");

		streamSupplier.get().forEach(a -> System.out.println(a));

		streamSupplier.get().forEach(a -> System.out.println(a));

	}

}
package streamMethods.flatMap9;

import java.util.Arrays;
import java.util.List;
import java.util.function.Supplier;
import java.util.stream.Stream;

public class Example4 {

	public static void main(String[] args) {

		List<Integer> list1 = Arrays.asList(20, 11, 28, 54, 51);

		Stream<Integer> newList = list1.stream();

		// it will throw exception
		//newList.forEach(x -> System.out.println(x));
```

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

		List<Integer> list2 = Arrays.asList(20, 11, 28, 54, 51);

		//Stream<Integer> stream = list2.stream();

		Supplier<Stream<Integer>> supplier = () -> list2.stream();

		supplier.get().forEach(x -> System.out.println(x));

		supplier.get().forEach(x -> System.out.println(x));

	}

}
```

Returns an {@code DoubleStream} consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

10. DoubleStream flatMapToDouble(Function<? super T, ? extends DoubleStream> mapper);

```
package streamMethods.flatMapToDouble10;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.ToDoubleFunction;
import java.util.stream.DoubleStream;

public class Example1 {

	public static void main(String[] args) {

		ToDoubleFunction<Integer> doubleFunction = a -> Double.valueOf(a);

		List<Integer> list1 = Arrays.asList(20, 11, 28, 54, 51);

		List<Integer> list2 = Arrays.asList(10, 11, 15, 14, 25);

		List<List<Integer>> list = new ArrayList<List<Integer>>();
		list.add(list1);
		list.add(list2);
```

```
		System.out.println(list);


		DoubleStream newData = list.stream().flatMapToDouble(x ->
x.stream().mapToDouble(doubleFunction));


		newData.forEach(x ->System.out.println(x));

	}
}
package streamMethods.flatMapToDouble10;

import java.util.Arrays;
import java.util.List;
import java.util.stream.DoubleStream;

public class Example2 {

	public static void main(String[] args) {

		List<String> list = Arrays.asList("4.5", "7.7", "9", "90", "50.6");

		DoubleStream doubleStream = DoubleStream.of(70.98, 80.65,
Double.parseDouble("90.8776"));

		DoubleStream newData = list.stream().flatMapToDouble(a ->
DoubleStream.of(Double.parseDouble(a)));

		newData.forEach(System.out::println);

	}
}
package streamMethods.flatMapToDouble10;

import java.util.Arrays;
import java.util.List;
import java.util.stream.DoubleStream;

public class Example3 {

	public static void main(String[] args) {

		List<String> list = Arrays.asList("4.5", "7.7", "9", "90", "50.6");
```

```
		DoubleStream newData = list.stream().flatMapToDouble(a ->
DoubleStream.of(Double.parseDouble(a)));
		newData.forEach(System.out::println);

	}
}
```

11. IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper);

Returns an {@code IntStream} consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

```
package streamMethods.flatMapToInt11;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Function;
import java.util.function.ToIntFunction;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example1 {

	static ToIntFunction<Integer> toIntFunction = a -> a;

	public static void main(String[] args) {

		List<Integer> list1 = Arrays.asList(20, 11, 28, 54, 51);

		List<Integer> list2 = Arrays.asList(10, 11, 15, 14, 25);

		List<List<Integer>> list = new ArrayList<List<Integer>>();
		list.add(list1);
		list.add(list2);

		System.out.println(list);
```

```
		Stream<List<Integer>> stream = list.stream();

		Stream<Integer> stream2 = list1.stream();

		Function<Integer, Integer> function = (a) -> a * 2;

		IntStream intData = list.stream().flatMapToInt(x ->
x.stream().mapToInt(toIntFunction));


		List<Integer> listData = intData.boxed().collect(Collectors.toList());

		System.out.println(listData);

	}

}
package streamMethods.flatMapToInt11;

import java.util.ArrayList;
import java.util.List;
import java.util.function.ToIntFunction;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

import streamMethods.Student;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example2 {

	static ToIntFunction<Integer> toIntFunction = a -> a;

	public static void main(String[] args) {

		List<Student> list1 = Student.getListOfStudents();

		List<Student> list2 = Student.getListOfStudents();
```

```
		List<List<Student>> list = new ArrayList<List<Student>>();
		list.add(list1);
		list.add(list2);

		IntStream intData = list.stream().flatMapToInt(x -> x.stream().mapToInt(y ->
y.getAge()));

		List<Integer> listData = intData.boxed().collect(Collectors.toList());

		System.out.println(listData);

	}

}
package streamMethods.flatMapToInt11;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.function.ToIntFunction;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

import streamMethods.Student;
/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example3 {

	static ToIntFunction<Integer> toIntFunction = a ->a;


	public static void main(String[] args) {

		List<Student> list1 = Student.getListOfStudents();

		List<Student> list2 = Student.getListOfStudents();

		List<List<Student>> list = new ArrayList<List<Student>>();
		list.add(list1);
```

```
		list.add(list2);

		IntStream intData = list.stream().flatMapToInt(x ->x.stream().mapToInt(y ->
y.getAge()));

		Set<Integer> listData = intData.boxed().collect(Collectors.toSet());

		System.out.println(listData);

	}

}
```

12. LongStream flatMapToLong(Function<? super T, ? extends LongStream> mapper);
Returns an {@code LongStream} consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

```
package streamMethods.flatMapToLong12;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.LongStream;
import java.util.stream.Stream;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example1 {


	// Always watch Video in sequence otherwise you will loose concept
	public static void main(String[] args) {

		List<Integer> list1 = Arrays.asList(20, 11, 28, 54, 51);

		List<Integer> list2 = Arrays.asList(10, 11, 15, 14, 25);

		List<List<Integer>> list = new ArrayList<List<Integer>>();
		list.add(list1);
```

```
		list.add(list2);

		Stream<List<Integer>> stream;

		System.out.println(list);
		LongStream intData = list.stream().flatMapToLong(x ->
x.stream().mapToLong(y ->y*2));


		List<Long> listData = intData.boxed().collect(Collectors.toList());

		System.out.println(listData);

	}

}
package streamMethods.flatMapToLong12;

import java.util.ArrayList;
import java.util.List;
import java.util.function.ToIntFunction;
import java.util.stream.Collectors;
import java.util.stream.LongStream;

import streamMethods.Student;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example2 {

	static ToIntFunction<Integer> toIntFunction = a -> a;

	// Always watch Video in sequence otherwise you will loose concept
	public static void main(String[] args) {

		List<Student> list1 = Student.getListOfStudents();

		List<Student> list2 = Student.getListOfStudents();
```

```
		List<List<Student>> list = new ArrayList<List<Student>>();
		list.add(list1);
		list.add(list2);

		LongStream intData = list.stream().flatMapToLong(x ->
x.stream().mapToLong(y -> y.getAge()));

		List<Long> listData = intData.boxed().collect(Collectors.toList());

		System.out.println(listData);

	}

}
```

13. Optional<T> max(Comparator<? super T> comparator);

```
package streamMethods.max13;

import java.util.Comparator;
import java.util.List;
import java.util.Optional;

import streamMethods.Student;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example1 {

	static Comparator<Student> comparator = (a, b) ->
a.getName().compareTo(b.getName());

	// Make sure you are watching video in sequence. because these topics are depends
on previous
	// Comparator topic
	public static void main(String[] args) {

		List<Student> list1 = Student.getListOfStudents();

		Optional<Student> maxData = list1.stream()
```

```
                .max(Comparator.comparing(Student::getAge));

        System.out.println(maxData.get().getAge()); //50

        Optional<Student> maxData1 = list1.stream().max(comparator);

        System.out.println(maxData1.get().getName()); // Test5

        Optional<Student> maxData2 = list1.stream().max((a,b) ->
a.getName().compareTo(b.getName()));

        System.out.println(maxData2.get().getAge()+" "+maxData2.get().getName());
    }

}
```

14. Optional<T> min(Comparator<? super T> comparator);

```
package streamMethods.min14;

import java.util.Comparator;
import java.util.List;
import java.util.Optional;

import streamMethods.Student;

/**
 * https://www.youtube.com/c/learnbybhanu
 * https://www.udemy.com/javabybhanu
 * https://www.facebook.com/learnbybhanupratap/
 *
 * @author Bhanu Pratap Singh
 *
 */
public class Example1 {

    static Comparator<Student> comparator = Comparator.comparing(Student::getAge);

    // Make sure you are watching video in sequence. because these topics are depends
on previous
    // Comparator topic
    public static void main(String[] args) {

        List<Student> list1 = Student.getListOfStudents();
```

```
		Optional<Student> minData = list1.stream().min(Comparator.comparing(Student::getAge));

		System.out.println(minData.get().getAge());

		Optional<Student> minData1 = list1.stream().min(Comparator.comparing(Student::getName));

		System.out.println(minData1.get().getName());

		Optional<Student> minData2 = list1.stream().min((a,b) -> a.getName().compareTo(b.getName()));

		System.out.println(minData2.get().getAge()+" "+minData2.get().getName());
	}

}
```

CompletableFuture

What is CompletableFuture?

A **CompltableFuture** is used for asynchronous programming. Asynchronous programming means writing non-blocking code. It runs a task on a separate thread than the main application thread and notifies the main thread about its progress, completion or failure.

In this way, the main thread does not block or wait for the completion of the task. Other tasks execute in parallel. Parallelism improves the performance of the program.

A CompletableFuture is a class in Java. It belongs to java.util.cocurrent package. It implements CompletionStage and Future interface.

- **Class CompletableFuture<T>**
- java.lang.Object
-
    - java.util.concurrent.CompletableFuture<T>
- **All Implemented Interfaces:**
    - CompletionStage<T>, Future<T>

Constructor and Description
CompletableFuture() Creates a new incomplete CompletableFuture.

Modifier and Type	Method and Description
CompletableFuture<Void>	acceptEither(CompletionStage<? extends T> other, Consumer<? super T> action) Returns a new CompletionStage that, when either this or the other given stage complete normally, is executed with the corresponding result as argument to the supplied action.
CompletableFuture<Void>	acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action) Returns a new CompletionStage that, when either this or the other given stage complete normally, is executed using this stage's default asynchronous execution facility, with the corresponding result as argument to the supplied action.
CompletableFuture<Void>	acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action, Executor executor) Returns a new CompletionStage that, when either this or the other given stage complete normally, is executed using the supplied executor, with the corresponding result as argument to the supplied function.
static CompletableFuture<Void>	allOf(CompletableFuture<?>... cfs) Returns a new CompletableFuture that is completed when all of the given CompletableFutures complete.
static CompletableFuture<Object>	anyOf(CompletableFuture<?>... cfs) Returns a new CompletableFuture that is completed when any of the given CompletableFutures complete, with the same result.
<U> CompletableFuture<U>	applyToEither(CompletionStage<? extends T> other, Function<? super T,U> fn) Returns a new CompletionStage that, when either this or the other given stage complete normally, is executed with the corresponding result as argument to the supplied function.
<U> CompletableFuture<U>	applyToEitherAsync(CompletionStage<? extends T> other, Function<? super T,U> fn)

	Returns a new CompletionStage that, when either this or the other given stage complete normally, is executed using this stage's default asynchronous execution facility, with the corresponding result as argument to the supplied function.
<U> CompletableFuture<U>	applyToEitherAsync(CompletionStage<? extends T> other, Function<? super T,U> fn, Executor executor) Returns a new CompletionStage that, when either this or the other given stage complete normally, is executed using the supplied executor, with the corresponding result as argument to the supplied function.
boolean	cancel(boolean mayInterruptIfRunning) If not already completed, completes this CompletableFuture with a **CancellationException**.
boolean	complete(T value) If not already completed, sets the value returned by **get()** and related methods to the given value.
static <U> CompletableFuture<U>	completedFuture(U value) Returns a new CompletableFuture that is already completed with the given value.
boolean	completeExceptionally(Throwable ex) If not already completed, causes invocations of **get()** and related methods to throw the given exception.
CompletableFuture<T>	exceptionally(Function<Throwable,? extends T> fn) Returns a new CompletableFuture that is completed when this CompletableFuture completes, with the result of the given function of the exception triggering this CompletableFuture's completion when it completes exceptionally; otherwise, if this CompletableFuture completes normally, then the returned CompletableFuture also completes normally with the same value.
T	get() Waits if necessary for this future to complete, and then returns its result.
T	get(long timeout, TimeUnit unit) Waits if necessary for at most the given time for this future to complete, and then returns its result, if available.
T	getNow(T valueIfAbsent)

	Returns the result value (or throws any encountered exception) if completed, else returns the given valueIfAbsent.
int	**getNumberOfDependents**() Returns the estimated number of CompletableFutures whose completions are awaiting completion of this CompletableFuture.
<U> **CompletableFuture**<U>	**handle**(**BiFunction**<? super **T**,**Throwable**,? extends U> fn) Returns a new CompletionStage that, when this stage completes either normally or exceptionally, is executed with this stage's result and exception as arguments to the supplied function.
<U> **CompletableFuture**<U>	**handleAsync**(**BiFunction**<? super **T**,**Throwable**,? extends U> fn) Returns a new CompletionStage that, when this stage completes either normally or exceptionally, is executed using this stage's default asynchronous execution facility, with this stage's result and exception as arguments to the supplied function.
<U> **CompletableFuture**<U>	**handleAsync**(**BiFunction**<? super **T**,**Throwable**,? extends U> fn, **Executor** executor) Returns a new CompletionStage that, when this stage completes either normally or exceptionally, is executed using the supplied executor, with this stage's result and exception as arguments to the supplied function.
boolean	**isCancelled**() Returns true if this CompletableFuture was cancelled before it completed normally.
boolean	**isCompletedExceptionally**() Returns true if this CompletableFuture completed exceptionally, in any way.
boolean	**isDone**() Returns true if completed in any fashion: normally, exceptionally, or via cancellation.
T	**join**() Returns the result value when complete, or throws an (unchecked) exception if completed exceptionally.
void	**obtrudeException**(**Throwable** ex)

	Forcibly causes subsequent invocations of method **get()** and related methods to throw the given exception, whether or not already completed.
void	**obtrudeValue**(T value) Forcibly sets or resets the value subsequently returned by method **get()** and related methods, whether or not already completed.
CompletableFuture<Void>	**runAfterBoth**(CompletionStage<?> other, Runnable action) Returns a new CompletionStage that, when this and the other given stage both complete normally, executes the given action.
CompletableFuture<Void>	**runAfterBothAsync**(CompletionStage<?> other, Runnable action) Returns a new CompletionStage that, when this and the other given stage complete normally, executes the given action using this stage's default asynchronous execution facility.
CompletableFuture<Void>	**runAfterBothAsync**(CompletionStage<?> other, Runnable action, Executor executor) Returns a new CompletionStage that, when this and the other given stage complete normally, executes the given action using the supplied executor.
CompletableFuture<Void>	**runAfterEither**(CompletionStage<?> other, Runnable action) Returns a new CompletionStage that, when either this or the other given stage complete normally, executes the given action.
CompletableFuture<Void>	**runAfterEitherAsync**(CompletionStage<?> other, Runnable action) Returns a new CompletionStage that, when either this or the other given stage complete normally, executes the given action using this stage's default asynchronous execution facility.
CompletableFuture<Void>	**runAfterEitherAsync**(CompletionStage<?> other, Runnable action, Executor executor) Returns a new CompletionStage that, when either this or the other given stage complete normally, executes the given action using the supplied executor.
static CompletableFuture<Void>	**runAsync**(Runnable runnable)

	Returns a new CompletableFuture that is asynchronously completed by a task running in the **ForkJoinPool.commonPool()** after it runs the given action.
static **CompletableFuture**<**Void**>	**runAsync**(**Runnable** runnable, **Executor** executor) Returns a new CompletableFuture that is asynchronously completed by a task running in the given executor after it runs the given action.
static <U> **CompletableFuture**<U>	**supplyAsync**(**Supplier**<U> supplier) Returns a new CompletableFuture that is asynchronously completed by a task running in the **ForkJoinPool.commonPool()** with the value obtained by calling the given Supplier.
static <U> **CompletableFuture**<U>	**supplyAsync**(**Supplier**<U> supplier, **Executor** executor) Returns a new CompletableFuture that is asynchronously completed by a task running in the given executor with the value obtained by calling the given Supplier.
CompletableFuture<**Void**>	**thenAccept**(**Consumer**<? super **T**> action) Returns a new CompletionStage that, when this stage completes normally, is executed with this stage's result as the argument to the supplied action.
CompletableFuture<**Void**>	**thenAcceptAsync**(**Consumer**<? super **T**> action) Returns a new CompletionStage that, when this stage completes normally, is executed using this stage's default asynchronous execution facility, with this stage's result as the argument to the supplied action.
CompletableFuture<**Void**>	**thenAcceptAsync**(**Consumer**<? super **T**> action, **Executor** executor) Returns a new CompletionStage that, when this stage completes normally, is executed using the supplied Executor, with this stage's result as the argument to the supplied action.
<U> **CompletableFuture**<**Void**>	**thenAcceptBoth**(**CompletionStage**<? extends U> other, **BiConsumer**<? super **T**,? super U> action) Returns a new CompletionStage that, when this and the other given stage both complete normally, is executed with the two results as arguments to the supplied action.
<U> **CompletableFuture**<**Void**>	**thenAcceptBothAsync**(**CompletionStage**<? extends U> other, **BiConsumer**<? super **T**,? super U> action)

	Returns a new CompletionStage that, when this and the other given stage complete normally, is executed using this stage's default asynchronous execution facility, with the two results as arguments to the supplied action.
<U> CompletableFuture<Void>	thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action, Executor executor) Returns a new CompletionStage that, when this and the other given stage complete normally, is executed using the supplied executor, with the two results as arguments to the supplied function.
<U> CompletableFuture<U>	thenApply(Function<? super T,? extends U> fn) Returns a new CompletionStage that, when this stage completes normally, is executed with this stage's result as the argument to the supplied function.
<U> CompletableFuture<U>	thenApplyAsync(Function<? super T,? extends U> fn) Returns a new CompletionStage that, when this stage completes normally, is executed using this stage's default asynchronous execution facility, with this stage's result as the argument to the supplied function.
<U> CompletableFuture<U>	thenApplyAsync(Function<? super T,? extends U> fn, Executor executor) Returns a new CompletionStage that, when this stage completes normally, is executed using the supplied Executor, with this stage's result as the argument to the supplied function.
<U,V> CompletableFuture<V>	thenCombine(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn) Returns a new CompletionStage that, when this and the other given stage both complete normally, is executed with the two results as arguments to the supplied function.
<U,V> CompletableFuture<V>	thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn) Returns a new CompletionStage that, when this and the other given stage complete normally, is executed using this stage's default asynchronous execution facility, with the two results as arguments to the supplied function.
<U,V> CompletableFuture<V>	thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn, Executor executor)

	Returns a new CompletionStage that, when this and the other given stage complete normally, is executed using the supplied executor, with the two results as arguments to the supplied function.
<U> CompletableFuture<U>	thenCompose(Function<? super T,? extends CompletionStage<U>> fn) Returns a new CompletionStage that, when this stage completes normally, is executed with this stage as the argument to the supplied function.
<U> CompletableFuture<U>	thenComposeAsync(Function<? super T,? extends CompletionStage<U>> fn) Returns a new CompletionStage that, when this stage completes normally, is executed using this stage's default asynchronous execution facility, with this stage as the argument to the supplied function.
<U> CompletableFuture<U>	thenComposeAsync(Function<? super T,? extends CompletionStage<U>> fn, Executor executor) Returns a new CompletionStage that, when this stage completes normally, is executed using the supplied Executor, with this stage's result as the argument to the supplied function.
CompletableFuture<Void>	thenRun(Runnable action) Returns a new CompletionStage that, when this stage completes normally, executes the given action.
CompletableFuture<Void>	thenRunAsync(Runnable action) Returns a new CompletionStage that, when this stage completes normally, executes the given action using this stage's default asynchronous execution facility.
CompletableFuture<Void>	thenRunAsync(Runnable action, Executor executor) Returns a new CompletionStage that, when this stage completes normally, executes the given action using the supplied Executor.
CompletableFuture<T>	toCompletableFuture() Returns this CompletableFuture.
String	toString() Returns a string identifying this CompletableFuture, as well as its completion state.
CompletableFuture<T>	whenComplete(BiConsumer<? super T,? super Throwable> action)

	Returns a new CompletionStage with the same result or exception as this stage, that executes the given action when this stage completes.
CompletableFuture<T>	whenCompleteAsync(BiConsumer<? super T,? super Throwable> action) Returns a new CompletionStage with the same result or exception as this stage, that executes the given action using this stage's default asynchronous execution facility when this stage completes.
CompletableFuture<T>	whenCompleteAsync(BiConsumer<? super T,? super Throwable> action, Executor executor) Returns a new CompletionStage with the same result or exception as this stage, that executes the given action using the supplied Executor when this stage completes.

CompletableFuture is an extension to Java's Future API which was introduced in Java 5.You can create a CompletableFuture simply by using the following no-arg constructor

```
CompletableFuture<String> completableFuture = new CompletableFuture<String>();
```

All the clients who want to get the result of this CompletableFuture can call `CompletableFuture.get()` method

```
String result  = completableFuture.get();
```

The `get()` method blocks until the Future is complete. So, the above call will block forever because the Future is never completed.
We can use `CompletableFuture.complete()` method to manually complete a Future

```
completableFuture.complete("Future Result");
```

Running asynchronous using `runAsync()`

If you want to run some background task asynchronously and don't want to return anything from the task, then you can use `CompletableFuture.runAsync()` method. It takes a Runnable object and returns `CompletableFuture<Void>`.

```java
package completableFuture;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class Example2 {

    public static void main(String[] args) throws InterruptedException, ExecutionException {

        CompletableFuture<Void> future = CompletableFuture.runAsync(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    throw new IllegalStateException(e);
                }
                System.out.println("separate thread than the main thread.");
            }
        });

        future.get();

    }
}
```

```java
package completableFuture;

import java.util.concurrent.CompletableFuture;

public class Example3 {

    public static void main(String[] args) throws InterruptedException, ExecutionException {

        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                throw new IllegalStateException(e);
            }
            System.out.println("in a separate thread than the main thread.");
        });

    }
}
```

```
CompletionStage<Void> thenRun(Runnable action)
CompletionStage<Void> thenRunAsync(Runnable action)
CompletionStage<Void> thenRunAsync(Runnable action, Executor executor)
```

```java
package completableFuture.runAsync1;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;
import java.util.concurrent.ExecutionException;

public class Example4 {

        public static void main(String[] args) throws InterruptedException,
ExecutionException {
```

```
		CompletionStage<Void> cf = CompletableFuture.runAsync(() ->
performTask_1("first task"));

		cf = cf.thenRun(() -> performTask_2("second task"));

		cf = cf.thenRunAsync(() -> performTask_3("third task"));

		cf = cf.thenRunAsync(() -> performTask_3("third task"));

		((CompletableFuture<Void>) cf).join();

		System.out.println("main exiting");
	}

	private static void performTask_1(String task) {

		System.out.printf("task: %s, thread: %s%n", task,
Thread.currentThread().getName());
		try {
			Thread.sleep(2000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}

	private static void performTask_2(String task) {

		System.out.printf("task: %s, thread: %s%n", task,
Thread.currentThread().getName());
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}

	private static void performTask_3(String task) {

		System.out.printf("task: %s, thread: %s%n", task,
Thread.currentThread().getName());
		try {
			Thread.sleep(900);
		} catch (InterruptedException e) {
			e.printStackTrace();
```

```
		}
	}
}
task: first task, thread: ForkJoinPool.commonPool-worker-9
task: second task, thread: ForkJoinPool.commonPool-worker-9
task: third task, thread: ForkJoinPool.commonPool-worker-9
task: third task, thread: ForkJoinPool.commonPool-worker-9
main exiting
```

CompletableFuture, by default, uses an Executor created by ForkJoinPool.commonPool() to achieve the parallelism .
In which case, a new Thread is created to run each task.
In above example the two async tasks ran in the same thread (created by the default executor) i.e. ForkJoinPool.commonPool.
For a non-async stage, the task is executed in the thread that completes the previous stage.
Rather than relying on default Executor we can also supply our own executor by using overloaded method runAsync(Runnable runnable, Executor executor)

```
package completableFuture.runAsync1;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Example5 {

	public static void main(String[] args) throws InterruptedException,
ExecutionException {

		ExecutorService executor = Executors.newFixedThreadPool(3);

		CompletionStage<Void> cf = CompletableFuture.runAsync(() ->
performTask_1("first task"),executor);

		cf = cf.thenRun(() -> performTask_2("second task"));

		cf = cf.thenRunAsync(() -> performTask_3("third task"),executor);

		cf = cf.thenRunAsync(() -> performTask_3("fourth task"),executor);

		((CompletableFuture<Void>) cf).join();
```

```
		System.out.println("main exiting");
	}

	private static void performTask_1(String task) {

		System.out.printf("task: %s, thread: %s%n", task,
Thread.currentThread().getName());
		try {
			Thread.sleep(2000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.printf("task: %s, thread: %s%n", task,
Thread.currentThread().getName());
	}

	private static void performTask_2(String task) {

		System.out.printf("task: %s, thread: %s%n", task,
Thread.currentThread().getName());
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.printf("task: %s, thread: %s%n", task,
Thread.currentThread().getName());
	}

	private static void performTask_3(String task) {

		System.out.printf("task: %s, thread: %s%n", task,
Thread.currentThread().getName());
		try {
			Thread.sleep(900);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.printf("task: %s, thread: %s%n", task,
Thread.currentThread().getName());
	}
}
task: first task, thread: pool-1-thread-1
task: second task, thread: pool-1-thread-1
```

task: third task, thread: pool-1-thread-2
task: fourth task, thread: pool-1-thread-3
main exiting

Run asynchronously and want to return use supplyAsync()

```java
package completableFuture;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.function.Supplier;

public class Example4 {

    public static void main(String[] args) throws InterruptedException, ExecutionException {

        CompletableFuture<String> future = CompletableFuture.supplyAsync(new Supplier<String>() {
            @Override
            public String get() {
                return "Running asynchronous";
            }
        });
        String result = future.get();
        System.out.println(result);
    }
}
```

```java
package completableFuture.supplyAsync;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class Example5 {

    public static void main(String[] args) throws InterruptedException, ExecutionException {

        System.out.println("I am main thread");

        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> sum());

        Integer result = future.get();

        System.out.println(result);

        System.out.println(future.isDone());

    }

    public static int sum(int a, int b) {
        return a + b;
    }

    public static int sum() {
        int sum = 0;
        for (int i = 0; i < 1000; i++) {
            sum = sum + i;
            System.out.println("running Asynchronous="+i);
        }
        return sum;
    }
}
```

thenApply(Function<? super T,? extends U> fn)
Returns a new CompletionStage that, when this stage completes normally, is executed with this stage's result as the argument to the supplied function.

```
package completableFuture.thenApply;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class Example6 {

	public static void main(String[] args) throws InterruptedException,
ExecutionException {

		System.out.println("I am main thread");

		CompletableFuture<Integer> completableFuture1 =
CompletableFuture.supplyAsync(() -> {
			return sum(2, 4);
		});

		CompletableFuture<Integer> completableFuture2 =
completableFuture1.thenApply((a) -> {
			int sum = 0;
			for (int i = 0; i < a; i++) {
				sum = sum + i;
			}
			return sum;
		});

		Integer result = completableFuture2.get();

		System.out.println(result);

		System.out.println(completableFuture2.isDone());

	}

	public static int sum(int a, int b) {
		return a + b;
	}

}
package completableFuture.thenApply;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class Example7 {
```

```
	public static void main(String[] args) throws InterruptedException,
ExecutionException {

		System.out.println("I am main thread");

		CompletableFuture<Integer> completableFuture1 =
CompletableFuture.supplyAsync(() -> {
			return sum(2, 4);
		});

		CompletableFuture<Integer> completableFuture2 =
completableFuture1.thenApply((a) -> {
			int sum = 0;
			for (int i = 0; i < a; i++) {
				sum = sum + i;
			}
			return sum;
		});

		CompletableFuture<Integer> completableFuture3 =
completableFuture2.thenApply((a) -> {
			int sum = 0;
			for (int i = 0; i < a; i++) {
				sum = sum + i;
			}
			return sum;
		});

		Integer result = completableFuture3.get();

		System.out.println(result);

		System.out.println(completableFuture3.isDone());

	}

	public static int sum(int a, int b) {
		return a + b;
	}

}
package completableFuture.thenApply;

import java.util.concurrent.CompletableFuture;
```

```
import java.util.concurrent.ExecutionException;

public class Example8 {

	public static void main(String[] args) throws InterruptedException,
ExecutionException {

		CompletableFuture<String> completableFuture =
CompletableFuture.supplyAsync(() -> {
			try {
				Thread.sleep(200);
			} catch (InterruptedException e) {
				throw new IllegalStateException(e);
			}
			return "Bhanu";
		}).thenApply(name -> {
			return "Hello " + name;
		}).thenApply(request -> {
			return request + " make more Java video";
		});

		System.out.println(completableFuture.get());

	}
}
```

acceptEither(**CompletionStage**<? extends **T**> other, **Consumer**<? super **T**> action)
Returns a new CompletionStage that, when either this or the other given stage complete normally, is executed with the corresponding result as argument to the supplied action.

```
package completableFuture.acceptEither;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class Example1 {

	public static void main(String[] args) throws InterruptedException,
ExecutionException {

		System.out.println("I am main thread");

		CompletableFuture<Integer> completableFuture1 =
CompletableFuture.supplyAsync(() -> {
			try {
				return sum(2, 4);
			} catch (InterruptedException e) {
```

```
                        e.printStackTrace();
                }
                return null;
        });

        CompletableFuture<Integer> completableFuture2 =
CompletableFuture.supplyAsync(() -> {
                int sum = 0;
                for (int i = 0; i < 3; i++) {
                        sum = sum + i;
                }
                return sum;
        });

        CompletableFuture<Void> result =
completableFuture1.acceptEither(completableFuture2, x -> System.out.println(x));

        System.out.println(result);

        System.out.println(completableFuture2.isDone());

    }

    public static int sum(int a, int b) throws InterruptedException {
        Thread.sleep(2000);
        return a + b;
    }

}
package completableFuture.acceptEither;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class Example2 {

    public static void main(String[] args) throws InterruptedException,
ExecutionException {

        System.out.println("I am main thread");

        CompletableFuture<Integer> completableFuture1 =
CompletableFuture.supplyAsync(() -> {
                try {
                        return sum(2, 4);
```

```
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                return null;
        });

        CompletableFuture<Integer> completableFuture2 =
CompletableFuture.supplyAsync(() -> {
                int sum = 0;
                for (int i = 0; i < 3; i++) {
                        sum = sum + i;
                }
                return sum;
        });

        completableFuture1.get();

        CompletableFuture<Void> result =
completableFuture1.acceptEither(completableFuture2, x -> System.out.println(x));

        System.out.println(result);

        System.out.println(completableFuture2.isDone());

    }

    public static int sum(int a, int b) throws InterruptedException {
        Thread.sleep(2000);
        return a + b;
    }

}
```